Received 10 September 2022, accepted 18 October 2022, date of publication 26 October 2022, date of current version 2 November 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3217215

APPLIED RESEARCH

Implementation of a LoRa Mesh Library

JOAN MIQUEL SOLÉ, ROGER PUEYO CENTELLES, FELIX FREITAG, AND ROC MESEGUER
Department of Computer Architecture, Universitat Politècnica de Catalunya, BarcelonaTech, 08034 Barcelona, Spain

Corresponding author: Felix Freitag (felix.freitag@upc.edu)

This work was supported by the Ministry of Science and Innovation of the Spanish Government through the State Research Agency (AEI) under Project PID2019-106774RB-C21, Project PCI2019-111851-2 (LeadingEdge CHIST-ERA), and Project PCI2019-111850-2 (DiPET CHIST-ERA).

ABSTRACT LoRa is a popular communication technology in the Internet of Things (IoT) domain, providing low-power and long-range communications. Most LoRa IoT applications use the LoRaWAN architecture, which builds a star topology between LoRa end nodes and the gateway they connect to. However, LoRa can also be used for the communication between end nodes themselves, forming a mesh network topology. In this paper, we present a library that allows to integrate LoRa end nodes into a LoRa mesh network, in which a routing protocol is used. Thus, an IoT application running on these nodes can use the library to send and receive data packets to and from other nodes in the LoRa mesh network. The designed routing protocol is proactive, and maintains the routing table at each node updated by sending routing messages between neighboring nodes. The implemented library has been tested on embedded boards featuring an ESP32 microcontroller and a LoRa single-channel radio. By using our LoRa mesh library, nodes do not need to connect to a LoRaWAN gateway, but among themselves. This opens the possibility for new, distributed applications solely built upon tiny IoT nodes.

INDEX TERMS LoRa, mesh network, IoT routing.

I. INTRODUCTION

LoRa is a wireless communication technology designed for the interconnection of Internet of Things (IoT) devices. Its main features are its long range (hence its name), which can transmit data over several km of distance, low power consumption and low data rate. Configuration parameters like the Spreading Factor (SF) can extend the communication distance, although at the expense of lower data rates [1].

Most IoT applications that use the LoRa communication technology adhere to the LoRaWAN network architecture [2]. LoRaWAN defines a star topology where IoT end nodes, e.g., LoRa-equipped sensor nodes, transmit their data to a LoRaWAN gateway. Typically, a gateway covers an area with many end nodes, and listens for incoming LoRa packets from them. The gateway has two network interfaces: on the one hand, a multi-channel LoRa radio; on the other hand, a wired or a wireless connection to the Internet through which to communicate the received data to a higher layer of the application usually hosted in the cloud. While a gateway is expected to be always powered and operational, end nodes typically spend most of the time in sleeping mode, and periodically wake up to send a LoRa packet to the gateway. In LoRaWAN there is no direct communication between the end nodes.

In the last few years, a number of proposals have been made to extend the LoRaWAN star topology by means of multi-hop communication, which are reviewed in the form of a survey in [3]. Multi-hop in LoRaWAN allows extending the distance between the end nodes and the gateway, without increasing the density of gateways. The underlying principle is that LoRa end nodes far away from the gateway can communicate with intermediate end nodes, which act as forwarders of their messages towards the gateway. Overall, multi-hop for LoRaWAN provides mostly an extension of the geographical reach of the IoT layer, while it mostly maintains the principles of a LoRaWAN-based IoT application.

We identify several motivations for considering a more radical shift towards true LoRa mesh network-based solutions, which address 1) the architectural limitations of LoRaWAN, 2) the opportunity for application decentralization, and 3) the need for saving cost and energy consumption.

Applications following the LoRaWAN architecture typically consider the situation where end nodes are able to reach a LoRaWAN gateway with one hop. This condition can be

The associate editor coordinating the review of this manuscript and approving it for publication was Chunsheng Zhu.

a problem in scenarios where the geographic scale of the end nodes is large and the density of gateways is low. While increasing the number of gateways can be considered as a solution to the problem, it is not always possible due to the economic cost or the geographic conditions.

Devices used for event-based or periodic data transmission to the gateway are categorized as class A end node [4]. Therefore, the LoRa traffic from applications using devices of such class is mostly on the uplink channel, i.e., LoRa packets from the data sources (end nodes) sent via radio to the LoRaWAN gateway and, from there, over the Internet towards the cloud. After sending a LoRa packet, these end nodes open two short reception windows to get any downlink messages from the gateway, which are optional. Since in LoRaWAN the downlink messages from the gateway are linked to the uplink message from the sensor, instantaneous pushes of notifications to the sensor node are not immediately available. Also, the fact that in LoRaWAN the field for the Cyclic Redundancy Check (CRC) of the payload is only available in the uplink LoRa packet makes the downlink packet less of a reliable element in the data transmission. IoT application based on LoRaWAN often aim at centralized decision-making, with the purpose of the sensor nodes solely being to send data to cloud-based services. Decisions are taken after the cloud-based processing of the sensor data. Other application scenarios, however, conceive more autonomous IoT nodes where decisions need to be taken at the device. Indeed, the current trend in performing machine learning on low capacity devices only increases this need [5]. For such autonomous IoT nodes, a certain asynchronous communication capacity is needed, e.g., for being able to inform a node about a current application context to be taken into account for local decision-making. LoRa point to point communications between end nodes, as delivered by a LoRa mesh networks, can be a solution. However, this type of communication is not considered in LoRaWAN.

The energy consumption of ICT applications has become a raising concern [6]. LoRaWAN-based IoT applications address a full software and hardware stack that covers from the IoT device to the cloud. Therefore, it involves the operation of a huge software and hardware infrastructure, which translates in significant environmental costs due to the bill of materials and the energy consumption. Nonetheless, most services forming an IoT application can be obtained from shared cloud-based computing infrastructures, which reduces the operational cost per application. IoT applications can leverage the already deployed components for LoRaWAN-based solutions, where providers often offer enterprise and community partnerships, such as The Things Network,[1] among others. Still, we can envision applications of interconnected LoRa devices which only use the computing infrastructure available on the devices themselves. Such gateway-less IoT applications, in terms of infrastructure costs, are much more affordable in regard to the needed hardware and their energy consumption.

[1]https://www.thethingsnetwork.org/

In this paper, we aim to make a significant step forward in the public availability of ready to use code that allows to deploy LoRa mesh networks. In particular, we present our library code-named *LoraMesher*, which we have developed in order to deploy LoRa mesh networks operating a routing protocol as proposed in [7]. The library is implemented in C++ and is conceived to be a part of an application code that runs on an IoT node consisting of an embedded microcontroller and a LoRa radio. Having the LoRaMesher library integrated within the IoT code running on such a node, an application is able to connect to a LoRa mesh network and send, route and receive LoRa packets from the nodes within the network.

The main contributions of this paper are:

- We describe the design and implementation of the LoRaMesher library for deploying LoRa mesh networks which include a routing protocol.
- We evaluate the library performance in experiments with different topologies using real hardware nodes.

The main potential of this library, which is freely available as open source code,[2] is to enable a new class of distributed applications that can run only at the IoT layer using LoRa interconnected nodes.

II. DESIGN AND IMPLEMENTATION

A. OVERVIEW

The LoRaMesher library is a C++ implementation of a proactive distance-vector routing protocol for enabling the communication among LoRa nodes that form a mesh network, as proposed in [7]. The target hardware for which the library is compiled is an embedded board with a System on a Chip (SoC) and a single-channel LoRa radio, like the popular ESP32 SoC-based development boards featuring an SX1276 LoRa transceiver. LoRaMesher uses FreeRTOS[3] to implement the task handlers of the receiver, sender, packet processing, routing protocol and application data bidirectional transmission.

For the interaction with the LoRa radio chip, LoRaMesher leverage RadioLib,[4] a versatile communication library which supports the SX1276 LoRa series radio available on the hardware we use. Features of RadioLib include an easy configuration of LoRa parameters, and the addition of CRC verification of the LoRa payload, if enabled. Moreover, LoRaMesher uses RadioLib to define an Interrupt Service Routine (ISR) which is executed every time a LoRa packet is detected.

In the following section, we describe in detail the design of the LoRaMesher library.

B. PACKET STRUCTURE

LoRa belongs to IoT communication technologies for building Low Power Wide Area Networks (LPWANs). LoRa targets applications in which remote sensors communicate

[2]https://github.com/LoRaMesher/LoRaMesher
[3]https://www.freertos.org/
[4]https://github.com/jgromes/RadioLib

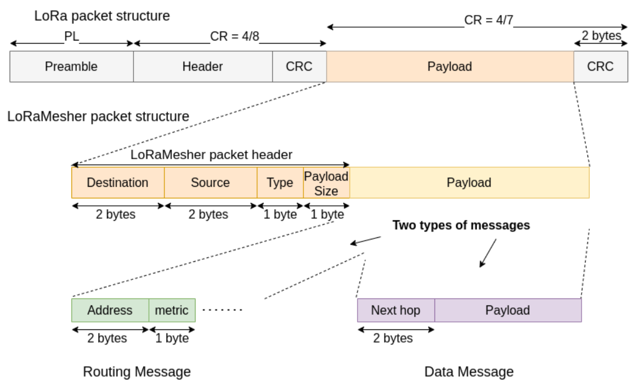

FIGURE 1. Standard LoRa packet Structure using the Explicit Header Mode (top) and the LoRaMesher packet frame inside the payload section (center). The two types of messages the library supports are also depicted (bottom).

data with low data rates. A LoRa packet is limited to 255 B. [8].

Figure 1 shows the structure of a LoRa packet and that of a LoRaMesher packet. It can be seen that the LoRaMesher packet is encapsulated within the payload field of the LoRa packet. The LoRaMesher packet itself contains a header and a payload. Following the design in [7], this header consists of 4 B which hold the address of the destination and the source node, 1 B to specify the message type and 1 B to indicate the payload size.

LoRaMesher is a library that implements a proactive routing protocol. Thus, it sends two types of messages, namely *routing messages* which allow nodes to update their routing table, and *data messages*, which contain the actual application data that is sent from one node towards another.

C. ROUTING AND DATA MESSAGES

A routing message in LoRaMesher contains one or multiple routing entries of the sender node's routing table. A data message contains an additional field of 2 B to indicate the next hop, and the actual application data as payload (Figure 1).

Routing messages are broadcasted by each node at configurable periods. The purpose of sending routing messages is to allow neighboring nodes to update their routing tables with the information from other nodes around (and, in turn, from those nodes beyond their direct reach). A routing message sets the broadcast address $0 \times \mathrm{FF}$ as the destination address. Thus, a node receiving a LoRaMesher routing packet can distinguish it from a data message. The received routing messages are not forwarded by the nodes, but are processed to update their local routing table.

Figure 2 describes the algorithm that processes a received routing packet. Once a received message is identified as a routing message, it is necessary for the node to check if the sender is already annotated in the routing table. The next step is to process the routing table entries received from the neighboring node and update the local routing table. After the

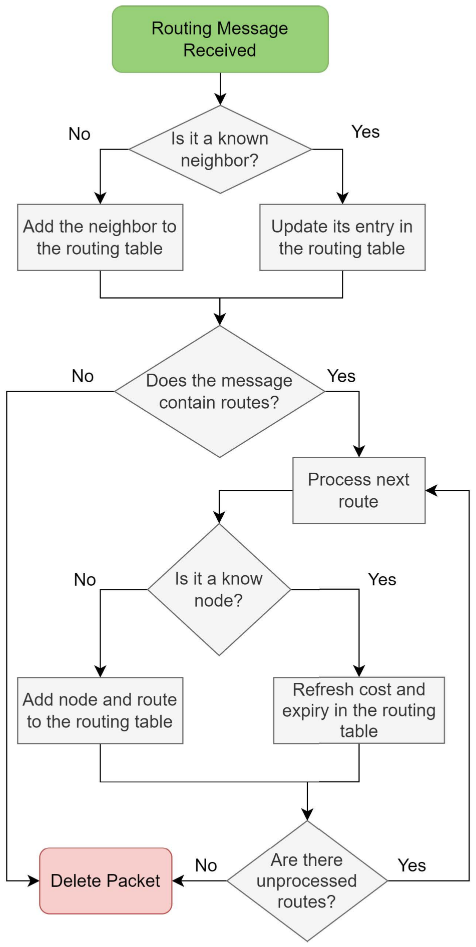
Routing Message Received
Is it a known neighbor?
No
Yes
Add the neighbor to the routing table
Update its entry in the routing table
Does the message contain routes?
No
Yes
Process next route
Is it a know node?
No
Yes
Add node and route to the routing table
Refresh cost and expiry in the routing table
Are there unprocessed routes?
No
Yes
Delete Packet

FIGURE 2. Routing message received diagram (based on [7]).

processing, the LoRaMesher packet containing the routing message is deleted.

Data messages contain the data from the application layer. The data message has a specific destination in the LoRa mesh network. When a node using the LoRaMesher library receives a data message, the following outcomes are possible: 1) the receiving node is the destination and thus the library will pass this data message to the application layer, 2) the node is not the destination, but it is the one specified in the *next hop* field, meaning that it is expected to forward the data message (for doing this, it will update the *next hop* field with the address found in its local routing table), 3) the node is not the destination of the data message nor the *next hop* and, in both cases, the node will delete the data message. A corner case might be found in the second situation above, where a node is expected to forward a data message headed to a destination not present in its routing table; in that case, the data message will also be deleted.

D. QUEUES

As explained before, a LoRaMesher packet can contain be a routing or a data message. Furthermore, received data messages may be re-routed to its destination or delivered to the node's application, while received routing messages are processed locally and generated routing messages at the node are broadcasted. This packet processing creates the need for having local structures where to store these packets. LoRaMesher implements for this purpose a set of *Packet Queues*.

Queues are also a means for the tasks in the LoRaMesher library, i.e., routines that carry out a certain type of processing of a packet, to have a data structure for sharing packets. Every time a packet is received and needs to be shared between tasks, the library creates a *Packet Queue element* that contains a priority number, the memory address of the packet and the next element. When adding this element to the queue, it is added at the first position such that its priority is higher than the next Packet Queue element.

The LoRaMesher library implements three different queues with the following purposes:

- **Received Packets (Q_RP)**: This queue is used for storing packets received from the LoRa radio and have them ready for local processing. A task is needed to fetch the received packet and add it to this queue.
- **Send Packets (Q_SP)**: This queue is implemented to store the packets that the node will send via the LoRa radio interface. A task is needed to take out the packets from this queue and transmit them.
- **User Received Packets (Q_URP)**: This queue is used by the application code of the node. When a packet is received and the destination is the application layer of the node itself, a task adds the packet to this queue.

E. TASKS

LoRaMesher is designed with six tasks to perform the different processing duties for the LoRa packets. Each task is implemented by a specific routine. The routines leverage the queues introduced in the previous section to operate with the packets. Thus, packets are added and deleted from the queues according to the operation of the task. Through the operations on the packets in the queues, tasks indirectly communicate with each other, and make requests between them.

1) RECEIVE TASK

The duty of the Receive Task is to receive a LoRa packet, get the LoRa packet payload, transform it into a LoRaMesher packet, create a Packet Queue element and add the previous packet to it, add this Packet Queue element into the Q_RP and finally notify the Process Task that a new packet needs to be processed.

This task works with an ISR. Thus, the microcontroller is switched to the Receive Task every time that a LoRa packet is detected. The ISR is provided by the RadioLib library and allows notifying the task upon every reception of a LoRa packet.

The Receive Task is designed to have the highest priority among the six tasks of LoRaMesher. This implies that the Receive Task is invoked whenever a packet is received,

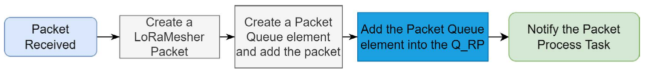
Packet Received
Create a LoRaMesher Packet
Create a Packet Queue element and add the packet
Add the Packet Queue element into the Q_RP
Notify the Packet Process Task

FIGURE 3. **Receiver task diagram. When receiving a LoRa packet, this is converted to a LoRaMesher packet and added to the Received Packets Queue.**

no matter whether the device is busy or idle, hence maximizing the number of received packets. Nevertheless, the time spent in this task is critical since, during its execution, no other packet can be received. Therefore, the operations performed by this task determine the maximum amount of packets that can be received in a certain period of time. Taking advantage of the queues, in LoRaMesher, the operations of the Receive Task consists of only the following steps (Figure 3).

1) Receive LoRa packet.
2) Get the LoRa packet payload and transform it into a LoRaMesher packet.
3) Create a Packet Queue element and add the received LoRaMesher packet to it.
4) Add the Packet Queue element to the Q_RP queue.
5) Notify the Process Task.

2) SEND TASK

The duty of this task is to get a Packet Queue element from Q_SP, send it and delete the Packet Queue element. In addition, if the packet contains a data message, it checks if the destination is inside the routing table; if so, it will update the next hop of the Data Message.

This task is notified and starts running every time a packet is added to Q_SP. In order to comply with the duty cycle regulations, every time a packet is sent, the library calculates the time on air depending on the payload size and the configuration of the library, and adds a mandatory delay before sending successive packets.

Before sending a packet, the task will start a Channel Activity Detection (CAD) routine, which will listen for LoRa preambles. If a LoRa preamble is detected, the task will wait a random delay and then start the CAD routine again. Only when no preamble is detected, the packet will be sent. This operation is an approximation to the Carrier Sense Multiple Access / Collision Avoidance (CSMA/CA) algorithm.

The Send Task is designed to have the second highest priority. Furthermore, every time and before a packet is sent, the ISR of the Receive Task is disabled, and thus it cannot receive any messages. Once the packet is sent, the ISR of the Receive Task is enabled. Figure 4 summarizes the following steps of the Send Task.

1) Notification that a packet has been added to the Q_SP.
2) Get the Packet Queue element from the Q_SP queue.
3) If the packet contains a Data Message and the destination is found in the routing table, update the Next Hop field.
4) Wait for a random delay.

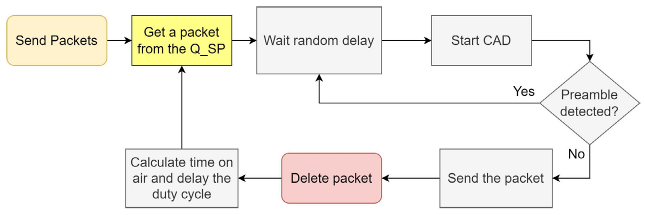

FIGURE 4. Send Task diagram. Automatic duty cycle calculation between packets and CSMA/CA.

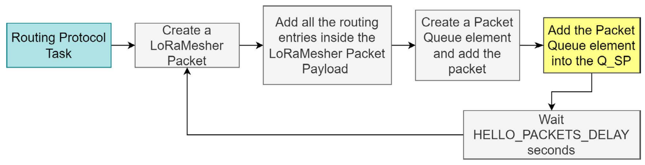
Routing Protocol Task
Create a LoRaMesher Packet
Add all the routing entries inside the LoRaMesher Packet Payload
Create a Packet Queue element and add the packet
Add the Packet Queue element into the Q_SP
Wait HELLO_PACKETS_DELAY seconds

FIGURE 5. Routing Protocol Task diagram. It creates a routing message with the actual routing table and adds it to the Q_SP to be sent as soon as possible.

5) Start CAD listening for LoRa preambles.
6) If a preamble has been detected, repeat 4.
7) Disable the Receive Task ISR.
8) Send the packet that is contained in the previous Packet Queue element.
9) Enable the Receive Task ISR.
10) Delete the packet and the Packet Queue element.
11) Calculate the time on air for the packet, calculate duty cycle and wait for it.

3) ROUTING PROTOCOL TASK

The Routing Protocol Task is periodically executed and builds a packet containing a Routing Message. This Routing Message is used to share the node's routing table with the neighboring nodes and to build the routing table in each node. A Routing Message contains the sending node's routing table, consisting of the addresses of nodes and a metric given by the number of hops with which the node can be reached. The priority of this task is less than that of the Send Task, but higher than that of the Process Task. Moreover, the periodicity of execution can be modified changing a variable named HELLO_PACKETS_DELAY. In the current implementation, the routing table maintains only the path with the least number of hops to reach a destination. Figure 5 summarizes the Routing Protocol Task.

4) PROCESS TASK

The execution of this task is triggered by the Receive Task. Every time a packet is received, the Receive Task notifies the Process Task. After receiving the notification from the Receive Task (Figure 3), the Process Task takes the first Packet Queue element of the Q_RP queue and determines whether the LoRaMesher packet is a Routing Message or a Data Message (Figure 6). In case it is a Routing Message, it will be processed as shown previously in Figure 2.

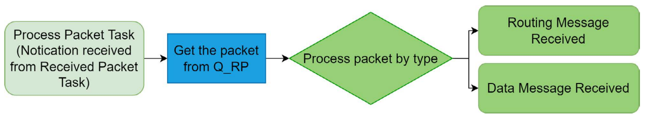

FIGURE 6. Packet Process Task diagram. When the Receiver task notifies the Packet Process task, then this task processes the first packet inside the Q_RP queue.

Otherwise, in the case of a Data Message, it will be processed as shown in Figure 7.

The Data Message process function will check first if the destination address match the address of this node. If that is the case, the Process Task will add this message into the Q_URP and will notify the User Receive Task; the user (i.e., the application layer) is ultimately responsible for deleting the packet. If the address of the LoRaMesher packet does not match the address of this node, it will check if the next hop is the address of this node; if it does not match, the packet and the Packet Queue element will be deleted. If it matches, it will be checked if the destination address is inside the routing table. In that case, it will add the Packet Queue element into the Q_SP to be forwarded. Otherwise, when the entry in the routing table with the destination of the packet does not exist, it will be deleted. Regarding the task's priorities, the Process Task has a lower priority than the Routing Protocol Task.

The following steps of the Process Task are summarized in Figure 6.

1) Get a notification from the Receive Task.
2) Get a Packet Queue element from Q_RP.
3) Process the packet by type.
   Routing Message case: if the source address is not inside the routing table then add a new route, otherwise, update the entry. If the packet contains a Routing Message, process each route. If the destination is contained in the node's routing table, update its cost. If the route does not exist, create it. Finally, delete the LoRaMesher packet.
   Data Message case: if the destination address is the node, add the packet to the Q_URP queue and notify the application level. Once processed, the application code needs to delete the LoRaMesher packet from the queue. If the destination is not the node, but the address in the next hop field exists as a destination address in the routing table, then the packet is added to the Q_SP queue and forwarded. In any other case, the packet is deleted.

5) USER SEND TASK

This task is implemented at the application level and corresponds to the code that is responsible for sending data using the LoRaMesher library. Every time the application wants to send a Data Message to another node, it needs to call the function *createPacketAndSend*, where it specifies the destination, the payload memory address and the number of bytes that have to be sent. If the application code uses

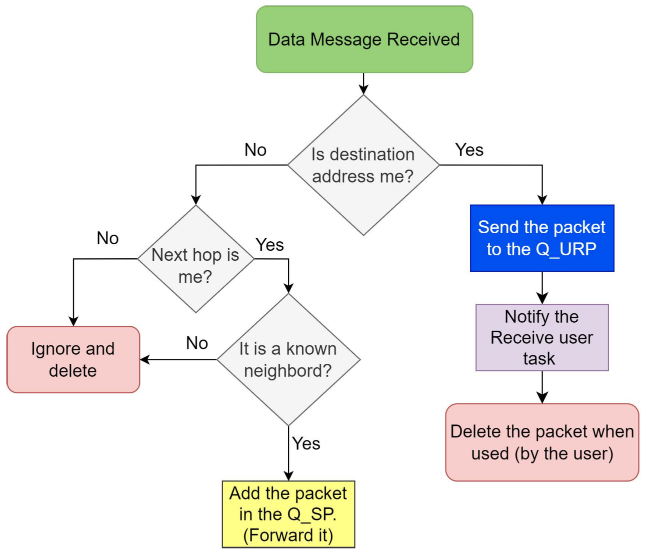
Data Message Received
Is destination address me?
No
Yes
Next hop is me?
No
Yes
Ignore and delete
It is a known neighbord?
No
Yes
Add the packet in the Q_SP. (Forward it)
Send the packet to the Q_URP
Notify the Receive user task
Delete the packet when used (by the user)

FIGURE 7. **Data Message received diagram.**

```
int dataTablePosition = 0;

for (;;) {
    //If the routing table is empty, wait 20000
        milliseconds and check again.
    if (radio.routingTableSize() == 0) {
        vTaskDelay(20000 / portTICK_PERIOD_MS);
        continue;
    }

    //If dataTablePosition is equal or greater
        than routingTableSize, start again from 0.
    if (radio.routingTableSize() <=
        dataTablePosition)
        dataTablePosition = 0;

    //Get the address from the routing table
    uint16_t addr =
        radio.routingTable[dataTablePosition].
            networkNode.address;

    //Create packet and send it.
    radio.createPacketAndSend(
        addr, userPayload, 1);

    dataTablePosition++;

    //Wait 120000 milliseconds to send the next
        packet
    vTaskDelay(120000 / portTICK_PERIOD_MS);
}
```

FIGURE 8. **Example of a User Send Task. In this example, the application code sends every 120 seconds a *userPayload* to a node contained in the node's routing table.**

structs or classes in C++, which contain the payload data, it is recommended to work with a one byte alignment in the payload data to prevent *empty* bytes inside it.

If the nodes available in a LoRa mesh network are not known beforehand and need to be discovered, then LoRaMesher offers to the application level the possibility to read the node's routing table. This way, an application can select among the valid addresses of nodes to which Data Messages will be sent to. The code fragment of Figure 8 shows how the application gets an address from the routing table and sends a specific payload to the corresponding node.

```
for (;;) {
    // Wait for the notification of Process Task
        and enter blocking
    ulTaskNotifyTake(pdPASS, portMAX_DELAY);

    // Iterate through all the packets inside the
        Q_URP
    while (radio.getReceivedQueueSize() > 0) {
        //Get the first element inside the
            Received User Packets FiFo
        AppPacket<dataPacket>* packet = radio.
            getNextAppPacket<dataPacket>();

        //Do something with the packet.
        printDataPacket(packet);

        //Delete the packet when used. It is very
            important to call this function to
            release the memory of the packet.
        radio.deletePacket(packet);
    }
}
```

FIGURE 9. **Example of a User Receive Task. Every time the application receives a packet, it gets the first Packet Queue element from Q_URP queue and processes it. Finally, it deletes the packet.**

6) USER RECEIVE TASK

The objective of this task is to get the elements inside the Q_URP queue and process the LoRaMesher packets as requested from the application level. The User Receive Task is notified by the library every time the Process Task receives and identifies a Data Message for the node. This task is implemented in the application code and needs to contain *ulTaskNotifyTake(pdPASS, portMAX_DELAY)*, which is a function of the FreeRTOS operating system that allows the task to stay in sleep mode until it is notified.

The application code has to take care of deleting the messages from the queue that are no longer needed. This deletion of messages is important for a correct memory management of the device. The LoRaMesher library provides the application with the function *deletePacket(packet)* to delete the packet.

The code example in Figure 9 shows how to integrate the User Receive Task into the application layer.

F. RELATIONSHIP BETWEEN TASKS AND QUEUES

There are a few different situations where LoRaMesher packets are created. Every time a Packet Queue element is added to a queue, a LoRaMesher packet must have been created previously. First, every time a packet is received, the library will create a LoRaMesher packet containing the payload of the packet. Second, when the application layer or the user wants to send a Data Message, the library will create a LoRaMesher packet with the payload specified by the application. Finally, every time the Routing Protocol Task is executed, a LoRaMesher packet containing a Routing Message will be created. Moreover, every time a packet shall be added

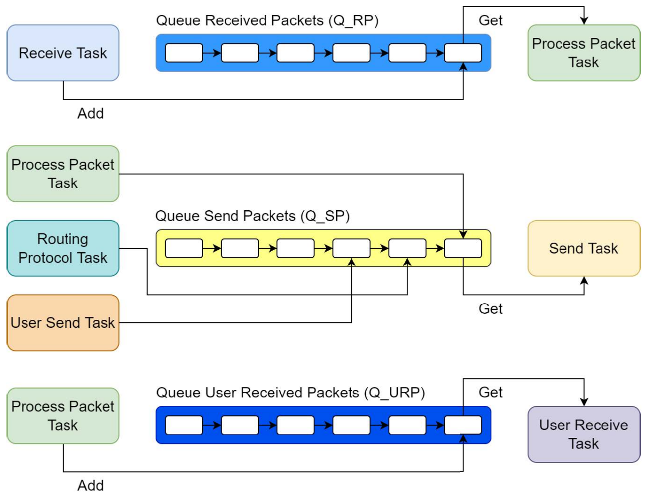

FIGURE 10. Queues diagram. The Receive task adds to the Q_RP queue, the Process Packet task gets from the Q_RP queue and adds to the Q_SP and Q_URP queue. The Routing Protocol task and User Send task add to the Q_SP queue, the Send task gets from the Q_SP queue and finally the User Receive task gets from the Q_URP queue.

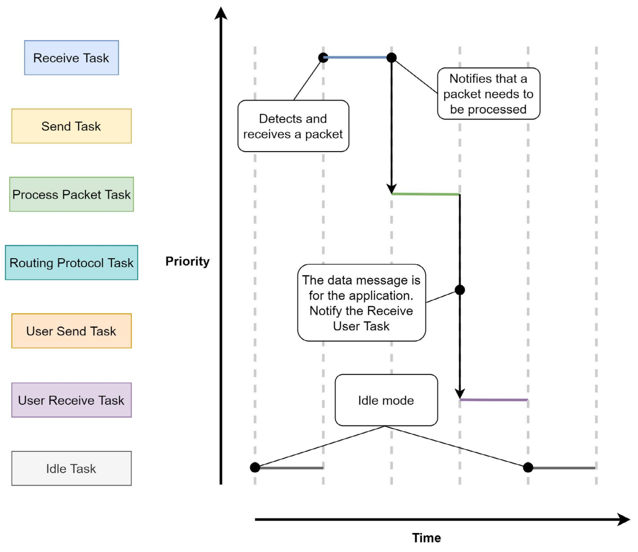
Receive Task
Send Task
Process Packet Task
Routing Protocol Task
User Send Task
User Receive Task
Idle Task
Priority
Detects and receives a packet
Notifies that a packet needs to be processed
The data message is for the application. Notify the Receive User Task
Idle mode
Time

FIGURE 11. LoRaMesher task sequence example when receiving a data message whose destination is the application of this node.

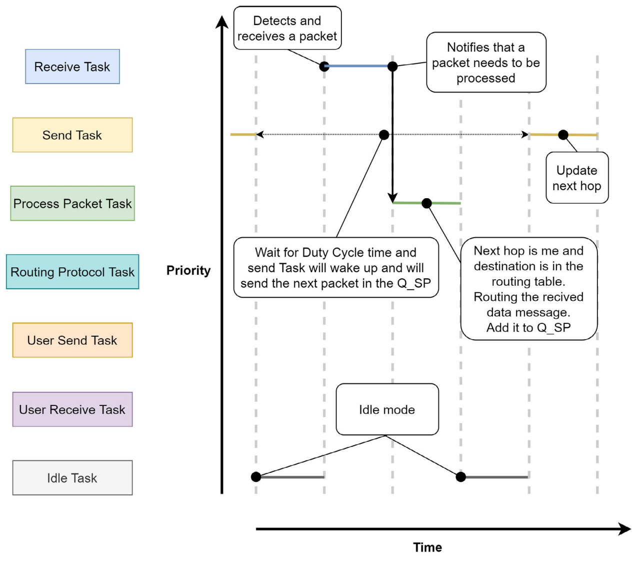
Detects and receives a packet
Notifies that a packet needs to be processed
Receive Task
Send Task
Update next hop
Process Packet Task
Routing Protocol Task
Priority
Wait for Duty Cycle time and send Task will wake up and will send the next packet in the Q_SP
Next hop is me and destination is in the routing table. Routing the recived data message. Add it to Q_SP
User Send Task
User Receive Task
Idle mode
Idle Task
Time

FIGURE 12. LoRaMesher tasks sequence when routing a received data message.

inside a packet queue, a Packet Queue element will be created containing the packet to be shared.

Figure 10 illustrates how the three queues introduced in section II-D are used by the different tasks. All packets from any origin that are received by the library are added to the Received Packets Queue. They are taken by the Process Packet Task for further packet classification. The Send Task periodically takes messages from the Send Packets Queue to transmit them to the LoRa radio. Packets to be sent can consist of new data messages created by the application at the node, routing messages created by the library itself, or received data messages which according to the routing protocol need to be forwarded by the node. Data messages that are received being the node itself the destination are added to the User Received Packets Queue, and these data messages pushed to the application layer.

Figures 11 and 12 show examples of task sequences that are executed when receiving a Data Message. It can be seen in Figure 11 that after performing the Receive task, the Process Packet task determines that the Data Message received is for the node itself. Therefore, the User Receive task is notified in order for the application to process the packet. In Figure 12 the Process Packet task determines that the Data Message received needs to be routed by the node. Therefore, the packet is added into the Q_SP queue. It is processed by the Send task that is periodically called to get and send the packets of this queue.

III. EVALUATION

A. METRICS

In this section, we describe a set of experiments that we have conducted with the LoRaMesher implementation flashed on real devices. In the experiment, we focus on the following metrics:

Packet Delivery Ratio (PDR): The ratio of the total number of packets received to the total number of packets sent from a source to the destination. This metric is obtained by logging at each node the number of received packets. The monitored packet information allows determining the PDR of each node.

Control Overhead: The ratio of the control information sent over the data received at each node. This metric is influenced by the configured periodicity for the sending of the routing messages. Dynamic networks require a higher number of routing messages in order to keep the routing tables up to date, while for stable networks this periodicity can be low.

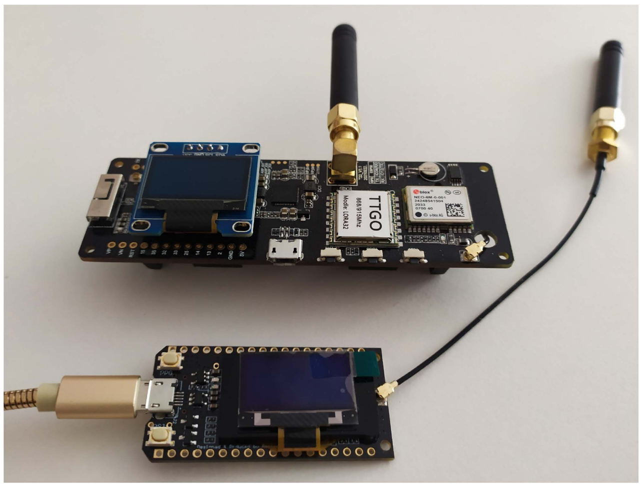

FIGURE 13. Boards with LoRa radio used for the experimentation. Top board: *TTGO T-Beam V1.1 LoRa ESP32*. Bottom board: *LoRa V1.3 ESP32*.

To determine the Control Overhead, we measure the total number of bytes of routing and data messages sent. We treat the headers of the two packet types as control information. Since both the periodicity of the routing message and that of the data messages is configurable, the metric reflects the specific experimental setting.

The following formula is used to calculate the Control Overhead: $ControlOverehead = (RoutingPacketSize + HeaderDataMessages)/DataMessagesPayload$

End-to-End Delay (EED): It is the time that it takes for an message from the source to completely arrive at the destination. This metric accumulates the time the LoRa packet is held at each node in the path to its destination. The time at each node is influenced by the timer configuration of the Send Task in the LoRaMesher library at each node. As detailed in section II-E, the Send Task is a periodic task and every time it is executed, it will send the next packet inside the Q_SP.

Hops: It is the number of nodes a packet passes from the source to the destination. This metric is relevant for validating the path of the data messages, but not for the routing messages, which are not forwarded by the nodes (see section II-C).

B. EXPERIMENTAL FRAMEWORK

For the experimentation, we use eight *TTGO T-Beam V1.1 LoRa ESP32* boards and two *LoRa V1.3 ESP32* (Figure 13). Both boards are equipped with the ESP32 SoC and a SX1276 LoRa transceiver. The boards are flashed with the developed LoRaMesher library.

In Table 1 the setting of the LoRa parameters common to all our experiments is shown. The parameters are configurable in RadioLib.

We evaluate the LoRaMesher library with different network topologies. In order to conduct controlled experiments, we chose to implement the geographic deployment of the nodes of each topology by software. Therefore, we determined for each node a set of visible nodes corresponding to the evaluated topology. When starting an experiment, upon receiving routing messages, the node constructs its routing table from the messages obtained by the visible nodes, while the routing messages received from other nodes are ignored.

The experimental configuration includes varying the number of LoRa packets sent from each node and the payload size of these LoRa packets. Both parameters influence in the number of collisions of the LoRa packets, which grows with an increasing number of packets and larger payloads.

TABLE 1. LoRa parameter setting for the experiments with LoRaMesher (EU863-870).

Parameter	Values
Spreading Factor (SF)	7
Bandwidth (BW)	125kHz
Preamble length (PL)	8
Transmission Power (P_{Tx})	+10dBm
Coding Rate (CR)	4/7
CRC checking	Header & Payload

C. RESULTS

For the experimentation, the ten nodes are started at the same time. This induces the worst case scenario for the PDR in which each node periodically sends the messages. We add an identifier attribute of 1 byte in the generic packet which, together with the source address, allows identifying a packet. This additional identifier attribute is included in the payload.

The packet size is calculated from the payload size and the header. In the experiments we use three different payloads: a 5 bytes payload (4 bytes of payload + 1 byte of the packet identifier), 105 bytes and 213 bytes. All three payloads have an 8 bytes header (6 bytes of the LoRaMesher Packet Header + 2 bytes of Next Hop of the Data Message type).

1) EXPERIMENT 1

We experiment the LoRaMesher library for a topology where the ten nodes are at one hop distance to each other (Figure 14). This experiment aims to assess from the results the correct operation of the queues and tasks in the implementation. The configuration of the experiment is presented in Table 2. All nodes send data messages to each other every 120 seconds.

Figure 15 shows the PDR obtained for the different payloads. It can be seen that the PDR decreases for larger sizes of the payload. This can be explained by the larger time on air of these packets, which increases the probability of collision with other packets.

Figure 16 looks at the detailed PDR of each of the ten nodes. A similar PDR pattern can be observed for the different nodes.

In Figure 17 the EED is compared for the different payloads. It can be seen that the EED depends on the payload

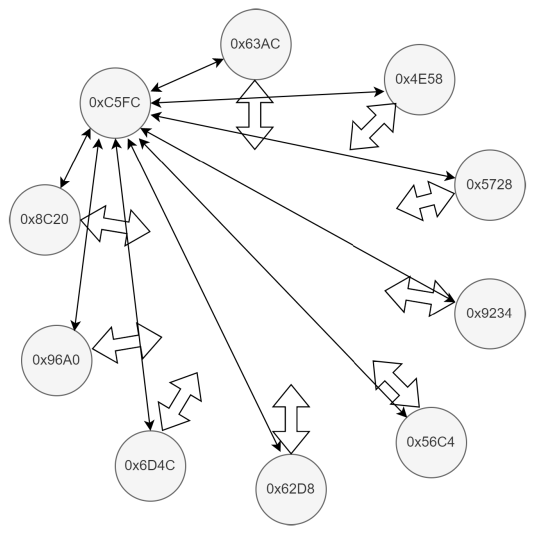

FIGURE 14. Topology of the first experiment, where all ten nodes are interconnected by one hop, as illustrated for node 0 × C5FC.

TABLE 2. Settings of the first experiment.

Configured parameter	Values
Data message payload	5 bytes first hour, 105 bytes second hour and 213 bytes the last hour
Periodicity send data message	every 120 seconds
Periodicity send routing message	every 300 seconds
Topology	1 hop connectivity between all nodes (Figure 14)
Duration of the experiment	3 : 35 hours

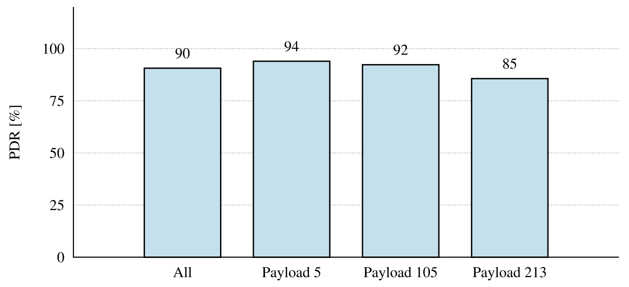

FIGURE 15. PDR by payload (experiment 1).

size. The larger packets have a larger time on air, which increases the EED.

Since all the nodes send the same amount of data, both in terms of routing and data messages, all nodes have the same Control Overhead, which is 19% in this experiment.

2) EXPERIMENT 2

We experiment a topology in which the nodes form a chain (Figure 18). This experiment aims to assess whether the routing tables built represent the given topology and that the routing of data messages at each node performs correctly. Only

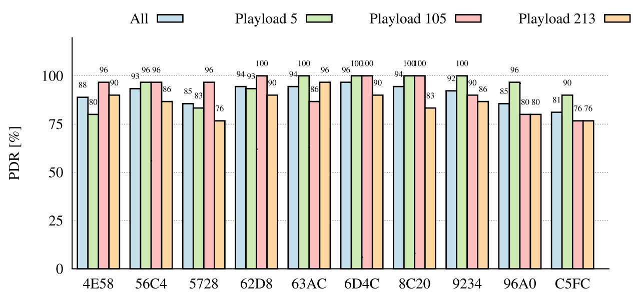

FIGURE 16. PDR by source node (experiment 1).

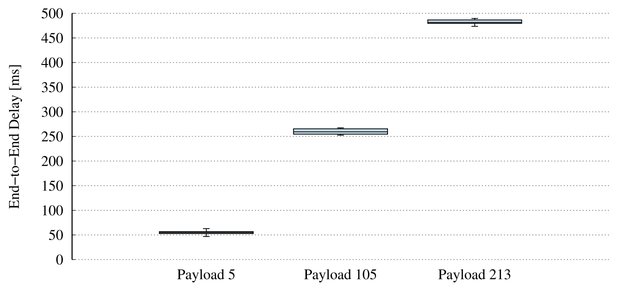

FIGURE 17. EED by payload (experiment 1).

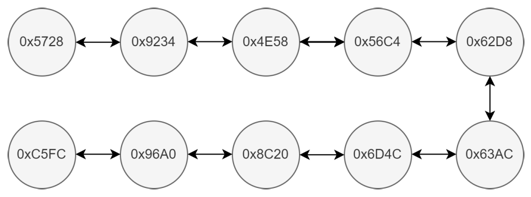

FIGURE 18. Topology of the second experiment with a chain of nodes.

the first node 0 × 5728, at one end of the chain, creates data messages that are sent to the last node 0 × C5FC at the other end of the chain. The experimental settings are described in Table 3. It needs to be mentioned that, before node 0 × 5728 can start sending data messages, it needs to wait for at least nine routing message exchanges between the neighboring nodes in order to have in its routing table the entry of the last node 0 × C5FC. With the configured periodicity of 300s for routing table exchanges, it adds 45 minutes to the experiment for the routing table updates to travel through the 9 hops of the middle nodes.

In Figure 19 it can be seen that, similar to the first experiment, the PDR is lower for the larger packets, which can be attributed to a higher number of collisions. Different to the first experiment, however, the number of data messages sent in the second experiment is lower since only the first node creates data messages. The reduced number of packets sent results in a higher PDR for the different payloads.

Figure 20 shows the obtained EED. The dependence between the payload size and the EED can be observed, where the time of the packet in the network increases with the payload size. In the configuration used in this experiment, the

TABLE 3. Settings of the second experiment.

Configured parameter	Values
Data message payload	5 bytes first 20 minutes, 105 bytes second 20 minutes and 213 bytes the last 20 minutes
Periodicity send data message	every 120 seconds
Periodicity send routing message	every 300 seconds
Topology	Chain topology (Figure 18)
Duration of experiment	1 : 45 hours

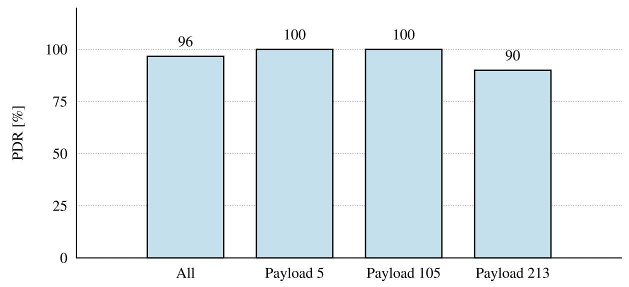

FIGURE 19. PDR by payload (experiment 2).

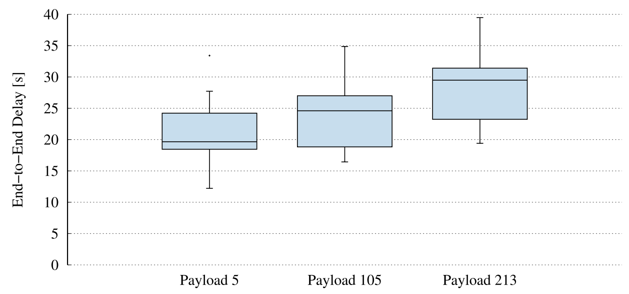

FIGURE 20. EED by payload (experiment 2).

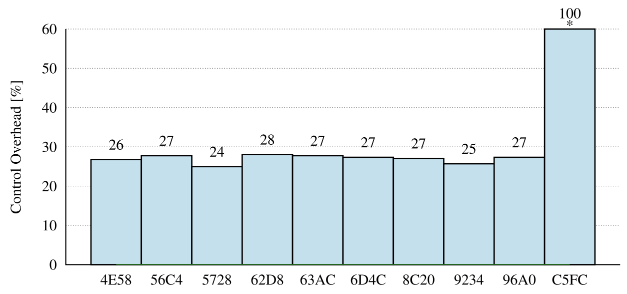

FIGURE 21. Control Overhead (experiment 2).

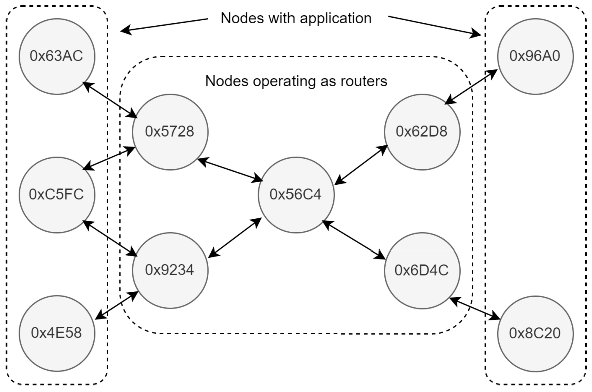

FIGURE 22. Topology of third experiment with incremental number of hops.

TABLE 4. Settings of third experiment.

Configured parameter	Values
Data message payload	5 bytes first hour, 105 bytes second hour and 213 bytes the last hour
Periodicity send data message	every 120 seconds
Periodicity send routing message	every 300 seconds
Topology	Star topology (Figure 22)
Duration of experiment	5 : 30 hours

EED is not affected by delays due to duty cycle limitations. The reason is that even the largest payload of 221 bytes can be sent every 48 seconds fulfilling duty cycle requirements, while the nodes are configured to send packets every 120 seconds.

The Control Overhead is shown in Figure 21. Compared to the first experiment, an increase of the Control Overhead in all nodes is observed. The reason is that in the topology of experiment 2 an additional 9 routing message exchanges are done to construct the routing tables before the data messages can be sent. The last node, $0 \times$ C5FC, shows a 100% Control Overhead, which is due to the fact that it is the destination of the data messages and does not send any data message itself.

3) EXPERIMENT 3

We experiment with a topology that integrates different connectivity situations of nodes (Figure 22). We configured that five nodes, i.e., $0 \times$ 63AC, $0 \times$ C5FC, $0 \times$ 4E58, $0 \times$ 96A0 and $0 \times$ 8C20, operate as hosts. Their application creates data messages to be sent to each of the other host nodes in the network. The software of the other five nodes, i.e., $0 \times$ 5728, $0 \times$ 9234, $0 \times$ 56C4, $0 \times$ 62D8 and $0 \times$ 6D4C, consists of the LoRaMesher library without application code. Therefore, these nodes operate as routers only.

The experimental configuration for this experiment is indicated in Table 4. The host nodes increases the payload size of the data packets every thirty messages. Specifically, the first payload is 5 bytes, the second 105 bytes and the last one is 213 bytes. This experiment first aims to gain insight in the performance of the middle node, i.e., $0 \times$ 56C4, which is exposed to the maximum traffic, and secondly, present the case of a distributed application hosted on IoT devices interconnected over a LoRa mesh network.

We configured the nodes to discard those messages which the radio receives but which are not part of the topology of the experiment. The PDR per payload is shown in Figure 23 and

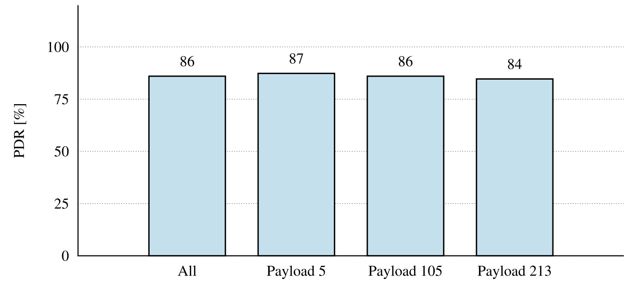

FIGURE 23. PDR by payload (experiment 3).

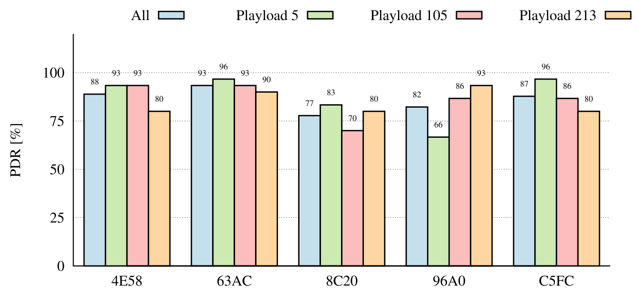

FIGURE 24. PDR by source node (experiment 3).

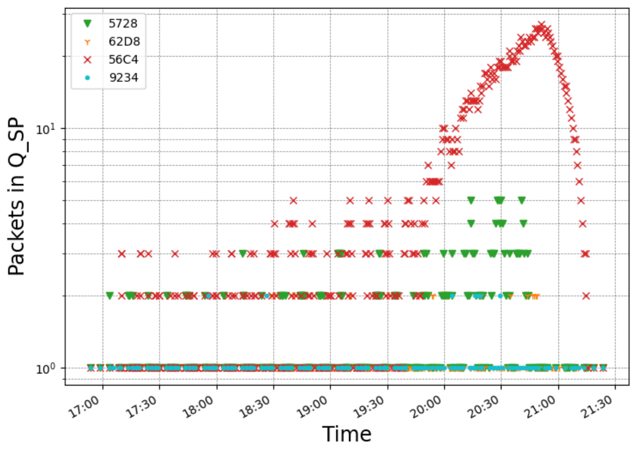

FIGURE 25. Number of packets inside the Q_SP to be sent to each node in the third experiment.

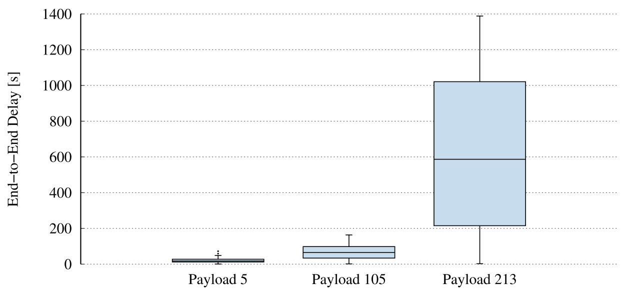

FIGURE 26. EED by payload (experiment 3).

that by node in Figure 24. In comparison with experiment 1, the PDR is lower, which can be explained by the fact that data messages are routed over several hops. Compared to experiment 2, there are more devices that create data messages. Both situations contribute to a higher number of collisions, which reduce the PDR.

Figure 25 shows the Send Packets Queue (Q_SP) of the router nodes that have more traffic, i.e., 0 × 5728, 0 × 62D8, 0 × 56C4 and 0 × 9234. As a consequence of the topology and from how the routing table is being built, node 0 × 5728 is used to forward the packets between the nodes 0 × 56C4 and 0 × C5FC. It can be seen in the figure how the packets are becoming hold in the queue of the nodes as the payload grows. That happens since the nodes that do forwarding of messages need to take into account the duty cycle. Every time a packet is prepared for sending, the nodes needs to wait for the duty cycle limitation to end before sending the next packet.

In Figure 26 the EED by payload is shown. We observe a much higher delay in comparison with Figures 17 and 20 of experiment 1 and 2, respectively. The reason is the duty cycle that affects the nodes with more traffic in the high payload scenario, in which packets need to be hold. With the low payload of 13 bytes packet (5 bytes of payload and 8 bytes of header) a packet can be sent every 6 seconds while fulfilling the duty cycle. However, if all the five host nodes send the packet through the node 0 × 56C4, one of the packets will need to wait for up to 35 seconds, which is what we experimented at some points in time. For the 113 bytes payload (105 bytes of payload and 8 bytes of header), the duty cycle allows the nodes to send this packet every 26 seconds. However, this already represents a situation that the nodes have more packets to send than the network can process without delays, and the time of the EED starts to increase. The increase is even more visible for the payload of 221 bytes (213 bytes of payload and 8 bytes of header), which the duty cycle allows sending every 48 seconds. In Figure 25 we can see for the higher traffic nodes that these packets are stored before being forwarded, which in some cases led to an EED of as much as 1400 seconds (∼ 23.33 minutes).

Figure 27 shows the Control Overhead of each node. It can be seen that the five nodes that create data message have the highest Control Overhead. This is due to the fact that, in this topology, these nodes actually send a lower amount of data messages compared to the nodes that route these data messages. Compared to experiments 1 and 2, when having finished sending the data messages, these nodes send only routing messages (which in the experiments is for a duration of more than 2 hours). The nodes 0 × 62D8, 0 × 6D4C and 0 × 9234 have a similar Control Overhead due to having similar data traffic. In comparison, the node 0 × 5728 has more data traffic due to how the routing table was built, resulting in that the packets of nodes 0 × 56C4 and 0 × C5FC pass through this node. Finally, node 0 × 56C4 has the highest data traffic in the network. Therefore, its Control Overhead is less than that of the other nodes.

IV. RELATED WORKS

Several proposals regarding multi-hop, mesh and routing for LoRa and LoRaWAN have been made in recent years. They

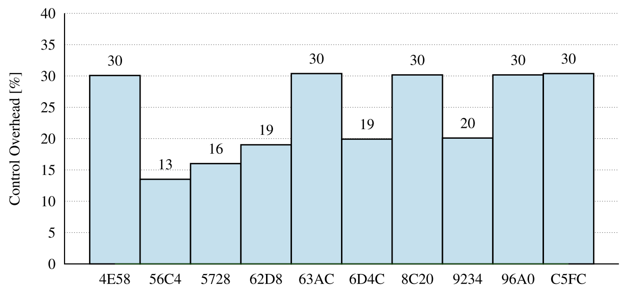

FIGURE 27. **Control Overhead (experiment 3).**

have been thoroughly classified and analyzed by researchers from different points of view: taking the application scenarios into account [9], focusing on the LoRaWAN architecture [3], or on specific implementation aspects like topology and routing [10]. The maturity and technology readiness level of these works are heterogeneous, and range from theoretical contributions to experimentally validated proposals in testbeds or real-world deployments. In this section, we analyze the proposals that are most relevant to our work.

Ebi et al. implemented a synchronous LoRa mesh protocol to extend LoRaWAN networks for end nodes monitoring underground infrastructures [11]. Their approach adds repeater nodes that bridge a synchronous LoRa mesh network segment with the regular LoRaWAN gateway. The results outperform a standard LoRaWAN network regarding the reliability of packet delivery when transmitting from range-critical locations like underground areas. The solution enhances transmission reliability, efficiency, and flexibility, but requires a precise time reference (e.g., using GPS or DCF77 time signaling) for synchronization.

Another work which investigates LoRaWAN with multi-hop is presented by Pueyo Centelles et al. [12]. Their work considers the scenario of an earthquake where principal communication infrastructures fail, and a communication system based on the LoRaWAN architecture extended by LoRa mesh networking provides an emergency network for end users. The performance of the system is evaluated extensively by simulations.

Osorio et al. [13] extended LoRaWAN with a multi-hop forwarding mechanism, which is based on a gossip algorithm. The solution improves the efficiency and flexibility of transmission, but requires intensive use of the communication channel due to the duplicated messages caused by the gossip algorithm. The number of messages is one of the limitations of LoRa and LoRaWAN, and the duty cycle further reduces the number of messages. The performance of the system was evaluated by means of a simple experiment with 8 nodes.

There are several proposals for multi-hop networks using LoRa which do not belong to, or extend, the LoRaWAN architecture. Using different strategies like routing, Time-Division Multiple Access (TDMA), clustering techniques, etc. they create tree and mesh topologies to build more decentralized and flexible networks. Most often, systems are built with single-channel radio nodes only, but some also combine them with multi-channel gateway hardware.

Sartori et al. addressed the LoRaWAN coverage extension topic with RLMAC, a Medium Access Control (MAC) layer protocol that enables Routing over Low Power and Lossy Networks (RPL) multi-hop communications based on LoRa [14]. They argue that the star topology is convenient for the ease of deployment and from a business perspective, though multi-hop could be the only option for covering very large areas with few base stations.

Lee and Ke designed and implemented a LoRa mesh networking system to ensure that indoor nodes can communicate with network servers without deploying more gateways [15]. Their design consists of a *data sink* broadcasting beacons to invite nodes to join the network. The authors state that while their solution extends the coverage of a network without installing more gateways, the number of serviced nodes would be smaller than with a conventional star topology because of the latency introduced by successive packet forwarding.

Zhu et al. improved the capacity of a multi-hop LoRa network by off-loading traffic into several subnetworks with different SFs [16]. This clustering technique results in a multiple-access dimension network where each subnetwork is rooted at a sink node with a specific SF. This enables packet transmission in parallel with multiple SFs to become feasible. The authors present a Tree-based SF Clustering Algorithm (TSCA) that conducts node allocation. Their solution requires a coordinated effort for the clustering decision-making tasks.

Mai and Kim proposed a collision-free multi-hop LoRa network protocol with low latency [17]. In their network, the sink node exchanges packets with the other nodes to construct a tree topology and assign a timeslot and a channel to each link. The authors state that their protocol provides high reliability, parallel transmissions, low latency and a minimized number of timeslots and packet size. However, it is only suitable for networks with static topology where all the collected data are targeted towards a single sink node.

Duong and Kim designed and implemented a protocol with multi-hop communication for LoRa networks covering large distances [18]. Their solution was intended for deployments where every monitoring node is placed along a line, such as a gas pipe or a high voltage line. Devices are synchronized and wake up at specific moments in time to receive data packets from their neighbors, which they can combine with their own data packets and send further along the line.

Similarly, Abrardo and Pozzebon designed a multi-hop LoRa linear network for underground environments, optimizing the nodes' sleep/wake cycles to reduce battery consumption [19]. They opted for a data propagation model with sensor nodes forming a transmission chain towards the gateway, including a synchronization mechanism when propagating data between pairs of nodes to maximize the sleep cycles' duration.

In regard to practical implementations, there are very few readily available commercial or open source products offering LoRa multi-hop and mesh possibilities.

Hester and several other contributors work on Meshtastic [20], a project for using inexpensive development boards with GPS, battery and a LoRa chip as secure mesh communicators. Meshtastic is intended for outdoor sport activities or any other situation with no Internet access. Users create a private mesh to exchange their location and send text messages to a group chat. Devices forward packets using a flooding algorithm to reach the furthest member.

Pycom provides commercial development boards and OEM products for IoT projects in the Python language. These devices can run Pymesh, a firmware for flexible LoRa mesh networking [21]. It provides encrypted ad-hoc communication over raw LoRa, implements Listen-before-talk (LBT) MAC, and supports multiple node roles (leader, router, child, and border router). The firmware also has some routing capabilities, as it claims to forward packets via the best link available. Unfortunately, Pymesh can only run on Pycom's products, making it incompatible with other vendors.

NiceRF commercially offers the SV-Mesh and LoRaStar range of LoRa transceivers. These products, available as embedded boards or packaged devices, provide serial TTL, RS232, or RS482 communication over LoRa links. They consist of a low power microcontroller and a regular LoRa transceiver. The manufacturer developed the proprietary LoRa-Pro mesh networking protocol, which defines a 2 byte addressing scheme, three network roles (node, router, node plus router), and a virtually unlimited number of routes.

Based on this review of related works, we can conclude that most of the works that propose a LoRa mesh network do not implement it, they are limited to simulations or analytical calculations. The few works that implement a network focus on addressing very specific and particular use cases. They do not perform a general implementation as the one done in this paper.

V. CONCLUSION

This paper presented the development of the LoRaMesher library, which implements a routing distance-vector protocol for the communication of nodes in a LoRa mesh network. The design of the library is explained in detail. The implementation is deployed on real devices, showing the library as ready to use code. The experimentation with LoRaMesher is done with different topologies and payloads and evaluates the packet delivery ratio, end-to-end delay and control overhead. The implementation of LoRaMesher is open source.

The results obtained from the experimentation showed that the LoRaMesher library worked correctly in the different topologies and for small, medium and large sizes of payloads. Configuration parameters such as the periodicity of data and routing messages allow the user of the library to adapt its operation to the specific application requirements. The library can be included in an IoT application code with a few steps, which can lead to distributed LoRa mesh-based applications at the IoT layer.

Future work will include extending the implemented protocols with reliable LoRa packet delivery and enabling the library to route in the mesh network application layer payloads beyond the size of LoRa packets. Having such capacities would allow the LoRaMesher library to become a communication substrate for distributed embedded machine learning applications.

REFERENCES

[1] M. A. M. Almuhaya, W. A. Jabbar, N. Sulaiman, and S. Abdulmalek, "A survey on LoRaWAN technology: Recent trends, opportunities, simulation tools and future directions," *Electronics*, vol. 11, no. 1, p. 164, Jan. 2022. [Online]. Available: https://www.mdpi.com/2079-9292/11/1/164

[2] J. Haxhibeqiri, E. De Poorter, I. Moerman, and J. Hoebeke, "A survey of LoRaWAN for IoT: From technology to application," *Sensors*, vol. 18, no. 11, p. 3995, Nov. 2018. [Online]. Available: https://www.mdpi.com/1424-8220/18/11/3995

[3] J. R. Cotrim and J. H. Kleinschmidt, "LoRaWAN mesh networks: A review and classification of multihop communication," *Sensors*, vol. 20, no. 15, p. 4273, Jul. 2020. [Online]. Available: https://www.mdpi.com/1424-8220/20/15/4273

[4] N. Sornin, M. Luis, T. Eirich, T. Kramp, O. Hersent. (2016). *LoRaWAN Specification; Version V1.0.2*. LoRa Alliance. Beaverton, OR, USA. Accessed: Oct. 20, 2022. [Online]. Available: https://lora-alliance.org/wp-content/uploads/2020/11/what-is-lorawan.pdf

[5] F. Sakr, F. Bellotti, R. Berta, and A. De Gloria, "Machine learning on mainstream microcontrollers," *Sensors*, vol. 20, no. 9, p. 2638, May 2020. [Online]. Available: https://www.mdpi.com/1424-8220/20/9/2638

[6] E. Ahvar, A.-C. Orgerie, and A. Lebre, "Estimating energy consumption of cloud, fog, and edge computing infrastructures," *IEEE Trans. Sustain. Comput.*, vol. 7, no. 2, pp. 277–288, Apr. 2022.

[7] R. P. Centelles, "Towards LoRa mesh networks for the IoT," Ph.D. dissertation, Dept. Comput. Archit., Universitat Politècnica de Catalunya, Barcelona, Spain, Nov. 2021.

[8] *LoRa Core sx1276/77/78/79 Documentation*. Accessed: Oct. 20, 2022. [Online]. Available: https://www.semtech.com/products/wireless-rf/lora-core/sx1276#documentation

[9] R. P. Centelles, F. Freitag, R. Meseguer, and L. Navarro, "Beyond the star of stars: An introduction to multihop and mesh for LoRa and LoRaWAN," *IEEE Pervasive Comput.*, vol. 20, no. 2, pp. 63–72, Apr. 2021.

[10] A. Osorio, M. Calle, J. D. Soto, and J. E. Candelo-Becerra, "Routing in LoRaWAN: Overview and challenges," *IEEE Commun. Mag.*, vol. 58, no. 6, pp. 72–76, Jun. 2020.

[11] C. Ebi, F. Schaltegger, A. Rust, and F. Blumensaat, "Synchronous LoRa mesh network to monitor processes in underground infrastructure," *IEEE Access*, vol. 7, pp. 57663–57677, 2019.

[12] R. P. Centelles, R. Meseguer, F. Freitag, L. Navarro, S. F. Ochoa, and R. M. Santos, "LoRaMoto: A communication system to provide safety awareness among civilians after an earthquake," *Future Gener. Comput. Syst.*, vol. 115, pp. 150–170, Feb. 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X20306063

[13] A. Osorio, M. Calle, J. Soto, and J. E. Candelo-Becerra, "Routing in LoRa for smart cities: A gossip study," *Future Gener. Comput. Syst.*, vol. 136, pp. 84–92, Nov. 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X22001996

[14] B. Sartori, S. Thielemans, M. Bezunartea, A. Braeken, and K. Steenhaut, "Enabling RPL multihop communications based on LoRa," in *Proc. IEEE 13th Int. Conf. Wireless Mobile Comput., Netw. Commun. (WiMob)*, Oct. 2017, pp. 1–8.

[15] H.-C. Lee and K.-H. Ke, "Monitoring of large-area IoT sensors using a LoRa wireless mesh network system: Design and evaluation," *IEEE Trans. Instrum. Meas.*, vol. 67, no. 9, pp. 2177–2187, Sep. 2018.

[16] G. Zhu, C.-H. Liao, T. Sakdejayont, I.-W. Lai, Y. Narusue, and H. Morikawa, "Improving the capacity of a mesh LoRa network by spreading-factor-based network clustering," *IEEE Access*, vol. 7, pp. 21584–21596, 2019.

[17] D. L. Mai and M. K. Kim, "Multi-hop LoRa network protocol with minimized latency," *Energies*, vol. 13, no. 6, p. 1368, Mar. 2020. [Online]. Available: https://www.mdpi.com/1996-1073/13/6/1368

[18] C. T. Duong and M. K. Kim, "Multi-hop linear network based on LoRa," *Adv. Sci. Technol. Lett.*, vol. 150, pp. 29–33, Apr. 2018.

[19] A. Abrardo and A. Pozzebon, "A multi-hop LoRa linear sensor network for the monitoring of underground environments: The case of the medieval aqueducts in Siena, Italy," *Sensors*, vol. 19, no. 2, p. 402, Jan. 2019.
[20] *Meshtastic: Open Source Hiking, Pilot, Skiing and Secure GPS Mesh Communicator*. Accessed: Oct. 20, 2022. [Online]. Available: https://meshtastic.org/
[21] *Pycom PyMesh*. Accessed: Oct. 20, 2022. [Online]. Available: https://docs.pycom.io/pymesh/

JOAN MIQUEL SOLÉ received the bachelor's degree in computer science from the Faculty of Informatics, Universitat Politècnica de Catalunya (UPC), in 2022. He is currently a Main Developer and Maintainer at the LoRaMesher Library. He is also working as a Research Engineer with UPC. His research interest includes networks and protocols for the IoT.

ROGER PUEYO CENTELLES received the Ph.D. degree from the Universitat Politècnica de Catalunya (UPC), in 2021. He is currently working as a Senior Researcher with the i2CAT Foundation. His research interests include community networks, mesh and ad-hoc networks, and the IoT.

FELIX FREITAG is currently an Associate Professor with the Department of Computer Architecture, Universitat Politècnica de Catalunya (UPC). His research interests include edge computing and federated machine learning.

ROC MESEGUER is currently an Associate Professor with the Department of Computer Architecture, Universitat Politècnica de Catalunya (UPC). His research interests include resource allocation for large-scale systems, decentralized systems applied to ambient intelligence, and bottom-up networks.