Course Project: Tree-based Any-to-Any Routing

Iuliia Sharipova (Juliya Sharipova)

Instructor Gian Pietro Picco, Teaching Assistant Enrico Soprana

Abstract—The project aims to implement a tree-based Any-to-Any routing protocol for low-power IoT networks, enabling efficient communication between nodes. It uses Contiki OS and Cooja simulations for protocol development and testing. Additionally, the author attempted deployment on a physical testbed to evaluate the real-world feasibility. The focus is on achieving efficient, reliable, and energy-efficient routing in constrained environments.

1. Introduction to the Problem

In Low-power IoT networks, enabling efficient and reliable communication between any pair of nodes is challenging due to limited energy, memory, and dynamic topologies. This project builds upon the data collection protocol developed in previous labs of the course to implement a routing protocol that supports any-to-any (point-to-point) communication. The goal is to balance **reliability**, **energy efficiency**, and **low latency** in simulated and real world testbed environments, which makes it suitable for IoT applications such as smart cities, environmental monitoring, and smart agriculture.

2. Protocol Overview

Tree-based Any-to-any Routing protocol uses a tree structure built for data collection to enable communication between any two nodes, even if they are not directly connected. The nodes maintain a routing table, updated with topology reports.

2.1. Beacon Messages and Propagation:

The sink initializes the routing protocol via `rp_open()`, setting up broadcast and unicast channels, timers, and the routing table. It periodically sends beacons using `send_beacon()` with an incremented sequence number (`beacon_seqn`) and metric 0.

Non-sink nodes receive beacons in `bc_recv()` and evaluate them based on RSSI, sequence freshness, and metric improvement. If a beacon comes from a better candidate (not in the subtree, better metric, sufficient interval since last switch), the node switches parent, updates its routing table, sends `REMOVE_CHILD` and `ADD_CHILD` messages, and triggers a topology report. Beacon intervals are reset and dynamically increased with stability via `parent_stable_counter`.

Forwarding of received beacons is delayed (`BEACON_FORWARD_DELAY`) or suppressed unless a timeout (`BEACON_SILENT_LIMIT`) triggers rebroadcasting. Neighbor nodes are added to the routing table if not selected as parent.

Route cleanup is handled by a periodic timer (`cleanup_timer`), and topology reports are explicitly triggered on parent changes or subtree updates.

2.2. Compact routing tables:

Non-sink nodes store compact routing tables with only parent (`ROUTE_PARENT`), children (`ROUTE_TOPOLOGY`), and neighbors (`ROUTE_NEIGHBOR`) by initializing their subtree with themselves (`ROUTE_SELF`) and adding routes via `add_route()`, which updates or creates entries based on the priority of the route; route lookup `lookup_route()` tries direct matches first, then falls back to the parent if not a sink. Only the sink stores the full routing table.

Routes are prioritized by type using `route_priority()`: self-routes are never overwritten, while parent, topology, and neighbor routes have decreasing priority. Existing routes are updated only if the new route has higher or equal priority, optionally replacing the next hop. If no matching route exists, `add_new_route()` inserts a new entry at the head of the list with the current timestamp.

2.3. Topology reports:

Non-sink nodes periodically send topology reports with subtree information to their parent using `send_topology_report()`, which includes the node's address, metric, and all known subtree nodes from the routing table. Reports are buffered upon reception in `tr_recv()` and applied in batches using `delayed_send_topology_report_cb()` after a delay, reducing redundant updates. Parents update their routing tables based on received reports and forward the report upward unless they are the sink. The subtree is dynamically maintained using `add_to_subtree()` and `remove_from_subtree()`.

Additionally, topology reports are sent in response to parent changes: the child sends a report to inform the new parent of its updated subtree, and the old parent sends a report after removing the child to reflect the updated structure.

Note: An earlier version of the implementation processed each report immediately and forwarded a new report upstream after each reception. This approach resulted in higher PDR in Cooja simulations, but was replaced with batched processing for efficiency and reduced message overhead.

2.4. Removal of outdated routes:

To keep the routing table accurate and lightweight, outdated routes are periodically removed by `purge_old_routes()`, excluding self and parent routes. Entries are checked based on `last_updated` timestamps and purged if expired. Memory is freed, and removed destinations are also excluded from the subtree.

The cleanup is triggered by a periodic timer via `cleanup_timer_callback()`, which reschedules itself using `ctimer_reset()`. The interval `cleanup_interval` is set higher than the beacon interval to reduce overhead. Cleanup is only executed on non-sink nodes to minimize redundant computation.

2.5. Parent selection:

Nodes evaluate received beacons based on signal strength (RSSI), freshness (sequence number), and path cost (metric). Beacons with weak signal or outdated sequence numbers are ignored. Non-sink nodes switch parents only if the new beacon offers a better metric, the sender is not in their subtree, and a minimum interval since the last parent change has passed. Upon parent switch, the node updates routing entries, sends `REMOVE_CHILD` and `ADD_CHILD` messages to old and new parents, respectively, and sends a topology report.

Stable parents trigger exponential backoff of beacon intervals and periodic topology reports after a silence timeout (`BEACON_SILENT_LIMIT`). Nodes not selected as parents are added as neighbors.

2.6. Data Forwarding:

Data packets are sent via `rp_send()`, which looks up the route to the destination and prepares a header containing source, destination, and hop count. The packet is forwarded via unicast to the next hop.

Upon receiving a unicast packet (`uc_recv()`), the node first checks for control messages: `ADD_CHILD` or `REMOVE_CHILD`, updating routes and subtree accordingly and sending topology reports after removals.

If a topology report is received, it is processed by `tr_recv()`.

For data packets, the hop count is incremented; if it exceeds `MAX_PATH_LENGTH`, the packet is dropped to avoid loops. If the node is the destination, the header is stripped and data passed to the application callback. Otherwise, the node looks up the next hop and forwards the packet.

If no route exists, the packet is dropped and an error is logged.

2.7. Loop Prevention

To avoid infinite forwarding loops, each data packet has a maximum allowed hop count. If a packet's hop count surpasses `MAX_PATH_LENGTH` (set to 10), it is dropped.

3. Testing and Evaluation

3.1. Cooja simulation: contikimac_driver & nullrdc_driver

3.1.1. Overall Analysis

The protocol was tested in Cooja using two different RDC drivers: ContikiMAC and NullRDC. The tests involved a 10-node network scenario evaluating reliability (PDR), latency, and radio duty cycling to compare the trade-offs between energy efficiency and communication performance.

ContikiMAC showed an overall PDR of 94.35% with moderate latency, while NullRDC achieved higher reliability at 96.99% with minimal latency but significantly increased radio activity. These results confirm expected behavior: ContikiMAC saves energy via duty cycling at the cost of higher latency, while NullRDC keeps radios mostly always on, trading energy for responsiveness.

3.2. Cooja - ContikiMAC

3.2.1. PDR

The Packet Delivery Ratio (PDR) under ContikiMAC averaged 94.35%, with some packet losses distributed unevenly across nodes (see Table 1). Nodes 2, 9, and 10 achieved perfect delivery, while others experienced minor losses.

Node	Sent	Lost	PDR (%)
1	30	3	90.00
2	4	0	100.00
3	27	1	96.30
4	26	3	88.46
5	28	1	96.43
6	30	1	96.67
7	23	3	86.96
8	26	2	92.31
9	27	0	100.00
10	27	0	100.00
Total	248	14	**94.35**

Table 1. ContikiMAC: Packet Delivery Ratio per node

The PDR stability indicates reliable communication but reflects occasional packet losses likely caused by the duty-cycled radio being off during transmissions.

3.2.2. Latency

Latency averaged 385.30 ms with high variation (standard deviation 527.33 ms), ranging from 19 ms up to spikes of 3922 ms. This happens because ContikiMAC uses a low duty cycle: nodes sleep most of the time and wake up periodically, so packets often have to wait until the receiver is awake.

3.2.3. Radio Duty Cycle

The energy consumption analysis shows that with the ContikiMAC driver, the radio modules of the nodes are active on average about 2.618% of the time (see the table below).

The per-node duty cycle ranges from 1.58% to 4.39%, with a standard deviation of 0.87%, indicating relatively stable load distribution across the network.

This low radio activity aligns with ContikiMAC energy-saving design, which periodically turns off the radio to reduce power consumption. However, nodes 7 and 10 show a slightly higher duty cycle (4.1–4.4%), likely due to the network position.

Node	Duty Cycle (%)
1	2.418
2	1.583
3	2.200
4	2.170
5	2.046
6	2.429
7	4.149
8	2.154
9	2.641
10	4.394

Table 2. ContikiMAC: Radio duty cycle per node

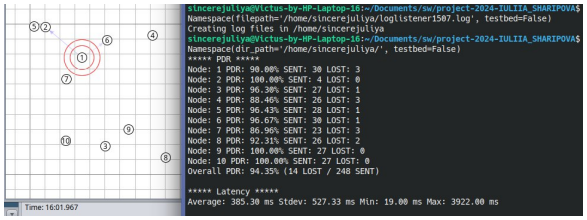

Figure 1. ContikiMAC: Screenshot with results

3.3. Cooja – NullRDC

3.3.1. PDR

NullRDC achieves higher PDR at 96.99%, with fewer lost packets overall (see Table 8). Most nodes delivered 100% packets, though some minor losses occurred on nodes 4-8, and 10.

Node	Sent	Lost	PDR (%)
1	30	0	100.00
2	29	0	100.00
3	24	0	100.00
4	19	1	94.74
5	28	1	96.43
6	30	1	96.67
7	23	1	95.65
8	29	1	96.55
9	25	0	100.00
10	29	3	89.66
Total	266	8	**96.99**

Table 3. NullRDC: Packet Delivery Ratio per node

3.3.2. Latency

Latency is dramatically lower and more consistent under NullRDC, with an average of 17.09 ms and standard deviation 8.22 ms, minimum 8 ms and maximum 38 ms. This is due to radios being almost always ON, allowing immediate transmission and reception.

3.3.3. Radio Duty Cycle

NullRDC keeps radios active almost 100% of the time (99.90%–99.96%), which eliminates wake-up delays but results in significantly higher energy consumption compared to ContikiMAC.

3.3.4. Summary

ContikiMAC provides significant energy savings through duty cycling, with trade-offs in latency and slightly lower PDR. NullRDC prioritizes minimal latency and higher reliability at the cost of much higher energy use.

The choice between them depends on application requirements for energy efficiency vs responsiveness.

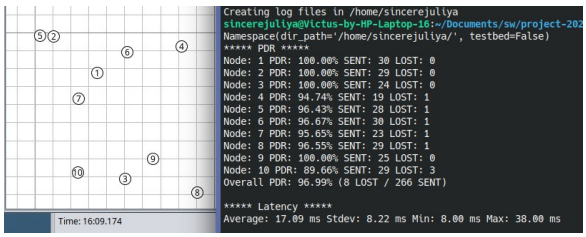

Figure 2. NullRDC: Screenshot with results

3.4. Testbed experiments

Preliminary testbed experiments were conducted at night to evaluate the protocol's real-world performance beyond simulation. The goal was to validate packet forwarding, route discovery, and basic stability under physical network conditions. Two setups were tested: one with nodes 1-36 and another with nodes 11–26.

3.5. Testbed - ContikiMAC

3.5.1. PDR

The overall Packet Delivery Ratio (PDR) was 63.43%, noticeably lower than simulation outcomes, showing real-world problems that affect wireless communication.

PDR values across nodes showed significant variability. While some nodes maintained high reliability - for example, node 3 and node 30 achieved 100% PDR - others experienced severe packet loss with the lowest being node 16 at 30.77%. Several nodes hovered around or below 50% PDR, indicating unstable links or high retransmission rates.

Node	Sent	Lost	PDR%	Node	Sent	Lost	PDR%
1	11	2	81.82	19	16	6	62.50
2	14	3	78.57	20	12	3	75.00
3	5	0	100.00	21	12	6	50.00
4	12	5	58.33	22	15	8	46.67
5	7	3	57.14	23	11	6	45.45
6	13	3	76.92	24	12	7	41.67
7	13	1	92.31	25	10	5	50.00
8	10	4	60.00	26	11	7	36.36
9	12	6	50.00	27	8	4	50.00
10	10	3	70.00	28	11	4	63.64
11	10	3	70.00	29	12	7	41.67
12	12	3	75.00	30	9	0	100.00
13	15	7	53.33	31	8	1	87.50
14	10	4	60.00	32	8	3	62.50
15	10	3	70.00	33	13	3	76.92
16	13	9	30.77	34	12	5	58.33
17	8	2	75.00	35	15	5	66.67
18	13	4	69.23	36	9	2	77.78
Overall PDR: 63.43% (147 LOST / 402 SENT)							

Table 4. Testbed. Network nodes 1–36. Packet Delivery Ratio (PDR) per node

Focusing on the subset of nodes 11–26, the overall PDR was 76.01%, showing improved reliability compared to the full network. Nodes ranged from 61.11% up to 93.33% PDR, showing persistent packet loss, but generally better performance.

Node	Sent	Lost	PDR%	Node	Sent	Lost	PDR%
11	17	4	76.47	19	18	4	77.78
12	16	5	68.75	20	15	3	80.00
13	15	1	93.33	21	21	2	90.48
14	20	7	65.00	22	23	4	82.61
15	15	4	73.33	23	18	7	61.11
16	21	5	76.19	24	18	5	72.22
17	20	5	75.00	25	23	6	73.91
18	16	4	75.00	26	20	5	75.00
Overall PDR: 76.01% (71 LOST / 296 SENT)							

Table 5. Testbed. Network nodes 11–26. Packet Delivery Ratio (PDR) per node

3.5.2. Duty Cycle

The duty cycle for nodes 11-26 averaged 4.23%, ranging from 1.96% to 9.19%, indicating moderate radio activity consistent with energy-conscious operation in the testbed environment.

Node	Duty Cycle (%)	Node	Duty Cycle (%)
11	3.299	19	9.191
12	1.955	20	5.308
13	2.368	21	5.765
14	4.073	22	7.814
15	2.570	23	3.596
16	4.615	24	3.640
17	3.796	25	4.229
18	2.064	26	3.455

Table 6. Testbed. Duty Cycle per node (nodes 11–26)

For the larger network (nodes 1–36), the duty cycle averaged 4.21%, with values between 1.57% and 16.62%.

Node	Duty Cycle (%)	Node	Duty Cycle (%)
1	4.503	19	5.073
2	16.616	20	3.338
3	3.461	21	3.681
4	3.314	22	5.440
5	3.228	23	3.955
6	3.396	24	5.350
7	4.339	25	3.744
8	2.398	26	5.684
9	4.100	27	4.011
10	2.842	28	5.785
11	8.420	29	4.141
12	2.036	30	3.168
13	2.220	31	3.794
14	2.432	32	4.992
15	2.586	33	3.036
16	3.770	34	6.726
17	3.149	35	2.838
18	1.566	36	2.304

Table 7. Testbed. Duty Cycle per node (nodes 1–36)

3.6. Testbed - NullRDC

3.6.1. PDR

The overall Packet Delivery Ratio (PDR) was 84.14%, showing better reliability compared to ContikiMAC in the testbed. Some nodes achieved perfect delivery, while others experienced noticeable loss. The lowest PDR was 50%, with a few nodes below 70%, indicating occasional interference or unstable links.

Node	Sent	Lost	PDR%	Node	Sent	Lost	PDR%
1	11	1	90.91	19	9	2	77.78
2	4	1	75.00	20	7	3	57.14
3	8	0	100.00	21	5	1	80.00
4	9	3	66.67	22	7	1	85.71
5	8	1	87.50	23	11	2	81.82
6	8	0	100.00	24	10	1	90.00
7	10	0	100.00	25	5	2	60.00
8	5	0	100.00	26	8	1	87.50
9	7	2	71.43	27	5	1	80.00
10	9	1	88.89	28	6	2	66.67
11	11	1	90.91	29	7	1	85.71
12	5	0	100.00	30	5	1	80.00
13	6	3	50.00	31	2	1	50.00
14	11	2	81.82	32	10	1	90.00
15	9	1	88.89	33	12	1	91.67
16	8	0	100.00	34	9	1	88.89
17	12	3	75.00	35	10	1	90.00
18	10	2	80.00	36	11	2	81.82
Overall PDR: 84.14% (46 LOST / 290 SENT)							

Table 8. Testbed. Network nodes 1–36. Packet Delivery Ratio (PDR) per node – NullRDC

For the subset of nodes 11-26, the overall PDR achieved 89.11%, indicating reliable packet delivery across the network. Individual node PDRs ranged mostly above 85%, with the best node achieving perfect delivery at 100%.

Node	Sent	Lost	PDR%	Node	Sent	Lost	PDR%
11	19	3	84.21	19	19	2	89.47
12	18	2	88.89	20	15	1	93.33
13	18	4	77.78	21	24	2	91.67
14	21	7	66.67	22	21	1	95.24
15	23	3	86.96	23	22	1	95.45
16	20	3	85.00	24	16	1	93.75
17	15	1	93.33	25	18	0	100.00
18	15	1	93.33	26	19	1	94.74
Overall PDR: 89.11% (33 LOST / 303 SENT)							

Table 9. NullRDC Testbed: Packet Delivery Ratio per node

3.6.2. Duty Cycle

All nodes showed a 100% duty cycle, meaning the radio was always on. This ensures minimal latency and high responsiveness but consumes maximum energy.

3.7. Summary

The real-world testbed results confirm that practical deployments introduce challenges not fully captured in simulations. Environmental factors, interference, and hardware limitations led to lower PDR and more variability across nodes, especially for ContikiMAC. NullRDC's always-on radio mitigated some packet loss but at the cost of maximum energy consumption.

These outcomes emphasize the importance of considering real deployment conditions when evaluating protocols, as simulations alone may overestimate performance and energy efficiency.

4. Design Trade-offs: Reliability vs. Energy Efficiency

Throughout the protocol design, key decisions were made to balance reliability and energy efficiency.

1. The tree-based routing method helps nodes choose stable parents and find routes efficiently, which improves packet delivery but causes extra work during network setup and changes.
2. Regular beacon messages keep the network updated and allow nodes to switch parents quickly, making the network more flexible but using more energy, shown by almost constant activity in simulations.
3. Topology reports share routing info in batches, reducing unnecessary messages but causing small delays in updates.
4. Cleanup timers remove outdated routes to keep the routing tables accurate, improving reliability but may increase control messages if set too short.

Overall, the protocol performs well due to stable parent selection and efficient route management. However, the protocol uses more energy to stay responsive and reliable. Frequent control messages and constant activity increase power use, especially in large or busy networks.

5. Conclusions

This project presents a tree-based any-to-any routing protocol designed for low-power IoT networks, balancing reliability, energy efficiency, and latency. Through the use of periodic beacons, compact routing tables, and dynamic topology updates, the protocol maintains efficient routes with minimal overhead.

Testing in both Cooja simulation and a real-world Firefly testbed demonstrated stable duty cycles and consistent network performance, confirming its practical applicability for IoT deployments. Future work will focus on enhancing scalability and improving adaptability to dynamic network conditions.

6. Additional materials

Some of the additional files include: code snippets and Cooja PDR graphs.

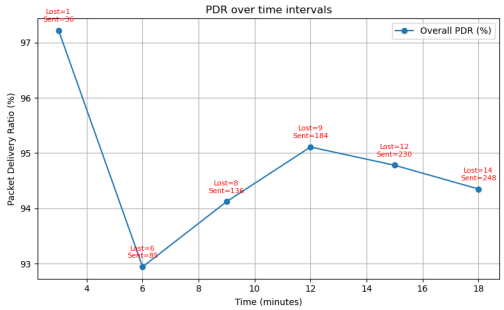

Figure 3. Cooja. ContikiMAC. PDR analysis

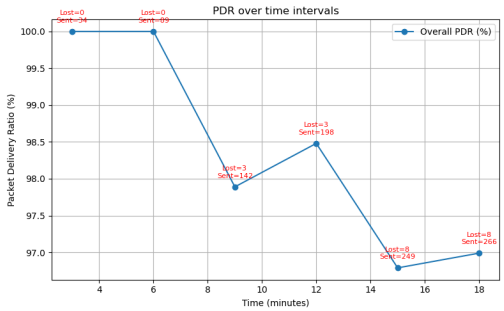

Figure 4. Cooja. NullRDC. PDR analysis

```c
/*---------------------------------------------------*/
#define RSSI_THRESHOLD -95
#define MAX_PATH_LENGTH 10  // Maximum number of hops
#define BEACON_SILENT_LIMIT 20 * CLOCK_SECOND // Max time node to be silent after parent switch(update )
/*---------------------------------------------------*/
/*               for the dynamic beacon              */
/*---------------------------------------------------*/
#define MIN_PARENT_SWITCH_INTERVAL (40 * CLOCK_SECOND) // min 40 sec for one parent

#define BEACON_FORWARD_DELAY ((random_rand() % CLOCK_SECOND)) // random delay to avoid collisions when forwarding beacons
#define BEACON_INITIAL_INTERVAL (15 * CLOCK_SECOND)
#define BEACON_MIN_INTERVAL     (10 * CLOCK_SECOND)
#define BEACON_MAX_INTERVAL     (70 * CLOCK_SECOND)
#define STABILITY_THRESHOLD     3   // number of beacons to consider parent stable
extern clock_time_t current_beacon_interval;

/*---------------------------------------------------*/
/*              for the topology reports             */
/*---------------------------------------------------*/
#define MAX_SUBTREE_SIZE 10 // Maximum number of nodes in the subtree
#define MAX_BUFFERED_REPORTS 7 // Max number of reports to store

struct topology_report {
  linkaddr_t node;
  uint16_t metric;
  uint16_t subtree_size;
  linkaddr_t subtree[MAX_SUBTREE_SIZE];
} __attribute__((packed));

/* to store reports before applying them, timer inside tr_recv
   every 6 sec apply reports from storage */
```

```
struct pending_topology_reports {
  struct topology_report reports[MAX_BUFFERED_REPORTS];
  int count;
};

/*---------------------------------------------------*/
/*              for the table cleaning               */
/*---------------------------------------------------*/
static const clock_time_t cleanup_interval =
    ↪ CLOCK_SECOND * 120;

/*---------------------------------------------------*/
```

Code 1. Variables

```
/*---------------------------------------------------*/
/*              routing table structure              */
/*---------------------------------------------------*/
/* Routing table */
typedef struct {
  linkaddr_t destination;
  linkaddr_t next_hop;
  route_type_t type;
  uint16_t metric;
  int16_t rssi;
  clock_time_t last_updated;
} routing_entry_t;

typedef struct routing_entry_node {
  routing_entry_t entry;
  struct routing_entry_node *next;
} routing_entry_node_t;
/*---------------------------------------------------*/
/*                  types of routes                  */
/*---------------------------------------------------*/
typedef enum {
  ROUTE_TOPOLOGY = 0,
  ROUTE_PARENT = 1,
  ROUTE_NEIGHBOR = 2,
  ROUTE_SELF = 3
} route_type_t; // just to distinguish routes --
    ↪ different from priority

/*---------------------------------------------------*/
/*               priority of routes                  */
/*---------------------------------------------------*/
/* Return priority of route types */
int route_priority(route_type_t t) {
  switch (t) {
    case ROUTE_SELF: return 4;     // highest (never
    ↪ overwritten)
    case ROUTE_PARENT: return 3;
    case ROUTE_TOPOLOGY: return 2;
    case ROUTE_NEIGHBOR: return 1;
    default: return 0;
  }
}
/*---------------------------------------------------*/
```

Code 2. Routing Table

```
/*---------------------------------------------------*/
/*                additional functions               */
/*---------------------------------------------------*/
/* Functions to delete routes */
void delete_route_by_next_hop(struct rp_conn *conn,
    ↪ const linkaddr_t *next_hop, bool is_sink);
void delete_route(const linkaddr_t *destination, const
    ↪ linkaddr_t *next_hop) ;

/* Return priority of route types */
int route_priority(route_type_t t);

/* to lookup a route in the routing table */
routing_entry_t *lookup_route(const linkaddr_t *
    ↪ destination, bool is_sink);

/* to add a route to the routing table */
void add_route(struct rp_conn *conn, const linkaddr_t *
    ↪ destination, const linkaddr_t *next_hop,
    ↪ route_type_t type,
              uint16_t metric,
              int16_t rssi);
void add_new_route(struct rp_conn *conn, const
    ↪ linkaddr_t *destination, const linkaddr_t *
    ↪ next_hop, route_type_t type,
              uint16_t metric,
              int16_t rssi) ;

/* receiving of a topology report */
void tr_recv(struct rp_conn* conn);

/* applying pending topology reports */
static void delayed_send_topology_report_cb(void *ptr);

/* to send a topology report */
void send_topology_report( void *ptr, char* lol);

/* to update the routing table based on a received
    ↪ topology report */
void update_routing_table(struct rp_conn *conn, const
    ↪ struct topology_report *report);

/*---------------------------------------------------*/
```

Code 3. Functions with logic