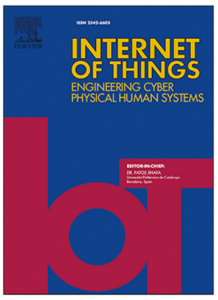
# Journal Pre-proof




A flexible Compilation-as-a-Service and
Remote-Programming-as-a-Service platform for IoT devices

Pavlos Charalampidis, Antonis Makrogiannakis,
Nikolaos Karamolegkos, Stefanos Papadakis, Yannis Charalambakis,
George Kamaratakis, Alexandros Fragkiadakis

PII: S2542-6605(22)00099-3
DOI: https://doi.org/10.1016/j.iot.2022.100617
Reference: IOT 100617

To appear in: *Internet of Things*

Received date : 13 April 2022
Revised date : 12 September 2022
Accepted date : 12 September 2022

Please cite this article as: P. Charalampidis, A. Makrogiannakis, N. Karamolegkos et al., A flexible Compilation-as-a-Service and Remote-Programming-as-a-Service platform for IoT devices, *Internet of Things* (2022), doi: https://doi.org/10.1016/j.iot.2022.100617.

Title Page

# A flexible Compilation-as-a-Service and Remote-Programming-as-a-Service platform for IoT devices

**Abstract**

The Internet-of-Things (IoT) presents itself as an emerging technology, which is able to interconnect a massive number of heterogeneous smart objects. Several complex data-driven applications, such as smart cities applications, home automation, health monitoring, etc., have been realized through the existence of these ubiquitous networks of smart objects. The ability to remotely update the devices forming an IoT network is of paramount importance, as it enables adding new functionality in their firmware, either for resolving software bugs and security vulnerabilities or for application re-purposing, without the need to physically access them. In this work, we present a flexible Compilation-as-a-Service and Remote-Programming-as-a-Service platform that jointly offers cloud-based compilation and Firmware-Over-The-Air (FOTA) update functionalities for deployed IoT devices, in a reliable and secure manner. Our system is capable of easily supporting various embedded operating systems and heterogeneous hardware platforms. We describe the system architecture and elaborate on the implementation details of all system components. In addition, we perform an extensive performance evaluation of a Proof-of-Concept (PoC) deployment of our system and discuss results in terms of system response, scalability and resource utilization.

*Keywords:*
Internet-of-Things, Cloud computing, Platform-as-a-Service, Cloud compilation, Over-the-air programming

## 1. Introduction

The Internet-of-Things (IoT) is a concept first introduced in the context of supply chain management by Kevin Ashton, who described it as a system able to interconnect physical objects through the Internet, by using ubiquitous sensors [1]. Since then, IoT has been an area of active research for providing solutions that support complex data-driven applications in a variety of domains, such as smart-cities [2, 3], healthcare [4, 5], industrial automation [6], manufacturing [7], etc. The advances in micro-electromechanical systems that enabled the development of low cost sensors, the progress of wireless communications in the field of Wireless Sensor Networks (WSNs), as well as the growing market demand for machine-to-machine interaction, have been identified as key factors that led to the remarkable popularity and adoption of the IoT technology [8].

In order to realize the IoT vision, a massive number of IoT devices, also known as *smart objects*, is commonly deployed for data collection and actuation upon the environment. These smart objects are typically of resource-constrained nature in terms of computational, storage, network and energy resources, thus an appropriate computing environment accounting for device

# A flexible Compilation-as-a-Service and Remote-Programming-as-a-Service platform for IoT devices

**Abstract**

The Internet-of-Things (IoT) presents itself as an emerging technology, which is able to interconnect a massive number of heterogeneous smart objects. Several complex data-driven applications, such as smart cities applications, home automation, health monitoring, etc., have been realized through the existence of these ubiquitous networks of smart objects. The ability to remotely update the devices forming an IoT network is of paramount importance, as it enables adding new functionality in their firmware, either for resolving software bugs and security vulnerabilities or for application re-purposing, without the need to physically access them. In this work, we present a flexible Compilation-as-a-Service and Remote-Programming-as-a-Service platform that jointly offers cloud-based compilation and Firmware-Over-The-Air (FOTA) update functionalities for deployed IoT devices, in a reliable and secure manner. Our system is capable of easily supporting various embedded operating systems and heterogeneous hardware platforms. We describe the system architecture and elaborate on the implementation details of all system components. In addition, we perform an extensive performance evaluation of a Proof-of-Concept (PoC) deployment of our system and discuss results in terms of system response, scalability and resource utilization.

*Keywords:*
Internet-of-Things, Cloud computing, Platform-as-a-Service, Cloud compilation, Over-the-air programming

## 1. Introduction

The Internet-of-Things (IoT) is a concept first introduced in the context of supply chain management by Kevin Ashton, who described it as a system able to interconnect physical objects through the Internet, by using ubiquitous sensors [1]. Since then, IoT has been an area of active research for providing solutions that support complex data-driven applications in a variety of domains, such as smart-cities [2, 3], healthcare [4, 5], industrial automation [6], manufacturing [7], etc. The advances in micro-electromechanical systems that enabled the development of low cost sensors, the progress of wireless communications in the field of Wireless Sensor Networks (WSNs), as well as the growing market demand for machine-to-machine interaction, have been identified as key factors that led to the remarkable popularity and adoption of the IoT technology [8].

In order to realize the IoT vision, a massive number of IoT devices, also known as *smart objects*, is commonly deployed for data collection and actuation upon the environment. These smart objects are typically of resource-constrained nature in terms of computational, storage, network and energy resources, thus an appropriate computing environment accounting for device

constraints is necessary. To this end, we have lately observed the integration of Cloud computing with the IoT domain. Cloud computing is a model for ubiquitous and flexible access to a large shared pool of resources (computational, storage, network, etc.) that can be provisioned dynamically with minimal management effort. Synthesizing the two worlds, has been beneficial for the IoT that can enjoy, in this way, ubiquitous device reachability and management, effective service composition, flexible data storage, sharing and knowledge extraction, scalability as well as improved security [9].

It is common for the smart objects of an IoT deployment to operate unattended for long periods in variable or even harsh environmental conditions. Irrespective of the care taken during the development phase, the IoT devices need regularly to be re-programmed, either for resolving post-installation identified bugs and security vulnerabilities or supporting different features and re-purposing applications [10, 11]. Failing to do so, may result in decreased performance, security breaches that could compromise the privacy and safety of users, and generally undermine the long-term sustainability of the IoT deployment.

The traditional solution for updating IoT devices includes physical presence of maintenance personnel at the IoT deployment and manual firmware loading, for example, through some hard-wired back channel. Obviously, this solution lacks scalability and suffers from time inefficiency, especially in cases of urgent updates that, for instance, arise when new security vulnerabilities are identified. Furthermore, there exist deployment scenarios that physical access of IoT devices is infeasible, e.g. parking sensors implanted into asphalt roads or medical sensors implanted into human bodies. In order to address the aforementioned limitations, modern IoT deployments commonly support Firmware Over-The-Air (FOTA) updates, by both embedding relevant capabilities into the devices and extending appropriately the functionalities of the device management cloud component. Apparently, the dynamic, heterogeneous and resource-constrained nature of IoT devices should be taken carefully into consideration for achieving dependable and efficient FOTA programming.

Building firmware for heterogeneous device hardware requires that an IoT developer installs and maintains several development environments corresponding to different subsets of target devices or embedded operating systems. The process of manually finding and assembling an environment of toolchains, compilers, and library dependencies to develop applications for specific targets, usually imposes serious overhead, therefore, distracting the IoT developer from the development task itself. In addition, when the device firmware is built locally in a personal machine, the developer must later either submit it manually to a device management server, which subsequently automates the FOTA update process, or directly upload it to the IoT device(s). To this end, cloud-based compilers have been introduced that can provide a platform-independent environment for compiling, debugging and building software and, consequently, remove the burden of manually installing and maintaining various development and building environments. Commonly, they are accompanied by an online Integrated Development Environment (IDE) that enable users to conveniently code and maintain source code versions by using their web browsers.

In this paper, we present a flexible Compilation-as-a-Service (CaaS) and Remote-Programming-as-a-Service (RPaaS) platform that provides a joint solution for building device firmware and managing the secure remote FOTA updates of device fleets. The proposed platform offers an easy and efficient deployment of isolated workspaces for user development and embeds source code versioning and management functionalities that provide a user-friendly development experience. Furthermore, it automates the remote programming of IoT devices by offering reliable and secure FOTA updates and, most importantly, by easily supporting heterogeneous hardware architectures and embedded operating systems. We provide the details of a Proof-of-Concept

(PoC) implementation of the proposed platform and, finally, perform an extensive evaluation for establishing a baseline performance profile of the deployed system.

The rest of the paper is organized as follows. In Section 2, we present background information and related work on IoT cloud platforms, cloud compilation and FOTA updates for IoT devices. In Section 3, we introduce the system architecture, and we further describe enabling technologies and implementation details in Section 4. Section 5 presents a web-based user interface, acting as a third-party application that consumes CaaS and RPaaS services of the proposed platform. In Section 6, we present and discuss the performance evaluation of our system and, finally, in Section 7, we conclude the paper and present directions of future work.

## 2. Background

### *2.1. IoT cloud platforms*

IoT networks usually comprise a substantially large number of sensors and devices, which are interconnected for offering diverse and complex smart applications. The large shared pool of resources (computational, storage, network, etc.) offered by Cloud computing has been beneficial for the IoT that can now enjoy ubiquitous device reachability, effective service composition and management, flexible data storage, sharing and knowledge extraction, as well as improved security. Thus, one of the most critical components in modern scalable and dependable IoT solutions is the IoT Cloud Service Platform.

In general, an IoT Cloud Service Platform provides the necessary abstractions across the multitude of diverse devices and data sources comprising the IoT system under consideration. In addition, it enables the management and configuration of a range of systems and processes. In summary, the most important functionalities offered by an IoT Cloud Service Platform are [12]: (1) device registration and discovery management, (2) communication and topology management, (3) data collection (including retrieval, aggregation and storage), (4) data analysis and visualization, (5) configuration and provisioning management, (6) fault management (including detection, isolation and correction), (7) performance monitoring, (8) security management, (9) device firmware life-cycle management, (10) billing and accounting, and (11) API management.

Currently, all major tech players host IoT Cloud Platforms with the aforementioned set of functionalities (even more enriched), as a part of the broader Cloud services they offer. For example, Google offers *Google IoT Core* [13] that mainly provides secure connection and management of IoT devices. More functionalities are provided through existing Google Cloud products, such as Pub/Sub, Cloud Data Flow, Cloud Bigtable, etc. Respectively, Amazon offers *Amazon Web Services (AWS) IoT Core* [14] for enabling a large number of IoT devices to interconnect and interact with other AWS cloud services (such as AWS Lambda, Amazon DynamoDB, Amazon S3, Amazon SageMaker, etc.) for supporting complex smart applications. Amazon also includes services specifically tailored for IoT, namely AWS IoT Events, for detecting and responding to events from IoT sensors and applications, and AWS IoT Site Wise for collecting, organizing, and analyzing industrial equipment data. Finally, *IBM Watson IoT Platform* [15], cooperates with a large number of IBM cloud services that facilitate the development of IoT applications, such as Event Streams for managing event streams through Apache Kafka, Cloudant for distributed storage of massive volume of IoT data, Cloud Object Storage for storing unstructured data as objects, and IoT Blockchain Service for interconnecting IoT devices with IBM Blockchain and enabling their interaction through smart contract transactions.

Resource management in IoT cloud systems is also a vital but challenging domain. Its complexity is attributed to several factors, such as the resource-constrained nature of IoT devices, the

system heterogeneity and limited standardization at protocol level, as well as the large volume and commonly irregular nature of data streams produced by the devices. There exists a significant amount of work in the literature related to open frameworks for IoT resource management and service provisioning/composition. In [16], the authors propose OpenIoT, an open service framework, which facilitates the establishment of an ecosystem that fosters the widespread adoption of IoT-related devices and software and benefits all the stakeholders involved. RERUM [17] is an architectural framework and platform which allows IoT applications to consider security and privacy mechanisms early in their design phase, ensuring a configurable balance between reliability and privacy in IoT systems. In [18], the authors describe the requirements and a reference architecture for IoT service platforms that follows the well-known four-layer architectural model, while in [19], resource-based strategies for optimal resource management of IoT ecosystems are reviewed. Finally, in [20], a common IoT service platform is designed for balancing connectivity and intelligent application support, by also employing an IoT mediator service (IoT broker) for achieving a high level of interoperability.

### *2.2. Cloud compilation*

Compilers are used to convert the source code from a high level language to machine language with executable format. Subsequently, the generated binary files can be executed to produce the program's output. Although, the compilation process itself is simple, developers have to setup their programming and building environment based on the languages they use. It is evident that the installation and configuration of all necessary prerequisites is platform-dependent, which makes the maintenance and the upgrade of the environment strenuous. To this end, cloud computing environments can offer an efficient way of source code compilation and software building in the cloud.

Cloud-based compilers provide an architecture-independent environment that is convenient for compiling, debugging and executing software programs. Specifically, most cloud-based compilers have the following characteristics:

- *Easy and quick deployment*: after a developer selects a programming language, a fully functional development environment, with all necessary libraries and dependencies, can be rapidly setup.
- *High usability*: a web-based UI allows the developer to write code in a user-friendly editor and get the output of compilation in her browser. If compilation succeeds, the executable file is generated and is available for downloading; otherwise, the compiler diagnostics are returned. The user is also commonly offered the choice to easily import a library per her needs.
- *Low latency*: the compilation is usually a computationally expensive process, necessitating a significant amount of time to complete. Cloud-based compilers run in distributed cloud clusters equipped with high performance CPUs, thus, the compilation latency can be substantially reduced.
- *Seamless collaboration:* Each development project can be accessible from multiple users who collaborate using version control tools like Git.

There are several works in the bibliography proposing cloud compilation solutions. In [21], a cloud compiler for C/C++ programming languages is proposed, which is built as a web-application using ASP-NET. The application acts as a centralized compilation endpoint, accessed

by authenticated users. In [22], an ahead-of-time cloud compiler is designed that builds Android applications with protection against bytecode-targeting. The authors of [23], design and implement a just-in-time multi-language and cross-platform cloud compiler for performing symbolic-numeric computations. Additionally, there are several multi-language cloud compilation services, such as Codeanywhere[1], Amazon Web Services (AWS) Cloud9[2], Koding[3], Codechef[4] and Microsoft Roslyn[5] that also provide an online Integrated Development Environment (IDE), accessible through web browsers. These are primarily general-purpose solutions that are not specifically tailored for compiling and building software for embedded IoT devices. Specifically, a popular online development environment for IoT devices is the mbed Compiler[6] which provides an online C/C++ IDE for developing IoT applications based on Arm Mbed OS. In contrast, our platform's focus is on providing a joint solution for source code management and firmware building for heterogeneous IoT devices and operating systems, by easily integrating various development toolchains and offering isolated workspaces for user development.

### *2.3. FOTA update for IoT devices*

As mentioned in Section 2.1, firmware life-cycle management is a core functionality provided by an IoT Cloud Platform. The firmware running on the IoT devices needs to be regularly updated during their lifetime to optimize existing features, fix software bugs and, more importantly, patch security vulnerabilities. Failing to do so may result in overall system performance degradation, as well as compromise the security and privacy of users. Due to the particularities of IoT networks, such as difficulty in accessibility of installed device and time-consuming nature of manual per-device firmware update, IoT devices commonly enjoy remote programming capabilities that enable FOTA updates. The most typical features of a reliable, robust and secure FOTA mechanism are: (1) atomic firmware installation, (2) easy rollback to previous version, (3) failure management (i.e. in case of power or connectivity loss), (4) short downtime, (5) simultaneous firmware update of multiple devices, (6) firmware authentication (the device should verify the origin of the firmware and the identity of the sender), (7) encrypted communication during the firmware transfer towards the devices, and (8) automatic deployment of the firmware.

Most FOTA mechanisms are based on server-client architecture, where the IoT device is the client, while the IoT Cloud platform hosts a server for deploying the new firmware. The communication between the IoT devices and the update manager takes place using technologies such as WiFi, LoRa, or IEEE 802.15.4 based protocols like Thread, ZigBee and 6LoWPAN. Several platforms, both open-source and commercial, aim at providing FOTA services with the aforementioned features. Here, we briefly discuss some of them.

*Mender* [24] is an open source over-the-air software update manager for embedded Linux devices, which considers security and reliability of the update process, enabling both application and full system update. Mender uses *Yocto Project* [25] as the software build system for generating new device software and the artifacts required by the target device. It is noted that although *Mender* is not a general purpose IoT device management platform, as it is solely focused on

[1] https://codeanywhere.com
[2] https://aws.amazon.com/cloud9
[3] https://www.koding.com
[4] https://www.codechef.com
[5] https://dotnetfiddle.net
[6] https://ide.mbed.com/compiler

managing and orchestrating the software updates, it has been successfully integrated into other major IoT platforms, such as Google Cloud IoT Core and Microsoft Azure IoT.

*Balena* [26] offers a set of tools for building, deploying, and managing fleets of connected embedded Linux IoT devices by leveraging Linux containers technology. IoT devices run *Balena* OS, a Yocto Project Linux-based OS, packaged with *balenaEngine*, a lightweight Docker - compatible container engine that manages containers executing *Balena* services or user applications. Reliability of the update process is based on pre-update sanity checks in terms of software-device compatibility and availability of the image in the container registry, double root partition approach (active/inactive partition) and rollback support.

*ARM Pelion* [27] IoT platform is a full-stack suite of management services that focuses on three core IoT components, namely connectivity, device, and data management, for devices running the ARM Mbed OS or Linux/Mbed Linux OS. Over-the-air firmware updates are performed in the form of *campaigns* that apply either to a single device or fleet of devices. Efficiency of the update process, especially for low-rate and low-power IoT devices (e.g. NB-IoT), is enhanced by delta updates, while reliability mechanisms include conditional updates, sanity checks on received firmware (bootloader verifies firmware integrity, by calculating its hash and comparing it with the one received as firmware metadata) and rollback support through dual partitions (active partition / candidate partition).

*Particle* [28] offers a full-stack IoT solution that includes different hardware platforms, various connectivity options and cloud services for device fleet management, over-the-air updates and device health monitoring. Reliability and resiliency of the firmware update process is supported by features, such as atomic updates, automatic rollbacks (in case of failure during firmware transfer), context-aware updates based on device operational status (e.g. update is postponed for a later time, if the device currently performs a critical task) and batch updates.

*Amazon FreeRTOS* [29] is an extension of FreeRTOS, a well-known preemptive real-time operating system for embedded devices, provided by Amazon. It includes libraries for the secure and reliable connection of IoT devices with *Amazon Web Services IoT* (AWS-IoT) cloud platform, for device management, data collection and application development. The *FOTA Update Manager* service, which is part of the AWS IoT backend, is responsible for notifying the device on existing updates, orchestrating the update process and maintaining the update log. In order to improve reliability and efficiency of the update process, the AWS IoT for FreeRTOS provides support for batch updates, continuous updates and update rollback.

Our platform offers a joint solution for compiling and building IoT device firmware and managing the secure remote programming of IoT devices. To this end, it incorporates functionalities for easy development through an online IDE, tools for source code management and versioning, scalable compilation, build and storage of device firmware and reliable FOTA update. With regards to the previously presented solutions, only ARM Pelion IoT platform offers similar functionalities, however limited to Mbed OS-based IoT devices. On the contrary, our platform aspires to bring increased flexibility by providing not only support for heterogeneous IoT devices (in order to avoid vendor lock-in) but also a containerization-based mechanism for easily supporting different IoT operating system, such as Contiki-NG [7] and Zephyr RTOS [8].

[7] https://www.contiki-ng.org
[8] https://www.zephyrproject.org

## 3. System architecture

In this section, we introduce the system architecture of the proposed flexible and reliable CaaS and RPaaS platform. At first, we define the system requirements and later, we present the high-level functional architecture of our platform, discussing its functional entities.

### *3.1. System requirements*

The requirements analysis should take carefully into account several critical aspects of the platform design. Following the common methodology, we distinguish between functional and non-functional requirements. Functional requirements describe the functions that the platform must be able to perform. On the other hand, non-functional requirements describe the attributes and constraints on the system.

#### *3.1.1. Functional requirements*

- **Heterogeneous device support:** The platform should account for native IoT device hardware platform heterogeneity, by supporting easy and flexible enrollment of different device flavors.
- **Device virtualization:** Physical devices and their resources should be appropriately virtualized, following a uniform information model that can be easily queried through standardized syntax.
- **Device life-cycle management:** A set of management resources should offer device configuration (e.g. firmware update configuration) and monitoring of device status. Device activity (e.g registration, de-registration etc.) should be appropriately logged.
- **Ubiquitous access and transparent communication:** Devices should be accessible irrespective of constraints imposed by end-node network topology and configuration. In case devices do not support IPv6 or are not IPv6 routable, devices should be accessed through (transparent) gateways that provide protocol translation or Network Address Translation (NAT).
- **Multi-tenancy:** Multiple tenants (platform users) should have customizable access to cloud and device infrastructure the platform provides. Adequate isolation between tenants is necessary for security and performance reasons.
- **Source code management:** Source code of users' development projects should be effectively and consistently stored, updated, versioned and automatically assessed for quality, in the cloud.
- **Compilation and build isolation:** User source code compilation and firmware image build should be performed in an isolated environment, both for compatibility (i.e. to avoid inconsistencies between different toolchains) and security reasons.
- **Multiple operating systems support:** Software heterogeneity of IoT devices dictates the availability of build toolchains for multiple embedded operating systems.
- **Reliable and robust remote programming:** FOTA programming should be safeguarded by reliability mechanisms, such as A/B firmware updates and automatic firmware rollback in case of a failure. The firmware image transmission should be robust, accounting for the lossy nature of the communication channel.

- **Firmware authenticity and integrity:** Devices should receive and install firmware originating from authorized sources. Firmware binaries should be cryptographically protected against unauthorized disclosure and modifications.

*3.1.2. Non-functional requirements*

- **Security and privacy:** Authentication and authorization of user and devices should be guaranteed. Control and application data communication should be encrypted and integrity-protected. The system should be able to support different levels of security, depending on IoT device capabilities.
- **High availability:** The platform should ensure the high availability of compilation service, remote-programming service, and IoT devices (regardless of network conditions).
- **Fault tolerance:** The system should be resilient and recover quickly form faults and failures, on both cloud and IoT device side.
- **Scalability:** The platform should be scalable to support a large number of users and devices.
- **Quality-of-Service (QoS):** QoS should be maintained as high as possible in terms of interaction with cloud services (e.g. low latency of compilation/build process) and IoT devices (e.g. low latency of firmware update process).
- **Energy efficiency:** Due to the resource constraint nature of many IoT devices, the energy efficient operation of the devices (especially the remote-programming process on device side) should be ensured.

*3.2. High-level functional architecture*

Here, we briefly describe the high-level functional architecture of the proposed CaaS and RPaaS platform. The interested reader is referred to [30] for more details.

Figure 1 displays the functional view of the platform architecture. The bottom part of the figure shows the *Device Adaptor* that is responsible for device registration, resource management, as well as FOTA update, through the *Firmware Installer* functional component.

The *Device Manager* is responsible for registering, managing and monitoring the devices. It consists of the *Device Template Store*, which stores a collection of unique device templates supported by the platform, the *Device Registry* that holds registered IoT devices, the *Device Discovery* component, which is responsible for discovering the devices based on the requests originating from the *Service Conductor*, and the *Device Management* that is responsible for device bootstrapping, service enablement and device monitoring.

The platform's Service Layer comprises the *Service Conductor*, the *Compilation Manager*, and the *Remote Programming Manager*. The *Service Conductor* acts as the main endpoint and service orchestrator for handling incoming requests, discovering and orchestrating the Compilation and Remote-Programming services in cooperation with the *Device Manager*, as well as monitoring the availability, status and load of cloud services.

The compilation service of the platform is provisioned through the cooperation of *Compilation Manager*'s functional components. These include, the *Source Code Manager* that maintains and manages the user's source code, the *Compilebox Orchestrator*, which is responsible for the configuration, provisioning and scaling of *Compileboxes* (isolated containerized environments,

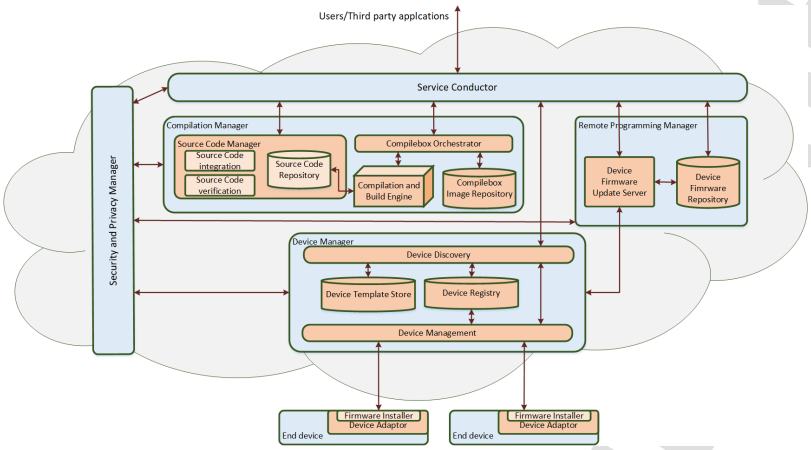



Figure 1: Platform architecture

where the user source code is compiled and the firmware image is built), the *Compilebox Image Repository* that stores the Compilebox images, and the *Compilation and Build Engine*, whose functionality includes fetching user source code from the *Source Code Repository*, compiling and building the device firmware, as well as processing the produced output, both in case of success and failure.

The *Remote Programming Manager* provisions the respective service through the cooperation of the *Device Firmware Repository*, which stores different versions of built firmware images, for different users and devices, and the *Device Firmware Update Server*, which is responsible for orchestrating the firmware update process, by fetching a firmware image from the *Device Firmware Repository* and invoking the *Device Adaptor*'s *Firmware Installer*, through the *Device Management* component, for transferring the image to the device and initiating the update process.

Finally, the *Security and Privacy Manager* is a cross-layer entity, responsible for all security-related operations of the platform, such as user authentication, authorization and access control, secure device bootstrapping and trusted credentials storage, device authentication, communication and data-at-rest encryption, firmware image and manifest integrity generation/verification, etc.

## 4. Implementation details

We hereafter present the technical details of our implemented solution. At first, we describe the tools and technologies adopted for the implementation of our platform. Then, we analyze and discuss all implemented system components that reside both on the cloud side and the device side.

### 4.1. Enabling technologies

#### 4.1.1. OpenStack

OpenStack[9] is a centerpiece of infrastructure cloud solutions for most commercial, in-house and hybrid deployments. It is an open-source virtual infrastructure manager that allows the management of large-scale virtualized computing, storage and network resources, by essentially playing the role of a cloud operating system. OpenStack follows a modular architecture, where each logical module is a concrete service related to the management of a specific category of virtualized resources that communicate both with each other and with external systems with well-defined and concrete RESTful APIs. As a result, the services are considered as loosely coupled and a cloud administrator is able to dynamically select an appropriate combination of them for satisfying the requirements of a specific cloud installation. In the context of our platform, we use the following OpenStack services:

- *Keystone*, which is the core OpenStack service that provides authentication, authorization and service discovery mechanisms. It is the common authentication system across the cloud operating system meaning that it is mandatory for all clients, other OpenStack services or user clients, to primarily use it in order to perform actions on the OpenStack infrastructure. The OpenStack Keystone service catalog allows clients to dynamically discover and navigate to cloud services. It supports standard username and password credentials, token-based systems and other modern authentication methods and can easily integrate with existing identity systems such as LDAP.
- *Glance*, which is the OpenStack image service. An image is a binary file that includes all requirements for running a single container, as well as metadata describing its needs and capabilities. In Glance, images are stored as templates, which are used for launching new container instances. Glance provides services to end-users such as discovering, registering, uploading and downloading the actual container images and of course their respective metadata.
- *Swift*, which is the OpenStack object storage service, which is a long-term storage system for large amounts of static data that can be retrieved and updated by users and other OpenStack services. Relevant pieces of information, such as content of documents, images, etc., as well as their metadata, are bundled together to form an object.
- *Zun*, which is the OpenStack component that acts as a container management service. Zun manages containers as OpenStack resources and provides a unified and simple RESTful API for users to interact with containers, abstracting the differences among different container technologies.
- *Barbican*, which is the secure key management service of OpenStack. Barbican exposes a RESTful API for the secure storage, provisioning and management of secrets such as passwords, encryption keys and X.509 Certificates.
- *IoTronic*, which is the IoT resource management service of OpenStack. IoTronic was initially designed and developed as the cloud-side component of the Stack4Things (S4T) framework [31], an infrastructure-oriented solution for implementing the Sensing-and-Actuation-as-a-Service paradigm [32].

[9] https://www.openstack.org

In the recent literature there exist works that combine the OpenStack Zun container service with S4T cloud architecture. In [33] the authors develop an extension of S4T for enabling container-based computing in fog nodes, while in [34] the authors combine S4T and Zun for implementing a Function-as-a-Service computing paradigm and a NodeRED-based dashboard for deploying workflow pipelines on Linux-based IoT devices. In both works, the emphasis is on executing containerized workload on the Linux-based IoT devices by hosting the container runtime and Zun components (Zun compute) on the IoT device's side. On the contrary, in our platform we employ a more typical cloud-oriented use case for combining Zun service with S4T. In particular, we deploy all Zun service components as well as the Docker runtime on the cloud side and instantiate docker containers (in essence Compileboxes) for hosting user workspaces and perform project compilations in the cloud. In addition, as the end IoT devices our platform hosts are mainly of resource-constraint nature, it is infeasible to install Docker runtime or Zun components on them.

#### *4.1.2. OMA LwM2M*

The Lightweight Machine to Machine (LwM2M) is a standard defined by the Open Mobile Alliance (OMA) [35] for remote device management in IoT applications. LwM2M is based on the Constrained Application Protocol (CoAP) [36] that makes it suitable for constrained devices, by taking into advantage CoAP packet-size optimization and its simple, stateless flow, supporting a REST API. Architecturally, LwM2M is an application layer communication protocol, which follows a client-server communication model and is developed either over User Datagram Protocol (UDP) secured with Datagram Transport Layer Security (DTLS) [37] in IP based networks, or over SMS in cellular networks.

LwM2M uses a simple data model, which relies on objects and resources defined in the specification; therefore, each LwM2M client that follows the standardized data model presents a symbolic representation of its configuration and state, which an LwM2M server can read and modify. The data model is organized in three levels, namely: (i) objects, (ii) object instances, and (iii) resources. Each object is a collection of resources representing a different data concept. For example, unique objects are specified for describing the device, the device's sensors, the security attributes or the firmware update process. Each of the standardized objects has a unique ID and can be mandatory or optional. An object can be instantiated one or multiple times, creating object instances that share the same resources. Each resource represents a well-defined concept with a unique ID, whose value may vary depending on the instance. On each resource, an LwM2M server may have: (i) Read or Write access, (ii) Read & Write access, or (iii) Execute access by referencing it through a Uniform Resource Identifier (URI) with the format "/objectID/instanceID/resourceI".

Communication between an LwM2M client and one or more LwM2M servers is feasible through four distinct interfaces defined in the standard:

- The *Bootstrap Interface* that enables a so called *LwM2M bootstrap server* to provision the LwM2M client with the necessary information to communicate with one or more LwM2M servers. The LwM2M bootstrap server cannot monitor the client's LwM2M resources but can only prepare the client for registering with the LwM2M server(s). It is noted that the use of this interface is redundant, when the device has been already pre-provisioned, e.g. during factory bootstrap.
- The *Client Registration Interface*, which describes the operations that are used by the client to inform the server about its presence and availability.

- The *Device Management and Service Enablement Interface* that defines the operations for interacting with the resources exposed by the LwM2M client. More specifically, these operations are *Read*, *Create*, *Delete*, *Write*, *Execute*, and *Discover*.
- The *Information Reporting Interface*, which is utilized by the LwM2M server to periodically receive notifications on resource value updates of a registered client.

### *4.2. Cloud-side components*

#### *4.2.1. Service Conductor*

Building a robust and secure cloud-based system raises the necessity of monitoring, managing and orchestrating the multitude of underlying heterogeneous technologies. In the heart of the cloud side of the platform resides the *Service Conductor*, which acts as a gateway between our platform and third party applications that wish to utilize the underlying provided services. Every request to the platform enters through the *Service Conductor*, which in turn verifies it, performs the corresponding actions and generates the appropriate response. *Service Conductor* is relatively easy to use, as it offers a rich RESTful API along with detailed documentation and instructions for communicating and exchanging data with it, and is an exceptionally scalable and easy to set up software component. Moreover, it is designed following a cloud-driven approach that enables installing it in multiple instances residing in different cloud computing nodes, thus providing locality and high availability. The system makes use of open-source software (e.g. Openstack, LDAP, Gitlab, etc.) that is widely used by the community for the development of a robust, scalable and easy-to-use environment.

Regarding its architecture, *Service Conductor* implements various software components called *Resource Modules* which are responsible for managing the various heterogeneous system resources that are available. When a request is received from *Service Conductor*, it is translated to serialized subsequent requests to Resource Modules, so that the necessary actions can be performed in a logical order, fulfilling the incoming request. We further describe the *Resource Modules*:

- *Identity module: Identity module* is the module, which is responsible for holding information about logical hierarchical entities on our system. These entities form the hierarchical model, which includes virtual representations of Users, Workspaces and Roles and their relevant connection graph. *Identity module* uses the *Security and Privacy Manager* to store these types of information in its internal database.
- *Container module: Container module* basically manages development and compilation projects on the platform, which are virtually represented by containers. Container technology is a modern method to package an application so that it can execute isolated from other processes. *Container module* uses the *Compilebox Orchestrator* to manage and monitor these containers and consequently is the module that controls how projects are maintained.
- *Image module: Image module* maintains the images that containers are built and based on. An image is a binary file that includes all of the requirements for running a single container, as well as metadata describing its capabilities. *Image module* securely configures, builds and stores these images as templates on the system and announces their availability. Under the hood, *Image module* appropriately exploits the *Compilebox Image Repository*.

- *Storage module: Storage module* is basically in charge of storing the compiled firmware images that are produced by the compilation process. Relevant pieces of information data concerning the binaries, as well as their metadata, are securely stored. To achieve this, *Storage module* takes advantage of the *Device Firmware Repository*.

- *Device module: Device module* interacts with the *Device Manager* and the *Device Firmware Update Server*, enabling it to operate on real devices and exploit their resources. Apart from the devices, *Device module* is able to manage device templates, which describe physical resources and compilation specifics of similar devices. Core operations include adding, removing and editing devices and templates, as well as updating devices using FOTA.

- *Version module: Version module* is responsible for tracking and providing control over changes to source code for every project on the platform, and communicates with the *Source Code Manager*.

*4.2.2. Compilation Manager*

*Compilation Manager* is the software component that was developed for creating, configuring and managing the virtual environment in which source code and corresponding firmwares are produced. It performs operations which include source code development, integration, compilation and versioning, as well as firmware storage and maintenance. Next, we discuss the individual components that comprise the *Compilation Manager* and the underlying technology used.

- *Source Code Manager*: *Source Code Manager* is responsible for maintaining source code written by users on the platform. It utilizes the installation of Gitlab which acts as the *Source Code Repository*, providing storage space for the source code files, being accessible either publicly or privately, keeping a history of revisions and versions of those files. More specifically, *Source Code Manager* allows users to control the source code versions they produce in their projects and perform operations such as committing source code to remote repository, cloning remote repository, pushing and pulling commits from remote repository, getting a history log of source code changes in time and comparing different commits with each other.

- *Compilebox Orchestrator*: *Compilebox Orchestrator* offers authorized users the means to manage their projects on the platform. Users can create, delete, list and retrieve information about their projects which are internally mapped to containers. The Zun Openstack Service is used to incorporate container technology to the platform and map internally user projects to containers, which are created from container images that reside in the *CompileBox Image Repository*. Authorized users can create, delete, enumerate, and retrieve information about images installed in the system. *Compilebox Orchestrator* is also responsible for compiling user projects. After compilation, it either stores the generated binaries in the *Device Firmware Repository* for further management using the *Compilation and Build Engine*, or in case of a failed compilation, it collects and returns compiler diagnostics to the requesting user.

- *Compilation and Build Engine*: *Compilation and Build Engine* performs operations inside the virtual isolated environment of a user project. It is a standalone command line application that is pre-installed in every container environment and is executed, on demand,

through the OpenStack Zun service, by various components of the system, in order to serve users requests. The output and the result of the actions is returned back to requesting components. For every project created, an instance of the *Compilation and Build Engine* is configured with options tailored to that project. In particular, these specific configuration options include the type of compilation environment, target device, firmware device and source code version control configuration. *Compilation and Build Engine* builds firmware according to the corresponding configuration, capturing the output of the compilation process. It is also responsible for source code version control operations (e.g. commit, push, etc.) and file operations (list directory, create/update file, etc.).

- *Compilebox Image Repository*: *Compilebox Image Respository* utilizes the Glance Openstack Service, storing information about container images used by the platform to create projects. Each of these container images offers the appropriate environment for developing and compiling firmware for a specific embedded operating system.

#### *4.2.3. Device Manager*

A centerpiece in the presented platform's architecture is the *Device Manager* component, which is responsible for storing unique device templates and information related to the IoT devices, as well as providing device discovery, device bootstrapping and service enablement. Since OpenStack was selected as the reference technology with regard to infrastructure management, we built this manager by leveraging (after major extensions) *IoTronic*[10], the OpenStack IoT resource management service. In general, IoTronic is characterized by the standard OpenStack service architecture. At its core, it consists of the *service database* (a relational database, e.g. MySQL or MariaDB) that stores all resource-related information (e.g. device templates, device unique identifiers and attributes, association with users and projects, etc.), the *service API* that exposes a REST interface for users or applications to interact with service through a web browser or a custom command-line client, and the *IoTronic Conductor* that is responsible for orchestrating the other components and dispatching remote procedure calls among them. Following the common OpenStack paradigm, all communication between the components is performed through Advanced Message Queuing Protocol (AMQP) queues, for increased scalability and reliability reasons.

It is noted that the S4T architecture follows a two-layer approach, discriminating between the *Cloud level* and the *Device level*. Communication between devices and the cloud backend is performed through the Web Application Messaging Protocol (WAMP)[11], a WebSockets (WS) subprotocol that specifies semantics for supporting the publish/subscribe and routed remote procedure call (rRPC) communication patterns. Its native support for WS technology facilitates the communication between S4T devices and the cloud backend, even behind NAT or strict firewall rules that may block all non-Web traffic. Thus, in addition to the aforementioned components, IoTronic employs also a *WAMP agent* component, which acts as a bridge between other components and the WAMP client running on the S4T devices, by translating WAMP messages to AMQP messages and vice-versa.

In our platform, contrary to the S4T architecture, we assume that the end (IoT) devices are of constrained nature and possess scarce computational, memory and storage resources. We concluded that the WAMP protocol is not a viable solution in our case, and, as a result, we

[10] https://opendev.org/x/iotronic
[11] https://wamp-proto.org

altered the southbound interface of the cloud backend, so that the LwM2M protocol is used for the communication between the cloud backend and the constrained end devices. Apart from that, several major changes where necessary in IoTronic implementation (including service database, API and WAMP agent) so that the data model and the functionality offered was enriched and adapted to the new requirements. The software components that constitute the *Device Manager* are depicted in Figure 2.

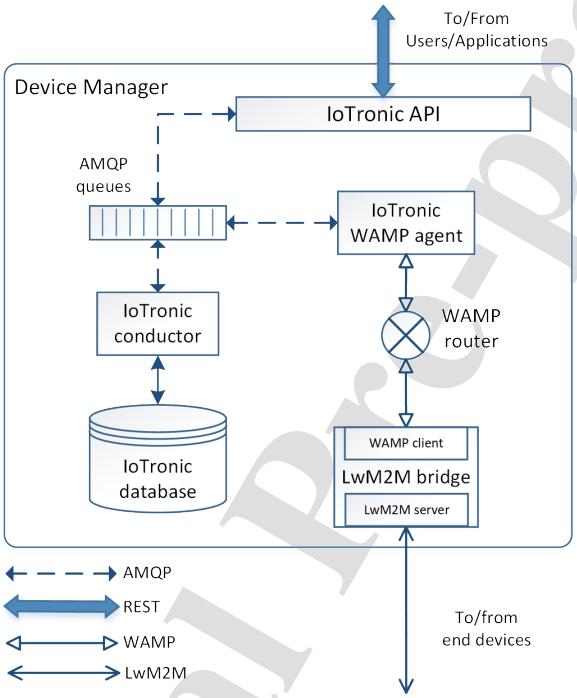



Figure 2: Device Manager software architecture

In brief, we extended the existing IoTronic functionality, as follows:

- We updated the database model in order to account for the description of the capabilities of constrained device types (named as *device templates*), supported by our platform. Information includes supported operating systems, DTLS ciphersuites, firmware binary and manifest signing algorithms (e.g. ECDSA), hardware acceleration crypto-engine and available device communication interfaces. In addition, the end device model was enriched with information related to registered device operational status, current configuration (operating system, DTLS ciphersuite, communication interface, crypto-engine, etc.) and firmware version. After the described modifications, the IoTronic database implements both the *Device Template Store* and the *Device Registry*, defined in platform's functional architecture (Figure 1).

- We added the *LwM2M bridge* component for interfacing constrained end devices through the LwM2M application protocol. The main responsibility of the LwM2M bridge is to act as a protocol and domain translator from WAMP to LwM2M, and vice versa. Thus, on one end it is connected to the WAMP router (and further to the WAMP agent) through a WAMP client, implemented in Java, through Autobahn[12] library. On the other end, it employs a LwM2M server for communicating with the end devices, implemented in Java through Eclipse Leshan[13] library. End devices communicate with the LwM2M Server over a secure channel provided by DTLS. Each LwM2M event related to end device life-cyle (registration, deregistration, status update etc.) is forwarded through the LwM2M bridge to Iotronic WAMP agent. Respectively, FOTA requests and image binaries are pushed from WAMP agent through the LwM2M server to the end devices. The LwM2M bridge implements the *Device Management* functional component, as defined in platform architecture. It is also noted, that although we could completely omit the WAMP related components of IoTronic and use the LwM2M bridge to directly bridge the end devices to the cloud backend through, e.g. AMQP, the use of WAMP may facilitate the implementation of a fog computing scenario, where fog nodes running the LwM2M bridge are installed between the constrained devices and the cloud backend.

- We extended the IoTronic API for handling requests of CRUD operations related to device templates and device instances, as well as requests for FOTA updates. The IoTronic API implements the *Device Discovery* component of the functional architecture. Additionally, the WAMP agent was suitably adapted by defining new WAMP rRPCs for LwM2M related events, as well as FOTA update steps.

#### *4.2.4. Remote Programming Manager*

*Device Firmware Repository:* The *Device Firmware Repository* (DFR) persistently stores the firmware binaries built per user's compilation request. Each produced binary stored, is accompanied by metadata, describing build options, such as device type, DTLS-related configuration (e.g. ciphersuite, ECC curve), FOTA verification configuration, etc. This metadata is of paramount importance for matching firmware available in the DFR with compatible IoT devices and thus avoiding the submission of invalid FOTA requests from the user's side. In our system, we use Swift, the OpenStack object storage service, as the implementation choice for the DFR, which is not a filesystem-based storage solution, rather it conveniently provides access to stored data through calls to a RESTful API. Among other features, Swift also provides encryption for data at rest. In this way, we guarantee the confidentiality of the stored firmware binaries, as well as their metadata.

*Device Firmware Update Server:* The *Device Firmware Update Server* (DFUS) prepares the *application image* for the FOTA update, and concomitantly is responsible for initiating, orchestrating and handling errors raised in any step of the FOTA update process. Specifically, the DFUS fetches the firmware binary from the DFR and generates and signs the manifest that is sent to the IoT device along with the raw firmware payload.

The manifest is one of the most significant parts of the firmware image because it contains all the information the devices need to validate and install the new firmware. More specifically,

[12] https://crossbar.io/autobahn
[13] https://www.eclipse.org/leshan

the devices download the manifest and firstly verify the manifest's authenticity and then analyze its fields in order to confirm that the firmware image is compatible. DFUS digitally signs the manifest using the ECDSA algorithm and the user's private key (also known as *signing key*); therefore, the receiving device can confirm the authenticity of the image. The signature verification is achieved by the device using the stored user's public key. The manifest fields are designed based on IETF instructions [38] and are summarized in Table 1.

The firmware payload can be either the full image stored in the DFR or a differential patch produced by employing a suitable differencing algorithm that extracts differences between the currently running and the new firmware. In our platform, we use Dfinder [39], a differencing algorithm for incremental FOTA updates that leverages enhanced suffix arrays and state-of-the-art construction techniques for producing (small) delta patches, thus saving energy. Dfinder supports two techniques for reconstructing the firmware by the differential image. The first technique is *out-of-place* update in which the current firmware is not erased from the device's memory, while the reconstruction of the new one takes place in a different memory address. The second technique is *in-place* update, in which the reconstruction takes place at the address where the current firmware is stored. Therefore, the *in-place* technique eliminates the usage of extra memory, while the *out-of-place* one provides the ability to roll-back to the old firmware.

In this implementation, we leverage the existing LwM2M Server as the DFUS, which, as described in Section 4.2.3, is used for the IoT device management, by utilizing the LwM2M Firmware Update object it offers. The object incorporates resources for informing the device on new firmware availability, transferring a firmware image, initiating and managing the update process, as well as performing post-update actions (e.g. reboot device).

#### *4.2.5. Security and Privacy Manager*

The *Security and Privacy Manager* is a cross-layer entity, responsible for all security-related operations of the platform, as described next.

*User identity management, authentication and authorization*: The *User Identity Manager* is responsible for providing and managing the users and the set of attributes that define a user identity, necessary for guaranteeing that only those authorized may access the platform services. In addition, it provides convenient administration interfaces for an administrator to manage the user attributes. Our system uses a token-based authentication system, where a bearer token is provided to the user after her identity is verified by submitting her credentials (username/password) to the authentication service. In terms of access control, we employ a role-based access control (RBAC) scheme. Platform resources are separated in mutually isolated tenants, in order to achieve the multi-tenancy requirement. Each user is further assigned a specific role inside a tenant that further defines her access to the tenant's resources. Our implementation utilizes the OpenStack Identity service (Keystone) with a Lightweight Directory Access Protocol (LDAP) backend for storing and managing the user directory.

*IoT device authentication and encrypted communication*: IoT device authentication is of utmost importance, so that only devices providing the necessary identification information are permitted to connect and register with our platform. In our platform, we uniquely identify each device through 128-bit Universally Unique Identifiers (UUIDs) and perform device authentication by using the DTLS protocol with Elliptic-Curve-Cryptography (ECC) ciphersuites. In addition to device authentication, DTLS offers encrypted communication (for ensuring confidentiality of data transfer between devices and cloud infrastructure) and integrity of the transmitted data.

*Device Trusted Credentials Store*: The *Device Trusted Credentials Store* (DTCS) is responsible for the secure storage of the credentials used for authenticating the IoT devices to the *Device*

Table 1: Manifest fields

| Field | Description |
|---|---|
| Manifest version ID | A unique ID which characterizes the manifest field in case multiple versions of a manifest exist. |
| Monotonic Sequence Number (MSN) | The UTC timestamp at the time of manifest generation, it is used to declare the freshness of the new image. |
| Vendor ID | Defines the device's vendor for which the new firmware is compatible with. |
| Class ID | Defines the device's class in which the new firmware is compatible with. |
| Precursor image digest | Contains the SHA-256 digest of the currently installed firmware. It is used by the device in differential updates to assure that it bears the appropriate installed firmware to apply the differential image. |
| Expiration time | The expiration time of the installed firmware. In that case, a new firmware should be installed. |
| Update type | Defines if the new firmware is a full or a differential image. |
| Reconstruction type | Declares the reconstruction technique in case of differential update. The technique can be either the in-place or out-of-place version of Dfinder differencing algorithm. |
| Storage location | Defines the startup flash address where the new firmware should be installed and boot from. |
| Reconstruction checksum | The CRC-16 reminder of the new firmware. It is used only in differential update to verify the correct reconstruction of the image. |
| Payload digest | The SHA-256 hash of the new firmware image (full or differential) in order the device to verify the integrity of the received image. |
| Payload size | The size of the new firmware image used by the device to confirm that it has enough space to install the firmware. |
| Reconstruction location | The reconstruction address for differential updates using the Dfinder algorithm's out-of-place technique. |
| Differential storage location | The flash address, where the differential image is written during download, before the application is reconstructed to the reconstruction location. |

*Manager*. The DTCS stores the necessary DTLS credentials (e.g. Pre-shared Keys, Raw Public Keys) depending on the ciphersuite(s) supported/used by each IoT device. The DTCS is currently implemented through Barbican, the OpenStack Secure Key Management service. IoT device credentials are automatically generated during the device creation process, based on the supported ciphersuites and user selection; then, the *IoTronic Conductor* stores the newly created credentials in a suitable Barbican container, through the RESTful API offered by the later. The credentials are retrieved and verified whenever an IoT device tries to authenticate to the *Device Manager*.

### 4.3. IoT device-side components

In this section, we present the architecture of FOTA functionality from the IoT device side. As mentioned before, the FOTA implementation is based on the corresponding Firmware Update object defined by the LwM2M protocol. Therefore, an IoT End Device (ED) should be initially prepared to support over-the-air updates using this object and then, exploit it to successfully complete the FOTA process. In the following paragraphs, we describe how the device preparation is performed using the base image (often referred as golden image). Moreover, we describe in details the over-the-air procedure for installing the new firmware image.

#### 4.3.1. Device preparation

Before the field deployment of an ED, the base firmware image has to be installed via USB, and it consists of three components, namely: (i) the *Firmware Installer*, (ii) the *Device Metadata*, and (iii) the *Application Image*.

The *Firmware Installer* (FI) is a stand-alone application for the different supported operating systems, designed to connect to the DFUS using the necessary components of the LwM2M client. FI is located at the beginning of the flash memory address; therefore, it boots, upon DFUS's request, each time a new firmware update is available for the ED. More specifically, FI securely downloads (over DTLS), verifies, and installs the firmware to the ED using the LwM2M firmware object.

The *Device metadata*, shown in in Table 2, contain device specific information that is essential for device management and the FOTA process. Particularly, the metadata is stored in the devices during the installation of the base image and is used for validating the manifest fields during the reception of the new firmware. This way, the EDs can verify the compatibility of the transmitted firmware. The *DTLS credentials*, *Firmware verification key*, *Vendor ID*, and *Class ID* are predefined for each device model and never change, while the *MSN*, *Firmware digest*, *Firmware expiration time*, and *Storage location* must be updated each time a new firmware is installed.

The *application image* refers to the firmware currently running in the device, and is stored in a pre-defined flash address space, contained in the *Storage location* field of the manifest. The device executes this firmware until an update request is sent by DFUS. Consequently, this firmware must always contain the LwM2M functionality in order to connect to the DFUS and expose the predetermined LwM2M resource for booting to FI in order to download the new firmware.

#### 4.3.2. FOTA image installation

Regarding the FOTA image installation, whenever a new firmware image is available, DFUS informs an ED accordingly. More specifically, DFUS uses a pre-defined LwM2M resource of LwM2M client to request ED to stop the execution of the currently installed firmware and boot the FI, and then, DFUS starts the transmission of the manifest. On the ED side, FI uses the public key stored in the metadata to verify the manifest's signature in order to validate its integrity and authenticity. If the signature is invalid, ED cancels the firmware update process and boots the existing (current) firmware. If the signature is valid, FI attempts to validate the manifest fields (i.e *MSN, Vendor ID*, etc.) against the corresponding metadata ones. As with the invalid signature case, if at least one of the manifest fields is not validated, the device abandons the firmware update process and uses the current firmware. On the other hand, if all manifest fields are valid, ED downloads the remaining image payload.

Table 2: Metadata fields

| Field | Description |
|---|---|
| DTLS credentials | Used by the ED to register to the DFUS using the DTLS protocol. For authentication based on the Pre-Shared Key (PSK), the metadata includes the paired identity and shared secret. On the other hand, for authentication based on the Raw Public Key (RPK), this includes the pair of the ED's private and public key. |
| Firmware verification key | The public key of the owner (user of the platform) of the ED and it is used to verify the integrity and authenticity of the new update image before downloading it. |
| Monotonic Sequence Number (MSN) | Used to ensure the image freshness. This denotes that the new (latest) application must have greater MSN than the installed (current) one |
| Vendor ID<br>Class ID | The Vendor ID and Class ID describe the class and family of the ED. These fields are important in order to avoid the installation of a firmware, which is not compatible to this family of ED |
| Firmware digest | The SHA-256 hash of the current installed *application image*. It is significant for differential updates as it guarantees that the differential update will be applied to the correct version of the *application image*. |
| Firmware expiration time | Describes the date after the device must reject the installed firmware and request a new one to be installed (by booting to the FI). |
| Storage location | Defines the boot address of the current *application image*. |

The image payload can be full or differential (generated by the Dfinder algorithm) and it is specified by the manifest's *Update type* field. In the first case, the FI writes the received payload (i.e. the entire firmware) to the device's flash starting from the address defined by the manifest's *storage location* field. In the latter case, the FI writes the received payload (i.e. a part of the firmware) to the device's flash starting from the address defined by the manifest's *differential storage location*, and subsequently use it to reconstruct the new firmware based on the already installed one and the selected reconstruction type (in-place or out-of-place).

## 5. A web-based user frontend as a third party application

In this section, we present the main functionalities of a web-based frontend, which is deployed as a third-party application and allows authorized users to take advantage of compilation-as-a-service and remote-programming-as-service functionalities, offered by the proposed platform. This system uses the PHP framework Laravel[14] for web development, the PostgreSQL for storing data, JSTree[15] for file management, Laravel's Fortify[16] for the two-factor authentication, and communicates with the platform through the Service Conductor (Section 4.2.1) and a well-defined REST API.

[14] https://laravel.com
[15] https://www.jstree.com
[16] https://laravel.com/docs/9.x/fortify

### 5.1. User dashboard

As soon as a user authenticates to the system, the dashboard page appears (Figure 3), which offers a current and convenient summary of several key resources of the system, in an intuitive and user-friendly form. In particular, the dashboard page summarizes: (i) the total number of workspaces (projects for firmware development) created by the user, (ii) the total number of user IoT devices registered to the platform, (iii) the total number of device firmware images built and stored by the user, and (iv) the total number of successful FOTA updates performed by the user. In addition, the dashboard illustrates the quotas and quota usage of the four aforementioned resources, according to the customer plan assigned to the authenticated user. By clicking on each widget displaying a resource quota and usage (except for performed FOTA updates), the user lands on the corresponding resource page, described in the next subsections.

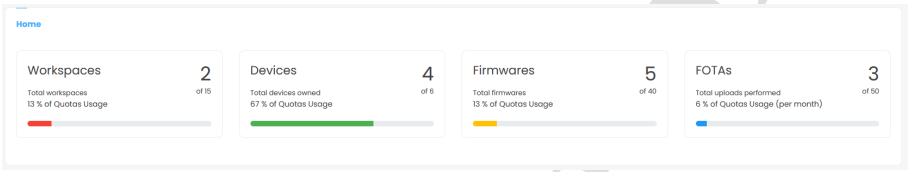



Figure 3: User dashboard

### 5.2. IoT device management

The IoT device management interface (IDM) allows a user to register his IoT devices to the platform. This is a prerequisite in order to use the offered services (compilation, FOTA). The device registration process consists of two phases: (i) the user adds a new device specifying various details through a web form, such as the device name, device serial number, device type (i.e. Launchpad CC2650), operating system (i.e. Contiki-NG), connectivity (i.e. IEEE 802.15.4 2.4GHz), cipher suite (i.e. TLS_PSK_WITH_AES_128_CCM_8), etc., (ii) the user requests the creation of the Base image, which is further downloaded from the platform and installed (manually) in the device for the actual registration to take place (see Section 4.2.3). As soon as a device is added to the platform, IDM displays several useful information (Figure 4): (i) device name, (ii) firmware (if any) installed in the device, (iii) the type of the device, (iv) the device vendor, (v) the operating system (OS), (vi) OS version, (vii) device status (initialized, registered, deregistered, updating), and other information regarding the time/date of the registration, etc. The FOTA process for a specific device is initiated using this interface, and the user selects the firmware to upload to the device through a list that contains only the compatible (to this device) firmware images.

### 5.3. Text editor

The text editor (TE) is based on CodeMirror[17] and is one of the most important components of the frontend, as it allows the user to: (i) create new workspaces, (ii) edit and save code in the platform, and (iii) compile the code and receive informative output, i.e. successful compilation, otherwise, a list with the errors. Regarding compilation, the user has two options: (i) compile

[17] https://codemirror.net

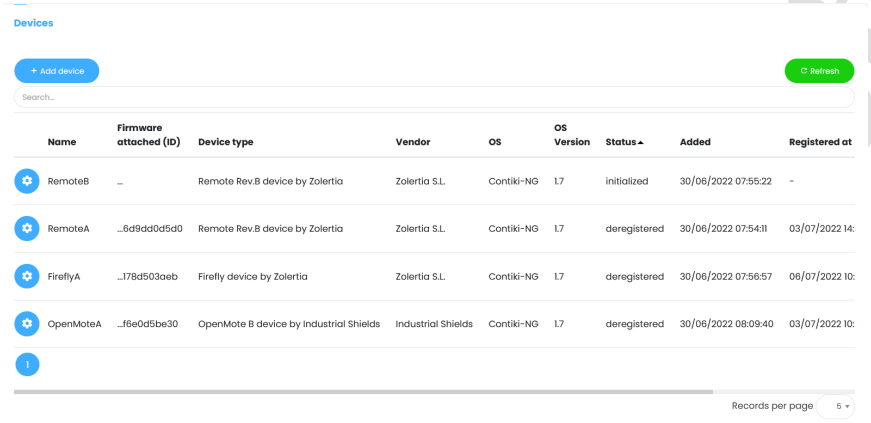



Figure 4: Added IoT devices

the code and correct any potential errors, and (ii) compile the code, and if successful, save the produced firmware image in the platform for later use (see next section). Figure 5 shows an example with code compiled for the Contiki-NG OS, and the compilation output after an error has been inserted in the code deliberately.


Figure 5: Text editor for code writing and compilation

### 5.4. *Firmware management*

As referred before, the user has the option to store the successfully compiled firmware images in the platform. Several information are provided such as (Figure 6): the firmware name,

the workspace is related to, an editable description text, if the firmware is FOTA-enabled (e.g. contains the bootloader to download a new firmware), the target device type, OS, size, date/time of creation, etc.

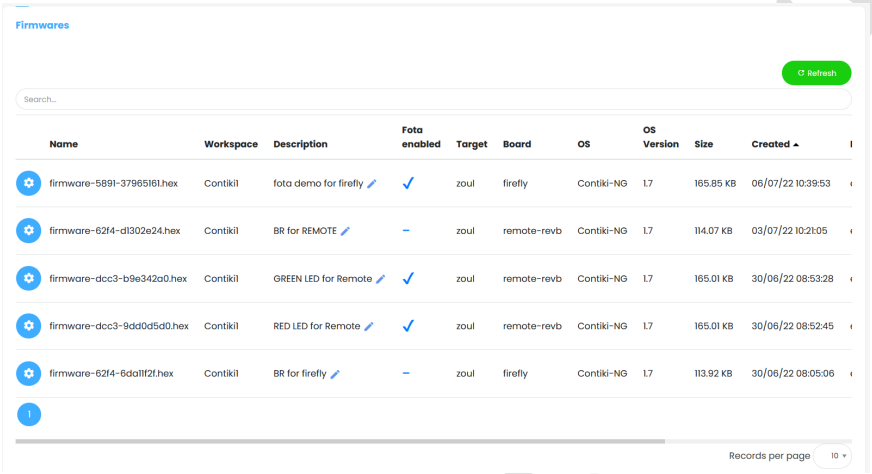



Figure 6: Firmware management

## 6. Performance evaluation

### *6.1. Cloud platform evaluation*

With the intended goal of acting as a multi-tenant solution offering jointly compilation and remote programming services for resource-constraint IoT devices, the cloud platform presented in this work, needs to efficiently handle a substantially large number of concurrent requests submitted by users. Therefore, in this section, we describe the methodology for evaluating platform' s performance under variable workload, and further present and discuss the results of the evaluation.

#### *6.1.1. Testbed and deployment*

Our testbed essentially comprises two distinct entities, namely: (i) the user infrastructure and (ii) the cloud infrastructure. The user infrastructure consists of the *Virtual End Devices* (VEDs) and the *Load Generator* (LG). The VEDs emulate the behavior of real IoT end devices, in terms of connectivity to the cloud-backend, service enablement and FOTA update capabilities. We implement the VEDs through a highly scalable standalone Java application, based on the Eclipse Californium (CoAP) and Eclipse Leshan (LwM2M) libraries.

Furthermore, the LG is responsible for generating configurable workload to the cloud-backend and collecting appropriate performance metrics. The LG is implemented as a scalable standalone Python application that leverages the *asyncio* Python framework for generating a variable number of asynchronous concurrent requests to the Service Conductor RESTful API.

On the other side, the cloud infrastructure hosts a complete deployment of our platform, including the *Service Conductor*, the *Device Manager*, the *Device Firmware Update Server*, the *Device Trusted Credentials Store* and the *Device Firmware Repository*. In this deployment, we used *Devstack*[18] tool (Ussuri version), which is a series of extensible scripts for conveniently building the necessary OpenStack environment. We note that this is a Proof-of-Concept (PoC) deployment, as it is based on a single-node installation of the OpenStack environment. Although our system's performance can benefit from deployment in a production-ready highly distributed cloud environment, we use the current setup for obtaining an initial baseline evaluation.

The user infrastructure and the cloud infrastructure have been deployed in two separate virtual machines (VMs), each one hosted on a unique workstation. The characteristics of each workstation (physical machine) and the corresponding VMs are as follows:

- User infrastructure:
  - **Physical machine:** 8 CPUs (Intel® Core™ i7-4790K CPU @ 4.00GHz), 16GB RAM, 256GB SSD.
  - **Virtual machine:** 2 vCPUs, 8GB RAM, 40GB disk, Ubuntu 20.04 LTS operating system.
- Cloud infrastructure:
  - **Physical machine:** 24 CPUs (Intel® Xeon® E5-2630 v2 @ 2.60GHz), 80GB RAM, 256GB SSD, 1TB HDD.
  - **Virtual Machine:** 4-8 vCPUs, 32GB RAM, 100GB disk, Ubuntu 18.04 LTS operating system.

From a performance point of view, we enabled CPU pinning (tying virtual CPUs to real physical CPUs of the host) for the VM hosting the cloud infrastructure, with KVM hypervisor set in "host-passthrough" CPU mode. Preliminary experimentation showed that this configuration offers substantial performance gains, compared to different configuration choices. In addition, we tuned the operating system limits so that our measurements reflect the capabilities of the applications, instead of the limits enforced by the operating system. Thus, the user limits for file size, max memory and CPU time were set to unlimited, and the open files limit was increased to 64000.

#### *6.1.2. Test plan*

In order to quantify the performance of our platform when handling different workload levels, we orchestrate two sets of experiments; the first one for evaluating the Golden Image (GI)[19] building task, and the second one for evaluating the FOTA update task. In both cases, we use the LG application for generating a varying number of concurrent requests towards the corresponding endpoints of the *Service Conductor* API. In addition to that, during the latest set of experiments (FOTA update task), we vary: (i) the number of VEDs connected to the platform for receiving FOTA updates, and (ii) the size of the firmware image sent over-the-air. It is noted once

[18] https://docs.openstack.org/devstack/latest

[19] GI is the base firmware image created for an IoT device in order to be registered to the platform, and is installed manually through the USB interface

again that the VEDs used in the FOTA update task emulate the behavior of real IoT end devices, in terms of connectivity to the cloud-backend, service enablement and FOTA update capabilities. In essence, the VEDs application incorporates multiple instances of a Leshan LwM2M client (equal to the number of VEDs used in each experimental point) that implements all necessary LwM2M interfaces (Registration, Device Management/Service Enablement and Information Reporting) and exposes, among others, the LwM2M Firmware object for controlling the FOTA update process, by following the corresponding Firmware Update State Machine described in [35]. In addition, it provides firmware integrity and authenticity verification by verifying the manifest's signature and the firmware image digest. After successful downloading and verification of the new firmware image, the client (representing a single VED) reboots, and the new firmware is executed.

In order to assess the impact of the cloud infrastructure resources on the performance, we also vary the number of vCPUs allocated to the corresponding VM. Table 3 summarizes the parameters used in our experiments.

Table 3: Experimental parameters

| Parameter | Values |
|---|---|
| Number of concurrent GI build requests | 2 – 14 |
| Number of concurrent FOTA update requests (equals number of VEDs) | 10 – 100 |
| Firmware image size | 128KB, 256KB, 512KB |
| Number of vCPUs allocated to cloud infrastructure VM | 4, 6, 8 |

Each of the two tasks under evaluation consists of several steps. More specifically, the GI building task includes the following steps: (i) credentials retrieval from the *Device Trusted Credentials Store*, (ii) creation and initialization of build environment, (iii) source code compilation and GI build, (iv) credentials embedding in GI, and (v) GI firmware storage in Device Firmware Repository. Accordingly, the FOTA update task embodies the following steps: (i) firmware image retrieval from Device Firmware Repository, (ii) manifest generation and payload construction (by concatenating the manifest and the firmware image), (iii) pushing payload to VED, and (iv) verifying, loading and executing the new image in VED.

We consider the following metrics for evaluating the performance of the platform: (i) *task execution time* (TET), defined as the total time elapsed between request submission and the requested task completion (GI build TET or FOTA update TET), and (ii) *resource utilization* of cloud infrastructure VM (CPU and active memory). Note that in terms of resource utilization, we report system-wide CPU and memory usage during the execution of each task and compare it with the respective utilization when the system is in idle status, namely none of the tasks under evaluation executes. We choose to report system-wide metrics because we prefer to acquire a general overview of our system's performance, instead of evaluating each of the several components separately.

In our experiments, we use the OpenStack Ceilometer telemetry service and the OpenStack Panko event storage service, which offer a convenient and fully integrated with our environment means for collecting and retrieving the necessary metrics. In particular, we use the RabbitMQ-based event collector of Ceilometer for collecting and processing event notifications generated at the beginning and the end of each task by the *Service Conductor*. The processed events are

stored in a MongoDB-based event storage database, where they are later retrieved from, through OpenStack Panko REST API for further analysis.

*6.1.3. Results*

We now present the outcome of our performance evaluation, which analyzes the behavior of the presented platform under different workload levels. We execute 10 Monte Carlo runs for each experimental scenario and report average values of the metrics defined in Section 6.1.2, for both tasks, along with the standard error of the mean (in the form of error bars).

*GI build task.* In this first set of experiments, we evaluate platform's performance during the GI build task. Figure 7 illustrates the TET for different number of concurrent GI build requests. As expected, the average TET increases as more requests have to be served by the system. For any number of vCPUs, the increase is less than proportional to the increase of the load. Nevertheless, this increase is more profound for a 4 vCPUs VM, where the average TET for serving 14 concurrent requests is almost 5x larger than TET for serving 2 concurrent requests (510s and 115s, respectively), compared to a less than 3x corresponding increase for 6 and 8 vCPUs. By comparing the TET between different vCPUs allocation choices for a given number of concurrent requests, we notice that in general there exist performance gains with increasing number of vCPUs, especially for a larger number of concurrent requests. For example, for 14 concurrent requests, we observe an almost 40% and 50% decrease in TET, when, instead of 4, we use 6 and 8 vCPUs, respectively.

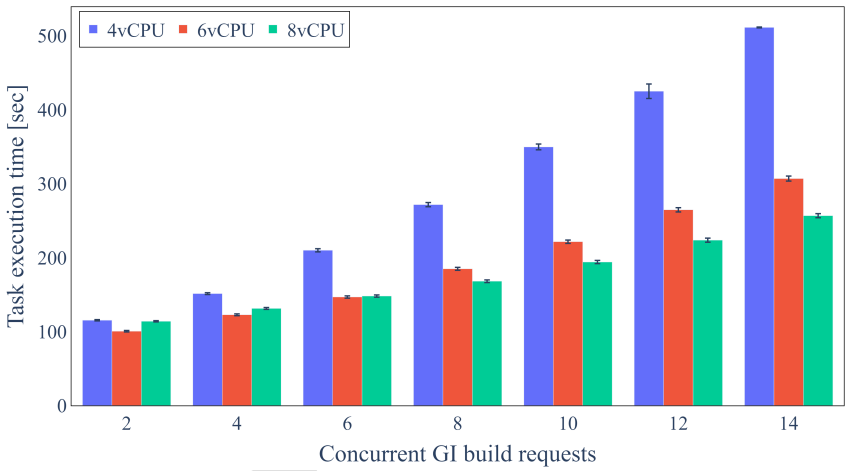



Figure 7: Execution time for GI build task

We display the average CPU utilization and active memory of cloud infrastructure VM for the GI task in Figure 8 and Figure 9, respectively. We also report CPU utilization and active memory at *idle state* that corresponds to zero value of the horizontal axis. Note that as "idle state", we define the state in which no GI build requests are served by the system.

The average CPU utilization, which is no more than 12% at idle state, quickly ramps up when the system serves concurrent requests. This can be mainly attributed to the source code compilation phase, which is particularly CPU-intensive. Observe that CPU utilization for 4 vCPUs rapidly reaches values higher than 90%, indicating an exhaustion of available computing resources, which further explains the high increase rate of TET, shown in Figure 7. For the other two values of vCPUs, the CPU utilization increases closer to linear with load, demonstrating the scalability of our system, in case there are enough computational resources.

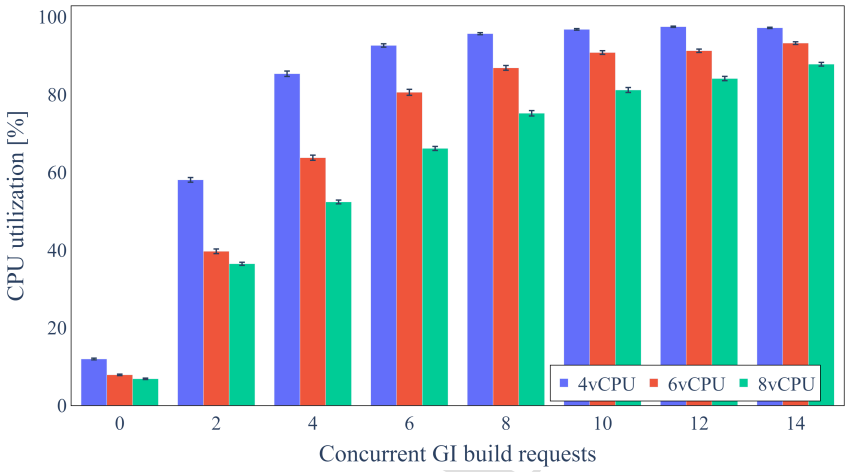



Figure 8: CPU utilization during GI build task

Finally, we perceive no substantial increase in memory usage as we increase the load to our system. In particular, we observe at most an increase of almost 15% from idle state to 14 concurrent requests for all values of allocated vCPUs.

*FOTA update task.* Next, we performed a set of experiments for evaluating the FOTA update task. The average TET under an increasing system load (concurrent FOTA update requests) and three different image sizes (128KB, 256KB, 512KB) is presented in Figure 10.

As expected, we notice an increase in average TET for all scenarios, as the number of concurrent requests increases. Similar to the GI build task, this increase is less than proportional to the increase in the load. For example, in case of 10 concurrent requests and 4 vCPUs, the FOTA update TET is 12s (128KB), 16s (256KB), and 22s (512KB), while for 100 concurrent requests the FOTA update TET raises to 76s (128KB), 89s (256KB), and 112s (512KB), respectively. For all firmware image sizes and number of concurrent requests we observe a decrease in average TET, as the number of vCPUs increases. In general, we again notice a more profound drop in TET when allocating 6 instead of 4 vCPUs, compared to the drop observed when transitioning from 6 to 8 vCPUs. This behaviour implies a high pressure to computing resources when 4 vCPUs are used, which is relieved when adding more vCPUs.

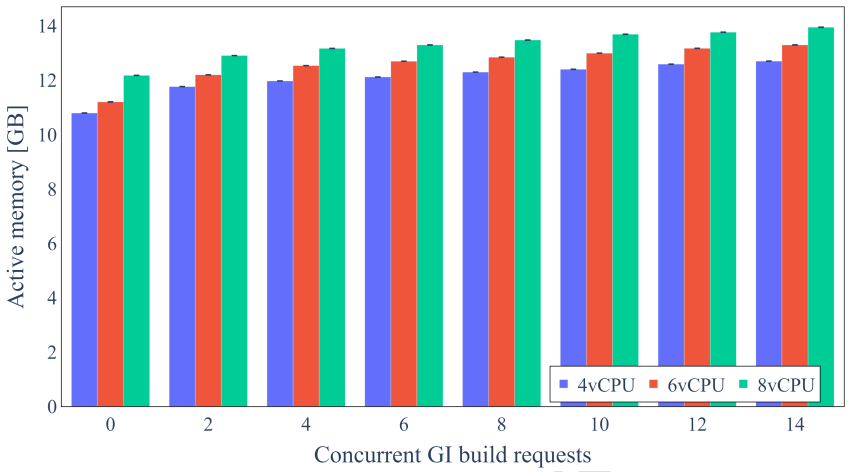



Figure 9: Active memory during GI build task

Figure 11, which depicts CPU utilization during FOTA update task, supports our aforementioned hypothesis. CPU utilization for 4 vCPUs quickly ramps up over 90% for more than 50 concurrent requests and an image size of 128KB or 256KB. On the contrary, CPU utilization is at most 86% and 70% when allocating 6 and 8 vCPUs, respectively. Our system demonstrates a scalable behaviour in terms of the FOTA update task, in the sense that an increasing system load can be handled only by adding more computational resources, without implying greater complexity or system redesign. Of course, we should keep in mind that this is a PoC platform deployment and further performance gains are feasible by employing a production-ready deployment, by e.g. deploying OpenStack cloud environment in a highly distributed fashion.

Finally, Figure 12 displays the active memory of cloud infrastructure VM during all experimental scenarios for the FOTA update task. Active memory is 13.2GB, 13.6GB and 14.6GB at idle state for 4, 6 and 8 allocated vCPUs, respectively. The largest value of active memory is observed for 100 concurrent requests and an image size of 512KB, reaching 18.7GB, 18.3GB and 19.1GB, respectively.

### *6.2. IoT devices overhead*

Here, we present the IoT device-side evaluation results in terms of memory utilisation, and time to verify the cryptographic signature during the FOTA process.

#### *6.2.1. Memory overhead*

When code is compiled for an IoT device, files (firmwares) in hexadecimal (*.hex*) and binary (*.bin*) formats are generated, which are further loaded into the device. Another type of files produced are the *ELF* ones that contain several useful information about the various memory segments present in the corresponding firmware file. An *ELF* files can provide information, among others, about the following segments: (i) *.text* that contains the executable code, (ii) *.data*

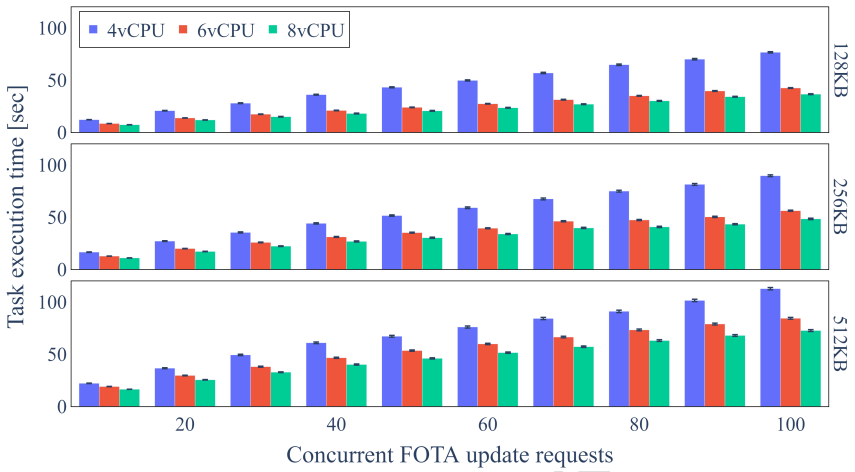



Figure 10: Execution time for FOTA update task

that contains the initialized static variables, and (iii) *bss*, which includes all static data. The size of the firmware depends mainly on the size of these segments (for the devices used later in the evaluation), so the smaller they are, the smaller the firmware size.

In order to demonstrate the memory overhead of the proposed platform for the IoT devices, we adopted the following approach:

- We used three different IoT devices: Firefly[20], Openmote B[21], and Remote[22]
- We selected the Contiki-NG operating system [40]
- We considered three cases: (i) Non FOTA-enabled/no DTLS, where the firmware produced does not contain any code to interact with the platform, (ii) Non FOTA-enabled/DTLS, where the firmware produced does not contain any code to interact with the platform, however, DTLS code is present, and (iii) FOTA-enabled, where the firmware contains code to interact with platform (FOTA, device registration, DTLS-based mechanisms, etc., are enabled). (iv) We used two applications and their corresponding firmwares: (i) COAP client, and (ii) COAP server. (v) When DTLS is enabled (Non FOTA-enabled/DTLS, FOTA-enabled), COAP client and server communicate over a secure channel with data encrypted in the transport layer. Moreover, two DTLS modes are used, which are suitable for IoT constrained devices: (i) *PreShared Key* (PSK) [41], where the symmetric encryption key used is derived using a secret shared by the two peers (client, server), and (ii) *Raw Public Key* (RPK) [42], where raw public keys are used along with a key agreement protocol (e.g. ECDH) for the encryption key sharing.

[20] https://zolertia.io/product/firefly
[21] https://www.industrialshields.com/shop/product/is-omb-001-openmote-b-721#attr=
[22] https://zolertia.io/product/re-mote

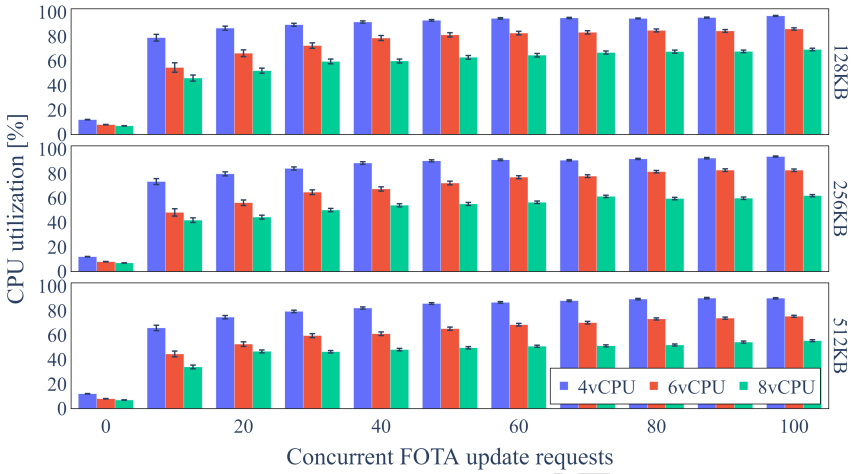



Figure 11: CPU utilization during FOTA update task

In Figure 13, we present the memory segments when compiling the COAP client application for the cases described previously. For all devices, The *.data* segment is mostly affected when DTLS or FOTA are enabled, with an overhead of less than 10Kbytes for the cases Non FOTA-enabled/DTLS, FOTA-enabled. This is true for both DTLS modes (PRK, RPK), while the RPK has a higher memory utilisation as more types of DTLS packets are exchanged during the handshake process, so code is longer. We observe similar results, shown in Figure 14, for the COAP server.

### *6.2.2. Signature verification overhead*

As we have mentioned in Section 4.2.4, during the FOTA update process, the DFUS digitally signs the manifest using the ECDSA algorithm and the user's signing key and then, transmits it to the IoT end device. On the other side, the device has to validate the manifest's authenticity using the ECDSA algorithm. The signature verification is an expensive process, especially if implemented in software. For this reason, we have designed the verification process in such a way that the device's hardware accelerator is used when available. In Table 4, we present the average signature verification time for the Zolertia Remote device using software or the hardware cryptographic accelerator.

Table 4: Signature verification time

| Method | Time |
|---|---|
| Software | 20.931 sec |
| Hardware | 698516 sec |

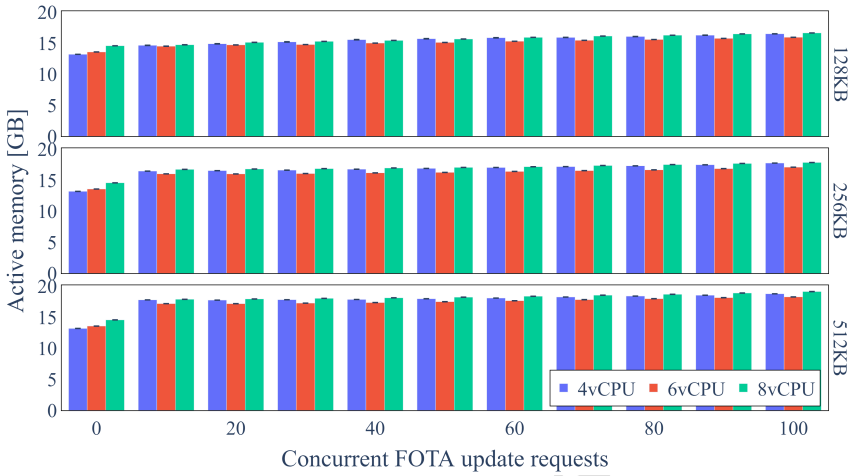



Figure 12: Active memory during FOTA update task

## 7. Conclusion - Future work

In this article, we introduced a flexible Compilation-as-a-Service and Remote-Programming-as-a-Service platform that jointly offers cloud-based compilation and Firmware-Over-The-Air update functionalities for deployed IoT devices. We presented the system architecture and thoroughly described implementation details of our platform. Furthermore, we introduced a web-based user frontend, deployed to act as a third party application that consumes the services offered. Later, we evaluated a Proof-of-Concept deployment of our system in terms of task execution time and resource utilization for two different task types, namely golden image build and FOTA update. The evaluation showed that our system enjoys scalability, provided that sufficient computational and memory resources are available. Future work includes: (i) re-evaluating the platform in terms of the selected tasks in a production-ready highly scalable deployment scenario (e.g. in a private cloud) and (ii) assessing the performance of the platform during FOTA update task in an extended testbed of real IoT device fleets.

## Acknowledgments

This research has been co-financed by the European Regional Development Fund of the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH – CREATE – INNOVATE (project code: T1EDK-03389).

## References

[1] J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami, Internet of things (iot): A vision, architectural elements, and future directions, Future Generation Computer Systems 29 (7) (2013) 1645–1660.

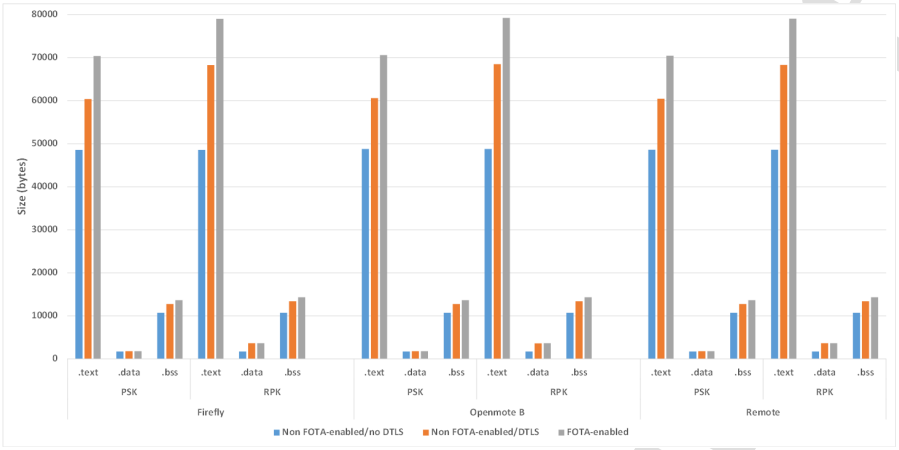



Figure 13: COAP client memory segments after code compilation for the various IoT devices


[2] W. Ejaz, M. Naeem, A. Shahid, A. Anpalagan, M. Jo, Efficient energy management for the internet of things in smart cities, IEEE Communications Magazine 55 (1) (2017) 84–91. doi:10.1109/MCOM.2017.1600218CM.

[3] A. H. Alavi, P. Jiao, W. G. Buttlar, N. Lajnef, Internet of things-enabled smart cities: State-of-the-art and future trends, Measurement 129 (2018) 589–606. doi:https://doi.org/10.1016/j.measurement.2018.07.067. URL https://www.sciencedirect.com/science/article/pii/S0263224118306912

[4] H. Habibzadeh, K. Dinesh, O. R. Shishvan, A. Boggio-Dandry, G. Sharma, T. Soyata, A survey of healthcare internet of things (hiot): A clinical perspective, IEEE Internet of Things Journal 7 (1) (2019) 53–71.

[5] Y. A. Qadri, A. Nauman, Y. B. Zikria, A. V. Vasilakos, S. W. Kim, The future of healthcare internet of things: A survey of emerging technologies, IEEE Communications Surveys Tutorials 22 (2) (2020) 1121–1167. doi:10.1109/COMST.2020.2973314.

[6] H. Xu, W. Yu, D. Griffith, N. Golmie, A survey on industrial internet of things: A cyber-physical systems perspective, IEEE Access 6 (2018) 78238–78259. doi:10.1109/ACCESS.2018.2884906.

[7] C. Yang, W. Shen, X. Wang, The internet of things in manufacturing: Key issues and potential applications, IEEE Systems, Man, and Cybernetics Magazine 4 (1) (2018) 6–15.

[8] J. Lin, W. Yu, N. Zhang, X. Yang, H. Zhang, W. Zhao, A survey on Internet of Things: Architecture, enabling technologies, security and privacy, and applications, IEEE Internet of Things Journal 4 (2017) 1125–1142.

[9] A. Botta, W. De Donato, V. Persico, A. Pescapé, Integration of Cloud computing and Internet of Things: A survey, Future Generation Computer Systems (2016). doi:10.1016/j.future.2015.09.021.

[10] K. Arakadakis, P. Charalampidis, A. Makrogiannakis, A. Fragkiadakis, Firmware over-the-air programming techniques for iot networks-a survey, ACM Computing Surveys (CSUR) 54 (9) (2021) 1–36.

[11] J. Bauwens, P. Ruckebusch, S. Giannoulis, I. Moerman, E. D. Poorter, Over-the-air software updates in the internet of things: An overview of key principles, IEEE Communications Magazine 58 (2) (2020) 35–41. doi:10.1109/MCOM.001.1900125.

[12] A. Rayes, S. Salam, Internet of Things From Hype to Reality: The Road to Digitization.

[13] Google Cloud IoT Core, Accessed: 2021-09-13. URL https://cloud.google.com/iot-core

[14] Amazon Web Services IoT Core, Accessed: 2021-09-13. URL https://aws.amazon.com/iot-core

[15] IBM Watson IoT Platform, Accessed: 2021-09-13. URL https://www.ibm.com/cloud/watson-iot-platform

[16] J. Kim, J.-W. Lee, OpenIoT: An open service framework for the Internet of Things, in: 2014 IEEE World Forum on Internet of Things (WF-IoT), 2014, pp. 89–93. doi:10.1109/WF-IoT.2014.6803126.

[17] H. C. Pöhls, V. Angelakis, S. Suppan, K. Fischer, G. Oikonomou, E. Z. Tragos, R. D. Rodriguez, T. Mouroutis, Rerum: Building a reliable iot upon privacy- and security- enabled smart objects, in: 2014 IEEE Wireless Commu-

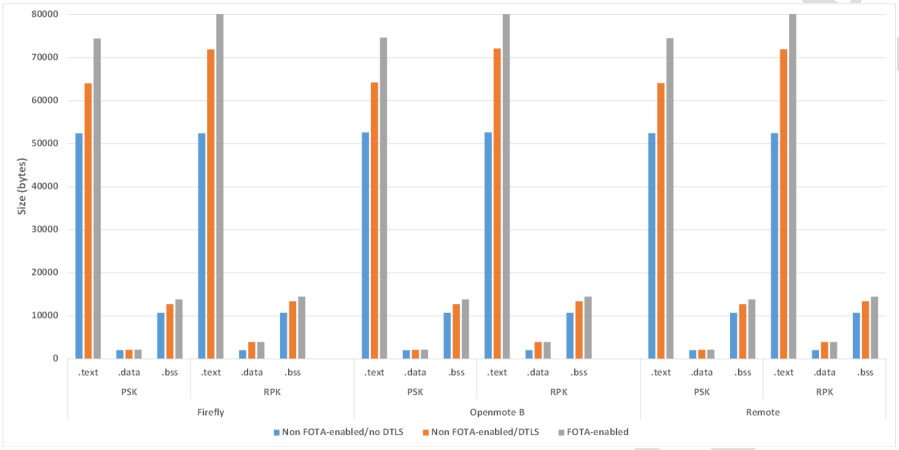



Figure 14: COAP server memory segments after code compilation for the various IoT devices

nications and Networking Conference Workshops (WCNCW), 2014, pp. 122–127. doi:10.1109/WCNCW.2014.6934872.

[18] P. Agarwal, M. Alam, Open service platforms for iot, in: Internet of Things (IoT), Springer, 2020, pp. 43–59.

[19] A. Souri, A. Hussien, M. Hoseyninezhad, M. Norouzi, A systematic review of iot communication strategies for an efficient smart environment, Transactions on Emerging Telecommunications Technologies 33 (3) (2022) e3736. doi:https://doi.org/10.1002/ett.3736.

[20] J. Kim, Y. Jeon, H. Kim, The intelligent iot common service platform architecture and service implementation, The Journal of Supercomputing 74 (9) (2018) 4242–4260.

[21] Aamir Nizam Ansari, S. Patil, A. Navada, A. Peshave, V. Borole, Online C/C++ compiler using cloud computing, in: 2011 International Conference on Multimedia Technology, 2011, pp. 3591–3594.

[22] M. Busch, M. Protsenko, T. Müller, A Cloud-Based Compilation and Hardening Platform for Android Apps, in: Proceedings of the 12th International Conference on Availability, Reliability and Security, 2017.

[23] P. Zhang, Y. Liu, M. Qiu, SNC: A Cloud Service Platform for Symbolic-Numeric Computation Using Just-In-Time Compilation, IEEE Transactions on Cloud Computing (2019).

[24] Mender - Open source over-the-air software updates for Linux devices, Accessed: 2021-09-13. URL https://mender.io/

[25] Yocto Project, Accessed: 2021-09-13. URL https://www.yoctoproject.org/

[26] Balena - The complete IoT fleet management platform, Accessed: 2021-09-13. URL https://www.balena.io

[27] ARM Pelion IoT Platform, Accessed: 2021-09-13. URL https://www.pelion.com

[28] Particle, Accessed: 2021-09-13. URL https://www.particle.io/

[29] FreeRTOS - Real-time operating system for microcontrollers, Accessed 2021-09-13. URL https://aws.amazon.com/freertos/

[30] P. Charalampidis, A. Fragkiadakis, A compilation-and remote-programming-as-a-service platform for iot devices, in: 2021 IEEE International Conference on Joint Cloud Computing (JCC), 2021, pp. 80–85. doi:10.1109/JCC53141.2021.00025.

[31] F. Longo, D. Bruneo, S. Distefano, G. Merlino, A. Puliafito, Stack4Things: An OpenStack-based framework for IoT, in: 2015 3rd International Conference on Future Internet of Things and Cloud, IEEE, 2015, pp. 204–211.

[32] F. Longo, D. Bruneo, S. Distefano, G. Merlino, A. Puliafito, Stack4things: a sensing-and-actuation-as-a-service framework for iot and cloud integration, Annals of Telecommunications 72 (1) (2017) 53–70.

[33] Z. Benomar, F. Longo, G. Merlino, A. Puliafito, Enabling container-based fog computing with openstack, in:

2019 International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), 2019, pp. 1049–1056. doi:10.1109/iThings/GreenCom/CPSCom/SmartData.2019.00181.

[34] G. Tricomi, Z. Benomar, F. Aragona, G. Merlino, F. Longo, A. Puliafito, A NodeRED-based dashboard to deploy pipelines on top of IoT infrastructure, in: 2020 IEEE International Conference on Smart Computing (SMARTCOMP), 2020, pp. 122–129. doi:10.1109/SMARTCOMP50058.2020.00036.

[35] O. M. Alliance, Lightweight machine to machine technical specification: Core (2018).

[36] Z. Shelby, K. Hartke, C. Bormann, The Constrained Application Protocol (CoAP), RFC 7252 (Jun. 2014). doi:10.17487/RFC7252.
URL https://www.rfc-editor.org/info/rfc7252

[37] E. Rescorla, N. Modadugu, Datagram Transport Layer Security, RFC 4347 (Apr. 2006). doi:10.17487/RFC4347.
URL https://www.rfc-editor.org/info/rfc4347

[38] B. Moran, H. Tschofenig, H. Birkholz, A Manifest Information Model for Firmware Updates in Internet of Things (IoT) Devices, RFC 9124 (Jan. 2022). doi:10.17487/RFC9124.
URL https://www.rfc-editor.org/info/rfc9124

[39] K. Arakadakis, N. Karamolegkos, A. Fragkiadakis, Dfinder—an efficient differencing algorithm for incremental programming of constrained iot devices, Internet of Things 17 (2022) 100482. doi:https://doi.org/10.1016/j.iot.2021.100482.
URL https://www.sciencedirect.com/science/article/pii/S2542660521001220

[40] G. Oikonomou, S. Duquennoy, A. Elsts, J. Eriksson, Y. Tanaka, N. Tsiftes, The contiki-ng open source operating system for next generation iot devices, SoftwareX, Elsevier (2022). doi:https://doi.org/10.1016/j.softx.2022.101089.

[41] P. Eronen, H. Tschofenig, RFC 4279 - Pre-Shared Key Ciphersuites for Transport Layer Security (TLS).
URL https://datatracker.ietf.org/doc/rfc4279

[42] P. Wouters, H. Tschofenig, J. Gilmore, S. Weiler, T. Kivinen, RFC 7250 - Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS).
URL https://datatracker.ietf.org/doc/7250

{Pavlos~Charalampidis}
{pcharala@ics.forth.gr}

{Antonis~Makrogiannakis}
{makrog@ics.forth.gr}

{Nikolaos~Karamolegkos}
{nkaram@ics.forth.gr}

{Stefanos~Papadakis}
{stefpap@ics.forth.gr}

{Yannis~Charalambakis}
{jchar@noveltech.gr}

{George~Kamaratakis}
{gkamaratakis@noveltech.gr}

{Alexandros~Fragkiadakis\corref{cor1}}
{alfrag@ics.forth.gr}
{Corresponding author}

{organization={Institute of Computer Science, Foundation for Research and Technology-Hellas (FORTH)},
city={Heraklion},
postcode={70013},
state={Crete},
country={Greece}}

{organization={Noveltech IKE},
city={Heraklion},
postcode={70013},
state={Crete},
country={Greece}}

**Declaration of interests**

☐ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☒ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Alexandros Fragkiadakis reports financial support was provided by General Secretariat for Research and Technology.