Customizing Arduino LMiC Library Through LEAN and Scrum to Support LoRaWAN v1.1 Specification for Developing IoT Prototypes

Juan M. Sulca, Jhonattan J. Barriga, Sang Guun Yoo, and Sebastián Poveda Zavala

Abstract The release of LoRaWAN in 2015 introduced specification v1.0, which outlined its key features, implementation, and network architecture. However, the initial version had certain flaws, particularly vulnerabilities to replay attacks due to encryption keys, counters, and nonce schema. To address these concerns, the LoRa Alliance subsequently released v1.1 of the LoRaWAN specification. This updated version aimed to enhance security by introducing new encryption keys, additional counters, and a revised network architecture. While the original LoRaWAN v1.0 specification spawned various device library implementations, such as IBM's LoRaWAN MAC in C (LMiC) from which Arduini-lmic was derived, none of these existing implementations adopted the improved security features of the LoRaWAN v1.1 specification. To address the lack of an open-source implementation for v1.1 end devices on open hardware platforms and to leverage the security enhancements of v1.1, a solution was devised and implemented to adapt the Arduino-lmic library. This adaptation process followed the principles of continuous improvement derived from the LEAN software development methodology, combined with the utilization of the Scrum framework.

Keywords LoRaWAN · Internet of Things · LoraWAN-MAC-in-C · Low power wide area network · Arduino · Library

J. M. Sulca · J. J. Barriga · S. G. Yoo (✉) · S. P. Zavala
Facultad de Ingeniería de Sistemas, Escuela Politécnica Nacional, Quito 170525, Ecuador
e-mail: sang.yoo@epn.edu.ec

J. M. Sulca
e-mail: juan.sulca@epn.edu.ec

J. J. Barriga
e-mail: jhonattan.barriga@epn.edu.ec

S. P. Zavala
e-mail: sebastian.poveda@epn.edu.ec

Smart Lab, Escuela Politécnica Nacional, Quito 170525, Ecuador

P. P. Joby et al. (eds.), *IoT Based Control Networks and Intelligent Systems*, Lecture Notes in Networks and Systems 789, https://doi.org/10.1007/978-981-99-6586-1_14

1 Introduction

LoRaWAN is a low-power wide area network protocol (LPWAN) focused on Internet of Things applications [1]. LoRaWAN has several benefits compared to other LPWAN technologies. LoRaWAN uses a free spectrum for transmission which represents no cost. In terms of development it is opened as it allows to customize solutions based on hardware and software. In 2015, LoRa Alliance published the first specification of LoRaWAN (v1.0) [2]. From this point onwards, the specification got revised several times, originating a division in the specification. On one side, the specification got overhauled with new encryption keys, and algorithms in specification v1.1 [3]. On the other side, the revisions of the original specification gave place to specifications v1.0.2 and v1.0.3 [4]. Although specification v1.1 is compatible by default with all v1.0.x specification family; most of the implementations of the specification only focused on v1.0.x [4].

Since the growth of IoT research in recent years, several implementations of the LoRaWAN specification have been released for the main development platforms like Arduino. In 2016 IBM released LoRaWAN MAC in C (LMiC) as an open-source implementation for the LoRaWAN v1.0 specification. Based on IBM's implementation, the code was ported to work with the Arduino environment giving birth to the Arduino-lmic which currently supports LoRaWAN v1.0.2 and v1.0.3 [5]. Even though LoRaWAN v1.1 has improved security characteristics compared to v1.0.x family; there are no open-source end-device implementations that work with such version, despite the existence of server-side deployments with support for both LoRaWAN v1.0.x and v1.1.

Some researchers, like [6–8] do not specify which end device implementation is being used, so most of these libraries are not released to the general public. Checking code repositories and IoT related forums like The Things Network [9], a list of end device implementations was found in [10]. From the listed implementations, only [11] supported LoRaWAN v1.1, with the limitation of only being able to use class A devices with OTAA and did not support rejoins. Due to the necessity of testing, developing, and creating new devices and sensors based on LoRaWAN there is an opportunity to take advantage of the improved features of v1.1. Taking this into account, in this work a re-engineered version of Arduino-lmic was developed to support the v1.1 specification of LoRaWAN. To the best of our knowledge this is the first available source code that implements LoRaWAN v1.1 for development over Arduino End-Devices.

The rest of the paper is organized as follows. Section 2 presents the methodology applied to this project. Section 3 describes the changes done to Arduino-LMIC in order to comply with the LoRaWAN v1.1 specification. Section 4 presents the results obtained during several tests to verify its functionality. Lastly, Sect. 5 presents the conclusions.

2 Methodology

The continuous improvement cycle characteristic of LEAN software development [12] will be the methodological foundation for the present work combined with the Scrum framework for the adaptation of the code. The project was decomposed into the following phases: identification, planning, execution, and review as shown in Fig. 1. The use of LEAN is key to provide a set of phases to carry out our project. LEAN was chosen as it has a common and generyc cycle that could be widely applied to software development or project management.

During the identification phase, LoRaWAN v1.0.x and v1.1 specifications were compared to extract similarities and differences between both them and abstract this to the Arduino-lmic code. Upon inspecting and understanding the code of the Arduino-lmic, a list of all the required changes for implementing LoRaWAN v1.1 was specified.

In the planning phase, the changes specified from the identification phase were organized based on how they are used. Based on activation methods, message types, and the expected behavior of LoRaWAN v1.1. With the organized changes, the project backlog was created. Each story takes the library as its user and focuses on solving the library's needs, e.g., having the new encryption schema to communicate with LoRaWAN v1.1 infrastructure.

The development environment was set during the planning phase. During this process, the main tools and infrastructure needed to adapt the library were installed and configured. Tools like Visual studio code and Platformio were key to the adaptation of the library. To experiment with the existing implementation and test the new features of the adapted library, a complete LoRaWAN network was implemented using AWS, Chirpstack, a raspberry pi 3, and a TTGO LoRa32 device.

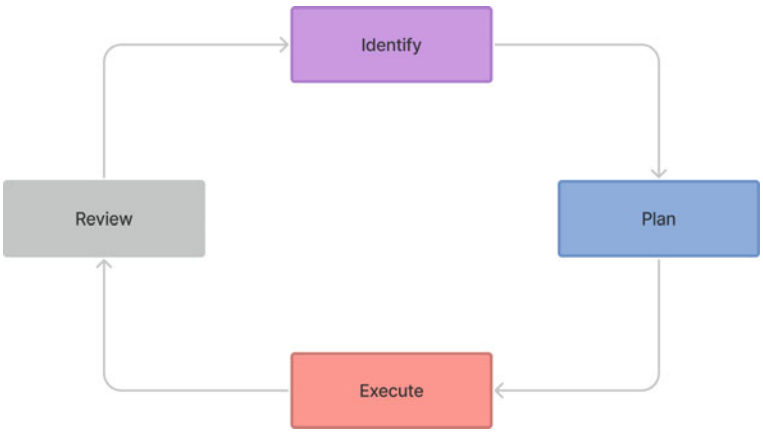

Fig. 1 Continuous improvement cycle

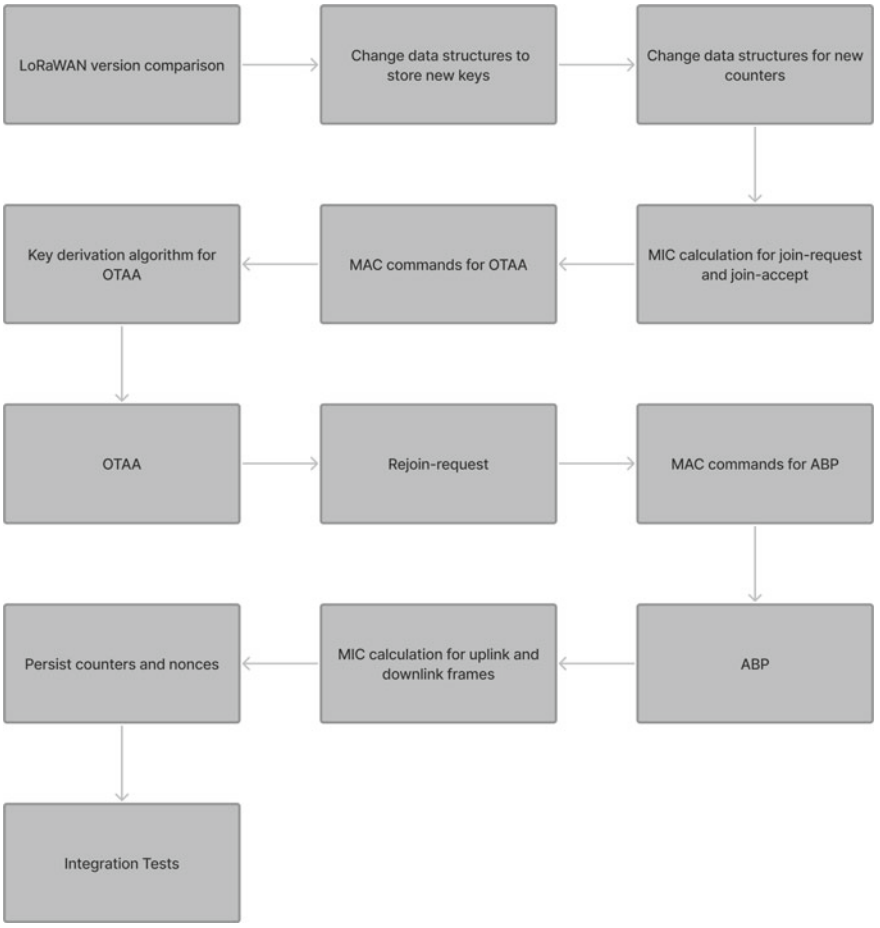

Fig. 2 Changes to Arduino-lmic to comply with LoRaWAN v1.1

Next, during the execution phase, all the changes found during the first and second phases were grouped based on the activation method. Figure 2 shows a general view of the changes implemented to the library based on the comparison done during the planning phase.

Lastly, in the review phase, a set of validation tests were proposed to check if the adapted version is capable of communicating with a LoRaWAN v1.1 network seamlessly and without a significant loss of performance related to the additional security measures required by LoRaWAN v1.1. The test suite aims to validate all the possible scenarios supported by the original implementation in which a LoRaWAN v1.1 device needs to work; these scenarios include, ABP and OTAA activation process, sending and receiving messages in both classes A and C, confirmed and unconfirmed messages.

3 Proposed Work

This section describes in detail the steps performed to apply the selected methodology (see Fig. 1) that let us develop the changes into LMiC library to support LoRaWAN v1.1. The proccess followed is described next.

3.1 Identification

During this phase, a comparison between both LoRaWAN technical specifications was performed alongside the reverse engineering of the Arduino-lmic implementation. The base for this comparison was LoRaWAN v1.0.3 [13] and LoRaWAN v1.1 [3]; LoRaWAN v1.0.3 was chosen because it was the latest supported by Arduino-lmic. The scope of the comparison was limited only to class A devices because class B have not been tested and class C are not supported at all.

Table 1 provides a list of acronyms used to describe some of the fields of the LoRaWAN v1.1 specification for LoRaWAN frames structure.

The first notable change between the two versions of LoRaWAN was the MAC message format. For LoRaWAN v1.0.3 the Message Integrity Code (MIC) of the join accept is not encrypted and in LoRaWAN v1.1 the MIC is encrypted. Also, a new message type called Rejoin-request is added in the specification.

The next changes are present in the physical payload format, where the minimum MAC payload size is increased from 1 byte in LoRaWAN v1.0.3 to 7 bytes in

Table 1 LoRaWAN acronyms list

Acronym	Description
MHDR	MAC header
MACPayload	MAC payload
MIC	Message integrity code
FHDR	Frame header
FPort	Frame port
FRMPayload	Frame payload
DevAddr	Device address
FCtrl	Frame control
FCnt	Frame counter
FOpts	Frame options
DevEUI	Device EUI
AppEUI	Application EUI
AppKey	Application key

Table 2 Differences between LoRaWAN specifications

Element	LoRaWAN v1.0.3	LoRaWAN v1.1
Rejoin-request	N/A	New message type
Join-accept	MIC not encrypted	MIC encrypted
MAC payload size	1 byte–M	7 bytes–M
MHDR Mtype	N/A	Rejoin-Request: 110
Frame counter (FCnt)	2-counter schema	2-counter and 3-counter schema, persisted in non-volatile memory
Frame options (FOpts)	Not encrypted	Encrypted with NwkSEncKey
MIC	Single algorithm for uplinks and downlinks	Separate algorithms for uplink and downlink
MAC commands	No MAC commands for ABP or OTAA	New MAC commands for OTAA and ABP
AES keys	2	6
OTAA	Requires DevEUI, AppEUI, AppKey	Require DevEUI, JoinEUI, AppKey, NwkKey
ABP	NwkSKey, AppSKey	FNwkSIntKey, SNwkSIntKey, NwkSEncKey, AppSKey
JoinNonce	AppNonce	Name changed and persisted in non-volatile memory

LoRaWAN v1.1. In addition, a new value for the MType of the MHDR is added to accommodate the new message type (Rejoin-request).

Another difference is in the frame counter (FCnt) schema which uses two counters in LoRaWAN v1.0.3. In addition to implementing the two-counter schema, V1.1 adds a second counter schema which uses 3 counters and is used only when the device interacts with a LoRaWAN v1.1 network. Also, all counters must be persisted in non-volatile memory of LoRaWAN v1.1 devices.

Other changes found between the two versions of LoRaWAN are the encryption of some fields of the message; e.g., the FOpts field is not encrypted in v1.0.3 but encrypted in v1.1. Likewise, the key derivation algorithms vary due to the additional encryption keys added to v1.1 changing from 2 session keys in v1.0.3 to 6 session keys. Also, the MIC calculation algorithm uses the new counter schema and keys.

Equally important new MAC commands are added in LoRaWAN v1.1 to adjust network parameters after the device joined a network either using OTAA or ABP; as well as new commands to adjust session parameters during the device execution.

Besides, the activation procedures have slight variations to support new encryption keys. Mainly key derivation algorithms and MAC commands sent after activation are changes introduced in v1.1. Table 2 summarizes the changes done to the Arduino-lmic implementation.

Table 3 User stories for the adaptation

As	I want	So that	Story points
End-device	To store the new encryption keys	I can implement the algorithms of LoRaWAN v1.1	13
End-device	To implement the new frame counter schema	I am able to communicate with a LoRaWAN v1.1 network	8
End-device	To implement the OTAA process	I interact with a LoRaWAN network securely	13
End-device	To implement the ABP process	I interact with a LoRaWAN network securely	13
End-device	To calculate the MIC of messages using the LoRaWAN v1.1 specification	I can guarantee the integrity of the messages	8
End-device	to store in non-volatile memory the nonces and counters	I am able to reconnect to the network and restore the session context after a restart	5

3.2 Planning

To create the project backlog, all the identified changes were written in form of user stories for feeding product backlog. To create user stories, the device assumed the user role; because Arduino-lmic exposes a low-level API oriented to other developers and should be completely transparent to the final user. The Table 3 shows the initial product backlog.

The operation of LoRaWAN v1.1 was divided into 5 parts: End-device activation, messages, MAC commands, encryption/decryption, and MIC calculation. After the activation of any device using either OTAA or ABP; the end device has all the session context to send messages including the encryption keys, counter values, and nonces.

Then, set up the development environment to adapt and test the library. For this, ChirpStack; an open-source LoRaWAN network server stack, was used to deploy the Network Server, Application Server, Join Server, and gateway firmware. When using ChirpStack it is not mandatory to use a Join Server because the Network Server can be configured to filter the join-request messages by JoinEUI or by connecting to an external Join Server.

In order for the ChirpStack infrastructure to work, some additional components need to be deployed. A Mosquitto Broker, Redis Cache, and two PostgreSQL databases need to be provisioned to ChirpStack to communicate and persist data and configurations. To host all the network infrastructure AWS was chosen due to its low cost, and experience with the platform. See Fig. 3

To implement the infrastructure, an AWS Lightsail instance was provisioned to host the Mosquitto Server, ChirpStack Application Server, Network Server, and Redis Cache. For the database, a PostgreSQL instance was provisioned using AWS RDS; a specialized service to host relational databases. The communication between the

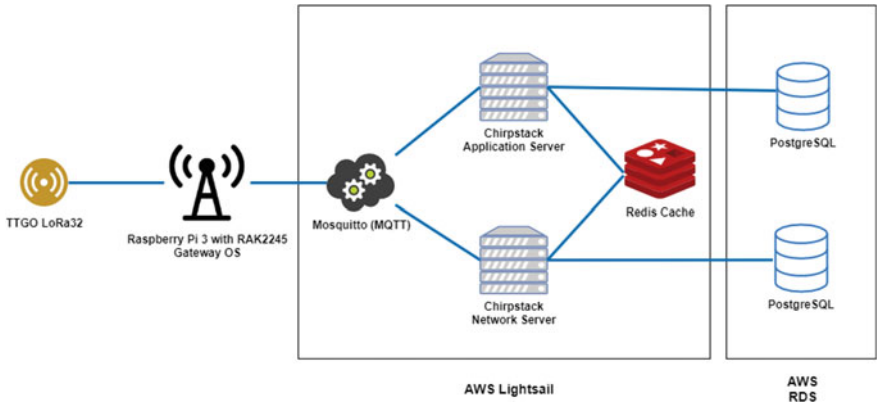

Fig. 3 LoRaWAN network deployed in AWS

Lightsail instance and the database was configured inside the same subnet and VPC in AWS to be secure and completely isolated from the Internet. Only the Mosquitto Broker and the administration web page from the network server were exposed to the internet. Figure 3 shows the full architecture in AWS.

A Raspberry Pi 3 with a RAK 2245 LoRaWAN shield and ChirpStack gateway OS was configured and deployed as gateway. It was configured to move messages from the end device to the cloud using MQTT; MQTT is a messaging protocol used to collect data under publisher/subscriber schema. The gateway was connected to the Internet through Ethernet.

Finally, for the end device, a TTGO LoRa32 which is supported by the original implementation of the Arduino-lmic library and because it relies on ESP32 platform which is widely documented and supported.

3.3 Execution

The existing implementation of Arduino-lmic is based on a single function called engineUpdate_inner which determines the current state of the library using a series of flags, and values on a structure called lmic_t, and callbacks to mutate the flags and the lmic_t structure.

For the library to work some functions need to be implemented by the client in order to set the encryption keys and EUIs in the case of OTAA devices or use the LMIC_setSession in the case of ABP devices. To configure other parameters like region and frequency, this needs to be done by setting constants during compilation. This makes it impossible to dynamically change the region during the execution of the device.

During the modification of the library, not only the functionality was changed, but the semantic of the implementation in order to use the same semantic as the technical specification. For example, the library used ArtKey for the (Application Key); which was changed to AppKey (as in the specification). Also, new function parameters were named using more descriptive names to help with the readability and maintainability of the codebase. The first step of the implementation was to store the counters and keys required by LoRaWAN v1.1 in the lmic_t data structure. All keys used by LoRaWAN are AES keys of 16 bytes length; in v1.1 the following keys were added: NwkSEncKey, SNwkSIntKey, FNwkSIntKey, and AppSKey which are needed for both ABP and OTAA. For OTAA operation, the JSIntKey and JSEncKey were also added to be used during the join process. The next step was to implement two flags on the lmic_t structure to handle sending and receiving ResetInd, RekeyInd, ResetConf, and RekeyConf MAC commands belonging to ABP and OTAA join procedures. These commands need to be sent after setting a session between the device with the server, and then the server has to respond with the corresponding Conf MAC command in order to acknowledge the join procedure.

The structure of the messages was updated with the MAC Header (MHDR) to adapt the new message types and length calculation of the messages which is needed for the MIC and message encryption.

Later, the MAC Payload was updated with the structure presented in LoRaWAN v1.1, FHDR (1 byte), FPort (1 byte), and FRMPayload (1–M byte). Under v1.1, a valid payload needs to have FHDR but can omit FPort, and FRMPayload.

Another change was an update to the FHDR with the following information DevAddr (4 bytes), FCtrl (1 byte), FCnt (2 bytes), FOpts (0–15 bytes), where the FOpts were encrypted using the NwkSEncKey. FCnt field was updated to include new counter schema presented in v1.1. The algorithms used to encrypt this field includes a specific block that needs to be generated using the device address, the frame counter, and the direction of the message.

Another key difference is the frame counter schema. In LoRaWAN v1.0.x, there are two counters for the messages, i.e., FCntUp for uplink messages and FCntDown for downlink messages. On the other hand, the v1.1 specification uses a three-counter schema where FCntUp increases with every uplink, NFCntDown for all messages with no port or targeted to port 0, and AFCntDown for messages with destination to all other ports. In addition, for updating the frame counter schema, two new counters were added in the OTAA join procedure. RJcount0 for rejoins of type 0 or 2 and RJcount1 for rejoins of type 1. The updated OTAA join procedure adds a new type of message called rejoin request with three different types to adjust the parameters of the device on the fly.

The next step was updating the format and implementation of the Join-request, Join-accept, and Rejoin-request messages. One of the key differences is that, in v1.1, some values like the DevNonce and JoinNonce need to be persistent in the device to successfully join a network and validate the Join-accept message from the network server. The DevNonce starts at 0 when the device is initialized and increases in

Table 4 LoRaWAN v1.1 frame header format

FPort	Frame type	Key
0	Uplink/downlink	NwkSEncKey
1–255	Uplink/downlink	AppSKey
–	Join-accept	NwkKey/JSEncKey

one with each Join-request. On the other hand, the JoinNonce will always need to be greater than the last valid received JoinNonce as it is used to calculate the End-device session keys.

Considering that the key derivation algorithms are almost the same between LoRaWAN v1.0.3 and v1.1; but with some variations, v1.1 utilizes 6 encryption keys instead of the 2 keys present in v1.1. Taking this into account, the original implementation was only used as a guide for the implementation of the derivation of the new keys.

Additionally, the OTAA join procedure was adjusted to incorporate the new keys, counters and message formats described in the LoRaWAN v1.1 specification. One specific aspect that needed to be adjusted is the MIC calculation since it depends on message length, the use of the newly added integrity keys, and the new counter schema. At the same time, the keys used to encrypt the FRMPayload before the MIC calculation were adjusted to follow the schema described in the new specification. The used encryption key depends on the frame type (Uplink, Downlink, or Join-accept) and the port where message is directed to (see Table 4).

In general, the session keys are used to encrypt and decrypt the messages except when the message is a Join-accept one. In case of the Join-accept responds to a Join-request, the key used is the NwkKey since the session keys are not yet derived. On the other hand, if the Join-accept responds to a Rejoin-request, the key used for encryption is the JSEncKey which corresponds to the Join Session Encryption Key.

Once the device has joined the network either using OTAA or ABP, the device can send frames to sever. These frames can be empty frames, frames with MAC commands, with application data, or with MAC commands and application data. Every payload sent through the network needs to be encrypted using AES with 128-bit keys.

Since the original implementation of the library already included an implementation of the AES algorithm, there was no need for implementing it or making any adjustments. The main changes made were the structure of the payloads, the keys used for encryption, and the A blocks for the encryption process

Then, refactor the ABP process, which uses the same keys as OTAA when the device has joined the network but needs the keys to be set before, so there is no key derivation procedure. One last difference between these two procedures is that a MAC command is sent to the server once the session is established, and a MAC command is used to acknowledge successful communication.

Four MAC commands were added to the library implementation, ResetInd and ResetConf for ABP devices, and RekeyInd and RekeyConf) for OTAA devices. The ResetInd and RekeyInd commands need to be sent on every frame after the device joins the network until the corresponding ResetConf or RekeyConf are sent as part of the server response.

In order to set all root keys, the original implementation of the library uses functions that need to be implemented by the client. Using this as a template, similar function declarations were created not only to set the keys of the device but also to persist counters, nonces, and data to know if the device has been restarted or not.

3.4 Review

In order to validate the adaptation of the library according to the requirements specified during the identification phase, a set of validation tests were proposed. Each test was performed on a real network deployed on AWS using ChirpStack (an open-source LoRaWAN server implementation), a Raspberry Pi 3 working as a gateway, and a TTGO LoRa32 as the end device.

The following test suite (Table 5) was proposed in order to test the correct execution of different join procedures, communication of different types of Uplinks or Downlink messages, and other possible use cases of the library.

Figure 4, details a summary of the outcomes obtained on every stage of the applied methodology.

Table 5 describes the way to test that our proposed solution complies with LoRaWAN v1.1 specification. We extracted the feautes from the specification that need to be present in the library code. The features to test are related to LoRaWAN v1.1 activation processess, for each feature we have described the type of the test that need to be performed. These tests aim to show that our implementation has not changed the activation process, also that the activation process complies to the new requirements of the specification and that uplink and downlink messages are able to be delivered included new security features as described in LoRaWAN v1.1 specification.

Table 5 LoRaWAN v1.1 validation test suite

Test	Feature to test
Class A OTAA unconfirmed uplinks	OTAA, unconfirmed uplink MIC
Class A ABP confirmed uplinks	ABP procedure, confirmed uplinkns MIC
Class A ABP confirmed downlinks	Downlink MIC
Key persistence and device restart using ABP	Restore ABP session
Key persistence and device restart using OTAA	Restore OTAA session params and join
LoRaWAN v1.0 Class A ABP versus LoRaWAN v1.1 Class A ABP	Performance test compared to the original implementation

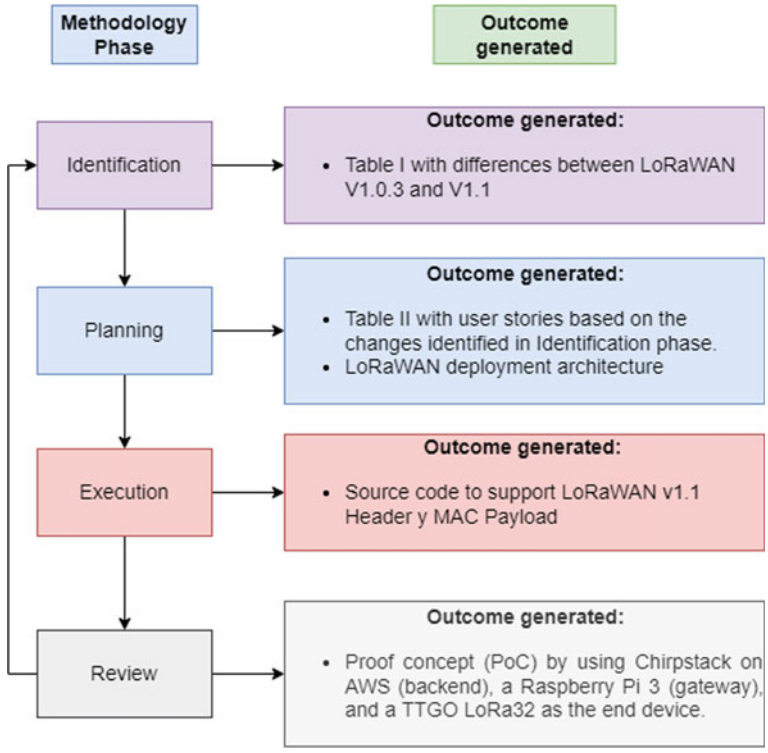

Fig. 4 Methodology and outcomes generated

Our solution was deployed in a Proof of Concept (PoC) where we used open hardware (Arduino) to validate that changes are not tied to a specific hardware platform. In contrast to other implementations, our solution is free to access and use, it uses open software libraries that can be deployed over any Arduino-based platform. To validate that LoRaWAN is working, we used an older version of LMiC library (supports LoRaWAN 1.0.x) and our proposed version to perform tests that validate ABP process and any performance changes within the used device.

4 Result Analysis

The use of LEAN allowed us to establish guidelines to carry out and to continously improve this project. Scrum, has allowed us to build during the execution phase of LEAN, a minimum viable product (MVP) by developing and implementing LoRaWAN v1.1 over a Proof of Concept scenario (PoC). User stories, were key to identify and prioritize the scope of this solution.

All the tests proposed during the review phase were executed successfully with promising results. For each of the tests, a new device was created on the server. Each device was started with all of the counters in the initial value and no established session. It means that devices are started out as "brand new" to derive all session keys in the case of OTAA or correctly beginning communication in the case of ABP. Each test consisted on a set of messages being sent (10, 25, 50, 75, 100), depending on the number of messages, each stage lasted 5–35 min approximately. The tests were performed for the following types of messages: Unconfirmed uplinks, Class A confirmed uplinks, Class A ABP confirmed downlinks, Key derivation and persistence (OTAA and ABP), and ABP for the two versions. To carry out the tests, two devices (TTGO LoRa 32) named as A and B were used to deploy the library built within this project. Device A was setup to use LoRaWAN V1.0.3. Device B used the version developed by us (LoRaWAN V1.1). After session key derivation, several messages were sent (10, 50, 100) to measure time taken (in seconds) for the whole infrastructure to proccess different messages sent.

4.1 LoRaWAN v1.1 Class A OTAA Unconfirmed Uplinks

The first test consisted in setting up a device to join the network using OTAA and then sending unconfirmed uplinks to the server. The device could send data messages containing the payload "Hello, World!" at an interval of approximately 30 s only using sub-band 0 (corresponding to channels 0–7), the only sub-band supported by the gateway. The first message sent by the device is the Join-request, followed shortly by a downlink containing the Join-accept.

During the test (Fig. 5), five experiments were conducted, each of them with a certaing number of messages (10, 25, 50, 75, 100), depending on the experiment, there was a slight amount of messages that were repeated; however the test successfully

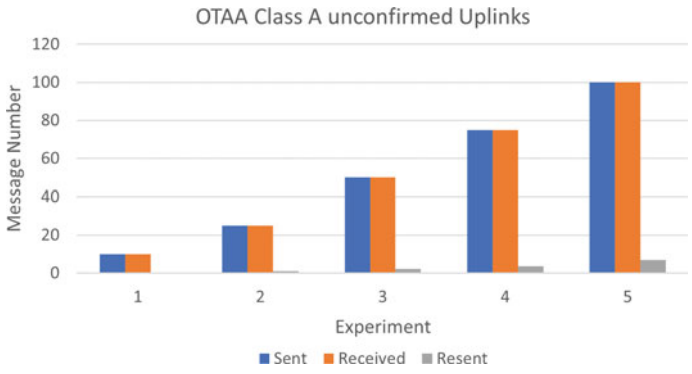

Fig. 5 Result OTAA Class A unconfirmed uplinks

received all messages that were sent. The server reported no errors or inconsistencies in the messages, meaning the MICs were correctly calculated and the payload correctly encrypted by the device. The time between messages was measured, having an average of 31.2 s, which is 1.2 s higher than the configured time. But this slight variation in timing is produced by the duty cycle of the library. As shown in Fig. 5, there is a small number of resent messages, it represents approximately 1% per every 10 messages for this scenario.

4.2 LoRaWAN v1.1 Class A ABP Confirmed Uplinks

For this test, the device was configured to join the network using ABP; for this, all the keys were generated in the server and configured with the device. In order to send confirmed uplinks, the function call in charge of sending the messages was updated to set the confirmation flag to 1 instead of 0.

During this test, no Join-request nor Join-accept messages were sent through the network as expected. The device started with the first data message, received by the server, and shortly later responded with an unconfirmed downlink sending the ACK to the original message.

On this test, 5 experiments were conducted with different lots of messages (10, 25, 50, 75, 100), some of the messages have to be resent in order to be delivered. Analyzing the logs, the ACK message from the server got lost so the device repeated the message until the ACK frame was received as shown in Fig. 6.

The mean time between messages was 36.3 s. The increase of the time was due to the repeated message. Excluding the repeated message, the mean time was 31.1 s, similar to the value gathered on the first test. As shown in Fig. 6, the number of resent messages approximately represents 2% per every 10 messages on each experiment.

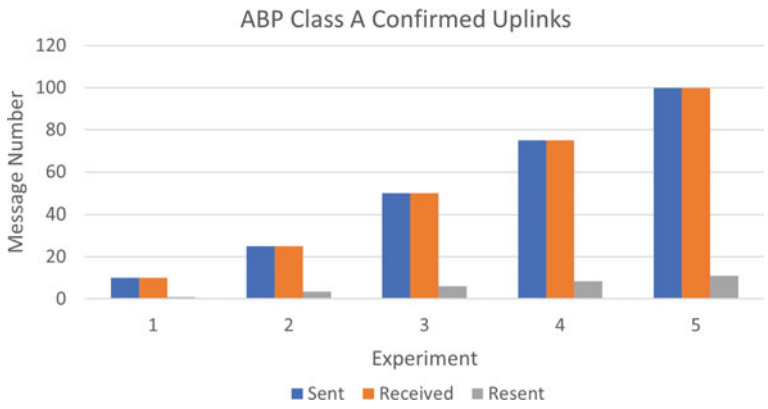

Fig. 6 Result ABP Class A confirmed uplinks

4.3 LoRaWAN v1.1 Class A ABP Confirmed Downlinks

During this test, a device was configured to send uplink messages periodically so the device opens the reception windows for downlinks. For this test, we conducted five rounds of experiments, each of them with a different amount of downlink messages (10, 25, 50, 75, 100), these downlink messages were queued on the server distributed between ports 1 and 2 of the device. The payload for the message was "SGVsbG8sIHdvcmxkIQ==" corresponding to "Hello, world!" encoded in base64.

The device started sending uplinks and after the server received the uplink message it sent the queued downlink with the configured payload of 13 bytes. Different from the other tests, the downlinks sent by the server are now all confirmed. This happens because the server uses the same frame to send the ACK to the device on the same frame data is sent. In the previous test, most of the downlink frames only contained the header with no payload or port.

Reviewing the messages sent by the end device, sometimes it responds immediately to the server with an unconfirmed uplink in order to send the ACK of the confirmed downlink. On this test, there was a slight number of messages that need to be retransmitted as the device did not acknowledge them. As shown in Fig. 7, there is a 100% of accuracy for sending and receiving messages, whilst the number of resent messages for this scenario at the last experiment represent at least 10%; however all messages were received without errors.

4.4 LoRaWAN v1.1 Key Persistence and Device Restart

The purpose of this test was to check if the device was capable of storing the session data in order to continue the communication after the device was restarted. Since

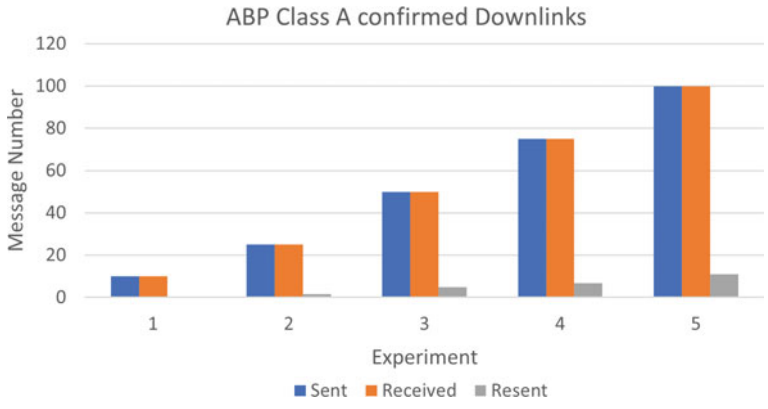

Fig. 7 Result ABP Class A confirmed downlinks

each activation procedure, i.e., OTAA and ABP needs to store different parameters, this test was divided into two parts.

ABP For this test, one of the devices from the previous test was reused but implemented the functions to restore the counters and the device restart indicator. The counters were incremented by one compared to the server since the library always stores the counters increased by one except if the counter is in 0. On this test, the device could resume communications with the server sending a frame where the counter FCnt was equal to 13. Similar to the other test, the device was capable of communicating with the server and sending a frame with FCnt set to 13 as expected.

OTAA Different from ABP, in this test, the device only needed to store the DevNonce and JoinNonce in order to start the Join-procedure again, derive the keys and restart all the counters back to zero. For the test, a device was configured to send ten messages. After the values for the DevNonce and JoinNonce were stored on the device and restarted again. The DevNonce starts at value 1 and the JoinNonce starts at value 0. The device is turned on again and it is now capable of joining the network again and sending messages again.

4.5 LoRaWAN v1.0 Class A ABP Versus LoRaWAN v1.1 Class A ABP

A final test was performed to compare the performance of the original implementation of the library compared to the adapted version for detecting any possible losses of performance. For both tests, ABP with unconfirmed Uplinks was used with the only difference of using different versions of LoRaWAN.

For the LoRaWAN v1.0.3 device, Table 6 was obtained. After the 10 messages, the average time of each message was 32.5 s with a standard deviation of 1.96 s.

The LoRaWAN v1.1 device produced the following results shown in Table 7. By conducting a test for blocks of 5, 10, 50, and 100 messages, the average time, standard deviation, minimum, and maximum values are shown in the following table.

Comparing the values of each device, the average time for the LoRaWAN v1.1 device was increased by 0.5 s compared to the LoRaWAN v1.0.3 which used the original implementation of the library. Comparing the average time increment to the standard deviation obtained on the original implementation (LoRaWAN v1.0.3 device) it can be concluded that the adapted version is having an insignificant perfor-

Table 6 Message times for LoRaWAN v1.0.3

Msg	0	1	2	3	4	5	6	7	8	9	Avg.	Std. dev.
Time (Seg)	30	38	32	32	32	33	32	32	32	32	32.5	1.96

Table 7 Min, Max, std. dev., avg. for 5, 10 50 and 100 messages scenarios

	5 messages	10 messages	50 messages	100 messages
Avg.	32.8	32.9	31.8	32.7
Std. dev.	1.73	1.34	2.71	1.93
Min	30	30	28	29
Max	34	34	25	33

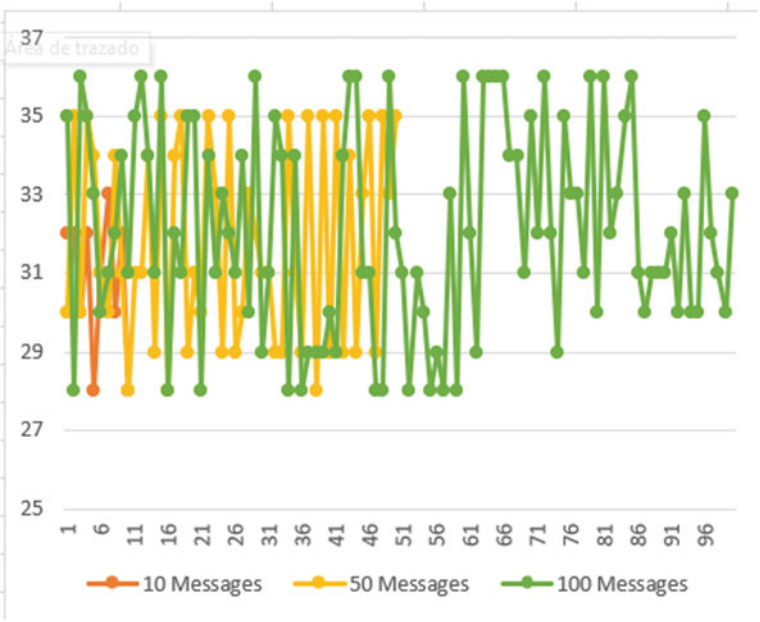

Fig. 8 Messages generated versus time (s) in LoRaWAN V1.1

mance loss of 1.53%, due to the modified algorithms to support the extra encryption keys and algorithms. In addition, Fig. 8 shows the times obtained for ten, fifty, and one hundred messages generated by Device B using LoRaWAN v1.1.

5 Conclusions

- The Arduino-lmic library was successfully adapted to work with LoRaWAN v1.1 specification limited to OTAA and ABP support for Class A devices.
- The comparison between LoRaWAN v1.0.3 and v1.1 specifications served as a theoretical base for the adaptation of the library.

- It was possible to incorporate v1.1 specification features into Arduino-lmic implementation as well as modify its behavior to communicate with LoRaWAN v1.1 network; without significatively impacting its performance.
- The operation of the adapted library was verified using a test suite described in Sect. 3.4. The tests were performed using an Arduino compatible device (TTGO LoRa 32) as well as a production-ready deployment of a LoRaWAN network using Chirpstack and hosted on AWS.

References

1. LoRa Alliance (2020) About LoRaWAN® LoRa Alliance®. Accessed 22 Jan 2020 [online]. Available https://lora-alliance.org/about-lorawan
2. LoRa Alliance (2019) LoRaWAN® back-end interfaces v1.0 LoRa AllianceTM (2017). Accessed 10 Nov 2019 [online]. Available https://lora-alliance.org/resource-hub/lorawanr-back-end-interfaces-v10
3. LoRa Alliance (2017) LoRaWAN® specification v1.1—LoRa Alliance®
4. Hunt D (2020) Selecting a LoRaWAN® specification. Accessed 22 Jan 2020 [online]. Available https://tech-journal.semtech.com/selecting-a-lorawan-specification
5. Terry M (2020) LMIC-v3.0.99. GitHub. Accessed 22 Jan 2020 [online]. Available https://github.com/mcci-catena/arduino-lmic
6. Maziero L et al (2019) Monitoring of electric parameters in the Federal University of Santa Maria using LoRaWAN Technology. In: 2019 IEEE PES innovative smart grid technologies conference—Latin America (ISGT Latin America), Sept 2019, pp 1–6. https://doi.org/10.1109/ISGT-LA.2019.8895425
7. Jeon Y, Kang Y (2019) Implementation of a LoRaWAN protocol processing module on an embedded device using secure element. In: 2019 34th international technical conference on circuits/systems, computers and communications (ITC-CSCC), June 2019, pp 1–3. https://doi.org/10.1109/ITC-CSCC.2019.8793333
8. Wang S-Y, Chen T-Y (2018) Increasing LoRaWAN application-layer message delivery success rates. In: 2018 IEEE symposium on computers and communications (ISCC), June 2018, pp 148-1-53. https://doi.org/10.1109/ISCC.2018.8538457
9. The things network—we are building a global open free crowdsourced long range low power IoT data network. https://www.thethingsnetwork.org/docs/. Accessed 10 Nov 2019
10. The Things Network (2020) Overview of LoRaWAN libraries [HowTo]. The Things Network. Apr 2019. Accessed 28 Jan 2020 [online]. Available https://www.thethingsnetwork.org/forum/t/overview-of-lorawan-libraries-howto/24692
11. Harper C (2020) cjhdev/lora_device_lib. Jan 2020. Accessed 28 Jan 2020 [online]. Available https://github.com/cjhdev/lora_device_lib
12. Poppendieck M, Cusumano MA (2012) Lean software development: a tutorial. IEEE Softw 29(5):26–32. https://doi.org/10.1109/MS.2012.107
13. LoRa Alliance (2015) LoRaWAN® specification v1.0—LoRa Alliance®