Delta-Based Compression and Multicast Synchronization for Efficient LoRaWAN FUOTA

Delta-Based Compression and Multicast Synchronization for Efficient LoRaWAN FUOTA

1st Andrea De Simone
Department of Electronics and Telecommunications
Politecnico di Torino
Turin, Italy
andrea.desimone@polito.it

2nd Giovanna Turvani
Department of Electronics and Telecommunications
Politecnico di Torino
Turin, Italy
giovanna.turvani@polito.it

3rd Mariagrazia Graziano
Department of Applied Science and Technology
Politecnico di Torino
Turin, Italy
mariagrazia.graziano@polito.it

4th Fabrizio Riente
Department of Electronics and Telecommunications
Politecnico di Torino
Turin, Italy
fabrizio.riente@polito.it

***Abstract*—Remote firmware updates in LoRaWAN are a fundamental component for maintaining edge devices. When energy is a limited resource and downtime must be minimized, standard over-the-air firmware updates may not provide an optimal solution. This paper proposes two approaches to address LoRaWAN's restricted data rate. The first strategy builds on an open-source tool tailored to resource-constrained microcontrollers, which reduces the amount of data to be transmitted by generating compact delta scripts. The proposed compression relies on flexible bit-length opcodes that reduce the delta-script size while keeping firmware reconstruction lightweight, achieving an average reduction of approximately 15–20 times compared to transmitting the full image, for minor code changes. The second strategy introduces a protocol that synchronizes nodes starting from Class A and activates the radio only during scheduled fragment transmissions, improving normal multicast sessions where devices remain continuously in Class C. Despite LoRaWAN latency and timing variability, experimental results show that devices can be synchronized to optimize radio activity. The two strategies can be combined to reduce both energy consumption and update duration, while the second method enables fine-grained tuning of the trade-off between campaign time and network load.**

***Index Terms*—LoRaWAN, IoT, FUOTA, multicast, delta-encoding**

I. Introduction

Maintenance of devices belonging to an Internet of Things (IoT) network is an essential system functionality. Firmware running on end nodes may need to be updated for several reasons, including the addition of new features, adaptation to changing operating conditions, or correction of bugs that compromise device operability. Manually reprogramming each device is not feasible when nodes are numerous and distributed across wide geographic areas. The widespread adoption of Long Range Wide Area Networks (LoRaWAN) in recent years demonstrates the efficiency and simplicity of this protocol, particularly for IoT deployments with many distributed nodes [1]. Moreover, LoRaWAN features extremely low energy consumption [2], which enables ultra-low-power applications and makes it particularly suitable for battery-powered devices. Remote reprogramming is supported by the Firmware Update Over-the-Air (FUOTA) standard [3]. However, despite being available and operational, the characteristics and limitations of LoRaWAN hinder the practical adoption of FUOTA in real deployments. In particular, duty-cycle (DC) constraints and Fair Use Policy (FUP) limitations, combined with the limited downlink payload size, render firmware image transmission both time- and energy-intensive, especially given the typical microcontroller (MCU) firmware sizes. Typically, the entire firmware image must be segmented into fragments and transmitted through multiple downlink messages. In the literature, most studies concentrate on minimizing the volume of transmitted data by employing delta updates [4], [5] or adopting modular code architectures in place of monolithic firmware images [6]. These approaches can significantly reduce both update time and radio activity, the latter being one of the dominant contributors to energy consumption.

Another relevant aspect is broadcasting updates to multiple nodes. Efficient dissemination is essential to limit network load, shorten the overall duration of a FUOTA campaign, and enhance scalability. However, maintaining synchronization across multiple nodes while keeping energy consumption under control remains a non-trivial challenge. Existing studies mainly investigate grouping strategies to minimize campaign time and energy consumption [7]–[10], whereas the standard FUOTA procedure is often executed.

This work proposes two strategies to address the high energy consumption and long completion times associated with standard FUOTA implementations, thereby improving the sustainability of firmware updates for battery-powered nodes:

1) **Delta-based compression.** The first strategy extends a previous work [11], which introduces an open-source tool named *bpatch* [12] that compresses the

new firmware image into a delta script encoding the differences between two firmware versions. The key innovation of this tool is the generation of a compact and simple script based on opcodes with flexible bit-length. It is designed to be hardware- and code-independent, to operate on arbitrary binary files, and to avoid specific constraints (e.g., fixed/absolute function placement). In addition, the reconstruction algorithm executed on the end node is lightweight and straightforward to integrate.

2) **Synchronized multicast and radio optimization.** The second strategy introduces a protocol to synchronize multiple nodes during multicast updates and optimizes the FUOTA fragmentation session by disabling the radio when nodes are not expected to receive data. Moreover, the protocol supports fine-grained tuning of airtime occupancy, enabling designers to explicitly trade off network load against the total firmware update completion time.

The rest of this article is organized as follows. Related works are discussed in Section II. Section III gives an overview of the LoRaWAN protocol and FUOTA. Section IV presents the two proposed strategies, while Section V reports and discusses the experimental results. Finally, Section VI draws conclusions.

II. Related Works

Several studies have investigated FUOTA optimizations in the context of LoRaWAN. The survey in [13] emphasizes the relevance of efficient dissemination mechanisms to reduce network load and energy consumption in battery-powered deployments. Given the high cost of transmitting a complete firmware image, delta scripting is a common and effective technique to reduce the volume of exchanged data. A key challenge for delta-based methods is achieving high compression efficiency while maintaining a lightweight reconstruction algorithm suitable for resource-constrained end devices. Even minor source-code modifications can lead to substantial binary differences due to address shifts in functions or global variables [14]. To mitigate this issue, techniques such as absolute address assignment and dedicated compression strategies have been proposed [4], [5]. In [4], the authors present a runtime update mechanism based on static addressing and fixed code sections, which facilitates the insertion of new features. The update is performed in LoRaWAN Class A to reduce energy consumption. However, this approach does not support multicast dissemination. In [5], the authors propose a suffix-array-based algorithm to generate delta scripts and apply an additional compression stage to reduce the impact of address shifts, with an overall compression declared of 2.3x. Alternatively, modular firmware approaches seek to avoid transmitting the entire binary image, as explored in [6], but they typically introduce additional overhead, may reduce efficiency, and often require more complex configuration compared to monolithic firmware.

With respect to dissemination, [8] investigates grouping strategies to reduce the overall energy spent by the network during a FUOTA campaign. In [7], the authors propose a scheme to balance energy consumption and update latency by dynamically varying the downlink Spreading Factor (SF). Scalability aspects are investigated in [9] through simulation, showing that a large number of gateways may be required to maintain low SF values and limit energy consumption. A complementary scalability study for multicast FUOTA in Class B is presented in [15], which highlights issues such as beacon blocking when the number of multicast groups grows because scheduling control of downlink is limited in LoRaWAN class B. While these works primarily focus on scalability at the network level, [10] proposes a protocol-level optimization aimed at reducing energy waste by preventing end devices from remaining active when receiving fragments that have already been successfully acquired.

The two strategies proposed in this paper address both dimensions: (i) an efficient delta-scripting tool that can be deployed on resource-constrained end devices, and does not rely on restrictive assumptions on firmware layout or coding practices; and (ii) a synchronization protocol for end nodes that optimizes the standard FUOTA process by reducing radio on-time while preserving group coordination.

III. Background

A. LoRaWAN Overview

LoRaWAN is a communication protocol well suited for a wide range of IoT deployments. It is built on the LoRa physical layer, which employs Chirp Spread Spectrum. In Europe, LoRaWAN commonly operates in the EU863–870 MHz band. LoRa's long-range and low-power characteristics are largely enabled by the selection of SF. Increasing the SF improves receiver sensitivity, at the cost of reduced data rates and increased time-on-air (airtime) per transmission. The other limitation is imposed by DC restrictions under FUP. In Europe, the 869.4–869.65 MHz sub-band allows a maximum DC of 10%, meaning that after transmitting a downlink with time-on-air equal to $airtime$, a gateway must wait approximately $9 \cdot airtime$ before sending another message, thereby limiting the achievable downlink rate. These properties are typically acceptable for end devices that exchange small payloads, where low throughput is not a limiting factor.

In a LoRaWAN network, end devices communicate with one or more gateways, which relay messages to a network server responsible for managing LoRaWAN protocol functions. To accommodate different application requirements, LoRaWAN defines three device classes that trade energy consumption for downlink latency:

- **Class A**: devices are energy-optimized. After each uplink transmission, they open two short receive windows during which downlink messages may be received.
- **Class B**: devices extend Class A by synchronizing to periodic network beacons and opening additional scheduled receive windows (ping slots), enabling more predictable downlink latency at the expense of increased energy consumption.

- **Class C**: devices keep the receiver almost continuously active (except during transmission), achieving the lowest downlink latency but at the cost of the highest energy consumption.

Classes A and B are typically preferred for battery-operated devices. However, downlink availability is constrained by the receive-window structure (Class A) or by beacon and ping-slot scheduling (Class B). Class C offers greater downlink flexibility, but its high power consumption generally limits its use to mains-powered devices or for short, time-limited intervals on battery-powered nodes.

B. FUOTA in LoRaWAN

Remote firmware updates in LoRaWAN are supported through the FUOTA protocol, the official LoRa Alliance documentation is available in [16]. In this work, we use The Things Network as the network infrastructure, while the end devices are based on STM32 microprocessors, which support FUOTA as documented in [17]. A key limitation is the constrained downlink payload size. In the EU863–870 band, the maximum application payload at the highest data rates (e.g., SF7/125 kHz) is 222 bytes. The "LoRaWAN Fragmented Data Block Transport" package allows the firmware to be split into multiple fragments and transmitted over a fragmentation session. The protocol defines a small overhead of three bytes to include a command identifier and a fragment index. Reliability is ensured by sending additional encoded fragments, enabling end devices to recover missing data and successfully complete the update even in the presence of packet loss.
As an indicative best-case example, consider a 100 kB firmware image transmitted at SF7 and using the maximum DC, i.e., 10%. The update requires about 457 downlink fragments. With a minimum inter-downlink spacing of a few seconds imposed by DC constraints, the overall campaign duration is on the order of tens of minutes, and end devices must generally operate in Class C to receive fragments continuously. Alternatively, Class B can be used, but it introduces additional constraints: not all gateways support Class B operation because beacon transmission requires accurate time synchronization (often via GPS), and ping-slot scheduling is less flexible, as also discussed in [15].

IV. Methodologies

A. Delta encoding

Differencing binary firmware on resource-constrained devices can be challenging. Even minor source-code changes can lead standard differencing tools to identify a large number of differences in the resulting compiled binaries. The main reason is the address shifting of functions and variables, which propagates changes throughout the whole file. When functions or sections change size, they may be relocated, shifting subsequent code forward or backward. As a result, most differences consist of small byte sequences, but they are highly dispersed across the entire image. A commonly adopted solution is to apply external compression after computing the delta between two firmware images. General-purpose compressors, such as *bzip2* and related alternatives, can effectively reduce the size of the delta script although dispersed and redundant changes. However, adopting such compressors on low-power end nodes—where a file system is typically absent and memory resources are limited—is challenging; consequently, some tailored alternatives have been proposed, albeit with reduced compression performance.
For this reason, an innovative open-source tool named *bpatch* [12] was proposed in [11] to generate delta scripts specifically for low-power MCUs. The main idea is to keep the operations that describe the differences simple, while optimizing the encoding of change positions using flexible bit-length words. The main characteristics of *bpatch* are summarized below:

- The *Unix diff* software is used to compare the two binary files. This tool is selected because it is based on the Myers algorithm, which finds a short sequence of edit operations to transform one text file into another [18].
- Differences are encoded into a delta script using only two opcodes: *COPY* and *ADD*.
- All byte positions within the opcodes are expressed as incremental values. In this way, positions are generally small numbers.
- The number of bits required to represent positions is determined for each opcode, and the corresponding maxima are encoded in the delta-script header.
- These maxima are required for reconstruction and decoding, as they define the bit-length of the opcode fields.
- No bits are wasted to indicate the opcode type: the first opcode is *COPY*, the next is *ADD*, then *COPY*, and so on.

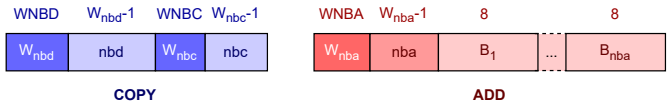


Fig. 1. *COPY* and *ADD* opcodes. The bit-length of the W_{nbx} fields is fixed by the constants *WNBX* (while the "new bytes" field is fixed to 8 bits), whereas the *nbx* fields have flexible bit-length.

The *COPY* and *ADD* opcodes are illustrated in Fig. 1. The values *WNBD*, *WNBC*, and *WNBA* are constants stored in the delta-script header, the first part of the patch, displayed in Fig. 2. They indicate the number of bits required to represent

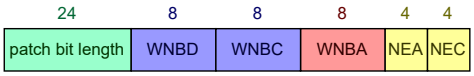


Fig. 2. Patch header. It contains the constants required to decode the patch and reconstruct the new firmware image. This header appears at the beginning of the patch.

nbd, *nbc*, and *nba*, which correspond to: (i) the number of bytes to skip in the old binary file, (ii) the number of bytes to copy from the old binary file, and (iii) the number of bytes to

add from the new binary file, respectively. The fields *nbx* are $W_{nbx} - 1$ bits long because representing a number with the minimum number of bits implies that the most significant bit is always 1 (this improvement is a feature of the extension proposed in this work). This encoding minimizes the bit-length of opcodes when displacement values are small, while still enabling the representation of longer opcodes without penalizing shorter ones, thanks to the flexible field length.

The second improvement deployed in this work consists of collecting the most frequently occurring opcodes into a dictionary and using a compact encoded representation for them. To avoid adding explicit overhead (i.e., extra bits) to indicate whether an encoded opcode is used, we exploit the unused values in the W_{nbd} and W_{nba} fields. To clarify the concept, assume that the maximum value of *nbd* is 10. In this case, the corresponding constant *WNBD* is 4. Values of *nbd* greater than 10 never occur in the delta script, leaving five unused values. The idea is to use these "free" values to encode the most common *COPY* opcodes. The same mechanism can be applied to *ADD* opcodes, where frequently used commands are encoded using the unused values of the W_{nba} field. To handle cases in which few (or no) free values are available, the tool evaluates the net bit saving obtained by increasing *WNBD* and *WNBA* by one. If the number of bits saved by dictionary encoding exceeds the overhead introduced by the additional bit in the opcode fields, the increased constant is selected. Further increments are not evaluated by the tool because we observed that they generally do not reduce the delta-script size.

The last two header fields (Fig. 2) indicate the number of encoded *ADD* and *COPY* opcodes. The dictionaries are inserted into the patch before the standard opcodes, using the format shown in Fig. 3. Opcodes are encoded according to their order

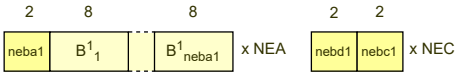


Fig. 3. Dictionaries used to represent encoded opcodes. Fields have fixed bit-length. After the header, the patch lists *NEA* encoded *ADD* entries and *NEC* encoded *COPY* entries.

of occurrence: the most frequent COPY or ADD opcode is mapped to the value $2^{WNBD} - 1$ or $2^{WNBA} - 1$, respectively, and so on, as illustrated in Fig. 4.

This improvement addresses cases in which repetitive changes

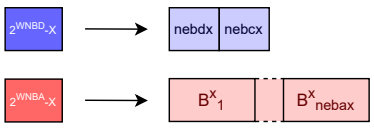


Fig. 4. Decoding of an encoded *COPY* value: when the W_{nbd} field matches an encoded value, the positions *nbd* and *nbc* are obtained from the corresponding dictionary entries. The same mechanism applies to *ADD*, which is associated with the corresponding "new bytes" entry.

occur in the delta script, further reducing its size.

B. Synchronized multicast and radio optimization

LoRaWAN end devices typically operate in Class A, which enables extremely low energy consumption and extends battery lifetime. End devices are active only for a small fraction of time compared to the sleep period, with a duty ratio that depends on the specific application. In most cases, they spend the majority of time in sleep mode. Firmware reprogramming events are infrequent, but they may still occur during the device lifetime. During firmware update campaigns, switching class is typically required. In particular, multicast is not supported in Class A, and downlink availability in Class A is tightly coupled to uplink transmissions. This coupling makes it difficult to control downlink timing and can result in excessively long fragmentation sessions. Class B is generally less widely supported and is not available by all gateways because it requires accurate time synchronization via GPS. In addition, ping-slot scheduling is constrained by the beaconing structure and does not readily support flexible downlink transmissions at arbitrary times. Class C is commonly supported and does not require time synchronization between gateways and end devices. Its primary drawback is energy consumption: devices must keep the radio active for the entire duration of the update. Consequently, the only practical way to reduce energy consumption is to transmit fragments at the highest available data rate, at the cost of increased network load. Even under these conditions, the radio remains active longer than necessary, leading to avoidable energy waste. The second strategy presented in the following aims to optimize radio activation on LoRaWAN end devices during the fragmentation session. An initial synchronization between all participating devices and the gateway is required to execute the FUOTA session.

An MQTT (Message Queuing Telemetry Transport) broker interacts with the LoRaWAN network server and manages uplinks and downlinks for all devices. The proposed protocol consists of two main phases: (i) a synchronization phase, performed one device at a time; and (ii) a fragmentation phase, during which nodes receive code segments concurrently. The two phases are described in detail below.

1) Synchronization phase: At the beginning, nodes operate in Class A. As a result, uplink times—and the corresponding downlink receive windows—are not easily predictable. While it is possible to infer them by analyzing network traffic and tracking device behavior, this approach is complex and impractical at scale. To synchronize all devices, the broker sends a unicast downlink to each device announcing the start of the FUOTA session. Prior to transmitting these downlinks, a fragmentation start time must be configured in the broker FUOTA settings. This time must be sufficiently long to ensure that all devices can transmit an uplink and thus open their Class A receive windows. Once a node sends an uplink, it becomes eligible to receive the corresponding the downlink. Upon recognizing the FUOTA start command, the device schedules a dummy uplink 30 s later, with the sole purpose of creating an additional Class A receive window. The broker then uses the timestamp of the first uplink to compute the

timing of the subsequent receive windows. Synchronization is completed by a second downlink that simultaneously sets up the fragmentation session and provides the FUOTA parameters listed in Table I. This procedure is repeated for all nodes

TABLE I
FUOTA PARAMETERS

FUOTA Time (FT)	fragment transmission period
FUOTA Start (FS)	delay before the session starts
airtime	time-on-air of a data-fragment downlink

involved in the update session, as illustrated in Fig. 5. The parameter FS is different for each device, as it depends on the timing of each node's uplink time.

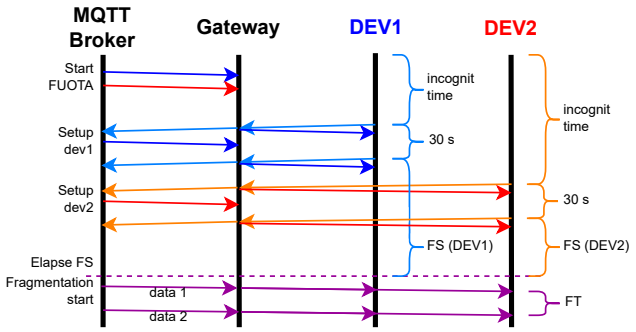


Fig. 5. Synchronization example with two devices. The broker sends FUOTA-start downlinks, which are scheduled by the gateway after each device uplink. Upon receiving an uplink, the broker computes FS and transmits a second downlink to configure the fragmentation session and the session start time. When FS elapses, the fragmentation session begins with multicast downlinks.

2) Fragmentation phase: After the FS time elapses for all devices (which should occur simultaneously), nodes switch to Class C and the broker starts sending data fragments with a period equal to FT. After receiving a fragment, devices return to sleep mode in Class A. End-device MCUs rely a low-power timer to switch class and re-enable the radio in time to receive the next fragment; this activation must occur at least $FT - airtime$ before the scheduled downlink time. The fragmentation-phase protocol is depicted in Fig. 6. During the session, some nodes may miss a fragment. To avoid remaining in Class C longer than necessary, devices revert to Class A after a timeout of 1 s—longer than the expected airtime—and then reactivate the radio after FT to attempt reception of the next fragment, as illustrated in Fig. 6(b). Recovery of lost fragments relies on standard FUOTA coding: after transmitting the last fragment, the broker sends an additional 15% of coded fragments. Nodes continue to receive fragments until they can reconstruct the entire firmware. Finally, devices reboot and complete the update. Due to LoRaWAN latency, nodes wake up with a 2 s margin for the first packet, while subsequent packets use a 55 ms margin. Reducing these margins may compromise synchronization and, consequently, the completion of the FUOTA session. An end-to-end latency analysis is presented in [19], which highlights the contributions of the

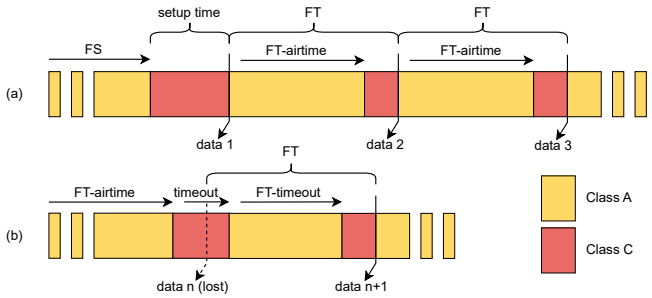


Fig. 6. Protocol overview. (a) Initial synchronization and fragmentation operation: after receiving a data fragment, the device returns to Class A and then switches to Class C to receive the next scheduled fragment. (b) Fragment-loss event: the device misses a downlink and returns to Class A after a timeout before attempting reception of subsequent fragments.

MCU, the LoRaWAN protocol, and the Internet connection, motivating the need for these margins.

The choice of FT enables fine-tuning of network load and update duration according to application needs. At the same time, varying FT does not, in principle, affect the total time spent in Class C, enabling significant energy savings compared to standard FUOTA, where nodes remain continuously in Class C.

V. RESULTS AND DISCUSSIONS

A. Delta encoding

Evaluating the compression performance of a delta-encoding algorithm is a delicate task. Testing performance only on auto-generated code and avoiding comparisons with other tools and external firmware binaries may lead to incomplete results. Differencing tools can behave very differently depending on the extent and nature of code modifications. Moreover, quantifying the "amount of change" between two firmware versions is inherently difficult and there is no general rule. Furthermore, the proposed tool aims to be hardware-independent, so, a diversified test set is necessary to support this claim. The dataset used in this study consists of multiple firmware families. First, 24 firmware versions written for the STM32WL series (the platform used in this work) are included. In addition, external firmware binaries are collected from an open-source repository containing compiled code for drones [20]. In particular, 40 binary versions targeting the STM32H7 family and 123 binary versions targeting Raspberry Pi devices are extracted and added to the dataset. To improve the quality of the evaluation, we compare our tool against two open-source patching tools. The first is *VCDIFF* [21], which uses a byte-level encoding and three opcodes: COPY, ADD, and RUN (for repeated sequences). The second tool, *HDiffPatch* [22], is based on suffix arrays and operates on byte sequences to identify similar segments across files. Both tools were originally designed for scenarios such as compressing Android application packages. Integrating these tools on MCUs like those used in this work is non-trivial due to the lack of a file system and limited memory resources. Moreover, these tools are typically used in

combination with external compression, but, to ensure a fair comparison, this functionality is disabled in our evaluation.

Compression results are reported in Table II. For each target platform and each incremental firmware version, a delta script is generated. Table II reports the average compression ratio, expressed as *new image size / delta size*. To better capture the tools' behavior under different levels of change between consecutive versions, delta scripts are grouped into three categories:

TABLE II
AVERAGE COMPRESSION RATIO

MCU family	Update type	Delta Script Tools			
		bpatch[a]	bpatch V2[a]	HDiffPatch	VCDIFF
STM32WL	NU	676	700	561	591
	MN	18.0	22.4	17.8	9.50
	MJ	2.33	2.40	2.38	2.10
STM32H7	NU	33844	37636	6793	14396
	MN	15.1	20.9	15.5	4.30
	MJ	3.31	3.97	4.04	2.06
Raspberry Pi	NU	39067	45011	6955	13568
	MN	9.37	12.5	10.1	3.43
	MJ	2.34	2.74	2.64	1.64

[a]bpatch is referred at the previous version of the tool, bpatch V2 at the extension proposed in this work.

- **No Updates (NU)**: consecutive firmware versions are almost unchanged; this may occur for documentation-only changes or minimal byte differences (e.g., macro or constant modifications).
- **Minor Updates (MN)**: changes due to bug fixes, feature enhancements, or small code adjustments.
- **Major updates (MJ)**: updates associated with significant code refactoring.

This categorization is not straightforward. To define an objective criterion, *HDiffPatch* is used as a reference, as it exhibits overall better performance than *VCDIFF*. Two thresholds are adopted: if the delta-script size is less than 0.5% of the new firmware image, the update is classified as NU, if it exceeds 20%, it is classified as MJ, otherwise, it is considered MN. This classification allows a more comprehensive assessment of the three differencing algorithms.

For NU updates, where differences are easier to compress, our tool consistently achieves the best performance, outperforming both baselines. For STM32H7 and Raspberry Pi targets, the improvement reaches several orders of magnitude. For all three platforms, *VCDIFF* overcomes *HDiffPatch* in the NU regime, indicating that a simpler encoding is advantageous when changes are minimal. For MN and MJ updates, the situation changes: *HDiffPatch* becomes clearly more efficient than *VCDIFF*, highlighting the benefits of a more sophisticated algorithm. The previous version of bpatch, despite using simple opcodes, achieves performance comparable to *HDiffPatch* for MN updates and only slightly worse results for MJ updates. This indicates that the flexible bit-length encoding recovers much of the efficiency typically obtained with more complex algorithms. The extension proposed in this paper adds limited complexity to the method and further improves performance for MN updates, with gains ranging from 24.4% to 38.4%, thereby outperforming *HDiffPatch*. For MJ updates, the improvement is more limited (13.3% on average), but it erases the gap between the original version of our tool and *HDiffPatch*. The results confirm that the proposed tool, which can be integrated on resource-constrained devices as claimed in [11], provides effective compression of the data to be transmitted for firmware updates. This reduction directly translates into lower transmission time and energy consumption, as reflected in Table II.

B. Synchronized multicast and radio optimization

The analysis and measurements to validate the second strategy are specifically conducted for the fragmentation phase. In particular, we measure the total time end devices spend in Class C using an MCU timer, and we compare the proposed protocol against standard FUOTA, where nodes remain continuously in Class C. The experiment consists of three series of FUOTA sessions involving multiple devices. Each series corresponds to a different file size (10 kB, 50 kB, and 100 kB), and each experiment is repeated five times. The session parameters are reported in Table III. Five devices participate in the experiment, after synchronization, one device remains in Class C until the end of the update. This reference device (labeled as *Ref*, while the other four are labeled as *Nodes*) is used to verify whether fragments are lost due to LoRaWAN latency. Indeed, if a downlink is transmitted before a device has switched to Class C, the fragment is missed. Clearly, fragment loss may also occur for reasons not directly attributable to the proposed protocol. The always-on reference device is also used to estimate the loss rate (LR) statistically. All devices are placed in the same location within a few centimeters, therefore they experience signal of comparable strength and are expected to exhibit similar LR. Regarding the FS and FT margins introduced in Section IV, we do not perform a systematic optimization study. Instead, margins are selected empirically to keep radio activity as low as possible while preventing excessive LR. A more focused analysis of LoRaWAN latency and downlink time variability would be valuable. It is outside the scope of this work, which aims to demonstrate the effectiveness of the proposed strategy and provide an initial estimate of the achievable savings compared to the standard approach.

TABLE III
PARAMETER LIST OF THE EXPERIMENTS

Type	Parameter	Value
LoRaWAN	Spreading Factor	7
	Bandwidth	125 kHz
	Coding Rate	4/5
	Data Rate	5
Fragmentation Session	Duty Cycle	2.4%
	Fragment Size	219 B
	Fragment Airtime	363.8 ms
Our Protocol	FUOTA Time	15 s
	File Sizes (kB)	10, 50, 100
	Fragment Number	47, 232, 463

The first measurement concerns FS. Specifically, we measure the duration of the initial window during which a node switches to Class C, which can be visualized in Fig. 6 as ”setup time”. The corresponding distribution (mean and percentiles) is shown in Fig. 7. The distribution exhibits a wide spread, ranging from about 0.5 s to almost 4 s across 60 FUOTA sessions. This variability is mainly due to the latency across the end device, gateway, and network server, which is high and difficult to predict. Although such variability affects both time and energy, its impact is limited compared to the overall fragmentation session duration.

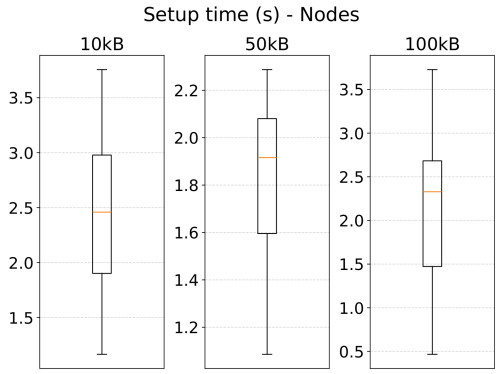
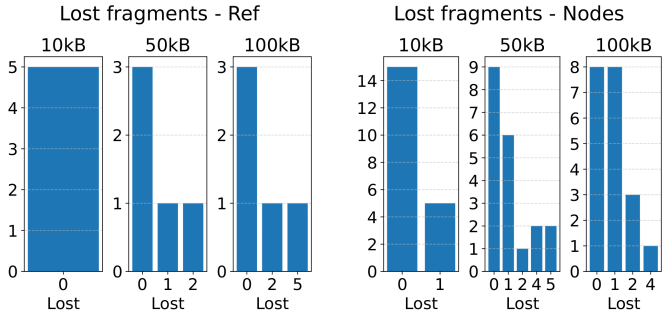


Fig. 7. Distribution of setup times for *Nodes* devices. The orange bar indicates the mean, while the box the inter-quartile range.

Fragment-loss behavior is illustrated in Fig. 8: Fig. 8(a) refers to the reference device (*Ref*), while Fig. 8(b) shows the distribution for the remaining devices operating with the proposed protocol (*Nodes*). The LR values for all cases are computed and reported in Table IV. Results indicate that increasing the file size increases the probability of missing fragments. In particular, for the smallest file size, the reference device does not miss any fragment in all five sessions, while the difference of LR between 50 kB and 100 kB is less marked. Comparing *Ref* and *Nodes*, a small increase in LR is observed for two firmware sizes, indicating that the proposed methodology generally introduces only a slight LR increment. The last rows of Table IV report the measured time spent in Class C. These values are provided for the *Nodes* only and represent the mean across sessions and devices. For each receive window, we subtract the downlink airtime from the measured Class C duration to obtain the extra time during which the radio is active. The measure is reported in the row ”Radio Overhead per fragment”. This overhead is strictly related to the chosen margin and is not constant because of LoRaWAN latency variability. The overhead increases with file size for two main reasons. First, the number of timeouts increases. As shown in Fig. 6, when a fragment is missed, the corresponding window becomes wider because no downlink triggers the class switch until the timeout expires. This effect becomes more frequent as the number of fragments increases, and it is amplified when LR is higher. Consistently, the 50 kB case, which exhibits the worst LR, also shows the largest overhead. To remove



(a) (b)

Fig. 8. Occurrences of lost fragments during FUOTA sessions: histogram of the number of lost fragments (x-axis) and the number of devices exhibiting that loss (y-axis) for (a) *Ref* and (b) *Nodes*.

TABLE IV
MEASURES ON FUOTA SESSIONS

Parameter measured	Device involved	Firmware size 10 kB	50 kB	100 kB
Loss Rate	Ref	0%	0.257%	0.302%
	Nodes	0.709%	0.747%	0.259%
Radio Overhead per fragment	Nodes	56.814 ms	63.942 ms	62.311 ms
	NLF	53.672 ms	60.690 ms	61.585 ms
Time in Class C per firmware size	Nodes	2.233 s/kB	2.037 s/kB	2.001 s/kB

the bias introduced by timeouts, the next row repeats the computation considering only FUOTA sessions with no lost fragments (NLF). Even in this case, overhead still increases with data size, mainly due to the variability of LoRaWAN latency: with a larger number of segments, timing variance increases the average per-segment overhead. Finally, the last row reports an estimate of the mean radio active time per file size. A small decrease with file size is attributed to the initial setup window, whose contribution is amortized over a larger number of fragments. To quantify the improvement enabled by the proposed protocol, we compare this last metric against a baseline corresponding to standard FUOTA, assuming the end device remains continuously in Class C during the whole transfer. This baseline is meaningful only when fragments are transmitted at the maximum DC allowed, i.e. using the minimum FT. Under this optimistic assumption, the “Time in Class C per firmware size” is approximately 17 s/kB. This value should be regarded as a lower bound, since it does not include either the mechanisms required to synchronize multiple nodes when operating in Class A, or the additional receive time needed to recover lost fragments. Moreover, increasing FT (e.g., to reduce network load) would directly increase this metric.

In summary, the proposed methodology enables the radio to be turned off and saves energy when no downlink is on air, at the cost of a minimal LR increase. Although setup time and window margins exhibit wide variance, the protocol

provides a safe mechanism to interrupt radio activity during the fragmentation session, including for large firmware sizes and high fragment counts.

VI. Conclusions

This work advances and evaluates two independent and complementary strategies to mitigate the limitations of standard LoRaWAN FUOTA for battery-powered devices.

First, we extend a delta-encoding tool designed for constrained MCUs to substantially reduce the amount of data that must be transmitted over-the-air during firmware updates. The tool's performance is evaluated using a diversified and, as far as possible, objective methodology across multiple firmware families and platforms. The results demonstrate that the proposed encoding achieves strong compression, particularly for minor code changes, which represent the most common update scenario in IoT deployments. On average, reducing the transmitted payload by approximately 15–20× yields a proportional reduction in both update time and radio energy consumption. Moreover, the proposed extension outperforms the considered open-source patching baselines, supporting the effectiveness of the algorithm. Finally, the hardware-agnostic design and the public availability of the implementation as an open-source project facilitate broad applicability across a wide range of MCU platforms and IoT networks.

Second, we propose a synchronization and fragmentation protocol that improves multicast FUOTA by reducing unnecessary radio on-time during the fragmentation session. By scheduling reception and allowing devices to deactivate the radio between expected downlinks, the protocol decreases the time spent in Class C compared to the standard approach in which devices remain continuously in Class C throughout the campaign. Experimental measurements show significant savings in radio active time, while maintaining a comparable loss rate under the tested conditions.

Future work will focus on further improving both strategies. For delta scripting, integrating optional lightweight compression stages tailored to MCU constraints may further reduce delta size at the cost of additional computational effort on end devices. For the multicast protocol, a deeper characterization of LoRaWAN downlink latency and timing variability could enable tighter margins and reduce setup overhead, further improving energy efficiency and robustness, especially for large-scale deployments.

References

[1] M. Centenaro, L. Vangelista, A. Zanella, and M. Zorzi, "Long-range communications in unlicensed bands: The rising stars in the iot and smart city scenarios," *IEEE Wireless Communications*, vol. 23, no. 5, pp. 60–67, 2016.

[2] W. Mao, Z. Zhao, Z. Chang, G. Min, and W. Gao, "Energy-efficient industrial internet of things: Overview and open issues," *IEEE transactions on industrial informatics*, vol. 17, no. 11, pp. 7225–7237, 2021.

[3] J. Catalano, "LoRaWAN Firmware Update Over-The-Air (FUOTA)," *Journal of ICT Standardization*, vol. 9, no. 1, pp. 21–34, 2021. [Online]. Available: https://journals.riverpublishers.com/index.php/JICTS/article/view/7017

[4] B. P. Neves, A. Valente, and V. D. Santos, "Efficient runtime firmware update mechanism for lorawan class a devices," *Eng*, vol. 5, no. 4, pp. 2610–2632, 2024.

[5] Z. Sun, T. Ni, H. Yang, K. Liu, Y. Zhang, T. Gu, and W. Xu, "Flora: Energy-efficient, reliable, and beamforming-assisted over-the-air firmware update in lora networks," in *Proceedings of the 22nd International Conference on Information Processing in Sensor Networks*, 2023, pp. 14–26.

[6] H. D. Nguyen, N. Le Sommer, and Y. Mahéo, "Over-the-air firmware update in lorawan networks: A new module-based approach," *Procedia Computer Science*, vol. 241, pp. 154–161, 2024.

[7] S. S. Borkotoky, "Balancing the energy consumption and latency of over-the-air firmware updates in lorawan," *IEEE Transactions on Industrial Informatics*, 2025.

[8] W. Mao, Z. Zhao, M. Kang, R. Cong, G. Min, Z. Chang, and X. Wang, "Reliable and energy-efficient reprogramming for smart lorawan," in *2023 IEEE Smart World Congress (SWC)*. IEEE, 2023, pp. 1–8.

[9] C. Charilaou, S. Lavdas, A. Khalifeh, V. Vassiliou, and Z. Zinonos, "Firmware update using multiple gateways in lorawan networks," *Sensors*, vol. 21, no. 19, p. 6488, 2021.

[10] H. D. Nguyen, N. Le Sommer, and Y. Mahéo, "Lorawan-based multicast and disruption-tolerant protocols for firmware update over-the-air," in *2025 21st International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT)*. IEEE, 2025, pp. 251–255.

[11] A. De Simone, G. Turvani, and F. Riente, "Incremental firmware update over-the-air for low-power iot devices over lorawan," *Internet of Things*, p. 101772, 2025.

[12] A. De Simone and F. Riente, "vlsi-nanocomputing/bpatch: version-2.0.0," Apr. 2026. [Online]. Available: https://doi.org/10.5281/zenodo.19384616

[13] S. Brown and C. J. Sreenan, "Software updating in wireless sensor networks: A survey and lacunae," *Journal of Sensor and Actuator Networks*, vol. 2, no. 4, pp. 717–760, 2013.

[14] N. Reijers and K. Langendoen, "Efficient code distribution in wireless sensor networks," in *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, 2003, pp. 60–67.

[15] Y. Shiferaw, A. Arora, and F. Kuipers, "Lorawan class b multicast scalability," in *2020 IFIP Networking Conference (Networking)*. IEEE, 2020, pp. 609–613.

[16] L. Alliance, "*FUOTA Process Summary*," accessed: Jan. 7, 2026. [Online]. Available: https://lora-alliance.org/wp-content/uploads/2020/11/tr002-fuota_process_summary-v1.0.0.pdf

[17] STMicroelectronics, "*AN5554*," accessed: Jan. 7, 2026. [Online]. Available: https://www.st.com/resource/en/application_note/an5554-lorawan-firmware-update-over-the-air-with-stm32cubewl-stmicroelectronics.pdf

[18] E. W. Myers, "An o (nd) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 1, pp. 251–266, 1986.

[19] A. Pötsch and F. Hammer, "Towards end-to-end latency of lorawan: Experimental analysis and iiot applicability," in *2019 15th IEEE international workshop on factory communication systems (WFCS)*. IEEE, 2019, pp. 1–4.

[20] ArduPilot, "*ArduPilot Firmware Download*," 2025, accessed: June 25, 2025. [Online]. Available: https://firmware.ardupilot.org/

[21] google, "*open-vcdiff*," 2025, accessed: Mar. 19, 2025. [Online]. Available: https://github.com/google/open-vcdiff

[22] sisong, "*HDiffPatch*," 2025, accessed: Mar. 19, 2025. [Online]. Available: https://github.com/sisong/HDiffPatch