RouMBLE BLE Mesh Simulator Documentation

RouMBLE BLE Mesh Simulator Documentation

This document describes both the RouMBLE BLE Mesh simulation library and its accompanying PyQt-based GUI application...

RouMBLE BLE Mesh Simulator Documentation

This document describes both the **RouMBLE BLE Mesh simulation library** and its accompanying **PyQt-based GUI application**. It covers installation, architecture, module/function references, configuration options, and usage examples.

Table of Contents

---

1. Overview

RouMBLE BLE Mesh Simulator is a Python-based discrete-event simulator implementing the RouMBLE sink-oriented routing protocol over a simulated BLE mesh network. It uses **SimPy** to model packet floods (BOM, RMS) and node behavior, and **PyQt5** for a real-time, zoomable GUI that visualizes:

- The network topology (static mesh + mobile "phone" nodes + sinks)
- Packet exchanges (highlights links briefly when packets traverse them)
- Node details (neighbors, routing tables, energy)
- Performance metrics (PDR, latency, hops, control overhead)
- An event log of packet-level actions

Users can also inject "external" packets (SMS→RMS or broadcast RMS) via a GUI form. Static mesh nodes are placed randomly (while avoiding overlapping circles) within a central sub-area and guaranteed at least one neighbor; "phone" nodes roam randomly.

---

2. Installation

1. **Clone or copy the project folder** into a local directory (e.g., `roumble_sim/`).
2. Create (or activate) a Python 3.7+ virtual environment.
3. Install dependencies with pip:
4. `pip install simpy PyQt5`
5. Ensure the following files are present in the same directory:
6. `packets.py`
7. `node.py`
8. `logger.py`
9. `engine.py`
10. `gui.py`

```
11. main.py
12. requirements.txt
```

13. To launch the application:

```
14. python main.py
```

A PyQt window appears with topology on the left and controls on the right.

---

3. Project Structure

```
roumble_sim/
├── packets.py      # Packet definitions (Packet class)
├── node.py         # Node behavior (Node class)
├── logger.py       # Logging & metrics (Logger class)
├── engine.py       # Topology creation & SimPy engine
(SimulationEngine class)
├── gui.py          # PyQt GUI (TopologyView, MainWindow)
├── main.py         # Entry point (creates engine, GUI, starts Qt loop)
└── requirements.txt # List of pip-installable dependencies
```

---

4. Simulator Architecture

4.1 Discrete-Event Core (SimPy)

- **SimPy Environment** (`self.env`) manages a priority queue of time-tagged events (transmission delays, mobility timers, BOM intervals, RMS generation).
- Each **`Node`** is a SimPy "process" (Python generator):
    - Sinks run a **BOM process** (`_sink_bom_process`) that floods BOMs every `Node.BOM_INTERVAL`.
    - Non-sink mesh nodes run an **RMS generation process** (`_generate_rms_process`) that creates RMS data every ~exp(1/`Node.RMS_INTERVAL`).
    - Phone nodes run a **mobility process** (`_mobility_process`) that moves them every `Node.MOVE_INTERVAL`.

4.2 Node Model (`node.py`)

- **Constructor**:
- `def __init__(self, env, node_id, is_sink, is_mobile, init_pos, engine)`
    - `env`: SimPy Environment
    - `node_id`: Unique integer
    - `is_sink`: True/False
    - `is_mobile`: True/False (phones only)
    - `init_pos`: `(x, y)` initial coordinates
    - `engine`: Reference to `SimulationEngine` for shared state & logger
- **Attributes**:

  - `self.ipv6`: Mock IPv6 string (e.g. `"fe80::1:00a"`)
  - `self.x, self.y`: Current coordinates
  - `self.neighbors`: List of neighboring Node objects (recomputed whenever `engine.update_neighbors()` is called)
  - **Routing state**:
    - `self.routing_table`: `Dict[sink_id ↦ (next_hop, hop_distance, seq_seen)]`
    - `self.best_seq`: `Dict[sink_id ↦ highest BOM seq seen]`
    - `self.seen_rms`: `set[(origin, seq)]` for duplicate suppression
  - **Counters/Energy**:
    - `self.seq_num`: In-node packet counter
    - `self.energy`: Float (initialized to 100.0; not decremented by default)
    - `self.tx_count`, `self.rx_count`: Number of transmit/receive events
- **Processes**:
  1. **`_sink_bom_process()`**:
     - Runs in an infinite loop:
       - `yield env.timeout(Node.BOM_INTERVAL)`
       - Increments `seq_num`, builds a `Packet('BOM', src=self.id, sink_id=self.id, seq=..., hop_count=0, origin=self.id, timestamp=env.now)`
       - Records a control-sent event (`logger.record_control_sent()`) and logs (`logger.log_event(env.now, 'BOM', self.id, -1)`)
       - Calls `self._broadcast(bom_packet)`
  2. **`_generate_rms_process()`** (non-sink only):
     - Loop:
       - `interval = random.expovariate(1.0 / Node.RMS_INTERVAL)`
       - `yield env.timeout(interval)`
       - `seq_num += 1`, create `Packet('RMS', src=self.id, sink_id=None, seq=..., hop_count=Node.MAX_HOPS, origin=self.id, timestamp=env.now)`
       - Record data-sent (`logger.record_data_sent()`) and log (`logger.log_event(env.now, 'RMS_GEN', self.id, -1)`)
       - Call `self._send_rms(rms_packet)`
  3. **`_mobility_process()`** (phones only):
     - Loop:
       - `yield env.timeout(Node.MOVE_INTERVAL)`
       - Random angle + distance → update `self.x, self.y`, clipped to `[0, width]×[0, height]`
       - Call `engine.update_neighbors()` to recompute adjacency
- **Methods**:
  - **`_broadcast(packet)`**:

- `yield env.timeout(Node.TX_DELAY)`
- For each neighbor in `self.neighbors`, clone the packet and schedule `neighbor.receive(clone)`
- Increments `self.tx_count`

- **`_deliver(neighbor, packet)`** (used for unicast):
  - `yield env.timeout(Node.TX_DELAY)`
  - Clone packet, `neighbor.receive(clone)`, increment `self.tx_count`
- **`receive(packet)`**: (SimPy process)
  - `yield env.timeout(Node.RX_DELAY)`
  - Increment `self.rx_count`
  - If `packet.pkt_type == 'BOM'`, call `_handle_bom(packet)`
  - If `packet.pkt_type == 'RMS'`, call `_handle_rms(packet)`
- **`_handle_bom(packet)`**:
  - Extract `(sink, seq, hop = packet.hop_count + 1)`
  - Compare `seq` vs. `self.best_seq.get(sink, -1)` and `hop` vs. existing `routing_table[sink][1]`
  - If `seq` is newer or `hop` is smaller:
    - Update `self.routing_table[sink] = (packet.src, hop, seq)`
    - Update `self.best_seq[sink] = seq`
    - `logger.record_routing_update()` and `logger.log_event(env.now, 'BOM_FWD', self.id, -1)`
    - Create `new_bom = Packet('BOM', src=self.id, sink_id=sink, seq=seq, hop_count=hop, origin=packet.origin, timestamp=packet.timestamp)`
    - `logger.record_control_sent()` and call `self._broadcast(new_bom)`
- **`_handle_rms(packet)`**:
  - If `(packet.origin, packet.seq)` already in `seen_rms`, drop it
  - Add to `seen_rms`
  - If `self.is_sink and packet.sink_id == self.id`:
    - `latency = env.now - packet.timestamp`
    - `hops_traveled = packet.hop_count`
    - `logger.record_data_delivered(latency, hops_traveled)`
    - `logger.log_event(env.now, 'RMS_DEL', packet.src, self.id)`
    - Return (stop forwarding)
  - Else if `packet.hop_count > 1`:
    - Decrement `packet.hop_count -= 1`
    - Call `self._send_rms(packet)`
- **`_send_rms(packet)`**:
  - If `routing_table` is non-empty:
    - Pick `sink_id` with smallest `hop_distance` from `routing_table`

        - Let `next_hop, dist, _ = routing_table[sink_id]`
        - Set `packet.sink_id = sink_id`, `packet.hop_count = dist + 1`
        - Find that neighbor in `self.neighbors` whose `id == next_hop`, then schedule `self._deliver(neighbor, packet)`
        - Log `logger.log_event(env.now, 'RMS_UNI', self.id, neighbor.id)` and increment `tx_count`
        - Return
    - Otherwise (no route), treat as broadcast:
        - `logger.log_event(env.now, 'RMS_BRD', self.id, -1)`
        - Call `self._broadcast(packet)`
  - **`logger()`**: Returns the shared `Logger` instance from `engine`.

4.3 Packet Definitions (`packets.py`)

- **`Packet`** class encapsulates both BOM and RMS packet types:
- `class Packet:`
- `    def __init__(self, pkt_type, src, sink_id, seq, hop_count=0, origin=None, timestamp=0.0):`
- `        self.pkt_type = pkt_type    # 'BOM' or 'RMS'`
- `        self.src = src`
- `        self.sink_id = sink_id`
- `        self.seq = seq`
- `        self.hop_count = hop_count`
- `        self.origin = origin if origin is not None else src`
- `        self.timestamp = timestamp`
- **Fields**:
  - `pkt_type`: `'BOM'` or `'RMS'`
  - `src`: Sending node's ID (int)
  - `sink_id`: For BOM: sink that originated; for RMS: sink destination
  - `seq`: Per-node sequence number (unique per originator)
  - `hop_count`: For BOM: distance so far; for RMS: TTL
  - `origin`: Original packet source (stays constant through relays)
  - `timestamp`: Creation time in simulated seconds

4.4 Logger (`logger.py`)

Collects global statistics and logs. All nodes and the engine share a single `Logger` instance.

- **Attributes**:
  - `control_sent`: Count of BOM/control packets transmitted
  - `data_sent`: Count of RMS/data packets generated
  - `data_delivered`: Count of RMS packets delivered to sinks
  - `delays`: List of end-to-end latency values for delivered RMS

    - `hops`: List of hop counts for delivered RMS
    - `routing_updates`: Count of times any node updated its routing table
    - `entries`: List of log strings (`"[time] TYPE from NodeX to NodeY"`)
- **Methods**:
    - `record_control_sent()`: Increment `control_sent`
    - `record_data_sent()`: Increment `data_sent`
    - `record_data_delivered(latency, hops)`: Append to `delays` and `hops`, increment `data_delivered`
    - `record_routing_update()`: Increment `routing_updates`
    - `log_event(time, pkt_type, src, dst)`: Append a formatted string to `entries`; uses `dst = -1` to denote broadcast
    - **Metric Queries**:
        - `packet_delivery_ratio()`: `data_delivered / data_sent`
        - `avg_latency()`: `sum(delays) / len(delays)` or `0` if none
        - `avg_hops()`: `sum(hops) / len(hops)` or `0` if none
        - `overhead_ratio()`: `control_sent / data_sent` (∞ if `data_sent=0` and `control_sent>0`)
        - `get_metrics()`: Returns a dict with all metrics

4.5 Topology Engine (`engine.py`)

Responsible for initial placement, neighbor maintenance, and connectivity repair.

- **Constructor**:
- `def __init__(self,`
- `             num_mesh_nodes=16,`
- `             num_mobile_nodes=3,`
- `             area_width=200,`
- `             area_height=200,`
- `             sink_positions=None,`
- `             min_dist=14.0):`
    - `num_mesh_nodes`: Number of static (relay) nodes
    - `num_mobile_nodes`: Number of mobile (phone) nodes
    - `area_width`, `area_height`: Dimensions of 2D simulation area
    - `sink_positions`: List of `(x, y)` if you want custom sink placement; otherwise two defaults at `(width×0.33, height×0.5)` and `(width×0.66, height×0.5)`
    - `min_dist`: Minimum center-to-center distance between static nodes (so circles of radius 6 do not overlap; $14 > 2\times6$)
- **Attributes**:
    - `self.env`: SimPy `Environment`
    - `self.logger`: Shared `Logger()`
    - `self.nodes`: List of all `Node` objects (sinks, static mesh, mobiles)
    - `self.node_map`: Dict mapping `node_id → Node`
    - `self.width`, `self.height`, `self.MIN_DIST`, `self.mesh_count`, `self.mobile_count`

- **Methods**:

1. **`_create_nodes()`**:
    - **Phase 1**: Place sinks at given positions, using up to 50 jitter attempts to ensure each sink is ≥ `MIN_DIST` from any already-placed static node.
    - **Phase 2**: Place `mesh_count` static relay nodes via rejection sampling in a central sub-rectangle (`80×80`) ensuring min-distance from all previously placed static nodes. Up to 200 tries. If it still collides, "snap" onto an anchor's circle at distance `MIN_DIST`.
    - **Phase 3**: Place `mobile_count` phone nodes anywhere uniformly in `[0, width]×[0, height]`. Phones have no min_dist constraint.
2. **`update_neighbors()`**:
    - Clears each `node.neighbors` list. For every pair `(i, j)`, compute Euclidean distance. If `<= Node.COMM_RANGE` (30.0), add each to the other's neighbor list.
3. **`_ensure_each_has_neighbor()`**:
    - Repeatedly:
        - Call `update_neighbors()`
        - For each static node with zero neighbors, pick a random "anchor" static node, then attempt up to 50 times to relocate the isolated node within radius `0.8 × Node.COMM_RANGE` of the anchor while preserving `MIN_DIST` from all other static nodes.
        - If still isolated after 50 tries, forcibly place it at radius `0.99 × Node.COMM_RANGE` from the anchor (ignoring `MIN_DIST`).
    - Repeat until no static node remains isolated.
4. **`step(dt=1.0)`**:
    - Advance the SimPy environment by `dt` simulated seconds:
    - `target = self.env.now + dt`
    - `self.env.run(until=target)`
5. **`run(until=None)`**:
    - Equivalent to `env.run(until=until)` (useful for headless runs).

4.6 GUI Frontend (`gui.py`)

Uses **PyQt5** to visualize and control the simulation.

- **`TopologyView` (subclass of `QGraphicsView`)**:
    - Overrides `mousePressEvent(...)`:
        - Maps mouse click to scene coordinates
        - Uses `scene().items(QPointF(x, y))` to retrieve all items under cursor
        - For the first item with `item.data(0) != None` (node ID), calls `parent_window.show_node_details(node_id)`
    - Overrides `wheelEvent(...)` to implement zooming.
- **`MainWindow`**:

- **Widgets**:
    - **QGraphicsScene** + **TopologyView** (left side) displaying nodes and links
    - Right panel containing:
        1. Start / Pause / Step `QPushButton`s
        2. Metric `QLabel`s:
            - PDR, Avg Latency, Avg Hops, Overhead, Routing Updates
        3. External-packet injection form (`QComboBox` + `QPushButton`):
            - Source (`ExternalDevice` or any NodeX)
            - Destination (`Broadcast` or any SinkX)
            - Type (`RMS` or `SMS`)
            - "Send Packet" button
        4. **QLabel:** "Selected Node: None"
        5. **QTableWidget** (1 row × 3 columns):
            - Neighbors (one ID per line)
            - Routing Table (each line: `D:<dest> NH:<next hop>`)
            - Energy (float)
        6. Spacer to push content upward
    - **Dockable "Event Log"** (`QPlainTextEdit`) at the bottom, showing protocol events
- **Signals & Slots**:
    - **btn_start.clicked → on_start()**: starts a **QTimer** that fires every 100 ms
    - **btn_pause.clicked → on_pause()**: stops timer
    - **btn_step.clicked → on_step()**: pauses (if running), then calls `engine.step(0.1)`, redraws, updates metrics/log
    - **timer.timeout → on_timeout()**:
        1. Calls `engine.step(0.1)`
        2. Calls `draw_network()` to re-plot all nodes & links
        3. Updates metric labels from `engine.logger.get_metrics()`
        4. Flushes any pending `logger.entries` into the log pane
    - **btn_send.clicked → on_send_packet()**:
        - Reads `cmb_src` and `cmb_dst` selections, determines `src_id` and `dst_id`
        - For type `SMS`: wrap as `RMS` targeted to `sink_id` (log `SMS→RMS`)
        - For type `RMS`: create an `RMS` packet directly, broadcast from the chosen source (log `RMS_INJ`)
- **Methods**:

1. **draw_network()**:
    - Clears the scene

- Draws all neighbor links (light gray lines) for each `(node, neighbor)` pair
- Draws each node as a circle:
  - Red if `node.is_sink`
  - Blue if static relay
  - Green if `node.is_mobile`
  - Radius = 6
  - `ellipse.setData(0, node.id)` tags each circle with its node ID

2. **`on_timeout()`**:
   - Steps sim by 0.1 s, calls `draw_network()`, updates metrics, flushes event log
3. **`on_start() / on_pause() / on_step()`**: Start/pause/step behavior as described
4. **`on_send_packet()`**: Build and inject "SMS→RMS" or RMS broadcast/unicast
5. **`show_node_details(node_id)`**:
   - Sets `selected_label` to "Selected Node: `node_id`"
   - Fetches `node = engine.node_map[node_id]`
   - Builds a newline-separated string of neighbor IDs
   - Builds a newline-separated string of routing entries formatted as `D:<dest> NH:<next hop>`
   - Shows energy as a single number
   - Populates the 1×3 `QTableWidget` with these values (top-aligned for multiline)

4.7 Entry Point (`main.py`)

```
# main.py

import sys
from PyQt5.QtWidgets import QApplication
from engine import SimulationEngine
from gui import MainWindow

def main():
    NUM_MESH_NODES = 16
    NUM_MOBILE_NODES = 3
    AREA_WIDTH = 200
    AREA_HEIGHT = 200
    SINK_POSITIONS = [
        (AREA_WIDTH * 0.33, AREA_HEIGHT * 0.5),
        (AREA_WIDTH * 0.66, AREA_HEIGHT * 0.5)
    ]

    engine = SimulationEngine(
        num_mesh_nodes=NUM_MESH_NODES,
        num_mobile_nodes=NUM_MOBILE_NODES,
        area_width=AREA_WIDTH,
        area_height=AREA_HEIGHT,
        sink_positions=SINK_POSITIONS,
        min_dist=14.0
```

```
    )

    app = QApplication(sys.argv)
    window = MainWindow(engine)
    window.show()
    sys.exit(app.exec_())

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

- **Configure** number of static mesh nodes, mobile nodes, area size, sink positions, and minimum separation.
- Creates a `SimulationEngine` and passes it to `MainWindow`.
- Starts the Qt event loop.

---

5. API Reference

5.1 `Packet` Class (packets.py)

```
class Packet:
    def __init__(self, pkt_type, src, sink_id, seq, hop_count=0,
origin=None, timestamp=0.0):
        """
        - pkt_type: 'BOM' or 'RMS'
        - src: ID of the node that sent this packet (int)
        - sink_id: For BOM: originating sink; for RMS: destination sink
(or None)
        - seq: Sequence number (int)
        - hop_count: For BOM: distance so far; for RMS: TTL (int)
        - origin: ID of original packet source (int). Defaults to src.
        - timestamp: Simulated creation time (float)
        """
        self.pkt_type = pkt_type
        self.src = src
        self.sink_id = sink_id
        self.seq = seq
        self.hop_count = hop_count
        self.origin = origin if origin is not None else src
        self.timestamp = timestamp

    def __repr__(self):
        return (f"<Packet {self.pkt_type} seq={self.seq} src={self.src}
"
                f"sink={self.sink_id} hops={self.hop_count}
origin={self.origin}>")
```

- **Usage**:
  - To create a BOM (beacon):
  - `bom = Packet('BOM', src=node_id, sink_id=node_id, seq=seq, hop_count=0, origin=node_id, timestamp=env.now)`
  - To create an RMS (data):

- o rms = Packet('RMS', src=node_id, sink_id=destination_sink, seq=seq, hop_count=Node.MAX_HOPS, origin=node_id, timestamp=env.now)

5.2 Node Class (node.py)

```
class Node:
    COMM_RANGE = 30.0
    MOVE_INTERVAL = 5.0
    MOVE_DISTANCE = 10.0
    BOM_INTERVAL = 20.0
    RMS_INTERVAL = 15.0
    TX_DELAY = 0.05
    RX_DELAY = 0.01
    MAX_HOPS = 3

    def __init__(self, env, node_id, is_sink, is_mobile, init_pos,
engine):
        """
        - env: SimPy Environment
        - node_id: Unique int
        - is_sink: bool
        - is_mobile: bool (True for phones)
        - init_pos: (x, y) starting coordinates
        - engine: SimulationEngine reference
        """
        self.env = env
        self.id = node_id
        self.is_sink = is_sink
        self.is_mobile = is_mobile
        self.engine = engine
        self.ipv6 = f"fe80::1:{node_id:04x}"
        self.x, self.y = init_pos
        self.neighbors = []
        self.routing_table = {}   # sink_id -> (next_hop_id, hop_dist,
seq)
        self.best_seq = {}        # sink_id -> highest seq seen
        self.seen_rms = set()     # {(origin, seq)}
        self.seq_num = 0
        self.energy = 100.0
        self.tx_count = 0
        self.rx_count = 0

        if self.is_sink:
            env.process(self._sink_bom_process())
        else:
            env.process(self._generate_rms_process())

        if self.is_mobile:
            env.process(self._mobility_process())

    def _mobility_process(self):
        """
        Runs only if is_mobile=True. Every MOVE_INTERVAL seconds,
        move randomly by up to MOVE_DISTANCE, then call
engine.update_neighbors().
        """
```

```
        while True:
            yield self.env.timeout(Node.MOVE_INTERVAL)
            angle = random.uniform(0, 2*math.pi)
            dx = Node.MOVE_DISTANCE * math.cos(angle)
            dy = Node.MOVE_DISTANCE * math.sin(angle)
            new_x = max(0, min(self.engine.width, self.x + dx))
            new_y = max(0, min(self.engine.height, self.y + dy))
            self.x, self.y = new_x, new_y
            self.engine.update_neighbors()

    def _sink_bom_process(self):
        """
        Runs if is_sink=True. Every BOM_INTERVAL, broadcast a new BOM.
        """
        while True:
            yield self.env.timeout(Node.BOM_INTERVAL)
            self.seq_num += 1
            bom = Packet('BOM', src=self.id, sink_id=self.id,
                         seq=self.seq_num, hop_count=0,
                         origin=self.id, timestamp=self.env.now)
            self.logger().record_control_sent()
            self.logger().log_event(self.env.now, 'BOM', self.id, -1)
            self.env.process(self._broadcast(bom))

    def _generate_rms_process(self):
        """
        Runs if is_sink=False. Every ~Exp(1/RMS_INTERVAL), generate and
send RMS.
        """
        while True:
            interval = random.expovariate(1.0 / Node.RMS_INTERVAL)
            yield self.env.timeout(interval)
            self.seq_num += 1
            rms = Packet('RMS', src=self.id, sink_id=None,
                         seq=self.seq_num, hop_count=Node.MAX_HOPS,
                         origin=self.id, timestamp=self.env.now)
            self.logger().record_data_sent()
            self.logger().log_event(self.env.now, 'RMS_GEN', self.id, -
1)
            self._send_rms(rms)

    def _broadcast(self, packet):
        """
        Wait TX_DELAY, then clone & deliver packet to each neighbor's
receive().
        """
        yield self.env.timeout(Node.TX_DELAY)
        for nb in list(self.neighbors):
            clone = Packet(packet.pkt_type, packet.src, packet.sink_id,
                           packet.seq, packet.hop_count, packet.origin,
                           packet.timestamp)
            self.tx_count += 1
            self.env.process(nb.receive(clone))

    def _deliver(self, neighbor, packet):
        """
```

```
        Wait TX_DELAY, then clone & deliver to specific neighbor
(unicast).
        """
        yield self.env.timeout(Node.TX_DELAY)
        clone = Packet(packet.pkt_type, packet.src, packet.sink_id,
                       packet.seq, packet.hop_count, packet.origin,
                       packet.timestamp)
        self.tx_count += 1
        self.env.process(neighbor.receive(clone))

    def receive(self, packet):
        """
        Wait RX_DELAY, then dispatch to _handle_bom() or _handle_rms().
        """
        yield self.env.timeout(Node.RX_DELAY)
        self.rx_count += 1
        if packet.pkt_type == 'BOM':
            self._handle_bom(packet)
        elif packet.pkt_type == 'RMS':
            self._handle_rms(packet)

    def _handle_bom(self, packet):
        """
        Update routing table if this BOM is new/better, then
rebroadcast.
        """
        sink = packet.sink_id
        seq = packet.seq
        hop = packet.hop_count + 1
        prev_seq = self.best_seq.get(sink, -1)
        prev_entry = self.routing_table.get(sink)
        prev_hop = prev_entry[1] if prev_entry else math.inf

        if seq > prev_seq or hop < prev_hop:
            self.routing_table[sink] = (packet.src, hop, seq)
            self.best_seq[sink] = seq
            self.logger().record_routing_update()
            self.logger().log_event(self.env.now, 'BOM_FWD', self.id, -
1)
            new_bom = Packet('BOM', src=self.id, sink_id=sink,
                             seq=seq, hop_count=hop,
                             origin=packet.origin,
timestamp=packet.timestamp)
            self.logger().record_control_sent()
            self.env.process(self._broadcast(new_bom))

    def _handle_rms(self, packet):
        """
        If destined to this sink, record delivery. Otherwise forward if
TTL >1.
        """
        key = (packet.origin, packet.seq)
        if key in self.seen_rms:
            return
        self.seen_rms.add(key)

        if self.is_sink and packet.sink_id == self.id:
```

```
            latency = self.env.now - packet.timestamp
            hops_traveled = packet.hop_count
            self.logger().record_data_delivered(latency, hops_traveled)
            self.logger().log_event(self.env.now, 'RMS_DEL',
packet.src, self.id)
            return

        if packet.hop_count > 1:
            packet.hop_count -= 1
            self._send_rms(packet)

    def _send_rms(self, packet):
        """
        If a route exists, unicast to the best sink; otherwise
broadcast.
        """
        if self.routing_table:
            sink_id, (next_hop, dist, _) = min(
                self.routing_table.items(), key=lambda kv: kv[1][1]
            )
            packet.sink_id = sink_id
            packet.hop_count = dist + 1
            for nb in self.neighbors:
                if nb.id == next_hop:
                    self.tx_count += 1
                    self.logger().log_event(self.env.now, 'RMS_UNI',
self.id, nb.id)
                    self.env.process(self._deliver(nb, packet))
                    return
        self.logger().log_event(self.env.now, 'RMS_BRD', self.id, -1)
        self.env.process(self._broadcast(packet))

    def logger(self):
        return self.engine.logger
```

5.3 Logger Class (logger.py)

```
class Logger:
    def __init__(self):
        self.control_sent = 0
        self.data_sent = 0
        self.data_delivered = 0
        self.delays = []
        self.hops = []
        self.routing_updates = 0
        self.entries = []

    def record_control_sent(self):
        self.control_sent += 1

    def record_data_sent(self):
        self.data_sent += 1

    def record_data_delivered(self, latency, hops):
        self.data_delivered += 1
        self.delays.append(latency)
        self.hops.append(hops)
```

```
    def record_routing_update(self):
        self.routing_updates += 1

    def log_event(self, time, pkt_type, src, dst):
        dst_str = f"Node{dst}" if dst != -1 else "All"
        entry = f"[{time:.2f}s] {pkt_type} from Node{src} to {dst_str}"
        self.entries.append(entry)

    def packet_delivery_ratio(self):
        if self.data_sent == 0:
            return 0.0
        return self.data_delivered / self.data_sent

    def avg_latency(self):
        return (sum(self.delays) / len(self.delays)) if self.delays
else 0.0

    def avg_hops(self):
        return (sum(self.hops) / len(self.hops)) if self.hops else 0.0

    def overhead_ratio(self):
        if self.data_sent == 0:
            return float('inf') if self.control_sent > 0 else 0.0
        return self.control_sent / self.data_sent

    def get_metrics(self):
        return {
            'pdr': self.packet_delivery_ratio(),
            'avg_latency': self.avg_latency(),
            'avg_hops': self.avg_hops(),
            'control_sent': self.control_sent,
            'data_sent': self.data_sent,
            'data_delivered': self.data_delivered,
            'routing_updates': self.routing_updates,
            'overhead': self.overhead_ratio()
        }
```

5.4 `SimulationEngine` Class (engine.py)

```
class SimulationEngine:
    def __init__(self,
                 num_mesh_nodes=16,
                 num_mobile_nodes=3,
                 area_width=200,
                 area_height=200,
                 sink_positions=None,
                 min_dist=14.0):
        self.env = simpy.Environment()
        self.logger = Logger()
        self.mesh_count = num_mesh_nodes
        self.mobile_count = num_mobile_nodes
        self.width = area_width
        self.height = area_height
        self.MIN_DIST = min_dist

        if sink_positions is None:
```

```
            sink_positions = [
                (area_width * 0.33, area_height * 0.5),
                (area_width * 0.66, area_height * 0.5)
            ]
        self.sink_positions = sink_positions

        self.nodes = []
        self.node_map = {}

        self._create_nodes()
        self.update_neighbors()
        self._ensure_each_has_neighbor()
        self.update_neighbors()

    def _create_nodes(self):
        node_id = 0
        static_positions = []

        # 1) Sink placement with min_dist enforcement
        for pos in self.sink_positions:
            x, y = pos
            for attempt in range(50):
                if not static_positions:
                    break
                good = True
                for sx, sy in static_positions:
                    if math.hypot(x - sx, y - sy) < self.MIN_DIST:
                        good = False
                        break
                if good:
                    break
                x = pos[0] + random.uniform(-self.MIN_DIST,
self.MIN_DIST)
                y = pos[1] + random.uniform(-self.MIN_DIST,
self.MIN_DIST)
                x = max(0, min(self.width, x))
                y = max(0, min(self.height, y))
            sink = Node(self.env, node_id, True, False, (x, y), self)
            self.nodes.append(sink)
            self.node_map[node_id] = sink
            static_positions.append((x, y))
            node_id += 1

        # 2) Static mesh placement with rejection sampling
        sub_w, sub_h = 80, 80
        x0 = (self.width - sub_w) / 2
        y0 = (self.height - sub_h) / 2
        for _ in range(self.mesh_count):
            placed = False
            for attempt in range(200):
                x = random.uniform(x0, x0 + sub_w)
                y = random.uniform(y0, y0 + sub_h)
                good = True
                for sx, sy in static_positions:
                    if math.hypot(x - sx, y - sy) < self.MIN_DIST:
                        good = False
                        break
```

```
                if good:
                    placed = True
                    break
            if not placed:
                ox, oy = random.choice(static_positions)
                angle = random.uniform(0, 2 * math.pi)
                x = ox + self.MIN_DIST * math.cos(angle)
                y = oy + self.MIN_DIST * math.sin(angle)
                x = max(0, min(self.width, x))
                y = max(0, min(self.height, y))
            mesh_node = Node(self.env, node_id, False, False, (x, y),
self)
            self.nodes.append(mesh_node)
            self.node_map[node_id] = mesh_node
            static_positions.append((x, y))
            node_id += 1

        # 3) Mobile phone placement (no min-dist)
        for _ in range(self.mobile_count):
            x = random.uniform(0, self.width)
            y = random.uniform(0, self.height)
            mobile = Node(self.env, node_id, False, True, (x, y), self)
            self.nodes.append(mobile)
            self.node_map[node_id] = mobile
            node_id += 1

    def update_neighbors(self):
        for n in self.nodes:
            n.neighbors.clear()
        for i, n1 in enumerate(self.nodes):
            for j in range(i + 1, len(self.nodes)):
                n2 = self.nodes[j]
                if math.hypot(n1.x - n2.x, n1.y - n2.y) <=
Node.COMM_RANGE:
                    n1.neighbors.append(n2)
                    n2.neighbors.append(n1)

    def _ensure_each_has_neighbor(self):
        static_nodes = [n for n in self.nodes if not n.is_mobile]
        changed = True
        while changed:
            changed = False
            self.update_neighbors()
            for node in static_nodes:
                if len(node.neighbors) == 0:
                    candidates = [n for n in static_nodes if n.id !=
node.id]
                    if not candidates:
                        continue
                    anchor = random.choice(candidates)
                    for attempt in range(50):
                        angle = random.uniform(0, 2 * math.pi)
                        r = Node.COMM_RANGE * 0.8
                        new_x = anchor.x + r * math.cos(angle)
                        new_y = anchor.y + r * math.sin(angle)
                        new_x = max(0, min(self.width, new_x))
                        new_y = max(0, min(self.height, new_y))
```

```
                        good = True
                        for other in static_nodes:
                            if other.id == node.id:
                                continue
                            if math.hypot(new_x - other.x, new_y - 
other.y) < self.MIN_DIST:
                                good = False
                                break
                        if good:
                            node.x, node.y = new_x, new_y
                            changed = True
                            break
                    if not changed:
                        angle = random.uniform(0, 2 * math.pi)
                        node.x = anchor.x + Node.COMM_RANGE * 
math.cos(angle) * 0.99
                        node.y = anchor.y + Node.COMM_RANGE * 
math.sin(angle) * 0.99
                        node.x = max(0, min(self.width, node.x))
                        node.y = max(0, min(self.height, node.y))
                        changed = True

    def step(self, dt=1.0):
        target = self.env.now + dt
        self.env.run(until=target)

    def run(self, until=None):
        self.env.run(until=until)
```

5.5 GUI-Related Classes (`gui.py`)

Refer to the previous section for the revised `MainWindow` and `TopologyView` code. Note especially:

- Routing table entries are now displayed as:
- `D:<destination>    NH:<next hop>`
- The “Selected Node” label shows which node ID was clicked.

---

6. Configuration & Customization

All configurable parameters are accessible in `main.py` (for overall simulation) or as class attributes (for protocol timing, mobility, etc.).

6.1 Topology Parameters

- **`SimulationEngine.__init__`**:
    - `num_mesh_nodes`: Number of static relay nodes.
    - `num_mobile_nodes`: Number of mobile (phone) nodes.
    - `area_width`, `area_height`: Size of the 2D simulation area.

- `sink_positions`: Specify a list of two `(x,y)` coordinates for sink placement. If omitted, defaults are used.
- `min_dist`: Minimum center-to-center distance between static nodes (ensures no circle overlap). Default = 14.0.

6.2 Routing/Protocol Parameters

- **`Node.COMM_RANGE`**: Communication radius (default 30.0). Any two nodes within this Euclidean distance are neighbors.
- **`Node.BOM_INTERVAL`**: Seconds between sink's BOM flood broadcasts (default 20.0).
- **`Node.RMS_INTERVAL`**: Mean inter-arrival time for non-sink RMS generation (default 15.0). Modeled as Exponential.
- **`Node.MAX_HOPS`**: TTL for RMS packets (default 3).

These can be adjusted by directly modifying class attributes in `node.py`.

6.3 Energy/Timing Parameters

- **`Node.TX_DELAY`**: Simulated transmission delay (default 0.05s).
- **`Node.RX_DELAY`**: Simulated reception delay (default 0.01s).
- **`Node.MOVE_INTERVAL`**: Seconds between phone node moves (default 5.0).
- **`Node.MOVE_DISTANCE`**: Maximum distance a phone moves each interval (default 10.0).

To simulate energy depletion, you can subtract fixed "costs" (e.g. `TX_COST = 1.0`) inside `_broadcast()`, `_deliver()`, and `receive()`.

6.4 GUI Parameters

- **Timer Interval**: In `MainWindow.__init__`, `self.timer.setInterval(100)` → GUI steps sim by 0.1 s every 100 ms. Adjust for faster/slower animation.
- **Node Circle Radius**: In `draw_network()`, radius = 6. Increase/decrease for larger/smaller circles.
- **Colors**:
  - Sink → red (`QColor('red')`)
  - Static mesh → blue (`QColor('blue')`)
  - Mobile phone → green (`QColor('green')`)
  - Links → gray (`QPen(Qt.gray)`)

---

7. Usage Examples

7.1 Running the Simulator

1. Ensure all `.py` files are in the same directory.
2. Open a terminal and navigate to that directory.
3. Run:
4. `python main.py`
5. A window appears:
    - **Left**: Topology (nodes + links)
    - **Right**: Controls + metrics + "Selected Node" label + details table + injection form
    - **Bottom**: Event log

At startup:

- Two red sinks, sixteen blue static nodes, and three green phones appear.
- Static nodes are randomly placed (no overlaps) in a central region; phones start anywhere.

7.2 Stepping & Pausing

- **Start**: Click "Start" → simulation timer begins → topology updates every 100 ms → metrics/log update.
- **Pause**: Click "Pause" → halts simulation.
- **Step**: Click "Step" → advances simulation by 0.1 s once → updates topology, metrics, log.

7.3 Inspecting Node Details

- **Zoom**: Use mouse wheel over topology→ zoom in/out.
- **Pan**: Click + drag anywhere on topology.
- **Select Node**: Left-click exactly on a node circle.
    - "Selected Node: X" appears in right panel.
    - The table's "Neighbors" column lists neighbor IDs (one per line).
    - The "Routing Table" column shows entries like:
    - `D:2    NH:5`
    - `D:4    NH:6`

      meaning this node has a route to sink ID 2 via next-hop 5, and sink ID 4 via next-hop 6.

    - "Energy" column shows the node's remaining energy (default 100.0).

7.4 Injecting External Packets

1. **Source**: Choose “ExternalDevice” (meaning inject from outside) or pick any “NodeX.”
2. **Destination**: Choose “Broadcast” or any “SinkX.”
3. **Type**:
   - **SMS**: Creates an RMS targeted to the chosen sink (or Sink0 if “Broadcast”).
   - **RMS**: Directly broadcasts an RMS from the chosen node (or first mobile for “ExternalDevice”).
4. Click **“Send Packet”**.
   - In the log, you’ll see “`SMS→RMS from Node0 to Sink0`” or “`RMS_INJ from Node12 to All`.”
   - The packet is injected into the SimPy environment and will be propagated/flooded accordingly.

---

8. Extending the Library

8.1 Adding a New Metric

1. **Define collection hooks** in `Logger`. For example, track number of BOM_FWD events:
   - Add attribute `self.bom_fwd_count = 0`.
   - Modify `_handle_bom` in `node.py`: after logging `'BOM_FWD'`, call `self.logger().bom_fwd_count += 1`.
2. **Expose via `get_metrics()`**:

```
3. def get_metrics(self):
4.     return {
5.         'pdr': self.packet_delivery_ratio(),
6.         'avg_latency': self.avg_latency(),
7.         'avg_hops': self.avg_hops(),
8.         'control_sent': self.control_sent,
9.         'data_sent': self.data_sent,
10.             'data_delivered': self.data_delivered,
11.             'routing_updates': self.routing_updates,
12.             'bom_fwd': self.bom_fwd_count,
13.             'overhead': self.overhead_ratio()
14.         }
```

15. **Update GUI**: In `on_timeout()` or `on_step()`, read `metrics['bom_fwd']` and display in a new `QLabel`.

8.2 Custom Mobility Models

- Currently, `_mobility_process()` performs random-walk steps every `MOVE_INTERVAL`.

- To implement, e.g., **grid-based waypoints** or **Gauss-Markov mobility**, replace `_mobility_process()` in `node.py`:
- `def _mobility_process(self):`
- `    while True:`
- `        yield self.env.timeout(Node.MOVE_INTERVAL)`
- `        # Compute new position (e.g., follow predetermined path or random on a grid)`
- `        self.x, self.y = new_x, new_y`
- `        self.engine.update_neighbors()`

8.3 IPv6 Payload Emulation

- The simulator treats `Packet` as carrying only routing fields. To emulate an IPv6 header:
    1. Extend `Packet` with an attribute `ipv6_payload` (e.g. a dictionary or string).
    2. When building an RMS, include `packet.ipv6_payload = {'src': self.ipv6, 'dst': 'fe80::1:0002', 'data': ...}`.
    3. In `Logger.log_event()`, optionally append IP addresses:
    4. `entry = f"[{time:.2f}s] {pkt_type} from {src_ipv6} to {dst_ipv6} (Node{src}→Node{dst})"`
    5. In the GUI, show this payload detail in a popup when inspecting a node.

9. Licenses & Acknowledgments

- **Dependencies**:
    - **SimPy** (BSD/MIT): For discrete-event simulation
    - **PyQt5** (LGPL/GPL): For GUI
- **Code Structure**:
    - Adapted from BMSim and sample mesh-networking templates, but fully rewritten to match RouMBLE specifications.
- **RouMBLE Protocol Paper**:
    - Ferrari, G. & Davoli, L., *"RouMBLE: A Sink-Oriented Routing Protocol for BLE Mesh Networks"*, IEEE IoT Journal, January 2025.
