

Meshtastic Python

API

Complete guide to the Meshtastic Python library: connection, messaging, automation, and API reference.

- [Getting Started with the Meshtastic Python Library](#)
- [Automating Meshtastic: Practical Scripts](#)
- [Meshtastic Python API Reference](#)

Getting Started with the Meshtastic Python Library

The Meshtastic Python library (`meshtastic` on PyPI) provides a clean API for connecting to Meshtastic devices, reading their state, sending messages, and reacting to received packets via callbacks. This page covers installation, all three connection methods (serial, TCP, BLE), and complete working code examples for the most common operations.

Installation

```
# Core library (serial + TCP + BLE all included)
pip install meshtastic

# Development / latest from GitHub
pip install git+https://github.com/meshtastic/python.git
```

The library requires Python 3.9+. As of recent releases, `bleak` (which provides BLE support) is a mandatory core dependency and is always installed - the old `[ble]` extra no longer exists, so `pip install meshtastic` gives you serial, TCP, and BLE. Its core runtime dependencies include `pyserial`, `protobuf`, `pypubsub`, `bleak`, `tabulate`, `requests`, `pyyaml`, and `packaging`.

Connecting via Serial

```
import meshtastic
import meshtastic.serial_interface

# Auto-detect first available Meshtastic device
iface = meshtastic.serial_interface.SerialInterface()

# Specify a port explicitly
iface = meshtastic.serial_interface.SerialInterface(devPath="/dev/ttyUSB0")
iface = meshtastic.serial_interface.SerialInterface(devPath="COM4") # Windows
```

When a `SerialInterface` is created, the library:

1. Opens the serial port at 115200 baud.
2. Performs a handshake to confirm the device is running Meshtastic firmware.
3. Downloads the full node database and device configuration from the node.

The constructor blocks until the download completes (typically 1 - 3 seconds). After that the `iface` object is fully populated.

Connecting via TCP

```
import meshtastic.tcp_interface

# Connect to a node by IP address
iface = meshtastic.tcp_interface.TCPInterface(hostname="192.168.1.42")

# Connect by mDNS hostname
iface = meshtastic.tcp_interface.TCPInterface(hostname="meshtastic.local")
```

TCP connection is useful for nodes deployed without USB access. The default port is 4403; you can override it with the `portNumber` parameter. The TCP interface requires the node to have Wi-Fi enabled and the TCP API enabled in Radio Configuration.

Connecting via BLE

```
import meshtastic.ble_interface

# Scan for available BLE devices.
# BLEInterface.scan() is a synchronous @staticmethod - do NOT await it.
devices = meshtastic.ble_interface.BLEInterface.scan()
for d in devices:
    print(d.name, d.address)

# Connect by device name or MAC address
iface = meshtastic.ble_interface.BLEInterface("Meshtastic_abcd")
```

Reading Device Info

```
import meshtastic
import meshtastic.serial_interface
import json

iface = meshtastic.serial_interface.SerialInterface()

# My node info
my_info = iface.getMyNodeInfo()
print("My node number:", my_info["num"])
print("My user:", my_info.get("user", {}).get("longName"))

# Full local config as a protobuf object.
# localConfig lives on the Node object, not on the interface.
config = iface.localNode.localConfig
print("Hop limit:", config.lora.hop_limit)
print("Region:", config.lora.region)

# Channel config (channels are held on the local Node)
for ch in iface.localNode.channels:
    print(f"Channel {ch.index}: role={ch.role}, name={ch.settings.name or 'default'}")

iface.close()
```

Listing Nodes

```
import meshtastic
import meshtastic.serial_interface
import time

iface = meshtastic.serial_interface.SerialInterface()

nodes = iface.nodes # dict keyed by "<hex_node_id>"

for node_id, node in nodes.items():
```

```

user = node.get("user", {})
pos = node.get("position", {})
metrics = node.get("deviceMetrics", {})
last_heard = node.get("lastHeard", 0)
age_minutes = (time.time() - last_heard) / 60 if last_heard else None

print(f"Node: {node_id}")
print(f" Name: {user.get('longName', 'Unknown')}")
print(f" Short name: {user.get('shortName', '???')}")
print(f" Hardware: {user.get('hwModel', 'Unknown')}")
if pos.get("latitudeI"):
    lat = pos["latitudeI"] / 1e7
    lon = pos["longitudeI"] / 1e7
    print(f" Position: {lat:.5f}, {lon:.5f} alt={pos.get('altitude', 0)}m")
if metrics:
    print(f" Battery: {metrics.get('batteryLevel', '?')}%")
    print(f" Chan util: {metrics.get('channelUtilization', '?')}%")
    print(f" SNR: {node.get('snr', 'N/A')} dB")
if age_minutes is not None:
    print(f" Last heard: {age_minutes:.1f} min ago")
print()

iface.close()

```

Sending a Text Message

```

import meshtastic
import meshtastic.serial_interface

iface = meshtastic.serial_interface.SerialInterface()

# Broadcast to all nodes on the primary channel
iface.sendText("Hello mesh!")

# Send to a specific node (direct message)
iface.sendText("Hello from Python", destinationId="!aabbccdd")

# Send on a secondary channel (index 1)

```

```
iface.sendText("Private message", channelIndex=1)

# Fire-and-forget with wantAck set (ACK is NOT checked here - see callbacks below)
import time
iface.sendText("Test with ACK", wantAck=True)
time.sleep(5) # gives the packet time to send; does not confirm delivery

iface.close()
```

The `sendText` method returns immediately after queuing the packet. The example above is fire-and-forget: setting `wantAck=True` and sleeping does *not* confirm delivery. For confirmed delivery you must listen for the ACK packet via a callback (see below).

A "direct message" is only end-to-end encrypted (PKC) when both nodes run firmware v2.5 or newer; otherwise the DM is encrypted only with the channel PSK and is readable by anyone sharing that channel.

Receiving Messages with Callbacks

The library uses `pypubsub` for its event system. Subscribe to topics before creating the interface, or add subscriptions after connection. The main topics are:

- `meshtastic.receive` - all decoded packets from the mesh.
- `meshtastic.receive.text` - text message packets only.
- `meshtastic.receive.position` - position packets.
- `meshtastic.receive.telemetry` - telemetry packets.
- `meshtastic.receive.user` - node-info (user) packets.
- `meshtastic.connection.established` - fired when connection and download complete.
- `meshtastic.connection.lost` - fired on disconnect.

```
import meshtastic
import meshtastic.serial_interface
from pubsub import pub
import time

def on_connect(interface, topic=pub.AUTO_TOPIC):
    # Called when the library finishes downloading node data.
    print(f"Connected! Nodes in mesh: {len(interface.nodes)}")

def on_receive(packet, interface):
```

```

# Called for every decoded packet from the mesh.
port_num = packet.get("decoded", {}).get("portnum", "UNKNOWN")
from_id = packet.get("fromId", "?")
to_id = packet.get("toId", "?")
hops = packet.get("hopStart", 0) - packet.get("hopLimit", 0)

if port_num == "TEXT_MESSAGE_APP":
    text = packet["decoded"].get("text", "")
    print(f"[TEXT] {from_id} -> {to_id} ({hops} hop(s)): {text}")

elif port_num == "POSITION_APP":
    pos = packet["decoded"].get("position", {})
    lat = pos.get("latitudeI", 0) / 1e7
    lon = pos.get("longitudeI", 0) / 1e7
    print(f"[POSITION] {from_id}: {lat:.5f}, {lon:.5f}")

elif port_num == "TELEMETRY_APP":
    tel = packet["decoded"].get("telemetry", {})
    dm = tel.get("deviceMetrics", {})
    # channelUtilization is often absent on a given packet - guard before formatting
    cu = dm.get("channelUtilization")
    cu_str = f"{cu:.1f}%" if cu is not None else "N/A"
    print(f"[TELEMETRY] {from_id}: battery={dm.get('batteryLevel')}, "
          f"chan_util={cu_str}")

def on_text_receive(packet, interface):
    # Called only for TEXT_MESSAGE_APP packets.
    text = packet.get("decoded", {}).get("text", "")
    print(f"Text message received: {text!r}")

pub.subscribe(on_connect, "meshtastic.connection.established")
pub.subscribe(on_receive, "meshtastic.receive")
pub.subscribe(on_text_receive, "meshtastic.receive.text")

iface = meshtastic.serial_interface.SerialInterface()

try:
    print("Listening for packets. Press Ctrl-C to exit.")
    while True:

```

```
    time.sleep(1)
except KeyboardInterrupt:
    pass
finally:
    iface.close()
```

Complete Minimal Example: Echo Bot

```
# echo_bot.py -- Meshtastic echo bot
# Responds to any text message with "Echo: <original message>"

import meshtastic
import meshtastic.serial_interface
from pubsub import pub
import time

iface = None

def on_receive(packet, interface):
    decoded = packet.get("decoded", {})
    if decoded.get("portnum") == "TEXT_MESSAGE_APP":
        text = decoded.get("text", "")
        from_id = packet.get("fromId")
        my_id = interface.getMyNodeInfo()["user"]["id"]
        # Don't echo our own messages
        if from_id != my_id:
            interface.sendText(f"Echo: {text}", destinationId=from_id)
            print(f"Echoed to {from_id}: {text!r}")

pub.subscribe(on_receive, "meshtastic.receive")
iface = meshtastic.serial_interface.SerialInterface()
print("Echo bot running. Press Ctrl-C to stop.")
try:
    while True:
        time.sleep(1)
except KeyboardInterrupt:
```

```
    pass
finally:
    if iface:
        iface.close()
```

Closing the Interface

Always call `iface.close()` when done. This cleanly shuts down the background receiver thread and closes the serial/TCP/BLE connection. Failing to close the interface can leave the serial port locked, preventing other tools (the app, another script) from connecting.

```
# Best practice: use a try/finally block
iface = meshtastic.serial_interface.SerialInterface()
try:
    # ... your code ...
    pass
finally:
    iface.close()
```

Automating Meshtastic: Practical Scripts

The Meshtastic Python library enables powerful automation workflows. This page provides four complete, ready-to-use scripts: a position logger, a message forwarder to Telegram, a battery monitor with alerts, and an automated network health reporter. Each script is self-contained and includes setup instructions.

Script 1: Position Logger to CSV

Logs GPS position updates from all mesh nodes to a CSV file in real time. Useful for tracking mobile assets, recording deployment surveys, or building a trace of network coverage over time. **Privacy note:** this logger writes every observed node's GPS coordinates to a plaintext file. Treat that file as sensitive location data and protect it accordingly.

```
# position_logger.py
# Logs all position packets from the Meshtastic mesh to a CSV file.
#
# Usage:
# pip install meshtastic
# python position_logger.py [--port /dev/ttyUSB0] [--output positions.csv]
import argparse
import csv
import datetime
import os
import sys
import time

import meshtastic
import meshtastic.serial_interface
import meshtastic.tcp_interface
from pubsub import pub

OUTPUT_FILE = "positions.csv"
```

```

CSV_FIELDS = ["timestamp", "node_id", "long_name", "short_name",
              "latitude", "longitude", "altitude_m", "speed_kmh",
              "heading_deg", "snr_db", "hop_count"]

def get_interface(args):
    if args.host:
        return meshtastic.tcp_interface.TCPInterface(hostname=args.host)
    port = args.port or None
    return meshtastic.serial_interface.SerialInterface(devPath=port)

def main():
    parser = argparse.ArgumentParser(description="Log Meshtastic positions to CSV")
    parser.add_argument("--port", help="Serial port (e.g. /dev/ttyUSB0 or COM4)")
    parser.add_argument("--host", help="TCP hostname or IP address")
    parser.add_argument("--output", default=OUTPUT_FILE, help="Output CSV file path")
    args = parser.parse_args()

    file_exists = os.path.isfile(args.output)
    outfile = open(args.output, "a", newline="", encoding="utf-8")
    writer = csv.DictWriter(outfile, fieldnames=CSV_FIELDS)
    if not file_exists:
        writer.writeheader()

    iface_ref = [None] # mutable container for the interface

    def on_position(packet, interface):
        decoded = packet.get("decoded", {})
        pos = decoded.get("position", {})
        if not pos.get("latitudeI"):
            return # no valid fix

        node_id = packet.get("fromId", "unknown")
        nodes = interface.nodes or {}
        node_info = nodes.get(node_id, {})
        user = node_info.get("user", {})

        lat = pos["latitudeI"] / 1e7

```

```

lon = pos["longitudeI"] / 1e7
alt = pos.get("altitude", 0)
speed = pos.get("groundSpeed", 0)
# NOTE: verify the groundTrack/heading scaling in the Position protobuf for your
# firmware version before dividing by 100; the field's units have varied.
heading = pos.get("groundTrack", 0)
snr = node_info.get("snr", "")
# Guard against missing hopStart/hopLimit; record None rather than a bogus 0.
if "hopStart" in packet and "hopLimit" in packet:
    hops = packet["hopStart"] - packet["hopLimit"]
else:
    hops = None

row = {
    "timestamp": datetime.datetime.now(datetime.timezone.utc).isoformat(),
    "node_id": node_id,
    "long_name": user.get("longName", ""),
    "short_name": user.get("shortName", ""),
    "latitude": f"{lat:.7f}",
    "longitude": f"{lon:.7f}",
    "altitude_m": alt,
    "speed_kmh": speed * 3.6 if speed else 0,
    "heading_deg": heading / 100 if heading else 0,
    "snr_db": snr,
    "hop_count": hops,
}
writer.writerow(row)
outfile.flush()
print(f"[{row['timestamp']}] {node_id} ({row['long_name']}) "
      f"@ {lat:.5f},{lon:.5f} alt={alt}m")

pub.subscribe(on_position, "meshtastic.receive.position")

iface = get_interface(args)
iface_ref[0] = iface
print(f"Logging positions to {args.output}. Press Ctrl-C to stop.")

try:
    while True:

```

```
time.sleep(1)
except KeyboardInterrupt:
    pass
finally:
    iface.close()
    outfile.close()
    print("Logger stopped.")

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

Script 2: Message Forwarder to Telegram

Forwards all text messages received on the mesh to a Telegram chat. Requires a Telegram bot token (create one via [@BotFather](#)) and a chat ID. This is a popular pattern for monitoring a community mesh from a smartphone without needing Bluetooth or Wi-Fi proximity to a node.

```
# mesh_to_telegram.py
# Forwards Meshtastic text messages to a Telegram chat.
#
# Setup:
# pip install meshtastic requests
# Set environment variables:
# TELEGRAM_TOKEN=<your bot token>
# TELEGRAM_CHAT_ID=<chat id, e.g. -1001234567890>
# python mesh_to_telegram.py [--host 192.168.1.42]
import os
import sys
import time
import datetime
import requests
import meshtastic
import meshtastic.serial_interface
import meshtastic.tcp_interface
from pubsub import pub
```

```

TELEGRAM_TOKEN = os.environ.get("TELEGRAM_TOKEN", "")
TELEGRAM_CHAT_ID = os.environ.get("TELEGRAM_CHAT_ID", "")

if not TELEGRAM_TOKEN or not TELEGRAM_CHAT_ID:
    print("ERROR: Set TELEGRAM_TOKEN and TELEGRAM_CHAT_ID environment variables.")
    sys.exit(1)

def send_telegram(text: str):
    # Send a message to the configured Telegram chat.
    url = f"https://api.telegram.org/bot{TELEGRAM_TOKEN}/sendMessage"
    data = {"chat_id": TELEGRAM_CHAT_ID, "text": text, "parse_mode": "HTML"}
    try:
        resp = requests.post(url, json=data, timeout=10)
        resp.raise_for_status()
    except requests.RequestException as exc:
        print(f"Telegram send failed: {exc}")

def on_receive(packet, interface):
    decoded = packet.get("decoded", {})
    if decoded.get("portnum") != "TEXT_MESSAGE_APP":
        return

    text = decoded.get("text", "")
    from_id = packet.get("fromId", "unknown")
    to_id = packet.get("toId", "^all")
    hops = (packet["hopStart"] - packet["hopLimit"])
    if "hopStart" in packet and "hopLimit" in packet else None)
    timestamp = datetime.datetime.now(datetime.timezone.utc).strftime("%H:%M:%S UTC")

    nodes = interface.nodes or {}
    node_info = nodes.get(from_id, {})
    long_name = node_info.get("user", {}).get("longName", from_id)

    dest_str = "broadcast" if to_id in ("^all", "4294967295") else to_id
    tg_msg = (
        f"Mesh Message [{timestamp}]

```

```

"
    f"From:
{long_name}
    ({from_id})
"
    f"To: {dest_str} | {hops} hop(s)
"
    f"
{text}"
)

print(f"Forwarding to Telegram: {text!r} from {long_name}")
send_telegram(tg_msg)

import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--host", help="TCP hostname or IP")
parser.add_argument("--port", help="Serial port")
args = parser.parse_args()

pub.subscribe(on_receive, "meshtastic.receive.text")

if args.host:
    iface = meshtastic.tcp_interface.TCPInterface(hostname=args.host)
else:
    iface = meshtastic.serial_interface.SerialInterface(devPath=args.port)

print(f"Forwarding mesh messages to Telegram chat {TELEGRAM_CHAT_ID}. Press Ctrl-C to stop.")
try:
    while True:
        time.sleep(1)
except KeyboardInterrupt:
    pass
finally:
    iface.close()

```

Script 3: Battery Monitor with Alerts

Monitors battery levels of all nodes in the mesh and sends a text-message alert over the mesh when any node's battery falls below a configurable threshold. Useful for solar-powered relay nodes where low battery means imminent network degradation.

```
# battery_monitor.py
# Sends a mesh alert when any node battery drops below a threshold.
#
# Usage:
# python battery_monitor.py [--threshold 20] [--interval 300]
import argparse
import time
import meshtastic
import meshtastic.serial_interface
from pubsub import pub

ALERT_COOLDOWN = {} # node_id -> last_alert_time to avoid spamming

def check_battery(iface, threshold, interval):
    # Periodically check all nodes and alert on low battery.
    while True:
        time.sleep(interval)
        nodes = iface.nodes or {}
        now = time.time()

        for node_id, node_data in nodes.items():
            metrics = node_data.get("deviceMetrics", {})
            battery = metrics.get("batteryLevel")

            if battery is None:
                continue # no telemetry available

            # Skip nodes that are charging (level > 100 indicates USB power on some firmware)
            if battery > 100:
                continue
```

```

if battery < threshold:
    last_alert = ALERT_COOLDOWN.get(node_id, 0)
    # Only alert once per hour per node
    if now - last_alert > 3600:
        long_name = node_data.get("user", {}).get("longName", node_id)
        alert_msg = (
            f"LOW BATTERY ALERT: {long_name} ({node_id}) "
            f"is at {battery}%"
        )
        print(alert_msg)
        # Broadcast alert on the primary channel
        iface.sendText(alert_msg)
        ALERT_COOLDOWN[node_id] = now

def main():
    parser = argparse.ArgumentParser(description="Meshtastic battery monitor")
    parser.add_argument("--threshold", type=int, default=20,
        help="Battery percentage threshold for alerts (default: 20)")
    parser.add_argument("--interval", type=int, default=300,
        help="Check interval in seconds (default: 300)")
    parser.add_argument("--port", help="Serial port")
    parser.add_argument("--host", help="TCP hostname")
    args = parser.parse_args()

    if args.host:
        import meshtastic.tcp_interface
        iface = meshtastic.tcp_interface.TCPInterface(hostname=args.host)
    else:
        import meshtastic.serial_interface
        iface = meshtastic.serial_interface.SerialInterface(devPath=args.port)

    print(f"Battery monitor started. Alert threshold: {args.threshold}%. "
        f"Check interval: {args.interval}s.")

    import threading
    t = threading.Thread(target=check_battery,
        args=(iface, args.threshold, args.interval),
        daemon=True)
    t.start()

```

```

try:
while True:
time.sleep(1)
except KeyboardInterrupt:
pass
finally:
iface.close()

if __name__ == "__main__":
main()

```

Script 4: Automated Network Health Reporter

Generates a periodic network health report and either prints it to the console or sends it as a mesh broadcast. Summarizes node count, online/offline status, average SNR, channel utilization, and identifies any nodes not heard in the last configured window.

```

# health_reporter.py
# Generates periodic Meshtastic network health reports.
#
# Usage:
# python health_reporter.py [--interval 3600] [--broadcast]
import argparse
import time
import datetime
import meshtastic
import meshtastic.serial_interface
import meshtastic.tcp_interface

def generate_report(iface, offline_threshold_minutes=60) -> str:
    # Build a health report string from current node data.
    nodes = iface.nodes or {}
    now = time.time()
    total = len(nodes)

```

```

online = []
offline = []
snr_values = []
util_values = []

for node_id, node_data in nodes.items():
    last_heard = node_data.get("lastHeard", 0)
    age_min = (now - last_heard) / 60 if last_heard else None
    long_name = node_data.get("user", {}).get("longName", node_id)
    metrics = node_data.get("deviceMetrics", {})
    snr = node_data.get("snr")
    util = metrics.get("channelUtilization")

    if snr is not None:
        snr_values.append(snr)
    if util is not None:
        util_values.append(util)

    if age_min is not None and age_min < offline_threshold_minutes:
        online.append((long_name, age_min, snr))
    else:
        offline.append((long_name, age_min))

avg_snr = sum(snr_values) / len(snr_values) if snr_values else None
avg_util = sum(util_values) / len(util_values) if util_values else None

lines = [
    f"=== Mesh Health Report {datetime.datetime.now(datetime.timezone.utc).strftime('%Y-%m-%d %H:%M UTC')} ===",
    f"Total nodes: {total} | Online (<{offline_threshold_minutes}m): {len(online)}",
    f" | Offline: {len(offline)}",
]

if avg_util is not None:
    # These bands are custom to this script. The app's conventional channel-utilization
    # bands are roughly green < 25% / orange 25-50% / red > 50%.
    health = "OK" if avg_util < 15 else "WARN" if avg_util < 25 else "HIGH"
    lines.append(f"Avg channel utilization: {avg_util:.1f}% [{health}]")

if avg_snr is not None:

```

```

lines.append(f"Avg SNR (last heard): {avg_snr:.1f} dB")

if offline:
lines.append(f"Offline nodes ({len(offline)}):")
for name, age in offline:
age_str = f"{age:.0f}m ago" if age is not None else "never"
lines.append(f" - {name}: last heard {age_str}")

return "
".join(lines)

def main():
    parser = argparse.ArgumentParser(description="Meshtastic network health reporter")
    parser.add_argument("--interval", type=int, default=3600,
        help="Report interval in seconds (default: 3600)")
    parser.add_argument("--broadcast", action="store_true",
        help="Broadcast report as mesh text message")
    parser.add_argument("--offline", type=int, default=60,
        help="Minutes without a packet to consider a node offline (default: 60)")
    parser.add_argument("--port", help="Serial port")
    parser.add_argument("--host", help="TCP hostname")
    args = parser.parse_args()

    if args.host:
        iface = meshtastic.tcp_interface.TCPInterface(hostname=args.host)
    else:
        iface = meshtastic.serial_interface.SerialInterface(devPath=args.port)

    print("Health reporter running. First report in", args.interval, "seconds.")

    try:
        while True:
            time.sleep(args.interval)
            report = generate_report(iface, offline_threshold_minutes=args.offline)
            print(report)
            print()

    if args.broadcast:
        # Mesh text messages cap at ~200 bytes of application payload; send a summary

```

```

nodes = iface.nodes or {}
now = time.time()
online = sum(
    1 for n in nodes.values()
    if (now - n.get("lastHeard", 0)) / 60 < args.offline
)
metrics = [n.get("deviceMetrics", {}).get("channelUtilization")
            for n in nodes.values()]
if n.get("deviceMetrics", {}).get("channelUtilization") is not None]
avg_util = sum(metrics) / len(metrics) if metrics else 0
short_report = (
    f"Mesh status: {online}/{len(nodes)} online, "
    f"chan util {avg_util:.1f}%"
)
iface.sendText(short_report)
print(f"Broadcast: {short_report!r}")
except KeyboardInterrupt:
    pass
finally:
    iface.close()

if __name__ == "__main__":
    main()

```

Meshtastic Python API Reference

This page documents the key classes, methods, and patterns of the Meshtastic Python library. It covers the interface classes (the `MeshInterface` base class plus three concrete transports: Serial, TCP, and BLE), the event system, protobuf message types, and error handling. Refer to the library's source on GitHub and the API docs at python.meshtastic.org for the full, authoritative method signatures as the API evolves with each firmware release.

Interface Class Hierarchy

```
MeshInterface (base class, meshtastic.mesh_interface)
├─ SerialInterface (meshtastic.serial_interface)
├─ TCPInterface (meshtastic.tcp_interface)
└─ BLEInterface (meshtastic.ble_interface)
```

The three concrete transports inherit the common message-sending, node-state, and configuration API from `MeshInterface`, but they are not identical: each subclass adds transport-specific constructor parameters (e.g. `SerialInterface` `devPath`, `TCPInterface` `hostname`, `BLEInterface` `address`) and some transport-specific helper methods. The shared send/read/config surface is inherited from the base class.

MeshInterface (Base Class)

Constructor Parameters

The `MeshInterface` base constructor takes exactly three parameters: `__init__(self, debugOut=None, noProto=False, noNodes=False)`. There is no `configTimeout` parameter.

Parameter	Type	Description
<code>debugOut</code>	file-like	Stream for debug output. Default None.

Parameter	Type	Description
<code>noProto</code>	bool	Skip protocol handshake (testing only). Default False.
<code>noNodes</code>	bool	Skip downloading node database on connect. Default False.

Key Properties

Note: configuration and channel data physically live on the local `Node` object, reachable via `iface.localNode` (e.g. `iface.localNode.localConfig`, `iface.localNode.moduleConfig`, `iface.localNode.channels`). The rows below describe those objects' shapes.

Property / Path	Type	Description
<code>nodes</code>	<code>Optional[dict[str, dict]]</code>	All known nodes keyed by <code>"!<hex_id>"</code> (None before connect). Each value is typically a dict with <code>num</code> , <code>user</code> , <code>position</code> , <code>snr</code> , <code>lastHeard</code> , <code>deviceMetrics</code> — though <code>position</code> / <code>deviceMetrics</code> are not guaranteed present on every node.
<code>myInfo</code>	protobuf <code>MyNodeInfo</code> (Optional)	Basic info about the local node. It is a protobuf object, not a plain dict; the node-number field is <code>my_node_num</code> .
<code>metadata</code>	protobuf <code>DeviceMetadata</code>	Firmware version, hardware model, capability flags (verify it is populated post-connect in your installed library version before relying on it).
<code>localNode.localConfig</code>	protobuf <code>LocalConfig</code>	Full device configuration: <code>.lora</code> , <code>.device</code> , <code>.position</code> , <code>.power</code> , <code>.network</code> , <code>.display</code> , <code>.bluetooth</code> , <code>.security</code> .
<code>localNode.moduleConfig</code>	protobuf <code>LocalModuleConfig</code>	Module configs: <code>.mqtt</code> , <code>.serial</code> , <code>.telemetry</code> , <code>.neighbor_info</code> , <code>.ambient_lighting</code> , etc.
<code>localNode.channels</code>	<code>list[protobuf Channel]</code>	List of channel objects (accessed via the local Node, not a public <code>localChannels</code> property on the interface). Each has <code>.index</code> , <code>.role</code> , <code>.settings</code> (name, PSK, etc.).

Key Methods

```

# Send a text message
iface.sendText(
    text: str,
    destinationId: Union[int, str] = BROADCAST_ADDR, # "^all" or "!aabbccdd"
    wantAck: bool = False,
    wantResponse: bool = False,
    onResponse: Optional[Callable] = None,
    channelIndex: int = 0,
    portNum: PortNum = portnums_pb2.PortNum.TEXT_MESSAGE_APP,
)

# Send raw data (any port number)
iface.sendData(
    data: bytes,
    destinationId: Union[int, str] = BROADCAST_ADDR,
    portNum: int = portnums_pb2.PortNum.PRIVATE_APP,
    wantAck: bool = False,
    wantResponse: bool = False,
    onResponse: Optional[Callable] = None,
    onResponseAckPermitted: bool = False,
    channelIndex: int = 0,
    hopLimit: Optional[int] = None,
    pkiEncrypted: bool = False,
    publicKey: Optional[bytes] = None,
    priority: MeshPacket.Priority = MeshPacket.Priority.RELIABLE,
)

# Get my node info (None if not connected)
iface.getMyNodeInfo() -> Optional[Dict]
# The node number is read from getMyNodeInfo()["num"] or iface.myInfo.my_node_num
# (there is no getMyNodeNum() method on MeshInterface).

# Send a traceroute request (hopLimit is required, no default)
iface.sendTraceRoute(
    dest: Union[int, str],
    hopLimit: int,
    channelIndex: int = 0,
)

```

```

# Node-database and config writes are methods on the Node object, NOT on
# the base MeshInterface. Obtain the local Node via iface.localNode (or
# iface.getNode("^local")):

# Remove a node from the local database
iface.localNode.removeNode(nodeId: str) # nodeId: "!aabbccdd"
# (or use the CLI: meshtastic --remove-node ...)

# Write a config change to the device
# (after modifying iface.localNode.localConfig protobuf object)
iface.localNode.writeConfig(config_name: str)
# config_name is one of: "device", "position", "power", "network",
# "display", "lora", "bluetooth", "security"

# Write module config
iface.localNode.writeModuleConfig(config_name: str)
# config_name is one of: "mqtt", "serial", "external_notification",
# "store_forward", "range_test", "telemetry",
# "canned_message", "audio", "remote_hardware",
# "neighbor_info", "ambient_lighting", "detection_sensor"

# Cleanly close the connection
iface.close() -> None

```

SerialInterface

```

class meshtastic.serial_interface.SerialInterface(
    devPath: Optional[str] = None, # e.g. "/dev/ttyUSB0", "COM4"; None = auto-detect
    debugOut=None,
    noProto: bool = False,
    connectNow: bool = True,
    noNodes: bool = False,
)

```

Auto-detection scans `/dev/ttyUSB*`, `/dev/ttyACM*`, `/dev/cu.usbserial-*`, and Windows COM ports for a device that responds to the Meshtastic handshake.

TCPInterface

```
class meshtastic.tcp_interface.TCPInterface(  
    hostname: str, # IP address or mDNS hostname  
    debugOut=None,  
    noProto: bool = False,  
    connectNow: bool = True,  
    portNumber: int = 4403, # TCP port; default is 4403  
    noNodes: bool = False,  
)
```

BLEInterface

```
class meshtastic.ble_interface.BLEInterface(  
    address: str, # Device name or MAC address  
    noProto: bool = False,  
    debugOut=None,  
    noNodes: bool = False,  
)  
  
# Static method: scan for devices (synchronous, do NOT await it)  
BLEInterface.scan() -> list[BLEDevice]
```

Event System (pypubsub Topics)

The library uses `pypubsub` for all asynchronous notifications. Subscribe before or after creating the interface; the subscription takes effect for all future events.

```
from pubsub import pub  
  
# All received packets  
pub.subscribe(callback, "meshtastic.receive")  
  
# Filtered by port (the trailing segment is the protocol name)  
pub.subscribe(callback, "meshtastic.receive.text") # TEXT_MESSAGE_APP
```

```

pub.subscribe(callback, "meshtastic.receive.position") # POSITION_APP
pub.subscribe(callback, "meshtastic.receive.user") # NODEINFO_APP
pub.subscribe(callback, "meshtastic.receive.telemetry") # TELEMETRY_APP (derived from the
# protocol registry name; verify against your library's dispatch path)
pub.subscribe(callback, "meshtastic.receive.routing") # ROUTING_APP; note ACK/NAK delivery
# to callbacks depends on the response (ackPermitted) handlers, not solely this topic
pub.subscribe(callback, "meshtastic.receive.data.<portnum>") # <portnum> is the integer
# port or PortNum enum name (e.g. meshtastic.receive.data.1); no "portnum_" prefix

# Connection lifecycle
pub.subscribe(callback, "meshtastic.connection.established") # on connect + data download
pub.subscribe(callback, "meshtastic.connection.lost") # on disconnect
pub.subscribe(callback, "meshtastic.node.updated") # when node DB entry changes

```

Callback Signatures

```

# For meshtastic.receive.*
def on_receive(packet: dict, interface: MeshInterface) -> None:
    ...

# For meshtastic.connection.established
def on_connect(interface: MeshInterface, topic=pub.AUTO_TOPIC) -> None:
    ...

# For meshtastic.connection.lost
def on_lost(interface: MeshInterface, topic=pub.AUTO_TOPIC) -> None:
    ...

```

Packet Dictionary Structure

```

{
  "id": 4012345678, # uint32 packet ID
  "from": 2864434397, # sender node number (integer)
  "fromId": "!aabbccdd", # sender node ID (hex string)
  "to": 4294967295, # destination (4294967295 = broadcast)
  "toId": "^all", # destination as string
  "hopLimit": 3, # remaining hops

```

```
"hopStart": 3, # original hop limit
"rxSnr": 4.25, # SNR at receiver (dB)
"rxRssi": -98, # RSSI at receiver (dBm)
"rxTime": 1714010000, # Unix time packet was received
"viaMqtt": False, # True if packet came via MQTT gateway
"channel": 0, # channel index
"decoded": {
  "portnum": "TEXT_MESSAGE_APP",
  "text": "Hello mesh!", # present for TEXT_MESSAGE_APP
  "position": { ... }, # present for POSITION_APP
  "telemetry": { ... }, # present for TELEMETRY_APP
  "user": { ... }, # present for NODEINFO_APP
  "routing": { ... }, # present for ROUTING_APP
},
"raw": <protobuf MeshPacket object>
}
```

Protobuf Message Types

The library exposes raw protobuf objects for advanced use. The most important generated modules in the package are:

Module	Key Message Types
meshtastic.mesh_pb2	MeshPacket, NodeInfo, User, Position, Data, Routing
meshtastic.config_pb2	Config, Config.DeviceConfig, Config.LoRaConfig, Config.PositionConfig, Config.NetworkConfig
meshtastic.module_config_pb2	ModuleConfig, ModuleConfig.MQTTConfig, ModuleConfig.TelemetryConfig
meshtastic.telemetry_pb2	Telemetry, DeviceMetrics, EnvironmentMetrics, PowerMetrics
meshtastic.portnums_pb2	PortNum enum (TEXT_MESSAGE_APP, POSITION_APP, TELEMETRY_APP, etc.)
meshtastic.channel_pb2	Channel, ChannelSettings

Example: Modifying a Config Setting

```

import meshtastic
import meshtastic.serial_interface

iface = meshtastic.serial_interface.SerialInterface()

# Get the local Node object (config reads/writes happen on the Node)
node = iface.getNode("^local")

# Read current value
print("Current hop limit:", node.localConfig.lora.hop_limit)

# Modify the protobuf object directly
node.localConfig.lora.hop_limit = 2
node.localConfig.lora.tx_power = 20 # dBm

# Write back to the device (triggers a reboot if radio settings changed)
node.writeConfig("lora")

iface.close()

```

Error Handling Patterns

```

import meshtastic
import meshtastic.serial_interface
from meshtastic.mesh_interface import MeshInterface

# Handle connection failures
try:
    iface = meshtastic.serial_interface.SerialInterface()
except Exception as exc:
    print(f"Failed to connect: {exc}")

# Common causes:
# - No Meshtastic device found on any serial port
# - Permission denied (Linux: add user to dialout group)
# - Device busy (app connected via BLE using same serial port implicitly)
raise

```

```

# Handle send timeout (device not responding)
import meshtastic.mesh_interface as mi
try:
    iface.sendText("test", destinationId="!aabbccdd", wantAck=True)
except Exception as exc:
    print(f"Send failed: {exc}")

# Handle TCP connection refused
try:
    import meshtastic.tcp_interface
    iface = meshtastic.tcp_interface.TCPInterface("192.168.1.99")
except ConnectionRefusedError:
    print("TCP connection refused. Is the node on Wi-Fi with TCP API enabled?")

# Handle device disconnect mid-session
from pubsub import pub

def on_lost(interface, topic=pub.AUTO_TOPIC):
    print("Connection to device lost. Attempting reconnect in 5 seconds...")
    import time, threading
    def reconnect():
        time.sleep(5)
        try:
            new_iface = meshtastic.serial_interface.SerialInterface()
            print("Reconnected successfully.")
        except Exception as e:
            print(f"Reconnect failed: {e}")
            threading.Thread(target=reconnect, daemon=True).start()

pub.subscribe(on_lost, "meshtastic.connection.lost")

```

Thread Safety

The Meshtastic Python library spawns a background reader thread on connection. All pubsub callbacks are invoked from this thread. If your callback modifies shared state, use a `threading.Lock` to prevent race conditions. Outbound publishing is handled by an internal deferred-execution thread; if you call send methods from multiple threads, guard your own shared state and consult the current `mesh_interface.py` source for the library's locking behavior rather than assuming a particular guarantee.

```

import threading

lock = threading.Lock()
message_log = []

def on_receive(packet, interface):
    decoded = packet.get("decoded", {})
    if decoded.get("portnum") == "TEXT_MESSAGE_APP":
        with lock:
            message_log.append({
                "time": packet.get("rxTime"),
                "from": packet.get("fromId"),
                "text": decoded.get("text"),
            })

```

Version Compatibility

Library Version	Firmware Compatibility	Notes
2.3.x	Firmware 2.3.x	Refined traceroute and neighbor-info handling (neighbor-info was introduced in firmware 2.2.0; traceroute predates 2.3).
2.4.x	Firmware 2.4.x	BLE improvements and config refinements.
2.5.x	Firmware 2.5.x	Public Key Cryptography (X25519) for direct messages and admin session keys; detection sensor module; power telemetry; new config fields.

Always match the library version to your firmware major/minor version. Mismatches can cause silent config field drops or protobuf decode errors. Run `pip install --upgrade meshtastic` after each firmware update.