Networked Software for Distributed Systems - Project 1

Abril Cano Castro - Emanuele Cimino - José Luis Del Río

Description

This project simulates a comprehensive IoT network monitoring system designed to analyze and visualize the dynamics of the RPL routing protocol in real time. The core of the simulation is built on the Contiki-NG operating system, where we configure a network of sensor motes, each running a custom application that integrates an MQTT client. Each node periodically compiles and publishes a detailed JSON snapshot of its internal IPv6 and RPL state to a dedicated MQTT topic, rplstats/json. To bridge the simulated low-power network with the IP-based backend, a dedicated border router node is employed, which acts as a gateway, routing all MQTT traffic from the 6LoWPAN mesh to a central Mosquitto MQTT broker running on the host machine. This broker serves as the critical data hub, decoupling the publishers (the motes) from the subscriber (our backend).

The backend processing and analytics layer is implemented in Node-RED, which subscribes to the broker's rplstats/json topic. As JSON messages arrive from every node in the network, Node-RED processes them through a series of custom functions that aggregate the data. This allows the system to reconstruct not only the current RPL routing tree, but also the underlying wireless topology. This aggregated network graph is stored in a global context object and is periodically updated with new snapshots, enabling the backend to track topological changes over time, calculate essential network statistics, understand the tree's shape, and assess network density and connectivity. The entire system is designed to function identically under both the OF0 and MRHOF objective functions, allowing us to compare the different properties of each.

1 Contiki-IoT

To be able to successfully retrieve and publish the network data, we base our firmware on Contiki-NG's `mqtt-client` example and extend it to periodically report each mote's IPv6/RPL state, mote metadata, and verified one-hop neighbors. On boot, the single Contiki process initializes RPL, loads a static MQTT configuration, constructs a client ID, sets the publish topic to `rplstats/json`, and registers the MQTT connection. A small state machine then drives connectivity: once a preferred global IPv6 address and a default route are available, the node connects to the broker, gives LED feedback while connecting, and transitions into the publishing phase. Each period, the node snapshots its current view of the network by combining the RPL neighbor table with the `ds6` neighbor table, while a lightweight link-local neighbor verification (LL-NV) service, implemented with `simple_udp`, actively probes and echoes link-local addresses to confirm true one-hop reachability. Only neighbors that are recently verified as link-local peers, or otherwise trusted such as the preferred parent or the default router, are included, and stale entries are suppressed by an age limit. In parallel, an ICMPv6 echo-reply callback periodically pings the default router to provide a measure of parent-link quality (RSSI). If connectivity drops, the client backs off exponentially and retries indefinitely; when packets are still in flight it defers new publishes to avoid buffer churn. Because the program is event-driven between periods, any RPL churn or neighbor updates are naturally reflected in the next snapshot, providing the backend with a clean, point-in-time view of the topology.

What it Publishes

To be able to reconstruct the topology and calculate the statistics of the network we collect the following data:

Top-level metadata

- `"Seq #"`: Monotonic per-mote counter that lets the backend order snapshots and detect gaps or resets.
- `"Timestamp"`: Seconds since boot on the mote, used to time-index snapshots and correlate events across nodes.
- `"Node ID"`: Stable per-mote identifier (COOJA `simMoteID` when present) for joining state across messages.
- `"IPv6 Address"`: Preferred global IPv6 of the mote (or `"unknown"` before SLAAC completes) for unique node identity in the graph.

RPL / DODAG context

- `"RPL Instance ID"`: Numeric identifier of the active RPL instance, used to namespace routing state.
- `"DODAG ID"`, `"DODAG Version"`: Root identifier and version counter that anchor each snapshot to a specific tree and detect global reconfigurations.
- `"RPL Rank"`, `"RPL DAG Rank"`: Raw rank and hop-like DAG rank that feed depth and diameter computations and reveal objective-function behavior.
- `"Preferred Parent"`: IPv6 address of the currently selected parent (or `"null"`) that defines the active tree edge for this node.

Routing context

- `"Def Route"`: Stringified default router address, used as a health indicator for upstream reachability and to cross-check parent selection.

Objective function and role

- `"objective_function"`: Text label (`"OF0"` or `"MRHOF"`) inferred at runtime, enabling side-by-side comparison of metrics under different objective functions.
- `"is_root"`: Boolean flag that marks the root and enables correct depth calculations and tree visualization.

Neighbor set (`"neighbors"` array) Each neighbor entry contains:

- `"addr"`: Neighbor IPv6 address that forms a candidate topology edge for degree and connectivity statistics.
- `"lladdr"`: Eight-byte link-layer address in hexadecimal (or `null`) for disambiguation and optional link-stats joins.
- `"rpl_rank_raw"`, `"rpl_dag_rank"`, `"rpl_link_metric"`: Neighbor rank information and link metric (e.g., ETX) when available, used to assess local structure and path cost.
- `"is_rpl_parent"`, `"is_preferred_parent"`: Flags that distinguish routing-tree edges from radio-only neighbors for accurate tree reconstruction.

How It Connects and Runs

- **Process & state machine**: `PROCESS(mqtt_client_process, "MQTT Client")` with states `INIT` → `REGISTERED` → `CONNECTING` → `CONNECTED/PUBLISHING`. Precondition for connecting: `have_connectivity()` (preferred global IPv6 and a default route). `DISCONNECTED` triggers exponential backoff; `CONFIG_ERROR`/`ERROR` idle until reset.
- **MQTT setup**: `mqtt_register()` then optional `mqtt_set_username_password()` if `org_id` is not `"quickstart"`. Client ID constructed as `d:{org}:{type}:{EUI64}`; `conn.auto_reconnect = 0` (manual retries). Clean session with keepalive derived from `pub_interval`.
- **Broker**: IP from `MQTT_CLIENT_BROKER_IP_ADDR` (default `fd00::1`); port `1883`. Publish topic fixed to `rplstats/json`.
- **Retry/backoff**: on disconnect, schedule reconnect after `RECONNECT_INTERVAL` left-shifted by the attempt count (capped after a few shifts); attempts limited by `RECONNECT_ATTEMPTS` or run forever if set to `RETRY_FOREVER`.
- **Flow control**: new publishes only when `mqtt_ready(&conn)` is true and `conn.out_buffer_sent` is set; otherwise the timer reschedules without sending.
- **Timers**:
  - `publish_periodic_timer`: publish cadence (default every 30 s).
  - `connection_life`: marks stable connection duration for retry logic.
  - `echo_request_timer`: periodic ICMPv6 echo to the default route (default 30 s) to read RSSI.
- **ICMPv6 echo RSSI**: `ping_parent()` sends an echo to the default router; `echo_reply_handler()` records `UIPBUF_ATTR_RSSI`.
- **LED feedback**: status LED (green by default) blinks while connecting and on publish.

Simulation

This project establishes a complete simulation-to-application pipeline by integrating the Contiki-NG COOJA simulator with a Node-RED backend via MQTT. The architecture hinges on a virtualized network where simulated sensor motes, running custom firmware with the Contiki-NG mqtt-client library, are configured to publish and subscribe to topics on an external MQTT broker. The critical gateway enabling this communication is a simulated border-router node, which acts as the root of the 6LoWPAN network, managing routing and providing connectivity to the outside world. To bridge the simulated network with the host machine's real network stack, the COOJA simulator directs the border-router's output to a virtual serial port, which is then processed by the tunslip6 utility. This utility creates a virtual network interface (tun0) on the host, effectively injecting the simulated motes' IPv6 traffic into the local machine's network routing table, making each simulated device appear as a native node on the network. This allows the Node-RED instance to connect seamlessly to the MQTT broker; its MQTT input nodes can receive telemetry published by the simulated sensors, while its MQTT output nodes can publish commands that are routed back through the tunnel to actuate virtual devices within the simulation, closing the loop between the simulated physical layer and the application logic.

Objective Functions Comparison

In RPL, the objective function is the routing policy engine: it takes the link/path metrics a node observes (e.g., hop count, ETX, latency, energy) and turns them into a single "Rank" and a parent choice toward the root. By defining how to compute path cost and when to switch parents (often with hysteresis), the OF shapes the entire DODAG.

OF0 (Objective Function Zero) OF0 steers the network toward the root by treating Rank as an abstract distance that closely tracks hop count under default settings. Each node sets its Rank as $\text{Rank}(c) = \text{Rank}(p) + \text{step\_of\_rank}$, where step_of_rank is a normalized increment defined by the implementation rather than by any advertised link metric. Because it relies only on base RPL objects and does not use metric containers, OF0 effectively chooses the parent that keeps the path as short as possible in hops. In practice this compresses the tree, reduces hop depth and hop diameter, and may tolerate links of mediocre quality if they preserve a shorter path.

MRHOF (Minimum Rank with Hysteresis) MRHOF minimizes an additive path cost built from link metrics. When a DIO Metric Container is present it uses the advertised additive metric (for example, latency); otherwise the common default is ETX. A node computes its path cost as $\text{Cost}(c) = \text{Cost}(p) + \text{Metric}(p \leftrightarrow c)$ and selects the parent that yields the smallest total. To avoid rapid parent switching due to small metric fluctuations, MRHOF introduces hysteresis and only changes parent if the new path improves the current cost by at least a threshold. In effect, MRHOF prefers higher-quality links even if they add an extra hop, which can increase hop depth and hop diameter while reducing the cumulative metric (for example, total ETX) along active routes.

More concisely, we can observe clear effects of the objective function in the backend statistics like:

- **Depth:** OF0 typically yields smaller hop depth, while MRHOF may use more hops to avoid lossy links, giving a smaller metric (ETX/latency) depth overall.
- **Diameter:** in hop terms, OF0 tends to minimize the network's hop diameter, whereas MRHOF can increase hop diameter when it bypasses poor links (yet it would generally reduce metric diameter).
- **Neighbor counts and degree distribution:** the raw radio neighbor counts per node are largely topology driven and similar under both objective functions; however, the routing-tree degree differs. MRHOF tends to funnel children onto a few high-quality parents, which produces a heavier tail and potential hot spots near the root. OF0 more often spreads children by hop distance, which yields a flatter degree distribution.

1.1 Test case

To prove this behavior in our simulation we did a basic test case in a diamond topology that allowed us to check the statistics for each objective function.

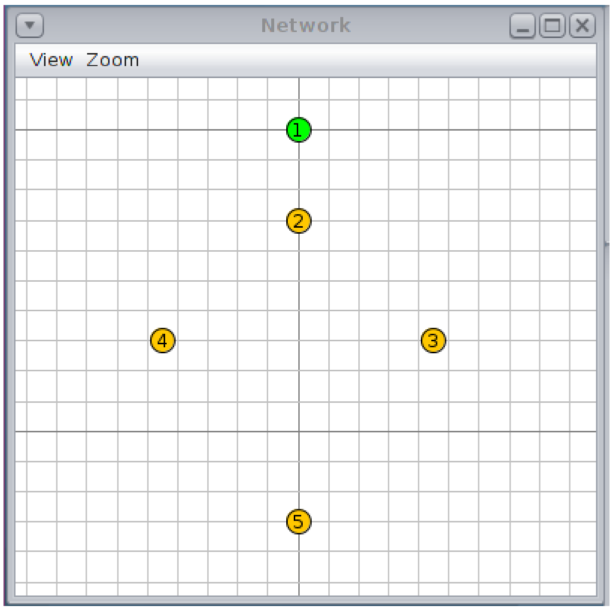


Figure 1: Diamond topology used for objective function comparison

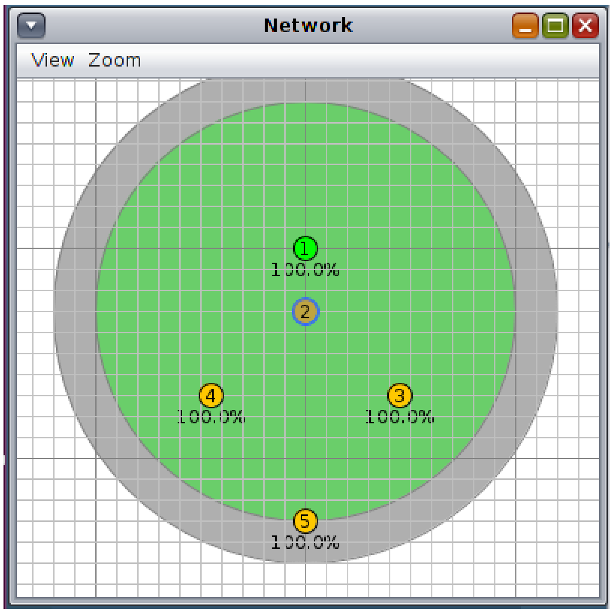
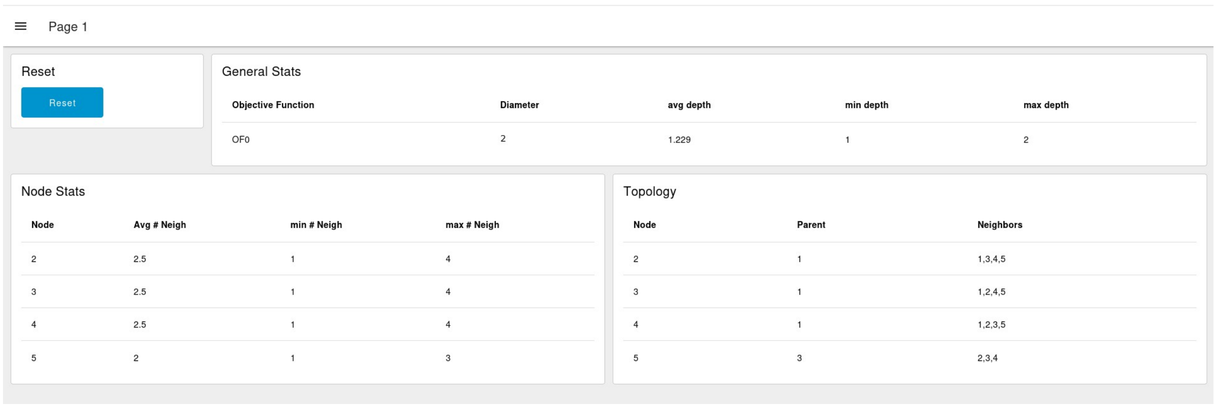


Figure 2: Mote 2 radio



General Stats

Objective Function	Diameter	avg depth	min depth	max depth
OF0	2	1.229	1	2

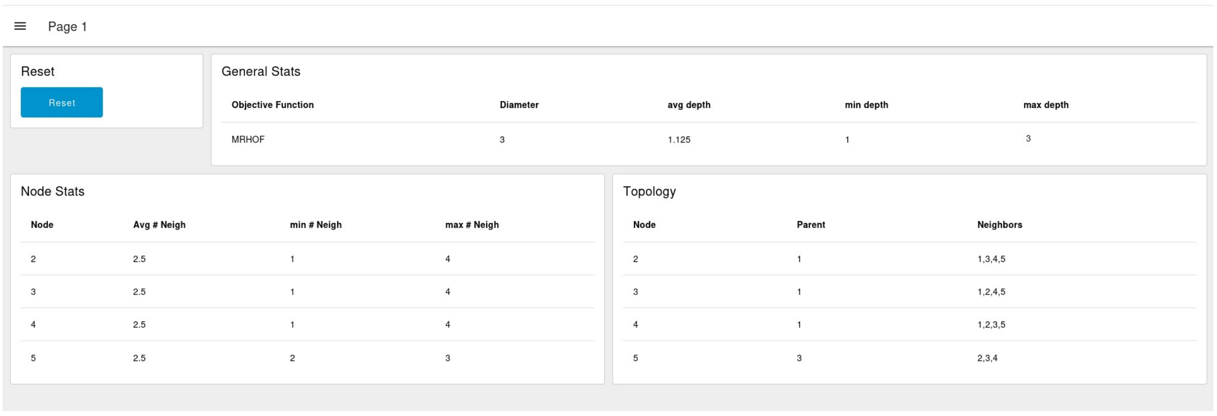
Node Stats

Node	Avg # Neigh	min # Neigh	max # Neigh
2	2.5	1	4
3	2.5	1	4
4	2.5	1	4
5	2	1	3

Topology

Node	Parent	Neighbors
2	1	1,3,4,5
3	1	1,2,4,5
4	1	1,2,3,5
5	3	2,3,4

Figure 3: Network statistics using OF0 objective function



General Stats

Objective Function	Diameter	avg depth	min depth	max depth
MRHOF	3	1.125	1	3

Node Stats

Node	Avg # Neigh	min # Neigh	max # Neigh
2	2.5	1	4
3	2.5	1	4
4	2.5	1	4
5	2.5	2	3

Topology

Node	Parent	Neighbors
2	1	1,3,4,5
3	1	1,2,4,5
4	1	1,2,3,5
5	3	2,3,4

Figure 4: Network statistics using MRHOF objective function

To validate the behavioral differences between objective functions in RPL, we conducted a simulation using a diamond topology (Fig. 1), where node 1 is the root, node 2 is directly connected to the root, nodes 3, 4 and 5 are connected to node 2 and nodes 3 and 4 are connected to node 5. We collected statistics for both OF0 and MRHOF, as shown in Fig. 3 and Fig. 4, respectively. The results align with the expected policies: With **OF0**, mote 5 chooses mote 2 as parent to minimize hop count, yielding the path $5 \to 2 \to 1$; the maximum depth is 2 and the tree diameter is 2 in hop terms. With **MRHOF**, mote 5 instead selects a relay (3 or 4) because the diagonal links produce a lower total ETX than the direct link to 2; the path becomes $5 \to 3/4 \to 2 \to 1$, so both the maximum depth and the diameter increase to 3. This illustrates the trade-off discussed earlier: OF0 prioritizes the shortest hop path and tends to compress the tree, whereas MRHOF may accept an extra hop when it reduces cumulative link cost, increasing hop-based depth and diameter while improving metric quality along the route, meaning reliability and overall network performance.

2 Node-RED flow

Overview

The Node-RED backend ingests data from the MQTT broker on `rplstats/json`, validates and normalizes each message, and updates a stateful model of the network in global context. From this live state, the flow reconstructs two views: the routing tree defined by each node's preferred parent, and the wireless neighbor graph formed from verified one-hop neighbors. A staleness policy removes nodes and edges that have not been refreshed within a configured interval, which keeps the model consistent with the simulator. On each update, the flow computes summary statistics since boot, including tree depth (average, minimum, maximum), tree diameter, and per-node neighbor counts (average, minimum, maximum). The results drive the dashboards and any downstream outputs. The design is event driven: each incoming publish triggers a short pipeline of parsing, state update, pruning, and metrics, so the UI reflects topology changes in near real time. Figure 5 sketches the Node-RED flow used in our implementation.

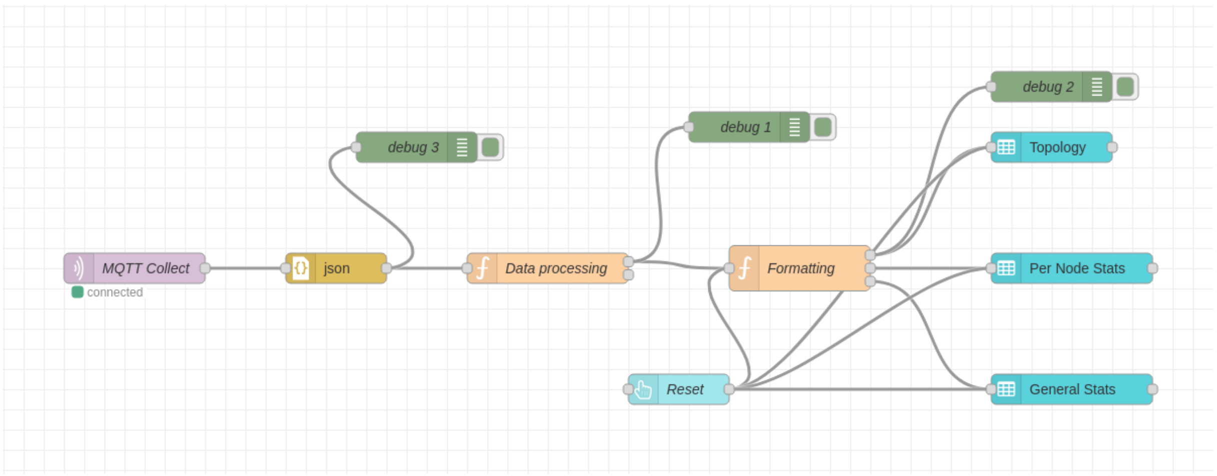


Figure 5: Node-Red flow

Data processing node

State Storage

The flow keeps a single, self-contained state object in global context under `rplStoreSimple`. This serves as the source of truth to rebuild both the routing tree and the wireless neighbor graph on every message with low overhead, to compute stable metrics without a database, and to keep memory bounded over long runs. Parent pointers and per-node neighbor sets allow fast reconstruction; `lastUpdateMs` with a staleness threshold prunes inactive nodes; running averages and a short history window provide useful trends without storing full time series; and per-node neighbor summaries highlight persistently sparse or dense regions.

Top-level structure

- `nodes:{}` Map from `nodeId` to the latest per-node snapshot that is needed to rebuild graphs and metrics at any time.
- `startedAt: number` Wall-clock timestamp (ms) when the backend started. Used to label since-boot averages and provide a stable reference for runs.
- `metricsAvg:{}` Online running averages since boot: `diameter`, `depthAvg`, `neighborAvg`, `apl`. This avoids storing full time series while still giving long-term trends.
- `history: array` Short FIFO history of `instant` metrics (time and a compact snapshot), capped to a small window for quick charts without a database.
- `nodeNeighborStats:{}` Per-node summary of neighbor counts over time (samples, min, max, and a midpoint-style average). Useful to characterize local density without retaining every tick.
- `currentSeq: number` Most recent sequence number seen. Helps detect gaps, resets, and to annotate rounds in the UI.

Per-node record `nodes[nodeId]` Each entry aggregates only what is needed to reconstruct graphs and compute statistics, while staying small so pruning and updates are cheap.

- `ipv6: string` Preferred global IPv6 for cross-checks and debugging.
- `of: string` Objective function label (`OF0` or `MRHOF`). Enables grouping and comparisons.
- `dag: string`, `version: number` DODAG identifier and version. Detects global reconfigurations and keeps snapshots tied to the correct tree.
- `rank: number`, `dagRank: number` Raw and DAG ranks. Inputs to depth statistics and sanity checks.
- `isRoot: boolean` Root marker. Needed to anchor depth computations and build the parent tree.
- `parentId: string|null` Current preferred parent as a normalized node ID. This single pointer defines the routing-tree edge for the node.
- `neighbors: Set<string>` Proven one-hop neighbors as node IDs. This set defines the undirected wireless topology used for neighbor statistics and connectivity checks.
- `lastSeq: number`, `lastTs: number` Latest mote counters for ordering and reset detection.
- `lastUpdateMs: number` Backend receive time. Used by the staleness policy to prune nodes that have not been refreshed within `STALE_MS`.
- `synthetic: boolean` Flag for a synthetic root record when needed to keep computations stable even if the true root has not reported yet.

Topology reconstruction

On each incoming message the backend derives two structures from the in-memory store `rplStoreSimple`: the undirected wireless neighbor graph built from per-node neighbor sets, and the directed RPL routing tree built from parent pointers. The reconstruction uses the following core and helper functions.

Core functions

- `addEdge(adj, a, b)` Inserts an undirected link between nodes `a` and `b`. The adjacency `adj` is a map `{nodeId → Set(nodeId)}`. Used to build both the wireless graph and the undirected view of the RPL tree.
- `computeDepths(parents, rootId, nodeList)` Walks the parent map `parents:{child → parent}` to compute integer depths from `rootId`. Detects and drops cycles or broken chains, so only nodes that can reach the root contribute to depth statistics.
- `basicStats(arr)` Returns average, min, and max for a numeric array. Used for depth and neighbor-count statistics after reconstruction.

Helper functions

- `nodeIdFromIPv6(addr, fallback)` Extracts the COOJA mote ID from an IPv6 address's IID. Normalizes neighbor and parent references into compact `nodeId` strings so graphs can be keyed consistently.
- `deriveRootIpv6FromDodag(dodagId)` Produces a usable root IPv6 when only a DODAG ID is known. Used to seed or refresh the synthetic root entry.

- `isFresh(id)` Checks if `nodes[id].lastUpdateMs` is within `LINK_FRESH_MS`. Useful when enforcing stricter link freshness policies for neighbor edges.
- `aHasB(a, b)` Tests whether node `a` currently reports `b` in its neighbor set. Enables optional mutual-neighbor validation.

Wireless neighbor graph

- **Inputs.** For each node `n`, `nodes[n].neighbors` is a `Set<string>` of normalized neighbor node IDs, populated from the firmware's "proven one-hop" list.
- **Construction.** For every pair (a, b) in those sets, call `addEdge(adj, a, b)` to build `adj:{nodeId` $\to$ `Set(nodeId)}`. The undirected edge count is computed as

$$|E| \;=\; \frac{1}{2}\sum_{v\in V}\deg(v).$$

- **Freshness and pruning.** Nodes are pruned before reconstruction if (`nowMs`$-$`lastUpdateMs`) $>$ `STALE_MS`. Optional link-level freshness can be enforced with `LINK_FRESH_MS` and `isFresh()`.

RPL routing tree

- **Inputs.** For each node `n`, `nodes[n].parentId` is the preferred parent as a normalized node ID.
- **Parent map.** If `parentId` exists and `parentId` $\neq$ `n`, set `parents[n] = parentId`.
- **Root selection.** Choose the root in order: a node with `isRoot = true`; else `ROOT_ID` if present (synthetic root); else any node that is not a child in `parents`; else the first available node. This guarantees an anchor for `computeDepths()`.
- **Tree adjacency.** Build an undirected adjacency `treeAdj` for distance metrics by calling `addEdge(treeAdj, child, parent)` for each parent relation.
- **Depths and distances.** Call `computeDepths(parents, rootId, nodeIds)` for per-node depths. Call `bfsAllPairs(treeAdj, treeNodes)` to obtain the tree diameter and the average path length.

Per-update outputs

- **Topology payload.** `topology.nodes` lists node attributes and neighbor counts; `topology.edges` contains undirected edges $[a, b]$ with $a < b$.
- **RPL tree payload.** `rpl_tree.root_id` identifies the root; `rpl_tree.edges` lists directed parent edges $[\mathit{child}, \mathit{parent}]$.
- **Instant metrics.** Depth statistics from `computeDepths()`, tree diameter and average path length from `bfsAllPairs()`, and neighbor-count statistics from `basicStats()` over degrees. Running averages are updated with `updateRunningAvg()`.

Metrics computation

The backend computes metrics on each update from the reconstructed graphs and parent map, then updates running averages since boot. The computations are deterministic and inexpensive, so the UI can refresh in near real time.

Functions used

- `computeDepths(parents, rootId, nodeList)` computes integer depths for nodes that can reach `rootId`. Nodes in cycles or with broken chains are excluded.
- `basicStats(arr)` returns `{avg, min, max}` for numeric arrays.
- `bfsAllPairs(adj, nodes)` Runs BFS from every node in the undirected tree adjacency to obtain the tree *diameter* and the average shortest-path length. Applied to the RPL tree view, not to the full wireless graph. Returns the *diameter* and the *average shortest-path length* (APL).
- `updateRunningAvg(obj, sample)` maintains online averages since boot without storing all samples.

Depth statistics (RPL tree)

- Compute depths: `depths = computeDepths(parents, rootId, nodeIds)`.
- Collect depth values: `depthVals = [depths[id]]` with optional exclusion of the root controlled by `COUNT_ROOT_IN_DEPTH_STATS`.
- Aggregate: `depthStats = basicStats(depthVals)` yielding `depth_avg`, `depth_min`, `depth_max`.

Neighbor statistics (wireless graph)

- Degree per node: `deg(n) = |adj[n]|` for non-synthetic nodes.
- Aggregate: `neighborStats = basicStats(degrees)` yielding `neighbor_avg`, `neighbor_min`, `neighbor_max`.
- Per-node evolution: `nodeNeighborStats[nodeId]` tracks `n`, `min`, `max`, and a compact mid-point average `avg = min + (max - min)/2` updated once per sequence number.

Tree diameter and path length

- Build undirected tree adjacency `treeAdj` by adding an undirected edge for each parent relation.
- Compute: `{diameter, avgPathLen} = bfsAllPairs(treeAdj, Object.keys(treeAdj))`.
- Emit as `diameter` and `avg_path_len` for the current tick.

Instant metrics payload For every processed message the flow emits an `instant` record that includes:

- Topology size: `node_count`, undirected wireless `edge_count`, and `tree_edge_count`.
- Depth statistics: `depth_avg`, `depth_min`, `depth_max`.
- RPL tree metrics: `diameter` and `avg_path_len`.
- Neighbor statistics: `neighbor_avg`, `neighbor_min`, `neighbor_max`.
- Context: `dag_id`, `objective_function`, `root_id`, `round_seq`, and `computed_at_ms`.

Since-boot running averages Online averages are updated per tick:

- `updateRunningAvg(store.metricsAvg.diameter, instant.diameter)`
- `updateRunningAvg(store.metricsAvg.depthAvg, instant.depth_avg)`
- `updateRunningAvg(store.metricsAvg.neighborAvg, instant.neighbor_avg)`
- `updateRunningAvg(store.metricsAvg.apl, instant.avg_path_len)`

They are exposed as `since_boot` fields (`diameter_avg`, `depth_avg_over_time`, `neighbor_avg_over_time`, `avg_path_len_over_time`) together with the sample count and `started_at_ms`.

Visualization node

This node receives the analyzer snapshot on `msg.payload` and formats three table outputs for the dashboard. A reset signal (`topic="rpl/reset"` or `payload.reset=true`) clears all tables.

Common inputs and helpers It expects `payload.topology` and `payload.rpl_tree`. From `topology.nodes` it derives a canonical per-node view `{id,is_root,parent,neighbors[]}`. The setting `INCLUDE_ROOT` controls whether the root appears in tables. Numeric KPIs are rounded with `round(v, d)` to `KPI_DECIMALS` places, and node lists are sorted numerically.

Output #1: Topology table Columns: **Node**, **Parent**, **Neighbors**. For each node (root optionally excluded), the node id is printed as `Node`, the preferred parent id becomes `Parent`, and the immediate neighbors are rendered as a comma-separated, numerically sorted string in `Neighbors`. Source: `topology.nodes[].parent` and `topology.nodes[].neighbors`.

Output #2: Per-node neighbor stats Columns: **Node**, **Avg # Neigh**, **min # Neigh**, **max # Neigh**. It joins the per-node summary emitted by the analyzer (`payload.per_node_neighbor_stats`) against the displayed node set, sorts by node id, and rounds the average to two decimals. If the summary is unavailable, it falls back to the current neighbor count for all three columns. This view highlights persistently sparse or dense nodes.

Output #3: General network stats Columns: a single row with **Diameter**, **avg depth**, **min depth**, **max depth**. `Diameter` and `avg depth` prefer since-boot running averages from `payload.since_boot` and fall back to the instant tick in `payload.instant`. `min depth` and `max depth` are taken from `payload.instant`. Values are rounded to `KPI_DECIMALS`.

Node outputs The node returns an array of three messages that feed three dashboard table widgets in order:

1. `msg[0].payload` = topology rows (`Node|Parent|Neighbors`)
2. `msg[1].payload` = per-node neighbor stats (`Node|Avg|min|max`)
3. `msg[2].payload` = general stats (single-row KPI table)

This keeps the UI reactive and stable. Repeated or out-of-order snapshots do not corrupt tables because the node is stateless and formats only the current `payload`.