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
`debugOut`	file-like	Stream for debug output. Default None.

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

Revision #3
Created 2026-05-03 05:39:41 UTC by Mesh America Admin
Updated 2026-06-09 01:59:54 UTC by Mesh America Admin