Internet of Things (IoT) Technology

(F24.285172U083.A)

Mini Project Report File

Submitted by:
Group: 2

May 27, 2024

Contents

Customized Routing Simulator

May 27, 2024

Lena Alargić AU758599
Purbak Sengupta AU759326
Kanak Rana AU759379

1 Introduction

This project evaluates the Routing Protocol for Low-Power and Lossy Networks (RPL) by designing and demonstrating a routing simulation using SimPy. It started with a simplified version of the RPL, demonstrating the establishment of the DODAG i.e, the DAO and the DIO messages. Then, the progress was made with distribution of network prefixes and updating routing tables for each node.

In this project, we have successfully developed a customized routing simulator to model and analyze the behavior of RPL in a simulated network environment. Afterwards, we have conducted some experiments, to show how the time of simulation of DODAG and network establishment increases as the number of nodes is increasing. The simulator is implemented using SimPy,[1] a discrete-event simulation library in Python, and NetworkX, a library that deals with the creation, manipulation, and study of complex networks. Pivot points of this project were:

- Establish a network and a node concept in SimPy.
- Implement and simulate a neighbor discovery mechanism that ensures that each node establish connectivity with its nearest neighbors.
- Implement the DIO message and simulate the DODAG formation with a varying number of nodes with different sets of neighbor nodes.
- Implement the DIS message (optional).
- Implement a trickle algorithm to control the period of DIS messages in the network(optional).
- Design a node addressing plan for the network. Make assumption on the use of a mesh under or a route over architecture.
- Implement the DAO message and demonstrate the distribution of network prefixes in the DODAG.
- Simulate a DODAG local repair mechanism (optional).
- Organize and document your code for the simulator in a software repository such as Git.

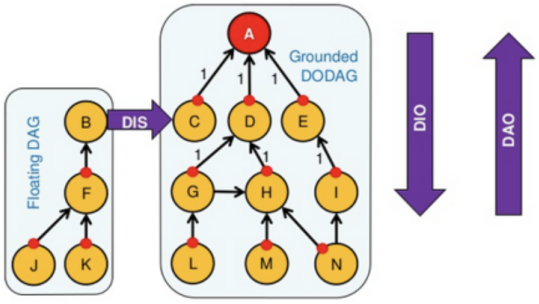

Figure 1: Example of the RPL routing protocol and its use of DIS, DIO and DAO messages to build and maintain the DODAG.

2 Background

RPL is a distance vector routing protocol used specifically in constrained environments in terms of power, memory and processing resources, such as sensor networks, smart grids, and home automation systems. It was Developed by the Internet Engineering Task Force (IETF) as RFC 6550[2]. RPL provides a robust and efficient routing solution for communication networks that are often unreliable and resources are limited.

RPL organizes the network topology into a hierarchical structure called Destination-Oriented Directed Acyclic Graph (DODAG). There is a central node called the DODAG root towards which routes need to be directed from other/leaf nodes. Nodes in the network use control messages to exchange information about their link quality and routing metrics, allowing them to dynamically establish and maintain optimal routes to the root.[5] These control messages include:

- **DIO (DODAG Information Object):** Message to advertise the presence of a DODAG and transfer information that is necessary for nodes to join and maintain the DODAG.
- **DAO (Destination Advertisement Object):** Messages that are used by nodes to inform their parents about their reachable addresses. It helps to construct a downward routing channel from root in DODAG.
- **DIS (DODAG Information Solicitation):** Messages that are sent by nodes to solicit DIO messages from their neighbours. Typically, new nodes requesting to join the network or nodes that have lost connectivity and in need to re-establish their route towards DODAG root, sends a DIS message.

Apart from these control messages, RPL has a feature called "Trickle Algorithm" that manages the propagation of DIO messages efficiently. It is able to adjust the frequency of DIO message transmissions dynamically based on some factors such as network stability, reducing control message overhead and ensuring rapid convergence in dynamic network conditions.

RPL also features a "Local Repair Mechanism" which address issues such as link failures or node mobility within the network. Usually, whenever a node detects that its parent node is unreachable or has become unavailable for any reason, it initiates the local repair mechanism so that it can successfully re-establish connectivity (typically, by finding a new parent) and maintain the DODAG structure. During the local repair process, the node may broadcast a repair request or recompute its routing information to find an alternative parent node. This ensures that the network can adapt to changes in topology and maintain continuous communication paths between nodes, even in the presence of dynamic conditions.

Another crucial thing is evaluation of performance of the RPL. Several key metrics related to the performance evaluation should be considered to evaluate its efficiency, reliability, and suitability for various network environments. These metrics help to understand the behavior of the protocol under different conditions and configurations. To evaluate RPL's performance, our customized routing simulator incorporates several of these metrics as follows:

1. **Control Message Overhead:** Count the total number of DIO, DAO, and DIS messages exchanged during the simulation.
2. **Convergence Time:** Measure the time from the start of the simulation until the DODAG is fully formed and stable.
3. **Average Hop Count:** Track the number of hops each packet takes during data transmission scenarios.
4. **Routing Table Size:** Report the size of the routing table for each node.
5. **Parent Changes:** Track the number of times each node changes its parent during the simulation.

These metrics allow us for a thorough evaluation of RPL's performance, identification of areas for improvement, and validation of the protocol's suitability for specific network scenarios[2].

3 Design, Implementation & Test Setup

We conducted the whole experiment on Jupyter Notebook environment using Python. We focused on creating a flexible system that could effectively model the behavior of the Routing Protocol for the RPL. We used Python libraries such as SimPy and NetworkX to build our simulation environment.

Our main idea was to design and create a network of nodes, implementing RPL message exchange mechanisms (such as DIO and DAO), and visualizing the network & DODAG structures. We assumed a static network topology for simplicity, but also designed a flexible system that can be extended for future improvements.

Our implementation includes node concepts (defining a Node class to represent individual nodes in the network), RPL message exchange (implementing functions for sending and receiving messages), visualization (visualizing the network topology and DODAG using Matplotlib) and simulation environment (modelling the discrete-event simulation using SimPy). We've achieved all these components through the process as described below:

3.1 Node and Network Setup

The simulation starts with the creation of a set of nodes, each assigned a random position within a predefined area size of 100x100 units. The following network initialization steps are performed:

- **Node Creation:** A specified number of nodes (e.g., 15 nodes) are created. Each node is given a unique name, rank, position, and IP address(IPv6).
- **DODAG Root Initialization:** The first node ('node1') is designated as the DODAG root and is assigned a rank of 0. All other nodes are initialized with an infinite rank, meaning that they have not yet joined the DODAG.
- **Address and Prefix Assignment:** Each node is assigned an IP address in the format **'10.0.0.x'** and a unique network prefix in the format **'2001:db8:i::/24'**, where 'i' is the node index.

3.2 Neighbour Discovery

Nodes discover their neighbours based on their positions and a maximum distance. The process includes:

- **Distance Calculation:** For each pair of nodes, the Euclidean distance between them is calculated.
- **Neighbour Addition:** If the distance between two nodes is less than a specified maximum distance (e.g., 65 units), they are considered neighbors, and an edge is added between them in the network graph.

3.3 DODAG Formation

Nodes exchange DIO (DODAG Information Object) messages to establish parent-child relationships, forming the DODAG structure. This process includes:

- **DIO Message Sending:** Each node sends DIO messages to its discovered neighbours.
- **DIO Message Reception:** As DIO message is recieved, a node evaluates the potential parent (the sender of the DIO) based on the received rank and the number of children the sender already has. Maximum number of children is defined as a constant, so that the DODAG has more tree-like structure.
- **Parent Selection:** If the conditions are met (i.e., the sender has a lower rank and can have another child), the receiving node sets the sender as its parent, updates its rank, and sends its own DIO messages.

3.4 DAO Message Exchange

Once the DODAG structure is formed, nodes exchange DAO (Destination Advertisement Object) messages to advertise their prefixes and update routing tables, which are used like a maps, to determine which way to forward traffic and to see which nodes are reachable. This process includes:

- **DAO Message Sending:** Nodes send DAO messages to their parents, informing them of their network prefixes.
- **Routing Table Updates:** Upon receiving a DAO message, a parent node updates its routing table with the prefix information received from the child node.
- **Continued DAO Exchange:** This process continues, propagating the network prefix information upward through the DODAG until the simulation ends.

3.5 Visualization and Analysis

The simulator provides visualizations of the network topology and the DODAG structure, along with detailed routing tables for each node.

- **Network Topology Visualization:** A graphical representation of the network showing all nodes and their connections.
- **DODAG Structure Visualization:** A hierarchical diagram showing the parent-child relationships within the DODAG.
- **Routing Table Display:** Printing the routing tables of all nodes, detailing the network prefixes and the corresponding next hops.

Our implementation was organized into adjustable components, making it easy to understand, modify, and extend. **The Git repository containing our project can be found here: Project Code.**

4 Experiments & Results

The results of the experiment demonstrate the successful initialization and operation of the RPL protocol. We're about to show the result with 15 nodes for now (simplicity purpose), however it can be adjusted and we've tried and experimented with various number of nodes from 5(the most simple) to 500 (too complex).

- Each node was assigned an IPv6 address and a network prefix as follows:

```python
class Node:
    def __init__(self, network, name, rank=float('inf'), position=None):
        self.network = network
        self.name = name
        self.rank = rank
        self.position = position or (random.uniform(0, AREA_SIZE[0]), random.uniform(0, AREA_SIZE[1]))
        self.neighbors = []
        self.parent = None
        self.children = []
        self.dao_sent = False
        self.address = f"10.0.0.{len(self.network.nodes)+1}"
        self.routing_table = {}
        self.prefix = None
        print(f"{self.name} assigned address {self.address}")
        self.dist_to_all_nodes = {child: float('inf') for child in self.network.nodes}
        self.dist_to_children = {child: float('inf') for child in self.children}
```

```
node1 assigned address 10.0.0.1
node2 assigned address 10.0.0.2
node3 assigned address 10.0.0.3
node4 assigned address 10.0.0.4
node5 assigned address 10.0.0.5
node6 assigned address 10.0.0.6
node7 assigned address 10.0.0.7
node8 assigned address 10.0.0.8
node9 assigned address 10.0.0.9
node10 assigned address 10.0.0.10
node11 assigned address 10.0.0.11
node12 assigned address 10.0.0.12
node13 assigned address 10.0.0.13
node14 assigned address 10.0.0.14
node15 assigned address 10.0.0.15
node1 assigned network prefix 2001:db8:0000::24
node2 assigned network prefix 2001:db8:0001::24
node3 assigned network prefix 2001:db8:0002::24
node4 assigned network prefix 2001:db8:0003::24
```

Figure 2: Node Initialization

- Each node sent DIO messages to all other nodes in the network. For example:

```python
def send_dio(self, recipient):
    print(f"{self.name} sends DIO to {recipient.name}")
    yield self.network.env.timeout(1)
    recipient.receive_dio(self)

def receive_dio(self, sender):
    if self.name == 'node1':
        return
    if (self.parent is None or sender.rank < self.parent.rank) and len(sender.children) < MAX_CHILDREN:
        if self.parent:
            self.parent.children.remove(self)  # Remove self from the old parent's children list
        self.parent = sender
        self.rank = sender.rank + 1
        sender.children.append(self)
        print(f"{self.name} receives DIO from {sender.name} and sets {sender.name} as parent (Rank: {self.rank})")
        self.network.env.process(self.send_dao())
```

```
node1 sends DIO to node2
node1 sends DIO to node3
node1 sends DIO to node4
node1 sends DIO to node5
node1 sends DIO to node6
node1 sends DIO to node7
node1 sends DIO to node8
node1 sends DIO to node9
node1 sends DIO to node10
node1 sends DIO to node11
node1 sends DIO to node12
node1 sends DIO to node13
node1 sends DIO to node14
node1 sends DIO to node15
node2 sends DIO to node1
node2 sends DIO to node3
node2 sends DIO to node4
node2 sends DIO to node5
node2 sends DIO to node6
node2 sends DIO to node7
node2 sends DIO to node8
node2 sends DIO to node9
node2 sends DIO to node10
node2 sends DIO to node11
node2 sends DIO to node12
node2 sends DIO to node13
node2 sends DIO to node14
node2 sends DIO to node15
node3 sends DIO to node1
node3 sends DIO to node2
```

Figure 3: DIO Exchange

- Nodes received DIO messages and selected their parents based on the received information. Subsequently, they sent DAO messages to their parents. For example:

```python
def send_dao(self):
    if not self.dao_sent and self.parent:
        print(f"{self.name} sends DAO to {self.parent.name}")
        self.dao_sent = True
        yield self.network.env.timeout(1)
        self.parent.receive_dao(self)

def receive_dao(self, sender):
    if sender.prefix:
        self.routing_table[sender.prefix] = sender.name
        self.update_routing_table(sender)

        print(f"{self.name} updated routing table with {sender.prefix} from {sender.name}")
        self.print_routing_table()
```

```
node2 receives DIO from node1 and sets node1 as parent (Rank: 1)
node2 sends DAO to node1
node3 receives DIO from node1 and sets node1 as parent (Rank: 1)
node3 sends DAO to node1
node4 receives DIO from node2 and sets node2 as parent (Rank: 2)
node4 sends DAO to node2
node5 receives DIO from node2 and sets node2 as parent (Rank: 2)
node5 sends DAO to node2
node6 receives DIO from node3 and sets node3 as parent (Rank: 2)
node6 sends DAO to node3
node7 receives DIO from node3 and sets node3 as parent (Rank: 2)
node7 sends DAO to node3
node8 receives DIO from node4 and sets node4 as parent (Rank: 3)
node8 sends DAO to node4
```

Figure 4: DAO Exchange

- Nodes constructed their routing tables based on the received DAO messages. For instance:

```python
def update_routing_table(self, sender):
    self.dist_to_all_nodes[sender] = 1
    self.dist_to_children[sender] = 1

    for child in sender.children:
        if child not in self.dist_to_all_nodes:
            self.dist_to_all_nodes[child] = self.dist_to_all_nodes[sender] + 1
            self.routing_table[f"{child.prefix}/{PREFIX_LENGTH+1}"] = child.name
            self.update_routing_table(child)

def print_routing_table(self):
    print(f"Routing Table for {self.name}:")
    print("{:<20} {:<20}".format("Network Prefix", "Hop"))
    for prefix, next_hop in self.routing_table.items():
        print("{:<20} {:<20}".format(prefix, next_hop))
    print("\n")
```

```
node1 updated routing table with 2001:db8:0001::24 from node2
Routing Table for node1:
Network Prefix       Hop
2001:db8:0001::24    node2
2001:db8:0003::24/25 node4
2001:db8:0007::24/25 node8
2001:db8:0008::24/25 node9
2001:db8:0004::24/25 node5
2001:db8:0009::24/25 node10
2001:db8:000a::24/25 node11


node1 updated routing table with 2001:db8:0002::24 from node3
Routing Table for node1:
Network Prefix       Hop
2001:db8:0001::24    node2
2001:db8:0003::24/25 node4
2001:db8:0007::24/25 node8
2001:db8:0008::24/25 node9
2001:db8:0004::24/25 node5
2001:db8:0009::24/25 node10
2001:db8:000a::24/25 node11
2001:db8:0002::24    node3
2001:db8:0005::24/25 node6
2001:db8:000b::24/25 node12
2001:db8:000c::24/25 node13
2001:db8:0006::24/25 node7
2001:db8:000d::24/25 node14
2001:db8:000e::24/25 node15
```

Figure 5: Routing Table Construction

- The final parent-child relationships and ranks in the network are as follows:

```
Node: node3, Parent: node1, Children: ['node6', 'node7'], Rank: 1
Routing Table for node3:
Network Prefix       Hop
2001:db8:0005::24    node6
2001:db8:000b::24/25 node12
2001:db8:000c::24/25 node13
2001:db8:0006::24    node7
2001:db8:000d::24/25 node14
2001:db8:000e::24/25 node15


Node: node4, Parent: node2, Children: ['node8', 'node9'], Rank: 2
Routing Table for node4:
Network Prefix       Hop
2001:db8:0007::24    node8
2001:db8:0008::24    node9


Node: node5, Parent: node2, Children: ['node10', 'node11'], Rank: 2
Routing Table for node5:
Network Prefix       Hop
2001:db8:0009::24    node10
2001:db8:000a::24    node11


Node: node6, Parent: node3, Children: ['node12', 'node13'], Rank: 2
Routing Table for node6:
Network Prefix       Hop
2001:db8:000b::24    node12
2001:db8:000c::24    node13
```

Figure 6: Parent-Child Relationships

- The network topology was visualized to illustrate the parent-child relationships and the routing paths more closely.

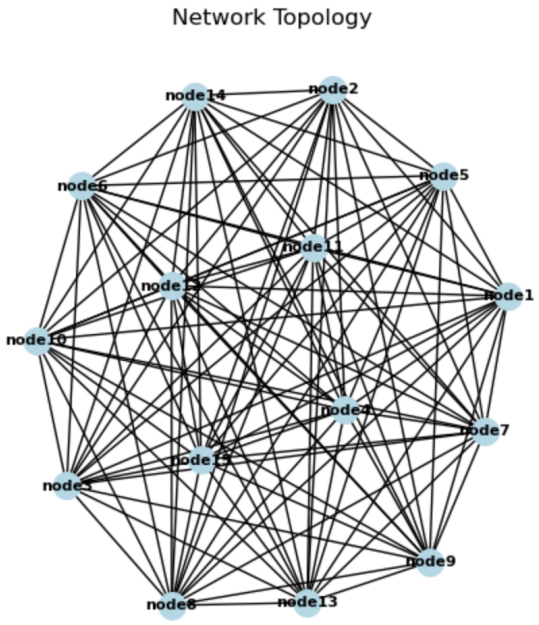

Figure 7: Network Topology

- Finally, the DODAG structure was created and visualized:
- Total time taken for this whole simulation is 0.0047 seconds, however it is much more time consuming when we experimented with a larger and complex network [fig 9].

  For 450 nodes it took 3.07 seconds, for 750 nodes it took 8.93 seconds and for 1533 nodes it took 38.6448 seconds.

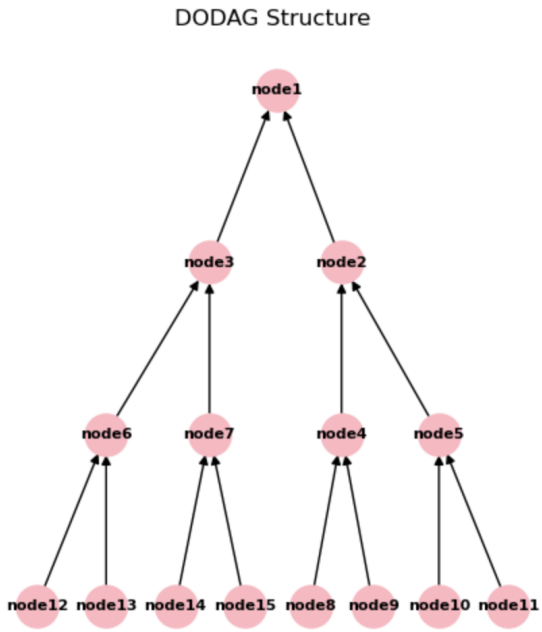

Figure 8: DODAG

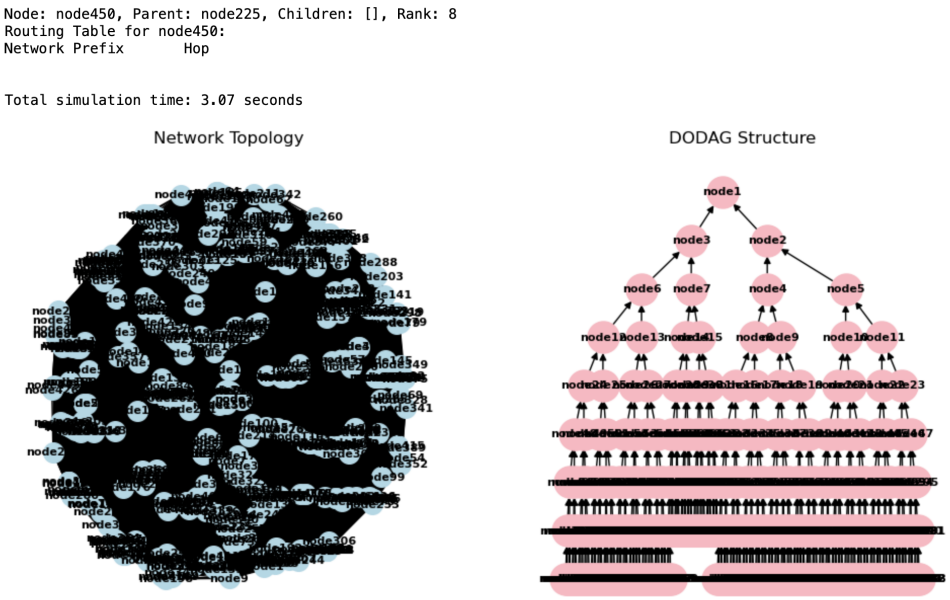

Figure 9: Experimentation

5 Workload Distribution

We distributed all the works among ourselves as:

1. Purbak: Handled visualization (DODAG and network). Also handled the network and node concepts and explored optional tasks.
2. Lena: Handled neighbour discovery, DIO and DAO exchange, and parent selection.
3. Kanak: Dealt with routing tables and their visualisation, network distribution and made the presentation.

6 Conclusion and Future Work

At the end of the whole process, we have structured a network fulfilling all the stated mandatory pivot points of the project. The simulation explains the process of control message exchange to establish parent-child relationships and construct routing tables. Key observations from the simulation are:

- **Dynamic Neighbour Discovery:** Nodes were able to identify their neighbours based on closeness and establish connections accordingly.
- **Hierarchical Structure Formation:** Through the exchange of DIO messages, nodes determined their ranks and selected appropriate parents, which has lead to the formation of a hierarchical DODAG structure.
- **Routing Table Updates:** The propagation of DAO messages ensured that nodes could dynamically update their routing tables.

We also attempted to implement additional features such as DIS messages, Trickle algorithm[3] and local repair mechanisms. Although we knew the theoretical concepts of these components, we faced challenges while trying to integrate them into our simulation code. We were not completely satisfied with the output of our implementation. As a result, we decided to drop these optional parts from the final implementation.

As our current implementation is flexible and adjustable, some of the future enhancements that could be done over the present simulation may include successfully integrating the omitted optional features such as the DIS messages, trickle algorithm[3], and local repair mechanisms[2]. On top of that, an extension to include more advanced metrics for parent selection[4], handling node mobility, and incorporating variable transmission ranges would provide more improved overall robustness.

```
node3 sends DIS
node4 sends DIS
node3 sends DIS
node4 sends DIS
node4 sends DIS
node5 sends DIS
node5 sends DIS
node1 triggers local repair
```

```python
def send_dis(self, env):
    if not self.dis_sent:
        print(f"{self.name} sends DIS")
        self.dis_sent = True
        self.dis_timer = env.timeout(self.dis_interval)
        yield self.dis_timer
        self.dis_sent = False
        self.dis_interval *= 2
        self.dis_interval += random.uniform(-self.dis_backoff, self.dis_backoff)

def trigger_local_repair(self, env):
    print(f"{self.name} triggers local repair")
    for child in self.children:
        env.process(child.send_dio(env))
    yield env.timeout(1)
```

Figure 10: Attempt on Implementing DIS & Local Repair

By addressing these aspects, future simulations can also provide deeper insights into the behaviour and performance of DODAG-based routing protocols, making them more applicable to real-world IoT (Internet of Things) deployments and other network scenarios. The continued development and refinement of these simulations will be crucial for advancing our understanding and implementation of efficient, reliable, and scalable network protocols.

Overall, this project has provided valuable insights into the workings of RPL and DODAG formation, and the collaborative effort has resulted in a solid foundation for further exploration and refinement.

References

[1] SimPy, Discrete event simulation for Python.

[2] T. Winter, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, JP. Vasseur, "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks", Internet Society, RFC 6550, March 2012.

[3] P.Levis, T.Clausen, J.Hui, O.Gnawali,and J.Ko, "The Trickle Algorithm", Internet Society, RFC 6206, March 2011.

[4] Pascal Thubert, "Objective Function Zero for the Routing Protocol for Low-Power and Lossy Networks (RPL)", RFC 6552, March 2012.

[5] Dominique Barthel , JP Vasseur , Kris Pister , Mijeom Kim , Nicolas Dejean, "Routing Metrics Used for Path Calculation in Low-Power and Lossy Networks", RFC 6551, March 2012