Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 241 (2024) 154–161

Procedia Computer Science

www.elsevier.com/locate/procedia

The 21st International Conference on Mobile Systems and Pervasive Computing (MobiSPC)
August 5-7, 2024, Marshall University, Huntington, WV, USA

Over-the-Air Firmware Update in LoRaWAN Networks: A New Module-based Approach

Huy Dat Nguyen*, Nicolas Le Sommer, Yves Mahéo

Université Bretagne Sud, UMR 6074, IRISA, F-56000 Vannes, France

Abstract

Over the last years, a huge number of things has been connected to the Internet to support applications for smart cities, smart industries, smart agriculture and smart environment, etc. In order to fix bugs, update features or perform dynamic adaptations on these things, efficient Firmware Update Over the Air (FUOTA) methods are required. In this paper, we propose a modular-based approach for the development of firmware of MCU-based IoT devices, and a new FUOTA method for devices communicating using LoRaWAN. This method enables partial and dynamic updates of modules forming the device firmware, and does not require a system reboot to apply these updates. This approach and this method have been implemented on STM32 MCUs. The experimental results show good performance in terms of update size and network traffic compared to the traditional FUOTA method and monolithic-based firmware development approach, which both impose full firmware updates.

Keywords: Firmware Update Over the Air; LoRaWAN; Module-based architecture

1. Introduction

With their capabilities of providing long-range and energy efficient communications (up to 50 km in rural zones and 10 km in urban zones [2]), Low Power Wide Area Networks (LPWANs) have emerged in recent years as networks for the Internet of Things (IoT) applied to various domains, such as smart agriculture, smart transportation, smart cities, environmental monitoring. Among existing standards for LPWANs, one of the most popular technologies is Long Range (LoRa). LoRa only defines the protocol for the physical layer, allowing to support different protocols on this layer. Long Range Wide Area Network (LoRaWAN) is a star-topology network protocol based on LoRa and specified by the LoRa Alliance. An IoT system relying on LoRa/LoRaWAN is composed of end devices (e.g. sensors, tracking devices), LoRaWAN gateways and a network server. Gateways are connected to the Internet. They forward

* Corresponding author. Tel.: +0-000-000-0000 ; fax: +0-000-000-0000.
E-mail address: Huy-Dat.Nguyen@univ-ubs.fr

1877-0509
10.1016/j.procs.2024.08.022

to the network server the packets they receive from the end devices, and reversely they forward to the end devices the packets they receive from the network server. The network server filters the duplicate packets, performs security checks and sends acknowledgments to the gateways.

When keeping up to date end devices in such an IoT system (i.e. to apply bug fixes, feature updates and performance optimizations for instance) without physical interventions, Firmware Update Over the Air (FUOTA) methods are essential. Several works have considered FUOTA methods in LPWAN networks [10, 11, 1, 6, 9], but most of them require an entire firmware update and a system reboot to apply the new firmware, which may not be suitable in some critical IoT applications, such as healthcare, autonomous vehicles or warning systems.

In order to facilitate the development of the firmware of LoRaWAN MCU-based end devices and their updates, we propose in this paper both a new module-based firmware design approach, and a new FUOTA method that supports partial and incremental firmware updates without requiring a system reboot, thanks to dynamic module loading and linking techniques. Allowing to update only the modules that are needed makes it possible to reduce the size of the updates, the network load and the energy consumption of end devices, while accelerating the update process itself. Moreover, with this approach, some parts of the code can be sped up by copying and executing the modules in RAM instead of in flash memory.

In this paper, we also present a concrete implementation of our approach. It consists in a minimal system (called "micro-system" in the paper) for STM32 M-Cortex end devices, these end devices being included in an IoT network architecture composed also of LoRaWAN gateways connected to The Things Network (TTN) platform[1], and an update server. This server ensures the fragmentation of the firmware modules in several pieces of code so that they can be embedded as payload in LoRaWAN packets. Fragments are uploaded on the TTN platform using the Message Queuing Telemetry Transport (MQTT) protocol so as to be transmitted to end devices. Experimental results obtained on a small testbed show that our proposition reduces the update size and the network traffic up to 17 times compared with the traditional monolithic approach.

The rest of this paper is organized as follows. Section 2 presents other works on FUOTA methods that leverages dynamic code loading and linking techniques or relying on LPWAN technologies. Sections 3 and 4 respectively detail the firmware modular design approach and the new FUOTA method we propose, and how they are currently implemented in a fully functional IoT system. Section 5 presents the evaluation results of this system. Section 6 summarizes our contribution and proposes future works.

2. Related work

Modular programming and dynamic linking techniques for sensor nodes have been investigated in limited number of works so far. In [3], Dunkels et al. (2006) present a run-time dynamic linker and loader for ELF (Executable and Linkable Format) and Compact ELF files that facilitate the reprogramming of wireless sensor nodes running on Contiki [4], an operating system for resource-constrained IoT devices. Dynamic linking techniques in Contiki are nevertheless limited as they rely on a strict binding model and on a static symbol table that contains all global symbols in the Contiki system. The symbol table is generated before deployment and cannot be extended afterward. Dynamic TinyOS [7] partially overcomes this problem by replacing the static symbol table by a dynamic symbol table that can be extended. In Contiki and Dynamic TinyOS, additions and updates are nevertheless restricted to the application level. In [8], Ruckerbusch et al. (2016) go a step further by proposing a generic extension that supports dynamic application and network level upgrades. They also describe an implementation and an evaluation of this extension in Contiki. In order not to resort to relocation tables that produce a runtime and memory overhead when loading modules, the SOS operating system [5], that mainly supports 8-bit devices, uses pieces of code that are compiled as position independent code, thus allowing to be executed from any address in memory. Such an approach is, in our opinion, the way to follow to enable the development of firmware composed of off-the-shelf modules and to perform dynamic Over-The-Air (OTA) updates. Except [7], the aforementioned works do not couple these modular programming and dynamic linking techniques with an OTA update process.

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

There is a limited number of research works investigating FUOTA in LPWANs. Recently, the LoRa Alliance has released FUOTA protocols based on the LoRaWAN specifications [10]. However, these protocols come with some limitations in terms of energy efficiency and transmission reliability [11]. In order to secure the firmware update, Anastasiou et al. (2020) proposed in [1] a blockchain-based framework, while in [6] the authors address these issues with a mechanism relying on an Adaptive Data Rate algorithm that enhances the speed, reliability and security of the update process. In general, these works perform a full image replacement, which requires to allocate at least three partitions in memory to store the active firmware, the downloaded firmware and a boot program. Nodes must remain awake to receive the entire firmware, resulting in huge energy consumption for power-constrained LoRaWAN nodes. Nodes must also reboot after having installed the new firmware, introducing interruptions in the collection, processing and transfer of data, which may not be suitable for some IoT systems. To perform incremental updates of end devices in LoRaWAN networks, the work presented in [11] uses a differencing algorithm that aims to calculate a patch based on differences between the old and the new versions of the firmware. Instead of transmitting a new full firmware, the patch is distributed, thus reducing the amount of data needed to be transferred. It must be noticed that this method requires an additional memory region compared to full image updates to store the patch. In some cases, the huge difference between the modified and the active image produces a large patch. Again, a system reboot is needed to activate the updated firmware.

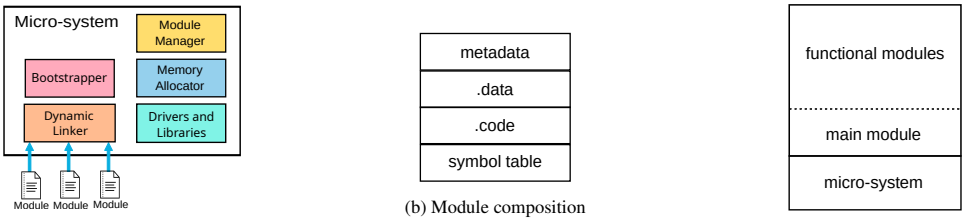
3. Dynamic module-based FUOTA approach

3.1. Overview

In order to overcome the shortcomings mentioned in the previous section, we propose an approach based on two pillars. The first one concerns the architecture of the software embedded on the end device: the monolithic firmware is replaced by a module-based software controlled by a micro-system. The second pillar is a novel FUOTA method that takes benefit from the splitting of the firmware into modules, and that has been implemented in a proof-of-concept Firmware Update Server (FUS) integrated in an usual LoRaWAN architecture, as depicted in Figure 2. In this architecture, end devices are embedded IoT components that come with a LoRa radio transceiver to reach gateways. LoRaWAN gateways listen to all channels and spreading factors at the same time. When a LoRa frame is received, the packet is processed before being transmitted over Internet to the LoRaWAN server that has been configured to be compatible with the gateways beforehand. The LoRaWAN server is responsible for receiving the messages transmitted by the gateways, removing duplicate packets and performing the authentication and encryption process. The FUS is responsible for producing the firmware updates and sending them to the end devices, via the LoRaWAN server. Our contribution on the module design and the associated update mechanism is detailed hereafter.

3.2. Module-based approach

In our approach, the code of the firmware that must be deployed on the end devices is divided into multiple smaller dynamically-loadable modules, which can be independently created, modified, replaced, or exchanged with other modules or between different systems. As illustrated on Figure 1a, these modules are installed together with the micro-system on the devices. The micro-system consists of 1) a Bootstrapper (BS) for installing the main module of



(a) General structure

(b) Module composition

(c) Memory layout

Fig. 1: Software on the end device

the system, 2) a Dynamic Linker (DL) for loading/unloading modules in an on-demand manner and resolving symbol references at run-time, 3) a Module Manager (MM) to globally control the operations on modules, 4) a Memory Allocator (MA) designed to optimize the memory usage, and 5) device drivers and additional libraries. A loadable module contains the native machine code of the program that is to be loaded into the system. The code usually contains symbol references that must be resolved to the physical address of the functions and variables before the execution. Unlike a pre-linked module which contains the absolute physical addresses of the referenced symbols and performed at compile time, in our approach, a module is self-contained and dynamically linked, in the sense that it maintains a symbol table consisting of all the system core functions and non-static variables that are referenced to in the module [3]. The linking process is done when the module is loaded at run-time by the dynamic linker. The module's .code and .data sections follow the symbol table as shown in Figure 1b. Modules also include additional metadata that specify the module's name, version, etc. The metadata, which are used only for the identification, verification and version management of the module, are removed before the module is loaded. Such a modular firmware design approach may open up a future market of reusable off-the-shelf modules, thus reducing the time and the cost to develop and maintain IoT systems.

Usually, microcontroller units (MCUs) embed a flash memory, which is dedicated to store the image of the entire firmware. The memory layout of devices is presented in Figure 1c. Here, our design requires to separate the flash memory into two sections. The first one is used for the micro-system, including additional libraries and drivers. The second one is dedicated to loadable modules, containing the main module and functional modules for specific tasks. Note that the way these components should be actually allocated strongly depends on the memory organization of the MCU. Hence it should be carefully chosen in order to avoid conflicts between them, conflicts that might lead to unexpected states. The DL enables modules to be loaded at run-time in two different ways. By default, modules are loaded and executed directly in flash memory by copying only the .data section into RAM, which improves the RAM usage. Alternatively, the entire module may be copied and executed in RAM instead of in flash memory (especially for small modules), thereby accelerating the execution of one or more performance-critical sections of the application. In order to specify which module should be entirely copied in RAM, the programmer can load a specific module called "configuration module", that contains configuration properties, such as the name of the modules that must be entirely loaded by the MM of the micro-system in the RAM. The MM also ensures on-demand loading/unloading modules so as to make it possible to dynamically disable some specific functionalities when they are not used, which may improve memory allocation.

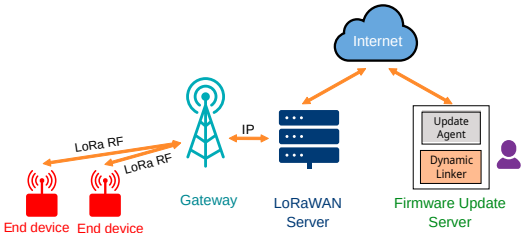


Fig. 2: FUOTA architecture

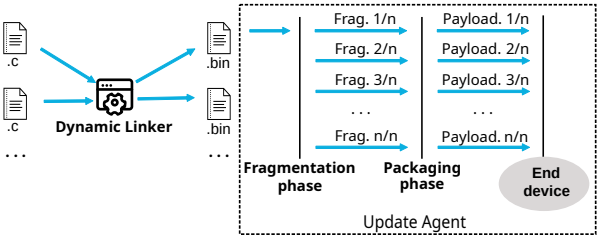


Fig. 3: FU server design

3.3. FUOTA method

We propose a new FUOTA method for module-based firmware on end devices communicating in LoRaWAN networks. The system design is driven by following features:

- *Dynamic update*. The FUOTA strategy is ensured without rebooting the system.
- *Low flash memory footprint*. The FUOTA method is performed module by module, so that end devices avoid allocating an additional partition in flash memory to store the entire downloaded firmware.
- *Size and network efficiency*. Only modified sections need to be transmitted over LoRaWAN instead of the entire firmware, which tends to reduce the size of the update and optimizes the network load.

LoRaWAN end devices support a modular architecture, with the capability of loading, replacing and executing independent modules. In terms of software updates, this enables some modifications in a single module to be down-

loaded and installed silently without disturbing other parts of the system. By adopting a dynamic linking technique in FUOTA methods, dynamic modifications can be applied without the necessity of a system reboot.

The Firmware Update Server (FUS) is designed as an "external connector" to the LoRaWAN server. It generates module images, collects/transmits data from/to the LoRaWAN server, and manages a set of module versions. The general design of the FUS is described in Figure 3. It is composed of a dedicated dynamic linker that is in charge of generating the module images from C source files and an update agent that is responsible for fragmenting, packaging and transmitting the module binaries. As modules contain a software code (i.e., a set of functions and procedures) that performs specific operations, such as reading data from a sensor or encrypting data, they can be compiled once and reused for different applications and different end devices. Our FUS server thus does not compile a module from its C files if a binary version of this module already exists. The binary module is directly passed to the update agent so as to be subsequently fragmented into several portions (fragmentation phase). Fragments are then packaged by appending various additional information to identify them (packaging phase) before being disseminated to reach end devices.

The main principle of the proposed FUOTA method is presented in Figure 4. On the FUS side, when some modifications are applied by the programmer to the source code of one or more modules, new binary images are generated. A request-to-update is transmitted to the end devices. Port 101 is reserved for firmware update purposes in LoRaWAN networks. The end devices verify the request and disseminate a ready-to-update signal. The FUS ensures the fragmentation of the firmware modules in several pieces of code so they can be embedded as payload in LoRaWAN packets. The fragment size is to be defined beforehand, which should strictly comply the LoRaWAN regulations. The ready-to-update message emitted by an end device implicitly requests all the fragments from the FUS. After having received all the fragments, the end device assembles them in a buffer, and removes the header data that are no longer useful. The old module image is replaced by the content of this buffer (i.e, the new module), followed by a process of unloading and reloading the module. A transmission error is detected by the end device when the sequence number of the fragment is not as expected, or when the checksum of a fragment is not correct. In this case a request to retransmit the fragment is sent to the FUS. All of these operations are performed dynamically and seamlessly, without a system reboot. Note that in this paper, we do not consider dependencies between modules. It is incumbent on the programmer to ensure that modules are updated in the correct order.

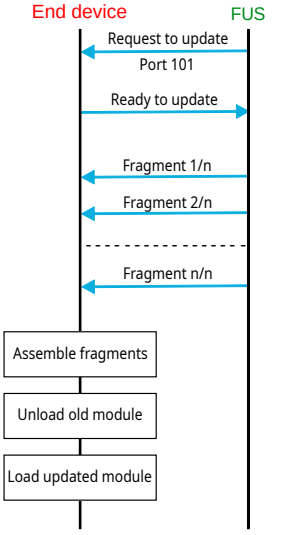


Fig. 4: FUOTA working principle

4. Implementation

To justify the feasibility of our approach, we propose a proof of concept operating under practical conditions. It exploits LoRaWAN end devices based on the STM32WLE5JC chip, a SX1302 868 MHz LoRaWAN gateway and The Things Network (TTN) platform. It is worth mentioning that even though a specific prototype is implemented for evaluation purposes, our approach remains compatible with other LoRaWAN network servers such as Actility or ChirpStack, and other STM32 MCU generations.

The general architecture of the FUOTA method is kept, as shown in Figure 2. We leverage Udynlink[2], a dynamic linker for ARM Cortex-M MCUs, that compiles code into a binary blob that can be loaded and executed at runtime on the MCU. The end device should be connected directly to the gateway and has the capability of performing the following functionalities: to handle received events such as a downlink frame or a join request, to process modules in an on-demand manner and to transmit uplink data for various purposes such as monitoring the temperature, humidity, or GPS data. On one hand, the gateway communicates with the end devices through the LoRa physical layer to send uplink messages and receive firmware fragments. On the other hand, it is also connected to the TTN LoRaWAN server through the Internet Protocol (IP). The LoRaWAN server is responsible for receiving module fragments transmitted by the FUS, performing the authentication and encryption process before forwarding them to the gateway. The FUS

[2] https://github.com/bogdanm/udynlink/

is composed of a Udynlink compiler to generate modified module images and an update agent being responsible for processing, transmitting and managing multiple module versions. The protocol used to communicate between the LoRaWAN server and the FUS is MQTT (Message Queuing Telemetry Transport), which is a lightweight, publish-subscribe, machine to machine network protocol for message queuing service. TTN exposes a MQTT broker to work with streaming events. We have implemented two MQTT clients: one is embedded in the LoRaWAN server and one is integrated into the update agent of the FUS. Typically, binary module images are fragmented, constructed by appending some additional information before being published to the broker through a dedicated topic. The supplementary data to be added in the beginning of the frame include the sequence number of the current fragment, the total number of fragments and a checksum, which enable the end device to verify the communication as well as to detect the final transmission. The LoRaWAN server can subscribe to the topic, listen and forward fragments. The fragment size depends on the configured data rate and also on the frequency band used in the application. Specifically, the LoRa frequency 863–870 MHz in Europe allows the payload size to vary between 51 bytes for the slowest data rates, and 222 bytes for higher rates.

As shown in Figure 5, the STM32WLE5JC MCU embeds 256 kB of single-bank flash memory, containing 128 pages of 2 kB each. This memory space is dedicated to storing the micro-system and the loadable modules (main module and functional modules). Here, we propose to allocate the first 50 pages (100 kB) for the micro-system and the last 78 memory blocks (156 kB) for storing module binaries. Again, the memory allocation strategy depends strongly on the memory organization defined by the manufacturer as well as the user intention.

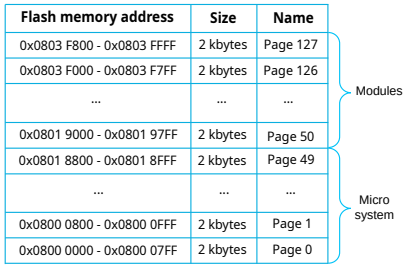


Fig. 5: Flash memory of the STM32WLE5JC MCU

5. Evaluation

Several experiments were carried out to evaluate the performance of the proposed prototype and to compare it with existing solutions. First, we measured the memory requirement of a FUOTA method based on a traditional monolithic design and on a dynamic module-based approach. Then, we assessed the impact of code changes on the size of the binary firmware to be updated and on the network load. The LoRaWAN parameters and the system configuration are listed in Table 1.

Table 1: Experiment parameters

Parameters	Value
LoRa frequency band	868 MHz
LoRa spreading factor	7
LoRa bandwidth	125 kHz
LoRa coding rate	4/5
Fragment size	200 bytes

Table 2: Memory consumption between monolith and modular design

	Section	Flash required (kB)	RAM required (kB)
Monolithic	Bootloader	17.96	2.16
	Active firmware	18.6	1.54
	Downloaded firmware	20	0
	Total	*56.56*	*3.7*
Modular	Micro-system	14.46	2.13
	Modules	11.27	2.78
	Total	*25.73*	*4.91*

Typically, the IoT device's firmware is composed of different sections such as application code, core, libraries, utilities, etc. It is possible to modularize every part of the program, but it is beyond the scope of this work. For our experiments, considering the fact that changes are mostly envisioned in the application code of an end device, we restricted the changes to the code section. We designed one that is composed of ten function blocks (encryption, wireless configuration, sensor data collection, etc). Various modifications were applied on this code during our experiments. For comparison purposes, the modifications are propagated to the end devices first through a monolithic firmware, and then through a set of modules whose size varies from 0.71 to 1.36 kB. A monolithic-based FUOTA method performs a full image replacement, and thus requires at least three separate sections in memory to store the active firmware, the firmware being downloaded, and a bootloader to handle them. Here, we assume that the partition in flash memory allocated for the downloaded firmware image is slightly larger that the active firmware, in order to accommodate a

small code extension (20 kB compared to 18.6 kB). In contrast, the module-based FUOTA approach requires only two partitions, one for the micro-system and one for the module images. In the process we described in section 3, modules can be loaded at any time, on demand. Moreover they can be entirely copied to RAM to speed up the execution. Here, to simplify the assessment, all the modules are systematically loaded from the start of the program and designed to execute directly from flash memory, that means only the .data section of a module is copied to RAM.

5.1. Memory requirement

The comparison in terms of memory requirement between the two approaches is presented in Table 2. The modular FUOTA approach demands a total of 25.73 kB of flash memory to store the firmware (micro-system and modules). The micro-system, that is not updated in a modular way, is responsible for dynamically loading/unloading the modules, resolving symbol values and managing modules. To ensure a proper update process, the flash consumption for the monolithic FUOTA method is 56.56 kB, including the bootloader, the active and downloaded firmware. The higher efficiency in flash memory usage witnessed in the modular approach is mainly explained by the fact that the update is done dynamically, one module at a time, without affecting the main routine or other parts of the system. So the buffer required for the downloading process simply hosts a single module, and not the entire firmware like in the monolithic approach.

In terms of RAM usage, the module-based update requires 4.91 kB, which is slightly more than the 3.7 kB used in a traditional monolithic FUOTA mechanism. This is due to the fact that it needs to allocate more RAM to dynamically load and execute modules at run-time. However, this could be mitigated by an on-demand loading strategy, in which unused modules are unloaded to free their allocated memory.

5.2. Update size and network load

To evaluate the impact of code changes on the size of the binary firmware to be updated and on the network traffic, we did the same modifications in the program in both configurations (modular and monolithic). The modifications consist in a number of small random changes in the code symbols, distributed evenly across the function blocks. We conducted a series of experiments, varying the percentage of symbols that were changed. Figure 6 gives, for each percentage of code changes, the update size (i.e., the cumulative size of the updated code portions), and the detailed network load (MQTT, LoRaWAN, LoRa). Indeed, the transmission of a (monolithic or modularized) firmware from the FUS to the end devices must take into account the MQTT, LoRaWAN and LoRa physical layer overheads. The MQTT protocol generates costs relative to connect, acknowledgment, subscribe and publish messages issued by the clients and the broker. The LoRaWAN overhead comes from embedding some additional information such as MAC header, frame header, frame port and message integrity code into a LoRaWAN frame payload before propagating it to the gateway through IP. Preamble, header, cyclic redundancy check are also automatically added to the physical payload, which raises the packet size to be broadcasted in the LoRa network. Note that the TTN imposes the behavior of the LoRaWAN server, that simply converts in LoRaWAN frames the MQTT payloads it receives, thus preventing us from implementing a fragmentation on the LoRaWAN server instead of on the FUS server, which would have reduced the MQTT overhead.

We can see in Figure 6 that in the monolithic approach, the update size and the network load remain roughly the same regardless of the percentage of code changes: an update size of around 19 kB, with a MQTT load of 81 kB, a LoRaWAN load of 20 kB and a LoRa load of 27 kB. Slight differences are presented because of the random nature of the code changes, differences that would disappear if the average on a large number of experiments were computed.

The important aspect lies in the gap between the results for the monolithic approach and those for the modular approach. The modular FUOTA method denotes a higher performance than the monolithic approach in terms of the update size, with a gain ratio ranging from 1.7 (with 100% of code changes) to 17 (with 1% of code changes). Even with 100% of code changes, the modular approach performs better because fixed parts of the firmware (i.e., the micro-system) do not need to be updated. The gain on the network load (which is approximately a linear function of the update size) is also spectacular. Obtaining a small load in the LoRa network is particularly beneficial: as shown in the figure, it passes from 27 kB for the monolithic approach to 1.6 kB when using dynamic modules, with 1% of code changes.

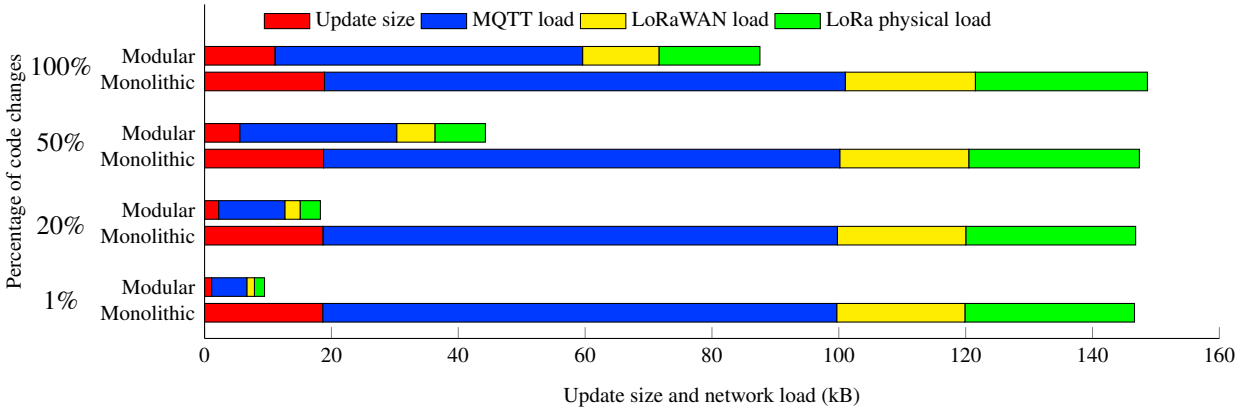


Fig. 6: Impact of code changes on the update size and the network load

6. Conclusion

In this paper, we have proposed a module-based approach for the development of IoT devices' firmware, and a new FUOTA method for devices communicating in LoRaWAN networks. This method enables partial and dynamic updates without requiring a system reboot. Experimental results obtained on a small testbed show that the solution we propose optimizes the update size and the network traffic up to 17 times compared to the traditional monolithic-based method.

In future work, we would like to improve the performance of this mechanism and conduct additional experiments to evaluate the update time and energy efficiency in the context of multiple end devices. In addition, we also investigate the feasibility of a dynamic modular FUOTA in opportunistic networks, considering that in some challenging networks, not only some end devices are not connected directly to the gateways but also there is no end-to-end paths between them.

References

[1] Anastasiou, A., Christodoulou, P., Christodoulou, K., Vassiliou, V., Zinonos, Z., 2020. IoT Device Firmware Update over LoRa: The Blockchain Solution, in: 16th International Conference on Distributed Computing in Sensor System (DCOSS 2020), IEEE. pp. 404–441.

[2] Centenaro, M., Vangelista, L., Zanella, A., Zorzi, M., 2016. Long-Range Communications in Unlicensed Bands: The Rising Stars in the IoT and Smart City Scenarios. Wireless Communications, IEEE 23, 60–67.

[3] Dunkels, A., Finne, N., Eriksson, J., Voigt, T., 2006. Run-Time Dynamic Linking for Reprogramming Wireless Sensor Networks, in: 4th International Conference on Embedded Networked Sensor Systems (SenSys 2006), ACM, Boulder Colorado USA. pp. 15–28.

[4] Dunkels, A., Gronvall, B., Voigt, T., 2004. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors, in: 29th Annual IEEE International Conference on Local Computer Networks (LCN 2004), IEEE, Tampa, FL, USA. pp. 455–462.

[5] Han, C.C., Kumar, R., Shea, R., Kohler, E., Srivastava, M., 2005. A Dynamic Operating System for Sensor Nodes, in: 3rd International Conference on Mobile Systems, Applications, and Services (MobiSys 2005), ACM, Seatle, WA, USA. pp. 163–176.

[6] Heeger, D., Garigan, M., Eleni Tsiropoulou, E., Plusquellic, J., 2021. Secure LoRa Firmware Update with Adaptive Data Rate Techniques. Sensors, MDPI 21, 1–17.

[7] Munawar, W., Alizai, M.H., Landsiedel, O., Wehrle, K., 2015. Modular remote reprogramming of sensor nodes. International Journal of Sensor Networks, Inderscience Publishers 19, 251–265.

[8] Ruckebusch, P., De Poorter, E., Fortuna, C., Moerman, I., 2016. GITAR: Generic Extension for Internet-of-Things ARchitectures Enabling Dynamic Updates of Network and Application Modules. Ad Hoc Networks, Elsevier 36, 127–151.

[9] Ruckebusch, P., Giannoulis, S., Moerman, I., Hoebeke, J., De Poorter, E., 2018. Modelling the Energy Consumption for Over-the-Air Software Updates in LPWAN Networks: SigFox, LoRa and IEEE 802.15.4g. Internet of Things, Elsevier 3-4, 104–119.

[10] Sornin, N., 2020. LoRaWAN®: Firmware Updates Over-the-Air. Technical Report. Semtech.

[11] Sun, Z., Ni, T., Yang, H., Liu, K., Zhang, Y., Gu, T., Xu, W., 2023. FLoRa: Energy-Efficient, Reliable, and Beamforming-Assisted Over-The-Air Firmware Update in LoRa Networks, in: 22nd International Conference on Information Processing in Sensor Networks (IPSN 2023), ACM, San Antonio, TX, USA. pp. 14–26.