ARCHIVIO ISTITUZIONALE
DELLA RICERCA

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

A Location-Aware WebAssembly-Based Software Update Framework for IoT End Devices

This is the final peer-reviewed author's accepted manuscript (postprint) of the following publication:

Published Version:
Zyrianoff, I., Montori, F., Trotta, A., Sciullo, L., Gigli, L., Kamienski, C., et al. (2025). A Location-Aware WebAssembly-Based Software Update Framework for IoT End Devices. 345 E 47TH ST, NEW YORK, NY 10017 USA : Institute of Electrical and Electronics Engineers Inc. [10.1109/ccnc54725.2025.10976056].

Availability:
This version is available at: https://hdl.handle.net/11585/1037103 since: 2026-01-14

Published:

DOI: http://doi.org/10.1109/ccnc54725.2025.10976056

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/).
When citing, please refer to the published version.

(Article begins on next page)

17 September 2026

A Location-Aware WebAssembly-based Software Update Framework for IoT End Devices

Ivan Zyrianoff*†, Federico Montori*†, Angelo Trotta*, Luca Sciullo*†,
Lorenzo Gigli*, Carlos Kamienski‡ Marco Di Felice*†
* *Department of Computer Science and Engineering, University of Bologna, Italy*
† *Advanced Research Center on Electronic Systems "Ercole De Castro", University of Bologna, Italy*
‡ *Federal University of ABC (UFABC), Brazil*

***Abstract*—The increasing computational capabilities of IoT end devices push the deployment of application logic tasks directly on the extreme edge rather than the cloud or edge nodes. However, there are still unresolved issues on the Over-The-Air (OTA) software update operations for IoT end devices: (*i*) the hardware heterogeneity in IoT settings requires custom code for each different device type; (*ii*) the growing complexity of microcontroller code couples the development of high-level processing tasks with low-level operations; (*iii*) efficient methods for updating target IoT devices in a specific geographical area are absent. To address these issues, we propose an OTA firmware update framework that utilizes WebAssembly (WASM) and incorporates location-aware features. We split the application logic in WASM from the rest of the firmware written in native code, in order to create a greater separation of concerns. WASM's platform independence creates an abstraction layer for the underlying hardware, allowing the same application logic to be deployed virtually to any IoT device. We integrate a location-aware extension of the MQTT protocol in our framework to enable software updates targeting devices in specific geographical areas. Finally, our experiments demonstrate that location awareness does not add significant overhead to the system and that the performance of WASM in a microcontroller is comparable to native code and superior to Micropython.**

I. Introduction

With the recent trend toward *softwarization* and the increasing computational capacities at the extreme network's edge, more processing tasks are being performed directly on IoT end devices [1]. This shift increases the complexity of the microcontroller code as application logic is offloaded to it.

However, three significant challenges hinder the widespread adoption of the softwarization trend in IoT large-scale deployments. The first is the lack of device interoperability [2]. Indeed, the hardware heterogeneity among IoT devices complicates software update operations, as each set of microcontrollers requires to be handled separately due to its variations in hardware and firmware configurations [3]. The second is the microcontroller code's undesirable coexistence of high-level processing tasks and low-level operations (e.g., reading electronic signals from a pin). These two sets of tasks require different skill sets, which leads to further complexity and maintainability of the embedded software. Finally, this shift exposes the need for a reliable and consistent pipeline to distribute code to embedded devices, leveraging features unique of IoT deployments, such as their geolocation. Modern software development requires more frequent code updates, and the current CI/CD methodologies are neither adequate for embedded devices nor location-aware since updates based on geographical positions are not required for traditional cloud-based deployments [4].

To address these challenges, we propose a location-aware firmware update framework for Over-The-Air (OTA) WebAssembly (WASM) code deployment to IoT devices. WASM is a platform-independent, assembly-like language with a compact binary format that delivers near-native performance [5]. Initially developed for web browsers, WASM has recently been ported to IoT devices due to its lightweight nature [6]. As a result of WASM platform independence, it can execute on diverse hardware architectures without customization, which allows deploying the same WASM code on different types of IoT devices, thus addressing the lack of interoperability mentioned. Our update framework keeps the application logic in WASM and the hardware-specific tasks in isolated native code modules; this way, we achieve a greater separation of concerns. Further, our framework either allows the update of only the application logic in a lightweight and rapid fashion; or the whole firmware code. To manage the update of IoT devices through their features and locations, we integrate our system with Location-Aware MQTT (LA-MQTT) [7], an extension of the traditional MQTT protocol that adds support for spatial-aware publish-subscribe communications. LA-MQTT enables group updates to devices in a target geolocation area, addressing the last challenge. A key advantage of this approach is that it utilizes one of the most widely used IoT protocols and does not require a custom broker, making it compatible with existing MQTT implementations.

We evaluate our proposed framework through experiments on a testbed of Espressif ESP32 devices. The goal of the experiments is twofold: (*i*) to quantify the overhead introduced by the localization features and (*ii*) to compare the execution delay of algorithms across WASM, native C++, and Micropython. Our results demonstrate that our framework's overhead is insignificant and that the WASM performance overcome Micropython in all scenarios.

II. Framework Architecture

The paper introduces a novel OTA software update framework tailored for IoT end devices, addressing the three key challenges outlined in the introduction: *interoperability*,

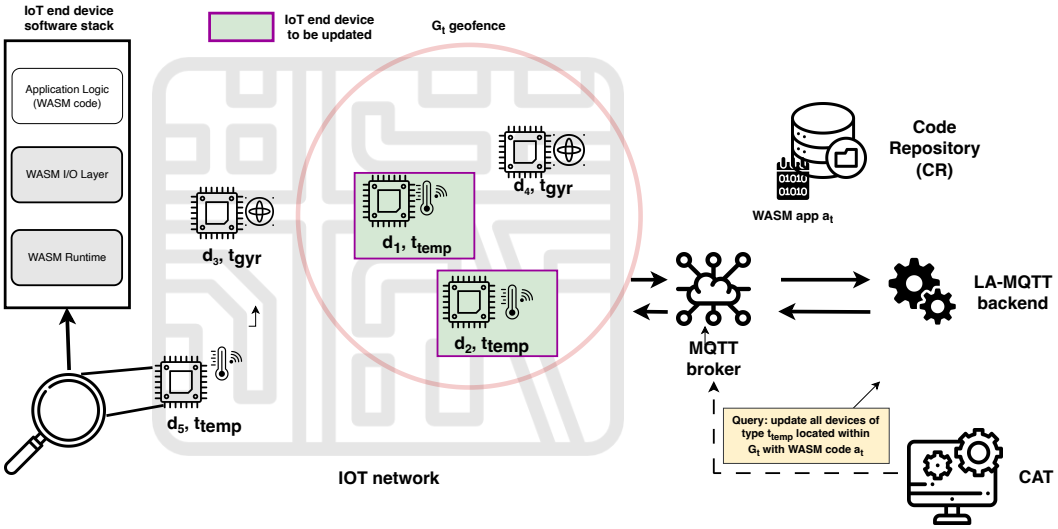

Fig. 1: Embedded software stack structure (on the left) and components of the proposed framework for the OTA group update of WASM applications (on the right).

greater separation of concerns and *location awareness*. Further, another requirement is *scalability*, which refers to the ability to implement software updates over a large-scale IoT network minimizing network overhead and latency.
Interoperability and the greater separation of concerns is achieved by leveraging the software stack proposed in [3]. The paper describes a reference software architecture for microcontrollers, which includes the three layers depicted in the left part of Figure 1:

1) *Application Logic*. The layer encapsulates the computational tasks executed by the device. Written in WASM [5], it ensures portability across various IoT devices and can be updated via the full procedure described in this paper. We will simply refer to it as "the application" in the following.
2) *WASM Runtime*. It allows the WASM code execution on constrained IoT devices, using any runtime available for extreme edge devices [8].
3) *WASM I/O Layer*: it handles interactions with the device's specific hardware. This layer is hardware-dependent and requires a complete firmware update for modifications, as detailed in [3]. The interaction between the WASM-based application logic and the native hardware is based on a WebAssembly System Interface (WASI).

Location awareness and scalability are achieved through the LA-MQTT protocol [7], an extension of the widely-used MQTT protocol for IoT. LA-MQTT enhances the traditional publish-subscribe model by incorporating geofencing capabilities. Devices can subscribe to topics as in MQTT, but with the added feature of being notified only when they are physically within a specified geographical area. We highlight that this extension maintains full compatibility with the standard MQTT protocol and requires no changes to the broker implementation.

Figure 1 depicts the system model and architecture of our framework. The core components are the following:

- *IoT end devices*: these microcontrollers form the target IoT network. They may vary in hardware architecture (e.g., CPU, memory size) but must support the previously mentioned software stack.
- *MQTT broker*: any legacy broker supporting the MQTT protocol.
- *LA-MQTT backend*: an external software acting as an MQTT client, enabling location-aware message distribution within a target area.
- *Code Repository (CR)*: it stores the WASM applications to be distributed to the IoT end devices.
- *Configuration and Administration Toolkit (CAT)*: this a software enabling the administrator to manage the IoT network and the applications installed on each device. Through the CLI, the administrator can upload new WASM applications to the CR and upgrade a subset of IoT devices, based on their types (e.g, upgrade all devices with accelerometer) and/or position (e.g., upgrade all devices located in a given area).

III. Firmware Update Procedure

In this section, we detail the application update procedure enabled by our framework. Without loss of generality, we assume that the IoT infrastructure consists of a fleet of N heterogeneous IoT devices. Each device d_i, where $0 \leq i < N$, has a unique identifier, denoted as l_i, which can be associated with the device's MAC address. Additionally, each d_i has a type label field, t_i, that describes its sensing and computational capabilities; for instance, the type label may indicate that the device is equipped with a temperature sensor. Furthermore, each device d_i can retrieve its current geolocation P_i, modeled as a pair of GPS coordinates corresponding to latitude and longitude. The localization

technology used by each device is beyond the scope of this paper. The software update procedure involves four stages, which are detailed in the following paragraphs.

1) Initialization & Periodic actions: After booting, each IoT end device d_i performs the following three MQTT actions:

- It issues a `PUBLISH` command to the default topic "*Registration*", providing its ID (l_i) as the payload.
- It issues a `SUBSCRIBE` command to a private topic (named after its own ID) to receive notifications about the initiation of a new update procedure.
- It issues a `PUBLISH` command to the default topic "*CPosition*" to notify the LA-MQTT backend of its current position, with P_i attached in the payload. This command is periodically executed every s seconds to account for mobile devices.

The initialization procedure also involves the LA-MQTT backend, which performs two complementary MQTT actions:

- It issues a `SUBSCRIBE` command to the topic "*Registration*" to be notified of the presence of IoT end devices.
- It issues a `SUBSCRIBE` command to the topic "*CPosition*" to track the current positions of all registered devices in the IoT network.

2) Update procedure start. At some point, the system administrators may decide to update the application logic of certain IoT end devices. To do this, they must provide three pieces of information through the CAT interface: (*i*) the application bytecode, compiled in WASM and referred to as a_t hereafter; (*ii*) the type of devices to be updated (e.g., all devices with accelerometer sensors), referred to as T_t; and (*iii*) the geofence area, referred to as G_t, where the update should be applied. The geofence area can be either circular or polygonal: in the first case, it is encoded as a JSON data structure with fields for the center and radius, while in the second case, it is encoded as a Polygon geometry using the GeoJSON format [9]. The CAT is responsible for uploading a_t to the CR through a REST API, which returns an HTTP endpoint ($e(a_t)$) from which the code can be downloaded. The CAT then issues a `PUBLISH` command to the "*Update*" topic, providing the triple $< T_t, G_t, e(a_t) >$ as the payload.

3) Code distribution. Whenever the LA-MQTT backend receives a new message on the "*Update*" topic, it constructs a list of potentially relevant IoT end devices, denoted as L. This list is defined as follows:

$$L = \{d_i \in N \mid t_i = T_t \wedge P_i \trianglelefteq G_t\} \quad (1)$$

Here, $\trianglelefteq$ is the geospatial operator that checks whether a point is contained within the geofence area. For each device $d_i \in L$, the backend issues a `PUBLISH` command to the device's private topic L_i, including the URL $e(a_t)$ as the payload.

4) Code installation. Each device d_i that receives a message on its private channel downloads the WASM code from the CR by issuing an HTTP GET request to the $e(a_t)$ URL. The downloaded WASM code is then loaded by the device's internal WASM interpreter without requiring a reboot. After loading the code, each device notifies the CR of the outcome of the procedure.

We highlight that the system operates agnostically, whether the IoT nodes are static (e.g., fixed IoT infrastructures) or mobile (e.g., embedded in autonomous vehicles). While the code distribution could be implemented using pure MQTT, as explored in [3], we opted to use HTTP for two main reasons: (*i*) the ability to use a secure transfer channel via HTTPS; and (*ii*) the potentially higher transfer speed, given the limitations on maximum payload size in some MQTT libraries for embedded systems. In terms of security, each WASM application stored in the CR must be signed using the administrator's private key, allowing each device d_i to verify the code by decrypting it with the corresponding public key. MQTT operations are further secured through authentication. However, there remains the risk of attackers stealing credentials to disseminate fake updates via MQTT topics or compromising the LA-MQTT backend. We plan to further investigate threat models and countermeasures in future work.

IV. Implementation and Preliminary Results

We developed a proof-of-concept for the custom architecture components depicted in Figure 1. The CAT and LA-MQTT backend components were implemented using Python. We deployed the embedded software stack for the popular Espressif ESP32-S2 microcontrollers, leveraging the PubSubClient[1] library for MQTT communication and the WASM3 library[2] as the WASM runtime. The CR component was implemented as a Node.js application.

In the previous section, we highlighted the novel features of the proposed OTA system, particularly its interoperability and location-awareness. In this performance evaluation, we demonstrate that these advantages do not incur excessive overhead in their implementation. We assess two types of overhead: (*i*) computational overhead, introduced by using WASM on embedded devices; and (*ii*) network latency overhead, resulting from the use of the LA-MQTT protocol and its backend. Regarding the first aspect, Figure 2 compares the execution delay of applications on microcontrollers, with varying coding languages: Native Arduino C++, WASM (via the interpreter), and Micropython. The focus is on feature extraction tasks, i.e.: an application processes an array of IoT data (potentially generated by embedded sensors) with a length of m items and produces f feature values in output. We considered three feature extraction tasks with different computational complexities: (*i*) Entropy with $f = 1$ and complexity $O(m)$, shown on the left; (*ii*) Interquartile Range (IQR) with $f = 1$ and complexity $O(m \cdot log(m))$, shown in the center; and (*iii*) Autocorrelation with $f = m$ and complexity $O(m^2)$, shown

[1]https://github.com/knolleary/pubsubclient

[2]https://github.com/wasm3/wasm3

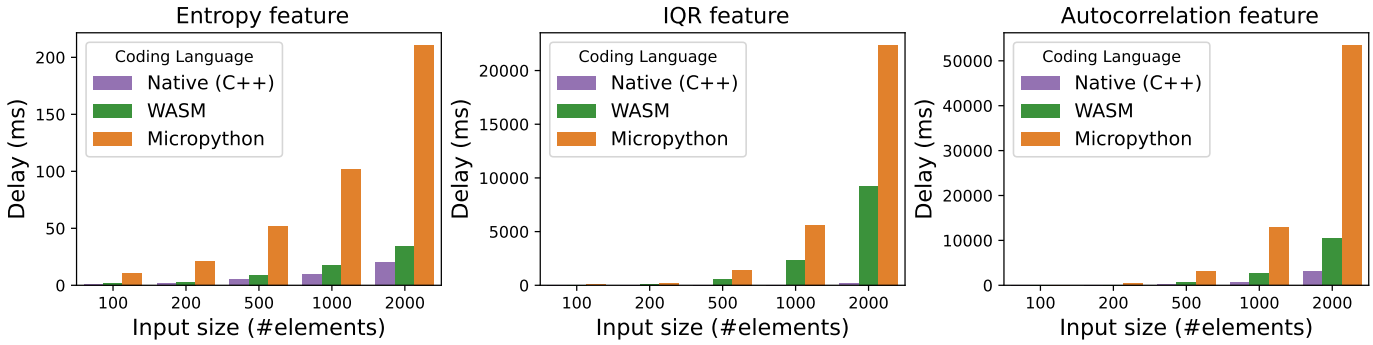

Fig. 2: Execution time for three feature extraction tasks: (*i*) Entropy (left), (*ii*) Interquartile Range (center), and (*iii*) Autocorrelation (right) when comparing three implementations: Native C++, WASM, and Micropython.

on the right. The x-axis of each plot varies the input size (m), while the y-axis shows the average computation time on the ESP32 device. As expected, the C++ implementation is the most efficient; however, WASM closely follows for small/medium input sizes and significantly outperforms Micropython for computationally intensive features.

Figure 3 shows the latency involved in completing the WASM update process as the number of ESP32 devices in the testbed varies. Latency is measured as the time interval from when the CAT initiates the procedure to when the last device in the IoT network notifies the successful loading of the updated WASM code. For this experiment, the geofence area was set to include all the ESP32 devices, and the IQR sketch (approximately 600B) was used as the WASM code to be updated. We compared two OTA update mechanisms: (*i*) the LA-MQTT protocol described earlier; and (*ii*) a legacy MQTT protocol without location awareness. In the second scenario, all IoT devices are associated with a default MQTT group. Each time the CAT initiates an update, all IoT devices are notified without any backend message filtering mechanism based on their locations. This approach generates network overhead proportional to the number of IoT devices located outside the target geofence, as thoroughly examined in [7]. The Figure demonstrates that the additional overhead introduced by the LA-MQTT backend is limited compared to the legacy MQTT approach and does not hinder system operations. We expect this overhead to become even less significant as the size of the WASM application being distributed increases.

Future work will involve more extensive experiments with heterogeneous microcontrollers and realistic applications, including scenarios like updating wearable IoT devices during physiotherapy sessions in indoor environments. Additionally, we plan to extend our framework by addressing crucial requirements such as security and reliability.

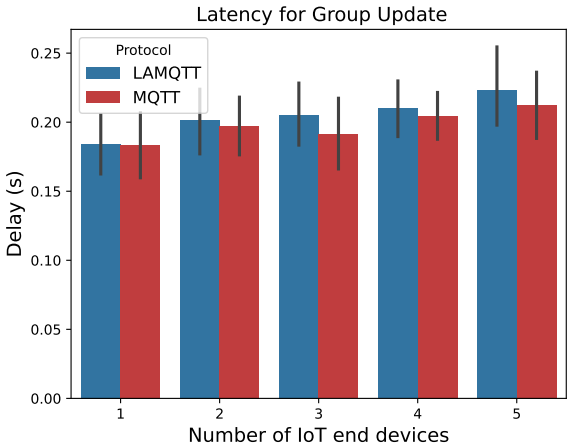

Fig. 3: Latency overhead introduced by location-awareness.

V. Acknowledgements

The work has been supported by the PRIN 2022 project I-TROPHYTS ("IoT and humanoid RObotics for autonomic PHYsio-Therapeutic monitoring, coaching and supervising in smart Spaces: a feasibility study", project id: 20224TAETP).

References

[1] M. A. Zormati, H. Lakhlef, and S. Ouni, "Review and analysis of recent advances in intelligent network softwarization for the internet of things," *Computer Networks*, vol. 241, p. 110215, 2024.

[2] L. Sciullo, L. Gigli, F. Montori, A. Trotta, and M. D. Felice, "A survey on the web of things," *IEEE Access*, vol. 10, pp. 47 570–47 596, 2022.

[3] I. Zyrianoff, L. Sciullo, L. Gigli, A. Trotta, C. Kamienski, and M. Di Felice, "An over the air software update system for iot microcontrollers based on webassembly," in *2024 20th International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT)*, 2024, pp. 331–338.

[4] F. B. Oliveira, M. Di Felice, and C. Kamienski, "Iotdeploy: Deployment of iot smart applications over the computing continuum," *Internet of Things*, p. 101348, 2024.

[5] A. Rossberg, "WebAssembly Specification - Release 2.0 (Draft 2024-01-17)." [Online]. Available: https://webassembly.github.io/spec/core/

[6] P. P. Ray, "An overview of webassembly for iot: Background, tools, state-of-the-art, challenges, and future directions," *Future Internet*, vol. 15, no. 8, p. 275, 2023.

[7] F. Montori, L. Gigli, L. Sciullo, and M. D. Felice, "La-mqtt: Location-aware publish-subscribe communications for the internet of things," *ACM Trans. Internet Things*, vol. 3, no. 3, jul 2022. [Online]. Available: https://doi.org/10.1145/3529978

[8] S. Wallentowitz, B. Kersting, and D. M. Dumitriu, "Potential of webassembly for embedded systems," in *2022 11th Mediterranean Conference on Embedded Computing (MECO)*, 2022, pp. 1–4.

[9] Internet Engineering Task Force (IETF), "The GeoJSON Format," https://tools.ietf.org/html/rfc7946, 2016, accessed: 2024-09-01.