LoRa meets IP: a Container-based Architecture to Virtualize LoRaWAN End Nodes

Antonio Cilfone, Luca Davoli, *Member, IEEE,* and Gianluigi Ferrari, *Senior Member, IEEE*

Abstract—In this work, a container-based architecture for the integration of Long Range Wide Area Network (LoRaWAN) end nodes—e.g., used to monitor industrial machines or mobile entities in specific environments—with Internet Protocol (IP)-based networks is proposed and its performance is investigated. To this end, we exploit the native service and resource discovery support of the Constrained Application Protocol (CoAP), as well as its light traffic requirements, owing to its use of User Datagram Protocol (UDP) rather than Transmission Control Protocol (TCP). This approach (i) adapts transparently (with no impact) to both private and public LoRaWAN networks, (ii) enables seamless interaction between LoRaWAN-based and CoAP-based nodes, through a logical "virtualization" of LoRaWAN nodes at server side, and (iii) enables routing among LoRaWAN end nodes, overcoming LoRaWAN's absence of inter-node communication and lack of compliance (at the end nodes side) with IP. Two virtualization approaches are proposed: (i) virtualization of a single end node (represented as a CoAP server) per container and (ii) virtualization of multiple end nodes (as CoAP servers) per container. Finally, deployments of the proposed virtualization architectures, using both a laptop and an Internet of Things (IoT) device (e.g., a Raspberry Pi), are considered, highlighting how the best solution relies on the use of several containers, with more than one CoAP server per container.

Index Terms—Internet of Things, LoRaWAN, Virtualization, CoAP.

1 Introduction

The Internet of Things (IoT) applies to heterogeneous ecosystems where a massive number of (typically) constrained devices is deployed and connected in order to cooperate for multiple purposes, such as data collection and actuation, in both Human-to-Machine (H2M)-oriented and Machine-to-Machine (M2M)-oriented ways. In this context, one of the main challenges is the seamless interaction between heterogeneous networks (e.g., in terms of transmission range capabilities [1] and especially targeting interactions at higher layers, rather than at physical layer) to improve plants' safety and operation.

Among IoT network technologies, Low-Power Wide Area Networks (LPWANs) are attracting a significant interest, having the advantage to meet almost all IoT requirements, such as: (i) easy and inexpensive deployment (i.e., unlike solutions based on cellular 4G/LTE communications, which incur costs related to SIM renting and traffic data plans), (ii) wide coverage, (iii) simple and scalable architecture, and (iv) low-power consumption. This comes at the cost of a few limitations such as: (i) limited data rate and (ii) restrictions on uplink and downlink capabilities. To this end, one of the most attractive LPWANs is Long Range WAN (LoRaWAN) [2], operating in the unlicensed Industrial, Scientific and Medical (ISM) bands [3] and emerging as a key enabler for typical (and heterogeneous) IoT contexts (e.g., smart city, smart farming, and Industrial IoT, IIoT) [4], with its appeal confirmed by extensive performance analysis [5]. As an example, future operational services offered by an integrated data collection from manufacturing machines and mobile vehicles will benefit from the possibility to expand their communication range [6,7].

An attractive approach to the design of LPWAN-based systems would be to leave the architecture of LPWANs unmodified (considering them as backbone networks), thus adopting self-organizing mechanisms enabling heterogeneous networks to interact with each other, in a transparent way from the point of view of the end nodes. This should be guaranteed even taking into account constraints on power consumption and (sometimes) *post-deployment* critical maintenance due to environmental limitations. In this context, devices' logical virtualization will likely play a key role in enabling the desired *trade-off* between network flexibility and performance (e.g., especially in the case LPWANs will not be natively IP-compliant and, thus, with nodes not natively addressable in a "standard" way). However, despite many implementations, LoRaWAN-based solutions are often considered as standalone, with very limited integrability with other protocols.

In this work, a novel networking architecture, based on the Constrained Application Protocol (CoAP) [8] and enabling the interaction between (non-IP-compliant) LoRaWAN end nodes and non-LoRaWAN IP-compliant nodes, is proposed and its performance is evaluated. More in detail, through one or multiple containers, LoRaWAN end nodes are virtualized [9]—they are represented with corresponding digital replicas, following a "digital twin" approach [10]—both *on-premise* (e.g., at the edge), as well as *in the cloud*. In this way, our solution enables a seamless interaction between LoRaWAN end nodes and external

- *A. Cilfone is with Tesmec Automation s.r.l., Italy. He was with the IoT Lab of the University of Parma when contributing to this work. E-mail: antonio.cilfone@tesmec.com*
- *L. Davoli and G. Ferrari are with the Internet of Things (IoT) Lab, Department of Engineering and Architecture, University of Parma, Italy. E-mail: luca.davoli@unipr.it, gianluigi.ferrari@unipr.it*

Manuscript received August X, 2023; revised October Y, 2023.

IP-compliant devices, without any impact on the existing LoRaWAN stack and on power requirements of the end nodes, which would still be unaware of this additional virtualization layer and would not be affected in their internal implementation.

The remainder of this work is organized as follows. In Section 2, a short introduction on LoRaWAN, Cayenne Low Power Payload (CLPP), and alternative data representation formats, is provided. In Section 3, we comment on research works available in the literature. In Section 4, the proposed architecture, where a container may allow the virtualization of one or more servers corresponding to LoRaWAN end nodes, is described. In Section 5, the system performance is investigated. Finally, in Section 6 conclusions are drawn.

2 Overview on LoRaWAN and Data Representation

LPWANs are designed to offer affordable connectivity to a large number of constrained devices distributed over large areas (e.g., different plants of the same production site). To this end, various LPWAN technologies are available, such as (just to name a few) Narrowband IoT (NB-IoT) [11,12], Sigfox [13], and LoRa [14]. LoRa refers only to the physical (PHY) layer in the LPWAN stack and does not provide sufficient reliability mechanisms to be used *as-is* in IoT contexts. In fact, as LoRaWAN operates in licence-free frequency bands (as further detailed in Subsection 2.1), coexistence challenges with other well-known radio standards (e.g., Zigbee, Bluetooth, etc.) in these frequency bands might emerge, leading to interference problems [15,16]. More in detail, a "background noise" may appear, due to the presence of devices exchanging data in parallel via heterogeneous radio communication protocols, as well as vehicles emitting electromagnetic waves through their electronic boards (e.g., in urban and sub-urban contexts). Finally, it is not possible to completely neglect environmental interference even caused by LoRa devices interfering with each other, because of simultaneous retransmissions (owing to the Aloha-like transmission behaviour of this protocol). With regard to the Sigfox protocol, in 2022 Sigfox (as a company) went into bankruptcy proceedings, so its adoption is not attractive. Finally, NB-IoT requires a cellular network coverage, which is not always available (depending on both location and environmental conditions), and bears costs due to the SIM card to be included in the IoT/IIoT devices.

Hence, among the available LPWAN protocols, one of the most interesting (as anticipated in Section 1) is LoRaWAN. More precisely, LoRaWAN has been defined by the LoRa Alliance [17], relies on LoRa modulation (based on Chirp Spread Spectrum, CSS, patented by Semtech Corporation [18]), and specifies the channel access method and the network architecture to be exploited in operational scenarios.

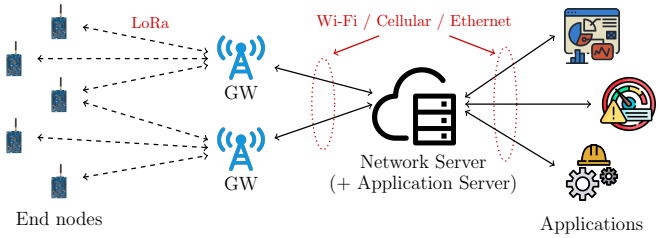

Fig. 1: General LoRaWAN architecture providing IoT-oriented information to external applications.

2.1 Basics of LoRaWAN

From an architectural point of view, as shown in Fig. 1, a "pure" LoRaWAN architecture (either private or public) is composed by (*on-field*) end nodes, intermediate gateways (GWs), and a Network Server (NS, often combined with an Application Server, AS) managing the overall LPWAN. In detail, the link between an end node and a GW is based on the PHY-layer LoRa protocol, while GWs are connected to the NS through the IP protocol. Focusing on *on-field* LoRaWAN end nodes, depending on their downlink capability, they can belong to one operational class among Class `A`, Class `B`, and Class `C` [2]. In detail, in Class `A`, which must be supported by all LoRaWAN devices, a node can receive downlink packets only inside two receive slots, following an uplink transmission act, thus resulting in the lowest energy consumption mode. Class `B` devices are allowed to open extra receive slots at scheduled intervals (regardless of uplink transmissions), identified by the reception of synchronization beacons from the GWs—therefore, they support applications requiring a more intense downlink traffic. Finally, Class `C` devices are always listening to the channel, thus presenting the highest energy consumption level among these operating classes, but, at the same time, being able to receive a downlink packet at any time, leading to the lowest downlink latency. The channel access mechanism is based on Aloha: once an end node wakes up, it sends a packet on a selected radio channel. At this point, one or more GWs (within the end node's transmission range) (i) receive the packet and (ii) forward it to the NS, which keeps only one instance of the received (potentially multiple times) packet. Since operations are carried out in unlicensed bands (in Europe, the European Telecommunications Standards Institute, ETSI [19]-defined 868 MHz band), end nodes and GWs must operate with a proper duty cycle (in Europe, between 0.1% and 10%, depending on the adopted frequency), unless end nodes perform Listen-Before-Talk (LBT) or frequency-hopping techniques. Hence, each time a frame is transmitted, the *airtime* is calculated and, subsequently, the time interval during which the transmitter cannot use the channel, denoted as *time off* (T_{OFF}), is derived.

Moreover, LoRaWAN end nodes are required to "join" the LoRaWAN network to participate to network operations. In detail, the join operation can be carried out through two activation methods: (i) Over-The-Air-Activation (OTAA) and (ii) Activation-By-Personalization

(ABP), with OTAA being the most secure method, as session parameters change at each session establishment. Thus, owing to this feature, OTAA is the join mechanism which will be considered in this work, being recommended by the LoRa Alliance because of its high security level, that perfectly fits scenarios where communication security is crucial. With ABP, end nodes already know all the configuration parameters required for activation. With OTAA, the LoRaWAN specifications [2] define the use of (i) static root keys (only provisioned in OTAA end devices) and (ii) session keys dynamically generated from the root keys and derived when an OTAA device executes a join procedure to the LoRaWAN network. Hence, once installed on the field, an OTAA-activated end device will protect its *over-the-air* traffic by using the session keys. At the opposite, end devices using the ABP activation procedure will not be provisioned with root keys, but they will feature only a set of session keys for a pre-defined network, with session keys remaining unmodified throughout the lifetime of the ABP end device itself.

2.2 Cayenne LPP (CLPP)

One of the main constraints of LoRaWAN (and, in general, of all LPWANs) is the available payload space inside a network packet: with reference to EU regulation, the smallest available payload is equal to 59 bytes. Nevertheless, the parameter of interest (in order to send as much information as possible) is the maximum payload size, which, in the case of LoRaWAN, depends on the chosen Spreading Factor (SF). In several scenarios, the focus would be on the possibility to shorten the payload to be sent through such a constrained protocol, but, at the same time, guaranteeing to transmit all the required information (e.g., through multiple packets), by devising a mechanism to "auto-describe" the data without the need to *a-priori* define a rigid packet format (e.g., avoiding the need to reserve a fixed space in the packet for each information). In order to meet these goals, one of the widely adopted formats is the Cayenne Low Power Payload (CLPP) [20], which can be "squeezed" to 11 bytes (thus fitting the available payload size given by any LoRaWAN SF), avoiding useless separators, and allows an end node to send multiple sensors' data at once by splitting the information across consecutive frames.

More in detail, each CLPP-formatted data packet has a prefix given by the following 2 bytes:

- *data channel* byte, used to identify the node's sensors across multiple frames (in the presence of multiple sensors of the same type);
- *data type* byte, used to identify the nature of the sensor inside the frame, as regulated by the Internet Protocol for Smart Objects (IPSO) guidelines [21].

As a clarification example, Table 1 summarizes the identifiers and data resolution (per bit) related to the sensors that will be assumed as reference *on-field* data collectors in this work. In Fig. 2, we show an illustrative CLPP frame sent by a LoRaWAN end node (e.g., located in a drying cell in a production plant) with three on-board sensors (pressure, temperature, and humidity) measuring $997.8\,\mathrm{hPa}$, $26.6\,^{\circ}\mathrm{C}$, and 55.5%, respectively. For each type of sensor a byte sequence is sent: data channel (one byte), data type (one bye), and "raw" sensor data (at least one byte). In this case, the CLPP-encoded payload is equal to `007326FA0167010A02686F`, where:

- `0x00`, `0x01`, and `0x02` are the indexes of the sensors;
- `0x73` identifies a pressure sensor (barometer);
- `0x26FA` is the pressure value that, once converted to decimal (i.e., 9978), has to be multiplied by $0.1\,\mathrm{hPa}$ (according to the rules shown in Table 1), leading to a final value equal to $997.8\,\mathrm{hPa}$;
- `0x67` identifies a temperature sensor;
- `0x010A` is the (2-byte) temperature value that, once converted to decimal, has to be multiplied by $0.1\,^{\circ}\mathrm{C}$, leading to a final value equal to $26.6\,^{\circ}\mathrm{C}$;
- `0x68` identifies a humidity sensor;
- `0x6F` is the humidity value that, once converted to decimal, has to be multiplied by 0.5 and expressed in $\%$, leading to a final value equal to 55.5%.

As will be discussed later, in this work the CLPP format will be exploited to perform automatic sensor discovery at each (static or mobile) end node, in order to simplify distributed environmental monitoring [22].

TABLE 1: LPP sensor codes.

Sensor	HEX Value	Length [bytes]	Data Resolution (per bit)
Analog input	`0x02`	2	0.01 Signed
Temperature	`0x67`	2	$0.1\,^{\circ}\mathrm{C}$ Signed MSB
Humidity	`0x68`	1	0.5% Unsigned
Accelerometer	`0x71`	6	$0.001\,\mathrm{G}$ Signed MSB per axis
Barometer	`0x73`	2	$0.1\,\mathrm{hPa}$ Unsigned MSB
Noise sensor	`0xE9`	2	$0.1\,\mathrm{dB}$ Unsigned MSB
Filling level	`0xEB`	2	0.01% Unsigned
Air quality	`0xE4`	2	1 $\mu\mathrm{g/m}^3$ Unsigned MSB
Location	`0x88`	9	Lat: $0.0001°$ Signed MSB Lng: $0.0001°$ Signed MSB Alt: $0.01\,\mathrm{m}$ Signed MSB

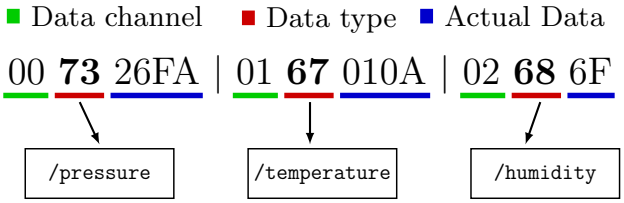

Fig. 2: Example of received CLPP-encoded uplink packet, with corresponding CoAP resources (as detailed in Section 4).

2.3 Alternative Data Representation Formats

Besides the CLPP format detailed in Subsection 2.2, there exist alternative ways to represent data to be exchanged between parties in a communication, obviously requiring proper encoding/decoding operations in order to encode

and decode information. In the following, we overview four possible alternative data representation formats in a comparative way with respect to CLPP.

2.3.1 JavaScript Object Notation (JSON)

JavaScript Object Notation (JSON) is a text-based, language-independent data interchange format based on JavaScript object syntax and on a set of formatting rules for the serialization and representation of structured data, commonly used for transmitting data in Web applications [23]. In detail, many programming environments can parse JSON (that is provided with the specific MIME type `application/json`) which allows to represent primitive (e.g., strings, etc.) and structured (e.g., objects and arrays) types. This leads to a serialized sequence of tokens, with *strings* being sequences of zero or more Unicode characters to be enclosed inside double quotes, and *objects* corresponding to an unordered collection of zero or more name/value pairs to be enclosed inside square and curly brackets. Colons and commas are used as name and value separators.

Owing to the above features, it is clear that JSON represents a good candidate in scenarios where a lightweight but structured data format is required, at the same time not presenting constraints on the available payload size. Therefore, JSON cannot be considered as an attractive candidate for constrained information exchanges via LoRaWAN packets (particularly with reference to CLPP). For the sake of comparison, with reference to Fig. 2, assume to represent each single CLPP block (composed by *data channel*, *data type*, and "raw" data) as an array of JSON objects, in which each single JSON object features (i) the CLPP *data type* associated with the key `t`, (ii) the unit of measurement being *a-priori* known thanks to a static mapping available in an external registry (similarly to CLPP), (iii) the CLPP *data channel* associated with the key `c`, and (iv) the "raw" data mapped as the value of a tuple with key `v`. A possible corresponding JSON-encoded LoRaWAN payload (with a length equal to 92 bytes) is shown in Listing 1 (expanded on multiple lines for readability).

Listing 1: JSON-encoded LoRaWAN payload

```
[
    {"c": "00", "t": "73", "v": "26FA"},
    {"c": "01", "t": "67", "v": "010A"},
    {"c": "02", "t": "68", "v": "6F"}
]
```

It can be thus concluded that CLPP is much more efficient, in terms of compact data representation, than JSON.

2.3.2 Extensible Markup Language (XML)

Extensible Markup Language (XML) is a text-based markup language (similar to HTML) defined for exchanging information between systems (in general, in M2M applications), and used to define the *so-called* XML documents [24]. In detail, each XML document is composed of XML entities and needs to begin with a unique root element. In order to encapsulate information, an XML document can have a starting (case-sensitive) XML declaration followed by several XML elements, denoted as XML nodes or XML tags, to be enclosed in triangular brackets. Moreover, each element can in turn contain multiple elements as its children, which need not to overlap (i.e., an element's end tag must have the same name as that of the most recent unmatched start tag). Finally, as for the XML declaration, even names and attributes (specifying single properties through name/value pairs) of an XML element are case-sensitive, too.

As highlighted for JSON, even XML is characterized by a long data representation, difficult to be enclosed in constrained payloads. Therefore, it cannot be considered as a valid alternative with respect to CLPP. For the sake of comparison, with reference to Fig. 2, assume that the XML root element is represented by the XML tag `<r>` and that each single CLPP block (composed by *data channel*, *data type*, and "raw" data) is represented as a list of XML elements, in which each single XML element `<i>` features (i) the CLPP *data type* associated with the key `t`, (ii) the unit of measurement being *a-priori* known, thanks to a static mapping available in an external registry (similarly to CLPP), (iii) the CLPP *data channel* associated with the key `c`, and (iv) the "raw" data mapped as the value of the XML element itself. A possible corresponding XML-encoded LoRaWAN payload (with a length equal to 80 bytes) is shown in Listing 2 (expanded on multiple lines for readability).

Listing 2: XML-encoded LoRaWAN payload

```
<r>
    <i c="00" t="73">26FA</i>
    <i c="01" t="67">010A</i>
    <i c="02" t="68">6F</i>
</r>
```

As for JSON, it can be concluded in this case as well that CLPP is more concise than XML.

2.3.3 Constrained RESTful Environments (CoRE) Link Format

The Constrained RESTful Environments (CoRE) Link Format is a serialization mechanism defined to describe (using a link-header style format [25]) relationships between entities (e.g., CoAP resources) in constrained nodes and networks, especially in M2M-like scenarios [26]. More in detail, CoRE Link Format has an associated MIME type (`application/link-format`), is encoded as UTF-8 (thus each character should be represented "wasting" a byte of space in the payload), and requires the use of commas to separate multiple link descriptions.

Therefore, being similar to XML (except for the attribute position inside the encoded payload), CoRE Link Format requires a non-negligible payload length, which is critical in the case payload size-constrained protocols have to used (e.g., LoRaWAN). For the sake of comparison, with reference to Fig. 2, assume to represent each single CLPP block (composed by *data channel*, *data type*, and "raw" data) as a list of link descriptors, in which each single link descriptor features (i) a resource name starting with `/r`

and followed by an incremental index (e.g., `/r0`, `/r1`, etc., required to distinguish among multiple links), (ii) the CLPP *data type* mapped as the value of a custom attribute `t`, (iii) the unit of measurement being *a-priori* known thanks to a static mapping available in an external registry (similarly to CLPP), (iv) the CLPP *data channel* associated with a custom attribute `c`, and (v) the "raw" data mapped as the value of the standard attribute `ct`. A possible corresponding CoRE Link Format-encoded LoRaWAN payload (with a length equal to 87 bytes) is shown in Listing 3 (expanded on multiple lines for readability).

Listing 3: CoRE Link Format-encoded LoRaWAN payload

```
</r0>;ct="26FA";t="73";c="00",
</r1>;ct="010A";t="67";c="01",
</r2>;ct="6F";t="68";c="02"
```

As in the previous cases, in this case as well CLPP is more efficient (in terms of representation compactness) than CoRE Link Format.

2.3.4 Custom Raw Data Encoding

As a final data representation approach, one may not exploit well-known, standardized, and structured data representation mechanisms, such as those mentioned in Subsection 2.2 and Subsections 2.3.1–2.3.3. In this case, one may encode the information to be sent from an *on-field* source node toward remote entities through a customized (*raw bytes*) representation. To this end, unlike CLPP, the definition of a custom data formatting mechanism requires to "manually" define encode/decoding policies for the data to be exchanged. These policies, in turn, often (i) strictly depend on the specific device's vendor, (ii) have a "position-based" significance (i.e., each byte has a specific meaning depending on its position inside the payload, with consequent decoding issues in the case of disturbed communications), and (iii) require the vendor itself to release documentation and specifications to allow the end user to encode and decode the collected data. Hence, these drawbacks may hinder the applicability of "custom-made" encoding/decoding strategies with respect to standardized (and constrained, in terms of payload length) solutions like CLPP.

3 Related Works

In [27], cloud and edge computing are combined with LoRaWAN to develop a campus air quality monitoring system. No seamless integration of LPWAN-based end devices is considered, as high-layer computing infrastructures are exploited only for visualization and processing needs.

In [28], a probabilistic approach is proposed to enable efficient sharing of LoRaWAN access networks between different services/slices and integration of admission control mechanisms (expedient to the devices from transmitting). In this case, the focus is on the physical layer of the chosen communication protocol, allowing multiple end nodes to send their messages toward natively defined LoRaWAN high-layer systems (i.e., NS and AS), rather than focusing on end nodes' virtualization for enhanced services.

In [29], the interest is on the integration of a 5G mobile network (with very high capacity) to provide backbone connectivity for the LoRaWAN architecture, thus focusing not on a seamless integration at service layer but, rather, on speeding up the transmission of information on the backbone layer.

In [30], a solution to seamlessly integrate LoRaWAN with 4G/5G mobile networks is proposed, claimed to be transparent to LoRaWAN end devices since only the LoRaWAN gateway needs to be modified. The integration thus requires the modification of a component of the architecture, thus being not completely seamless and without providing high-layer services to external consumers which might request data collected by *on-field* LoRaWAN end devices.

In [31], virtualization technologies are used to simulate LoRaWAN at the application layer (i.e., the design, development and testing for roaming in LoRaWAN networks in the context of future smart cities). The focus is on the simulation of such scenarios, rather than providing a way to seamlessly enable high-layer services' exploitation by possible external entities interested in interacting with LoRaWAN end nodes (in both uplink and downlink directions).

Given that the topics of the literature works summarized above differ from the main topic of our paper, in the following we detail and describe the design and implementation of a novel and effective virtualization architecture to allow LoRaWAN (non-IP-based) end nodes to interact seamlessly with IP-based nodes.

4 Container-based Virtualization Architecture

The main goal of the proposed architecture is to create replicas of LoRaWAN end nodes on top of the IP layer, in order to manage different *on-field* mobile and static sensing and actuating devices in a modular way (e.g., through an application layer control system). As detailed in Section 1, our focus is not on the chosen virtualization solution itself, but, instead, on the definition of a "high-layer digital twin" of a set of LoRaWAN end nodes, which natively do not support IP-based routing and inter-node communication. To this end, the virtual replica of a physical LoRaWAN end node is denoted as *virtual End Node* (vEN) and is implemented as a CoAP server (following an IoT-like approach). Then, following this CoAP-oriented approach, each sensor equipping a physical LoRaWAN end node will be represented as a virtual CoAP resource attached to its own vEN, thus allowing external CoAP-enabled entities to virtually interact with the (physical) LoRaWAN end nodes. In detail, CoAP has been chosen thanks to its light traffic requirements (e.g., being based on UDP instead of TCP) and native support to *service* and *resource discovery* mechanisms [32]–[34] (even if requiring additional information descriptors [35]), as well as

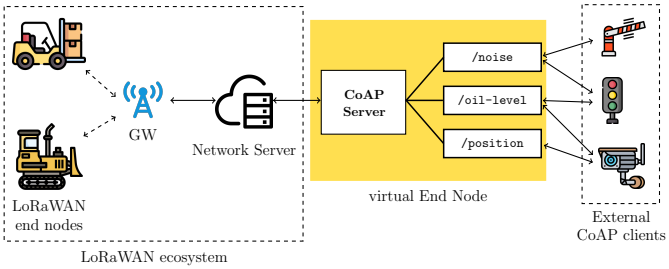

Fig. 3: LoRaWAN architecture extended with the Virtualizer.

to increase the number of application protocols (e.g., the traditional HTTP and MQTT protocols [36]) able to cooperate with the LoRaWAN's AS. This enables communication heterogeneity in several IoT scenarios. We remark that, as the aforementioned traditional protocols (HTTP, MQTT) to be possibly adopted in IoT/IIoT scenarios, even CoAP is not exempt from possible drawbacks (i.e., packet losses or delays). Nevertheless, CoAP represents an effective choice for the implementation of the "logical abstraction" framework proposed in this manuscript for the following main reasons: (i) "*readyness-by-design*" for IoT; (ii) lightweight service discovery mechanisms; (iii) binary nature; (iv) native support of the *observing* relation ("moving" the *pros* of the publish/subscribe paradigm, representative of MQTT, into a request/response-like protocol, like HTTP); and (v) support to real-time instantiation of operating CoAP resources on top of a CoAP server (unlike HTTP, in which the same resources should be HTML documents or predefined APIs).

As an example, consider an IoT context in which mobile LoRaWAN end nodes are installed in a mobile vehicle (e.g., an industrial forklift or a bulldozer), allowing them to seamlessly interact with other equipment (e.g., automatic barriers, safety cameras, and lights) during working hours, as shown in Fig. 3. Each LoRaWAN end node can send mobile vehicle-related data (e.g., oil level, *in-vehicle* conditions, position), as well as information related to the surrounding environment (e.g., air quality). The mobile vehicles may also be equipped with safety controllers to avoid certain areas or paths toward certain operational locations.

Regardless of the scenario of interest, the proposed virtualization architecture allows a seamless information exchange, guaranteeing a transparent compatibility between (non-IP) LoRaWAN end nodes and IP-compliant entities, thus leaving the "core" LoRaWAN architecture unmodified (i.e., compliant with the corresponding definitions and protocol specifications). Hence, the virtualization *add-on* can be seen as a "proxy" among heterogeneous networks, allowing to model a LoRaWAN "ecosystem" as a "black box" left unmodified by our virtualization architecture, which would be deployed on top of the LoRaWAN ecosystem itself. Since the LoRaWAN core infrastructure is left *as-is*, the latency among LoRaWAN nodes and IP-based nodes is the same experienced in a classical LoRaWAN network (with some possibly additional minor processing delays). In particular, the specific LoRaWAN end node operational mode (i.e., Class `A`, `B`, or `C`) is not influenced (in its internal transmission/reception intervals) by the proposed virtualization infrastructure. As an example, should a LoRaWAN downlink message be steered from the NS toward a specific LoRaWAN Class `A` end node, this will always correspond to an asynchronous operation (as downlink packets need to wait the first available LoRaWAN receive slot to be sent from the NS to the field). Hence, the downlink packets are queued in the *native* LoRaWAN AS by the proposed virtualization infrastructure as soon as its corresponding modules have received this enqueuing request from an external entity, but certainly *in advance* with respect to the reception slot of the LoRaWAN end node. In the case of LoRaWAN Class `C` end devices, the overall delay (from the time instant at which an external entity requires to send a downlink message to a LoRaWAN end node to the time instant at which this packet will be queued in the core LoRaWAN NS) would not be increased significantly by the presence of the virtualization infrastructure, e.g., by exploiting *native integrators* provided by the classical LoRaWAN architecture (e.g., via HTTP REpresentational State Transfer, REST, Application Programming Interfaces, APIs).

The same transparent compatibility introduced above holds for aspects depending on the LoRaWAN SF, which is not affected by the proposed virtualization. The adoption of the proposed virtualization architecture allows also data routing among LoRaWAN end nodes (through their CoAP-based virtual replicas). This would not be possible with a "pure" LoRaWAN architecture, as LoRaWAN requires that interactions should follow a "tree-like" structure, with messages following the classic LoRaWAN uplink (from end node to NS/AS) and downlink (from AS/NS to end node) sequence, thus preventing a direct interaction among end nodes.

In Fig. 4, we show the building blocks (and their interactions) of the proposed virtualization architecture, whose operations can be described as follows. Upon receiving new data from a LoRaWAN end node (through the NS/AS), the Data Manager (DM), shown in Fig. 4, looks for the corresponding vEN and updates the value of the proper resource, until new updates arrive. Therefore, the main advantage of this approach is that if a node receives two independent requests from an external IP-enabled entity within a time interval shorter than T_{OFF}, it can still use the latest available data to reply to these requests, thus providing a *caching* mechanism to the entire virtualization infrastructure.

Moreover, as highlighted above, the *resource discovery* functionality is another key aspect of the proposed virtualization architecture, since when a LoRaWAN end node is deployed *on-field*, the back-end architecture does not *a-priori* know which data types will be sent. In the same way, the LoRaWAN NS does not know in advance the amount and nature of the sensors each LoRaWAN end node is equipped with. Therefore, exploiting the CLPP format, it

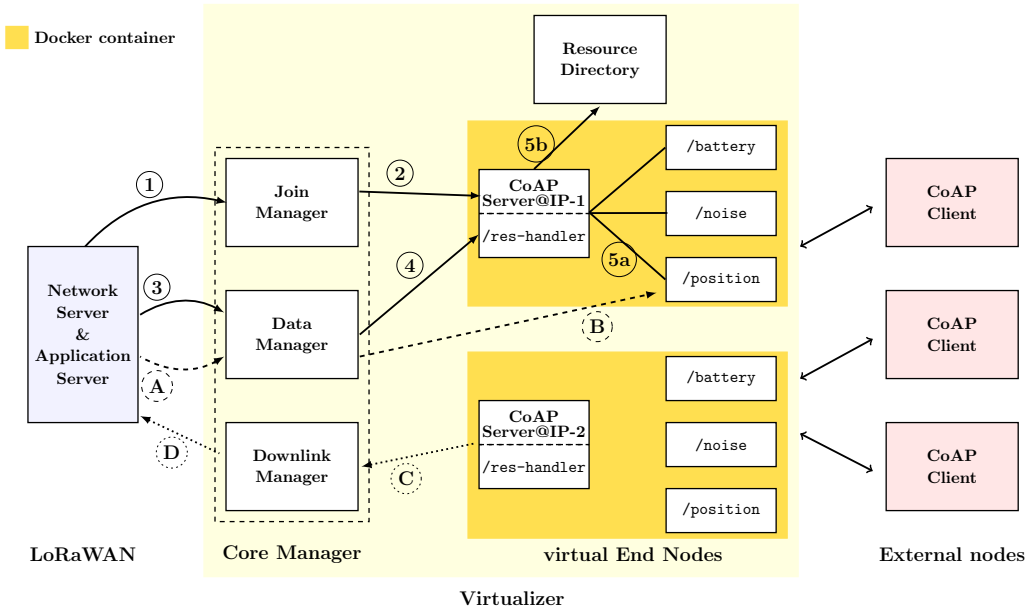

Fig. 4: Building blocks and interactions of the virtualization architecture following the "parallel container" approach.

is possible to introduce automatic sensors' discovery by simply parsing the LoRaWAN packet's payload. This is even truer since: (i) LoRaWAN end nodes know which kind of data they are producing, so they are aware of the CLPP data encoding to be applied before sending their information, while, on the other end; (ii) the proposed virtualization infrastructure will be able to know the exact amount of sensors maintained by each end node simply by parsing the CLPP payload. Since the virtualization infrastructure is *resourceful*, it might interact with external entities to retrieve additional information (e.g., the first time an "unknown" payload arrives and has to be parsed). According to this approach, the behavior of the LoRaWAN network becomes transparent to external (CoAP and, more generally, IP) clients.

Finally, from an implementation point of view, the proposed virtualization architecture relies on the use of Docker containers [37], thus ensuring isolation and independence between the applications (i.e., CoAP servers) executed inside them and possibly deployed *on-premise* in local data centers. We highlight that there may be alternative approaches to automatic sensor discovery (e.g., SLP [38], Zeroconf [39], mDNS [40], DNS-SD [41], etc.). However, as discussed and motivated in Subsection 2.2 and Subsection 2.3, CLPP is preferrable as it is compact and, thus, abides more effectively by LoRaWAN protocol's constraints.

We remark that future research activities will involve the analysis (and comparison) of *pros* and *cons* between an architectural deployment based on containers and a corresponding deployment based on a different approach, such as one based on microservices. In fact, both these approaches have recently emerged as attractive paradigms revolutionizing how applications are built, deployed, and scaled. In the following, we shortly highlight the benefits of each approach.

- *Containers* give the developer (and the end user) a lightweight and isolated environment in which applications and libraries can be packed and encapsulated into a single unit. This ensures a consistent behavior across different infrastructures and avoids drawbacks due to different operating systems' configurations, without worrying about underlying infrastructure variations. Therefore, containers enable (i) seamless deployment and scalability, (ii) high portability and isolation (leading to enhanced security and stability since changes or issues within a container do not affect others), (iii) efficiency, in terms of fewer consumed resources (if compared to traditional VMs), and (iv) fast startup times.
- Architectures based on *microservices* tend to be exploited when there is the need to "break down" complex applications into smaller and independent services, each one specifically responsible for a particular functionality and interacting with other microservices through APIs (as shared information channels). Therefore, microservice-based architectures can, in general, benefit from (i) granular scaling, with only high demand specific services having to be scaled (thus optimizing resources utilization), (ii) fault isolation, since if one service experiences criticalities or fails, it can be singularly rescued without affecting the overall application life cycle (thus enhancing the system's resilience), and (iii) technology heterogeneity, as deployed microservices may be based on different frameworks and programming languages, thus enhancing the system's flexibility.

It is clear that these two approaches cannot be con-

sidered as mutually exclusive, while, instead, as complementary to each other. A container-based approach seems preferrable for our proposed virtualization architecture, since the initial deployment may require to instantiate multiple clones (i.e., containers) of the same instance, in order to virtualize multiple *on-field* LoRaWAN end nodes: each container runs the same applications written with the same programming language. Nevertheless, the design, deployment, and orchestration of microservices within containers is an attractive research direction, expedient to enhance performance, security, and management of our proposed virtualization architecture. This goes beyond the scope of the current manuscript and is subject of future research.

4.1 "Parallel Container" Virtualization: One Server per Container

With reference to Fig. 4, a "parallel container" approach requires that one single Docker container is started for each vEN, i.e., only one CoAP server runs inside a container. More in detail, the operations of the virtualized architecture involve the following tasks: (i) solid arrows, describing the "discovery" phase; (ii) dashed arrows, representing "regular" operations carried out by LoRaWAN end nodes (e.g., uplink messages collected by vENs and sent, as CoAP response's payload, to external CoAP entities); and (iii) dotted arrows, denoting the data flows for LoRaWAN downlink messages.

As highlighted in Subsection 2.1, in the proposed architecture the OTAA join procedure mechanism is adopted for the LoRaWAN end nodes, in order to provide a higher self-configurability. More generally, each specific LoRaWAN NS may be properly implemented and uses a specific approach to make collected data available to external non-LoRaWAN networks. To this end, the most common approaches rely on (i) the REST paradigm, through specific APIs, (ii) the MQTT protocol, and (iii) the WebSocket protocol [42]. Although all these approaches are effective, in the proposed virtualization architecture MQTT will be considered due to its publish/subscribe nature, useful to notify the presence of a new data, instead of continuously polling an endpoint to check the availability of a new information (e.g., according to the REST paradigm).[1] In particular, the LoRaWAN NS exposes the following three MQTT topics:

- *join* events, on the `join/` MQTT topic;
- *uplink* messages, on the `uplink/` MQTT topic;
- *downlink* messages, on the `downlink/` MQTT topic.

In detail, once a LoRaWAN end node joins the network, the join event is published in the corresponding MQTT topic and the Join Manager, subscribed to this topic via a MQTT client (step ① in Fig. 4), retrieves the device address of the LoRaWAN end node and instantiates a corresponding Docker container where the vEN will be (i) executed (step ② in Fig. 4) with a certain IP address and (ii) associated with the "discovered" address. Then, for each LoRaWAN end node, a new container, associated with one vEN, is instantiated and started. Moreover, the vEN is added to the company's Resource Directory (RD), which acts as a sort of "white pages" service and is in charge of listing all the CoAP resources of the vENs corresponding to deployed physical nodes.

Once a vEN is created, an additional "reserved" CoAP resource, denoted as `/res-handler` and not visible outside the Virtualizer, is attached to the corresponding CoAP server. In detail, this specific CoAP resource is used to manage the sensors' discovery phase—as stated earlier, *resource discovery* is a key feature in IoT scenarios, avoiding an *a-priori* knowledge of the number and nature of the available LoRaWAN end nodes.

According to the approach presented in Subsection 2.2, the CLPP format will be exploited to allow automatic discovery of the sensors equipping a LoRaWAN end node by parsing the LoRaWAN packet's payload. In detail, for each sensor installed in the end node, the corresponding vEN generates a CoAP resource as a *virtual replica* of the specific sensor. Then, the MQTT client contained inside the DM listens to the `uplink/` topic and gathers data from the NS (step ③ in Fig. 4) and, once a packet is received through this MQTT topic, the Payload Parser Block (PPB) module is triggered. Depending on the type of received data and the resource list, the PPB can perform two different actions:

- if the sensor is *new*, no CoAP resource is associated with it and, therefore, a sensor discovery is performed;
- if the CoAP resource *already exists*, the received packet corresponds to an update packet and, therefore, the PPB (i) extracts the values from the packet and (ii) updates the related CoAP resources.

In detail, in the presence of a new sensor, the PPB parses the received payload, analyzing the CLPP data types found in the message, and, then, requires the DM to send a CoAP `POST` request to the specific vEN replica of the LoRaWAN end node (step ④ in Fig. 4) targeting the proper CoAP resource endpoint (as defined in Table 2) and the name of the CoAP resource that must be created, as `POST` payload. Then, as shown in step (5a) in Fig. 4, the vEN eventually creates the requested CoAP resource: as an example, with reference to Fig. 2, if the CLPP data type corresponds to `0x73`, then a `/pressure` CoAP resource is created and executed. In general, the syntax adopted in the proposed virtualization architecture is kept as simple as possible, with the name of the CoAP resource corresponding to the human-readable name of the sensor, as detailed in Table 2.

More in detail, considering the uplink CLPP payload shown in Fig. 2, for each sensor attached to the "real" LoRaWAN end node, the PPB obtains the data channel and type (in green and red colors in Fig. 2, respectively), and

1. Being WebSocket and MQTT similar (in terms of operational paradigm), the adoption of WebSockets in place of MQTT and the corresponding performance evaluation is left for future research.

TABLE 2: Sensors mapping on CoAP resources.

Sensor	CoAP Resource Endpoint
Battery voltage reader	`/battery`
Temperature	`/temp`
Humidity	`/humidity`
Accelerometer	`/accelerometer`
Noise sensor	`/noise`
Filling level	`/oil-level`
Air quality	`/air-quality`
GPS	`/position`

raw data. Then, the PPB decodes the values according to the bit resolution rules shown in Table 1 and, for each sensor, the DM sends a CoAP `PUT` request (step Ⓑ in Fig. 4) to the CoAP resource corresponding to the specific sensor, with the decoded values as CoAP `PUT`'s payload.

The following CoAP `POST` requests will be sent to the vEN (available at its `<vEN-IP>` IP address) in order to associate the proper CoAP resources with the vEN:

- `POST /pressure` to `coap://<vEN-IP>/res-handler`
- `POST /temp` to `coap://<vEN-IP>/res-handler`
- `POST /humidity` to `coap://<vEN-IP>/res-handler`

Regarding the update task on the different sensors, the following CoAP `PUT` requests will be sent to the specific vEN:

- `PUT` 997.8 to `coap://<vEN-IP>/pressure`
- `PUT` 26.6 to `coap://<vEN-IP>/temp`
- `PUT` 55.5 to `coap://<vEN-IP>/humidity`

If the vEN is still present in the RD, only the sensors' update is performed. Once vENs and their related CoAP resources are discovered and added to the RD, the rest of the message exchange corresponds to the classical exchange between CoAP nodes [8,43]: an external CoAP client looks for resources of interest in the RD and, then, sends CoAP-supported requests to the vENs.

Finally, if an external CoAP-enabled node is interested in sending a downlink message to a LoRaWAN end node (e.g., requesting a vehicle's position or an oil level), it simply needs to target a CoAP `PUT` request to the vEN's CoAP resource. In turn, this request will trigger the DM (step Ⓒ in Fig. 4) that, eventually, forwards it to the NS (step Ⓓ in Fig. 4) to reach the LoRaWAN end node with the specific downlink payload.

4.2 "Parallel vEN" Virtualization: Multiple vENs per Container

Unlike the virtualization approach proposed in Subsection 4.1, we now consider multiple vENs running in the same Docker container. In this case, instead of executing vENs in separate Docker containers (one vEN per container, with different IP addresses), a few vENs run in the same container, sharing the same IP but being identified by different ports (at transport layer). Hence, this solution makes vENs less independent but, as will be discussed in Section 5, it represents a good compromise for its use in constrained devices (e.g., because of deployments in remote areas far from a stable energy supply).

With respect to the "parallel container" approach discussed in Subsection 4.1, in the "parallel vEN" approach there is a new entity, implemented as a CoAP server and denoted as *vEN Handler*, which (i) has to handle the generation of new vENs, (ii) is accessible only from the Join Manager, and (iii) exposes (only internally to the Virtualizer) the `/ven-handler` CoAP resource. As in Subsection 4.1, when a LoRaWAN end node joins the LoRaWAN network, the join event is published in the `join/` MQTT topic and the Join Manager, subscribed to this topic (step ① in Fig. 5), retrieves the LoRaWAN end node's device address and sends a CoAP `POST` request to the `/ven-handler` CoAP resource, with the LoRaWAN device's address as payload (step ② in Fig. 5). Then, the vEN Handler starts a Docker container with a specific IP address (step ③ in Fig. 5). In this container, a vEN is created on a UDP port associated with the "discovered" address and added to the RD. Thanks to this approach, once the vEN is started, its reserved `/res-handler` CoAP resource is also started. Therefore, in the case new LoRaWAN end nodes are discovered (e.g., new mobile nodes are deployed to monitor specific environments for safety reasons), their vENs will run inside the same Docker container, with each vEN associated with a specific port.

Finally, the following steps remain the same as discussed in Subsection 4.1: CoAP resources are added sending a CoAP `POST` request to `/res-handler` (step ⑤ in Fig. 5) and updated sending a CoAP `PUT` request to the corresponding resource (step Ⓑ in Fig. 5).

5 Experimental Performance Evaluation

5.1 Programming Language Evaluation

The proposed virtualization architecture has been developed in Java. This makes the Virtualizer compatible with a generic computing platform where a Java Virtual Machine (JVM) can run. Moreover, the choice of Java is also motivated by the availability of a complete CoAP implementation (based on the Californium library [44]) and the Paho library [45], in detail enabling to use CoAP and to implement the MQTT client needed to subscribe to the MQTT topics described in Section 4, respectively.

For the sake of completeness, it should be mentioned that alternative programming languages might be adopted to develop and implement the virtualization architecture detailed in Section 4. The choice of alternative languages represents a future research extension, possibly targeting a performance comparison (among different languages) in terms of (i) amount of code statements required to define the same behavior, (ii) execution time, (iii) required host resources (e.g., CPU, RAM, persistent storage, etc.), and (iv) end-user experience. To this end, the *pros* in favor of the Java language can be summarized as follows (as anticipated above): (i) its widespread use in several fields and contexts, making it attractive to developers who can carry out supplementary developments and, usually,

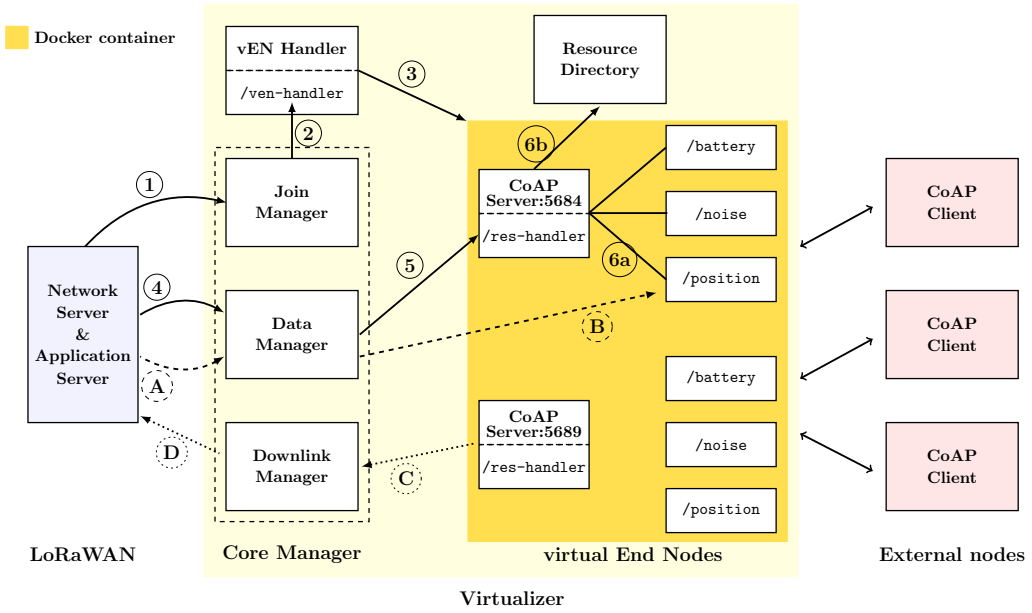

Fig. 5: Building blocks and interactions of the virtualization architecture following the "parallel vEN" approach.

allowing faster updates [46]; (ii) its portability across heterogeneous hardware devices, especially thanks to the presence of a run-time system (i.e., the JVM) able to make use of hardware resources without the developers' intervention—exploiting the traditional "*write once, run everywhere*" philosophy of Java [47]. The *cons* against the use of Java can be summarized as follows: (i) the need of a JVM for running the source code, instead of the execution of a compiled executable file; (ii) the availability of particular data structures and code statements that may speed up (at least at low levels) the execution of a particular task. The outlined *pros* and *cons* highlight the fact that there no programming language can be *a-propri* chosen as the best, since the choice of the programming language in a certain scenario may depend on several factors. Nevertheless, as discussed before, an analysis and comparison of different approaches and possibilities (even developing different parts of our proposed virtualization infrastructure through different programming languages, and establishing an inter-process data sharing through internal communication buses) represents an interesting research direction.

5.2 Experimental Setup

The Virtualizer has been tested on two different systems: (i) a laptop with an Intel i7-7700HQ CPU, running Ubuntu 20.10 OS; and (ii) a Raspberry Pi 3 Model B (RPi3) Single Board Computer (SBC), running Raspbian OS. In detail, the RPi3 has been chosen to evaluate the feasibility of the proposed approach with an IoT node, which could be potentially deployed *on-premise* in the scenarios mentioned in Section 1. In fact, the RPi3 is widely adopted in the IoT arena and guarantees a very attractive trade-off among complexity, performance, and cost. In other words, the RPi3 can be interpreted as a "technology enabler" for the IoT. The LoRaWAN end nodes are based on STM STEVAL-STRKT01 LoRa devices [48], equipped with a Cortex M0+ CPU and different sensors (e.g., temperature, humidity, accelerometer) and interfaces (e.g., GNSS). Due to their portability, the LoRaWAN end nodes (both static or mobile) can be placed flexibly inside a production plant, as well as worn by workers for safety monitoring purposes. Finally, both LoRaWAN GW and NS run on top of two RPi3 boards, with one acting as NS running an open source software implementation denoted as *lorawan-server* [49].

Although the proposed virtualization architecture has been tested also using the *public* LoRaWAN network "The Things Network" (TTN) [50] (thus confirming the modularity of the proposed container-based approach and demonstrating its independence from a specific network provider), in order to fully control the LoRaWAN network's settings the performance evaluation has been carried out on a *private* LoRaWAN network. This also highlights how a company may benefit (from both technical and economical sides) from deploying its own LoRaWAN network to support its transition toward IoT.

In order to complement the experimental performance analysis compliant with the duty-cycle limitations of the EU ISM bands, a simulation environment has been developed (at application layer) to investigate the behavior of the proposed virtualization approach. This has been achieved by deploying a Java-based application (later denoted as *node simulator*) composed of the following software modules: (i) a module (identified as *Creator* in Fig. 6) corresponding to a CoAP client triggering the Virtualizer to start a variable number of new vENs as CoAP servers; and (ii) a module (identified as *Updater* in Fig. 6) updating the values of the vENs' resources.

As shown in Fig. 6, the experimental setup is composed of the node simulator (shown on the left side of Fig. 6) con-

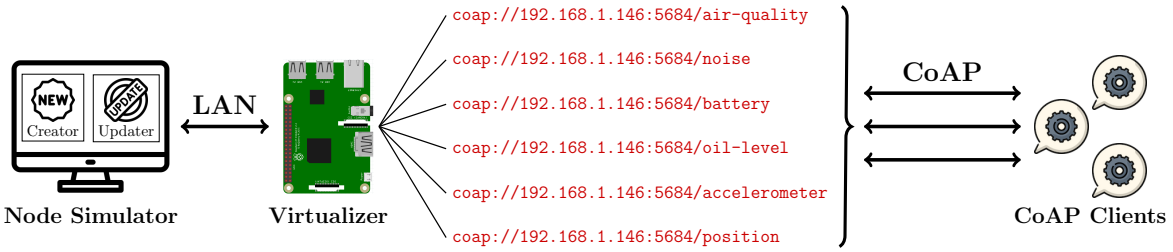

Fig. 6: Setup for the experimental evaluation of the proposed virtualization architecture, with the Virtualizer hosted on a RPi3. The experimental performance evaluation has also been carried out hosting the Virtualizer on a laptop.

nected via an Ethernet-based Local Area Network (LAN) to the Virtualizer. In Fig. 6, the Virtualizer is implemented on a RPi3, but the system performance with the Virtualizer running on a laptop has also been evaluated (this is not shown in Fig. 6 for simplicity). Then, external CoAP clients (shown on the right side of Fig. 6) are started with two settings: (i) set in *observing* mode on the CoAP resources or (ii) set in *polling* mode on the CoAP resources, thus emitting periodic CoAP `GET` requests. In *observing* mode, the value of a CoAP resource is automatically sent from the vEN to the external "listening" CoAP client each time the vEN's CoAP resource is updated—in detail, "stimulated" by the Updater (inside the node simulator) triggering the Virtualizer. In the experimental evaluation, we compare (i) the CPU percentage utilization by the Virtualizer hosted on laptop or RPi3 and (ii) the RAM utilization by the Virtualizer hosted on the laptop—as motivated in the following, due to some restrictions the RPi3-based implementation of the Virtualizer does not allow to evaluate the RAM utilization. In particular, the performance differences between *observing* and *polling* strategies are *first* evaluated and, *then*, the different behaviors of "parallel container" and "parallel vEN" approaches are investigated. In detail, both performance indicators are retrieved through the `docker stats` command-line tool given by the Docker daemon—this tool does not provide information on the RAM on the RPi3. In order to evaluate the CPU percentage utilization, in all the experimental campaigns the vENs associated with the LoRaWAN-enabled IoT nodes (based on STM STEVAL-STRKT01) are considered, with the sensors exposed as resources according to Table 2.

5.3 CPU Percentage Utilization

The first performance evaluation has been carried out analyzing the CPU percentage utilization on both laptop and RPi3. As shown in Fig. 7, the containerized architecture does not require significant resources. In detail, three phases can be identified in both experimented processing platforms: (i) JVM loading; (ii) resource discovery, in which the CoAP resources are added to the vEN; and (iii) observe notifications, in which a vEN sends updates to the external *observing* CoAP client(s)—we only refer to *observing* because both *observing* and *polling* modes lead the same CPU utilization. From the obtained experimental results, it is clear how the first phase is the most expensive for

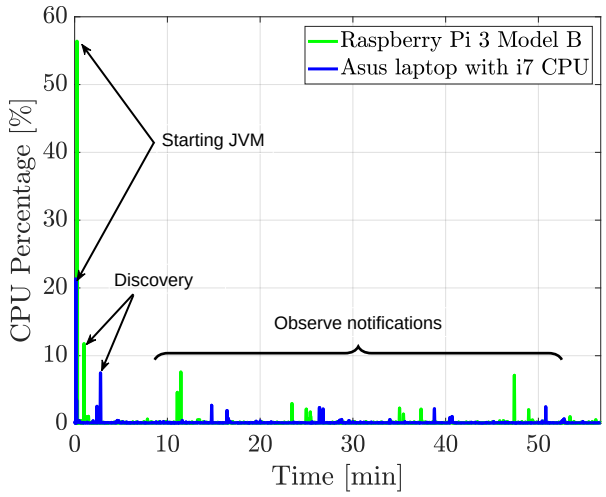

Fig. 7: Experimental CPU percentage utilization on laptop (ASUS with i7 CPU) and RPi3.

both laptop and RPi3, as both the devices present CPU percentage utilization's peaks. Moreover, considering the CPU peaks associated with JVM loading (20% on the laptop and 58% on the RPi3), we can conclude that if multiple physical nodes are virtualized at the same time and one container is started for each vEN, the CPU percentage utilization should be higher than 100% and, thus, the system would immediately crash.

Focusing on the JVM loading time, one can observe that it is up to 3 s on the laptop and up to 8 s on the RPi3. Therefore, on the RPi3, if two nodes are discovered in less than 10 s, then the CPU would saturate (it should reach a CPU percentage utilization equal to 116%). Hence, this motivates the need to define a "guard (time) interval." During performance evaluation, we discovered that the laptop (given the specific CPU equipping the laptop itself) can start up to 20 containers/JVMs without any problem; the RPi3 begins instead to be unstable if we run 3 containers/JVMs at the same time. Therefore, in order to be more conservative, the guard interval for each newly discovered device has been set to 5 s on the laptop and to 10 s on the RPi3. Finally, considering that a LoRaWAN network is "quasi-static" (with sporadic transmissions and almost static network topology, even in the presence of mobile

nodes in a limited region), this delay in the discovery phase should not represent a relevant issue and could be tolerated in real IoT deployments.

We make the following final observations. The high CPU consumption (e.g., in terms of percentage utilization peaks) in the RPi3 is likely due to the chosen programming language: Java guarantees good performance at the cost of higher resource consumption. If small networks (handled by a RPi3-like board) are considered, then the number of devices to be discovered is likely to be smaller. In the presence of denser and wider networks, with hundreds of devices, the Virtualizer needs to run on a dedicated server machine (either in the Cloud or *on-premise*) with higher computational power. Considering the discovery phase, the results are comparable: the laptop's CPU percentage utilization ratio is around 8%, whereas the RPi3's CPU percentage utilization is around 14%. Finally, focusing on the *observing* notifications, the interactions with the external CoAP clients result in a CPU percentage utilization in the interval $2\% \div 8\%$ for both laptop and RPi3.

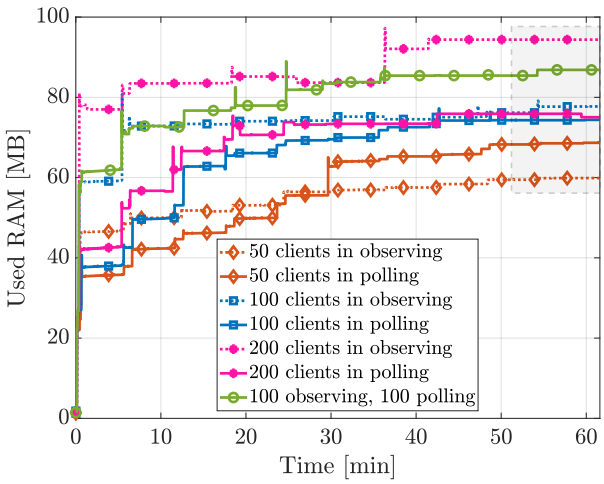

Fig. 8: Experimental RAM utilization comparison (on an ASUS laptop with i7 CPU), splitting the CoAP clients in *observing* and *polling*.

5.4 Comparison between Observing and Polling Strategies from External CoAP Clients

Given the experimental setup shown in Fig. 6, we investigate the scalability of the proposed container-based virtualization architecture. In detail, we set a variable number of external CoAP clients (selected among 50, 100, and 200) and we assume that each of them retrieves all the CoAP resources from the Virtualizer. Hence, we exploit both *observing* and *polling* approaches with the same update interval: (i) in *observing* mode, the *Updater* module (with reference to Fig. 6) updates the value of each CoAP resource every 10 min (which may be a reasonable interval in IoT scenarios, in the presence of non-critical monitoring tasks), whereas (ii) in *polling* mode, CoAP `GET` requests are sent by external clients every 10 min—this allows to perform a fair comparison with the *observing* mode.

As shown in Fig. 8, with 50 external CoAP clients, *observing* and *polling* modes return different results. In the beginning of the experimental evaluation, the *observing* setup requires an increased RAM utilization, while after three rounds of updates—we recall that, every 10 min, the CoAP resources are updated—the required RAM amount basically converges (there are small increments from then on). At the opposite, RAM utilization increases at each round with CoAP resource *polling*: after three rounds, it is higher than in *observing* mode. Considering the experiment with 100 external CoAP clients, in the beginning the difference between *observing* and *polling* modes is quite relevant. However, also in this case the RAM required in *observing* mode oscillates, with limited fluctuations, around 73 MB. In *polling* mode, instead, the RAM utilization slowly increases and, eventually, reaches the same value as in *observing* mode. Finally, the evaluation with 200 external CoAP clients shows that *polling* mode leads to the same performance as in the experiment with 100 clients, while the *observing* mode requires an higher RAM occupation with 200 external CoAP clients then with 100.

Therefore, the results shown in Fig. 8 highlight that: (i) with 50 external CoAP clients, it is preferrable to set the *observing* mode for all the clients; (ii) with 100 CoAP clients, there are no significant differences between *observing* and *polling* modes; and (iii) with 200 (or more) CoAP clients, it is convenient to adopt the *polling* mode. A good trade-off, e.g., because the resources are not regularly updated (we recall that, in *observing* mode, a notification is sent only when the CoAP resource is updated), would be to set a portion of the external CoAP clients in *polling* mode and the remaining ones in *observing* mode. In fact, looking at the case with 200 external CoAP clients, since the performance in both *polling* and *observing* modes is the same as in the case with 100 nodes, the best solution would be to evenly split the 200 clients, as shown in Fig. 8. As expected, this "network optimization" leads to a performance between the one with all 200 CoAP clients set in *observing* mode and the one with all 200 CoAP clients set in *polling* mode.

5.5 Comparison Between Virtualization Approaches: "Parallel Container" versus "Parallel vEN"

The risk of CPU saturation due to simultaneous virtualization of multiple nodes can be solved running multiple vENs in the same container. As shown in Fig. 7, most of the CPU is required during the JVM loading. Therefore, an effective approach is to exploit the already running JVM and start several vENs in the same JVM. In this experimental evaluation, we *first* start one container with one vEN. Then, with a pre-set activation interval of 5 min (expedient to highlight the "RAM steps" in Fig. 9), we start a new vEN, reaching the final configuration with 5 vENs. This has been repeated with 10, 15, and 20 vENs.

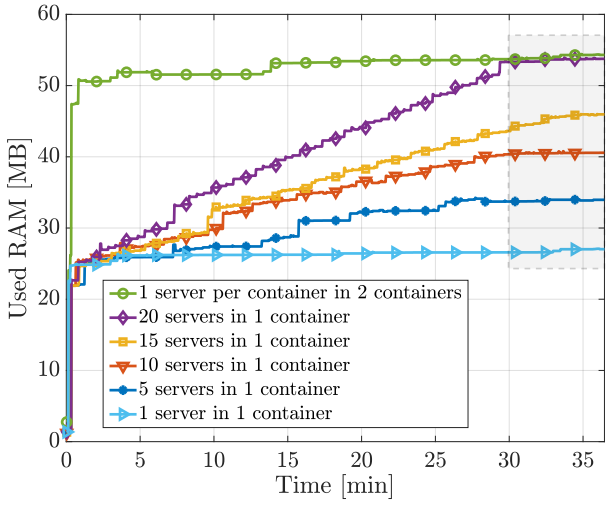

Fig. 9: Experimental comparison between RAM memory occupation (on an ASUS laptop with i7 CPU) with "parallel container" and "parallel vEN" virtualization approaches.

In order to extend our analysis, we start two containers, each running a vEN, and evaluate the total occupied memory by adding the occupied RAM of each single container. The results, shown in Fig. 9, can be commented focusing on the steady-state regime (highlighted in gray). The setup with 20 vENs per container is comparable to the scenario with two vENs running in two separated containers. This is reasonable, since, when a vEN is started, most of the memory (around 25 MB) is occupied by the JVM: running two containers would thus lead to a utilization of 50 MB only for the JVMs. In order to obtain the same RAM utilization with a single container, indicating the memory occupied by a vEN as $\mathrm{vEN}_{\mathrm{RAM}}$, the optimal number of vENs per container can be expressed as follows:

$$N_{\mathrm{vEN}} = \left\lfloor \frac{2\mathrm{JVM}_{\mathrm{RAM}} - \mathrm{JVM}_{\mathrm{RAM}}}{\mathrm{vEN}_{\mathrm{RAM}}} \right\rfloor . \quad (1)$$

Under the assumption that $\mathrm{vEN}_{\mathrm{RAM}} = 1.2$ MB, then $N_{\mathrm{vEN}} = 20$.

Therefore, one can conclude that, in the case of constrained environments (e.g., in IoT contexts), a feasible approach would be to start more containers, each of them running multiple vENs, and possibly grouping vENs with the same subset of functionalities/services.

Finally, we comment on the challenges at each "sub-layer" of the proposed container-based virtualization architecture (e.g., private/public "core" LoRaWAN network, virtualization layer, and CoAP-based layer) [51]. No particular challenges are envisioned in any of them, since: (i) known security aspects and vulnerabilities of "pure" LoRaWAN networks [52,53] are not worsened by the virtualization layer and have still to be managed in the "core" LoRaWAN layer; (ii) threats and vulnerabilities related to virtualization platforms and *orchestrators* [54] do not affect the LoRaWAN component; and (iii) the application layer suffers from well-known CoAP-related vulnerabilities [55], that should be dealt with at this layer. In other words, the proposed virtualization architecture does not introduce vulnerabilities in any "sub-layer."

5.6 Performance Comparison Between Data Formats

We now investigate, in a comparative way, how the data formats discussed in Section 2 (namely: CLPP, presented in Subsection 2.2; JSON, presented in Subsection 2.3.1; XML, presented in Subsection 2.3.2; and CoRE Link Format, presented in Subsection 2.3.3) adapt to the constraints of the LoRaWAN protocol, in terms of information encoding/decoding between *on-field* LoRaWAN end devices and high-layer entities.

In order to compare the considered data formats, we assume (according to the assumptions adopted for CLPP in Subsection 2.2 and with reference to the information contained in Fig. 2) that each "raw packet" contains 1 byte for the data channel, 1 byte for the data type, and 1 byte for the sensor data (e.g., humidity). We denote the number of required bytes of data format $d \in \{\mathrm{CLPP}, \mathrm{JSON}, \mathrm{XML}, \mathrm{CoRE}\}$ as L_{bytes_d}. In detail, since CLPP does not require any delimiter, its corresponding number of required bytes $L_{\mathrm{bytes}_{\mathrm{CLPP}}}$ can be expressed as

$$L_{\mathrm{bytes}_{\mathrm{CLPP}}} = 3 \cdot n_{\mathrm{raw}} . \quad (2)$$

With regard to JSON, as shown in Listing 1, it requires 2 external square brackets. Then, for each raw packet to be encoded, JSON needs 2 curly brackets (enclosing the raw packet as a JSON object), 26 bytes for encoding the raw packet itself, and, optionally, a comma for each represented raw packet (except for the last one), should more raw packets be encoded. Then, its corresponding number of required bytes $L_{\mathrm{bytes}_{\mathrm{JSON}}}$ can be expressed as follows:

$$\begin{aligned} L_{\mathrm{bytes}_{\mathrm{JSON}}} &= 2 + 2 \cdot n_{\mathrm{raw}} + 26 \cdot n_{\mathrm{raw}} + n_{\mathrm{raw}} - 1 \\ &= 29 \cdot n_{\mathrm{raw}} + 1 . \end{aligned} \quad (3)$$

Concerning XML, as shown in Listing 2, it requires: a 3-byte opening XML root element and a corresponding 4-byte closing XML root element; and 23 bytes for encoding each raw packet. Then, its corresponding number of required bytes $L_{\mathrm{bytes}_{\mathrm{XML}}}$ can be expressed as follows:

$$\begin{aligned} L_{\mathrm{bytes}_{\mathrm{XML}}} &= 3 + 4 + 23 \cdot n_{\mathrm{raw}} \\ &= 23 \cdot n_{\mathrm{raw}} + 7 . \end{aligned} \quad (4)$$

Finally, with regard to CoRE Link Format, as shown in Listing 3, given the notation assumption on the resource name starting with `/r` and followed by an incremental index (as detailed in Subsection 2.3.3), it requires a number of bytes $\alpha \in \{27, 28\}$ for encoding each raw packet—$\alpha = 27$ for the first 9 raw packets, and $\alpha = 28$ for the following ones—and, optionally, a comma for each represented raw packet (except for the last one), should more raw packets

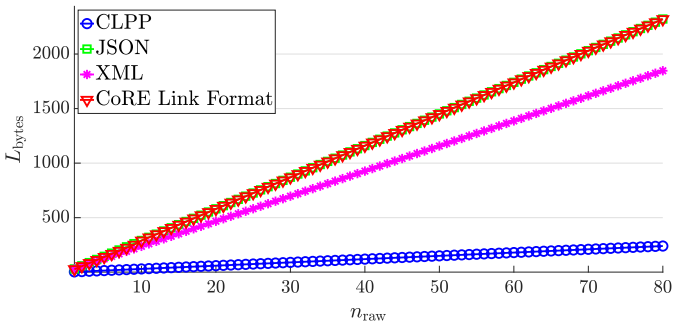

Fig. 10: Number of required bytes as a function of the number of raw packets to be sent, for the chosen data formatting mechanisms: CLPP, JSON, XML, CoRE Link Format.

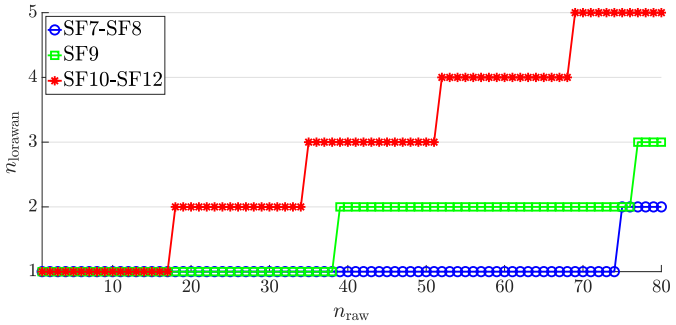

Fig. 11: Number of CLPP-encoded LoRaWAN packets as a function of the number of raw packets to be sent, for various values of the SF.

be encoded. Then, its corresponding number of required bytes $L_{\text{bytes}_{\text{CoRE}}}$ can be expressed as follows:

$$\begin{aligned} L_{\text{bytes}_{\text{CoRE}}} &= \alpha \cdot n_{\text{raw}} + n_{\text{raw}} - 1 \\ &= (\alpha + 1) \cdot n_{\text{raw}} - 1 \,. \end{aligned} \tag{5}$$

In Fig. 10, the number of required bytes L_{bytes} is shown as a function of the number of raw packets n_{raw} to be sent, for the chosen data formatting mechanisms (CLPP, JSON, XML, CoRE Link Format), according to Eqs. (2)–(5). Fig. 11, Fig. 12, Fig. 13, and Fig. 14 show the total number of LoRaWAN packets needed (denoted as n_{LoRaWAN}), as a function of of the number of raw packets n_{raw} to be sent, in the cases with CLPP, JSON, XML, and CoRE Link Format, respectively. For the sake of comparison, we show them considering the chosen LoRaWAN SFs, denoting the maximum allowed payload for each LoRaWAN SF as ℓ_{SF}, and taking into account that: SF7 and SF8 allow a maximum 222-byte payload ($\ell_{\text{SF7}} = \ell_{\text{SF8}} = 222$); SF9 allows a maximum 115-byte payload ($\ell_{\text{SF9}} = 115$); SF10, SF11, and SF12 allow a maximum 51-byte payload ($\ell_{\text{SF10}} = \ell_{\text{SF11}} = \ell_{\text{SF12}} = 51$). Therefore, the total number of LoRaWAN packets, as a function of the number of raw packets to be sent, for the chosen data representation format $d \in \{\text{CLPP}, \text{JSON}, \text{XML}, \text{CoRE}\}$, and the chosen LoRaWAN SF $\in \{\text{SF7}, \text{SF8}, \ldots, \text{SF12}\}$, can be expressed as follows:

$$n^{(\text{SF})}_{\text{LoRaWAN}_d} = \left\lceil \frac{L_{\text{bytes}_d}}{\ell_{\text{SF}}} \right\rceil . \tag{6}$$

In order to further investigate the impact of the considered data formats, we compare directly the encoding efficiencies of the various data formats for a given SF. In Fig. 15, Fig. 16, and Fig. 17, the number of LoRaWAN packets is shown as a function of the number of raw packets in the cases with SF7-SF8, SF9, and SF10-SF12, respectively.

From all the results presented, it clearly emerges that CLPP is the most efficient data formatting strategy, requiring the smallest number of LoRaWAN packets regardless of the chosen LoRaWAN SF, especially for a large number of raw packets to be sent.

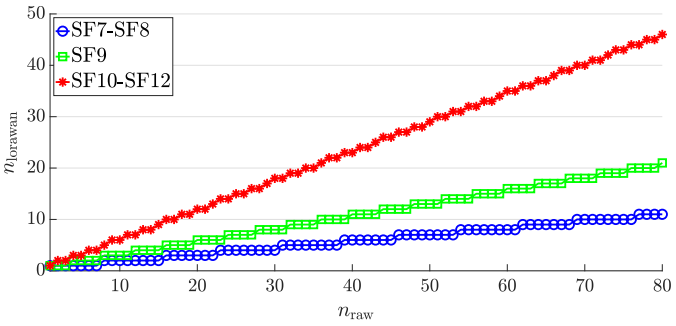

Fig. 12: Number of JSON-encoded LoRaWAN packets as a function of the number of raw packets to be sent, for various values of the SF.

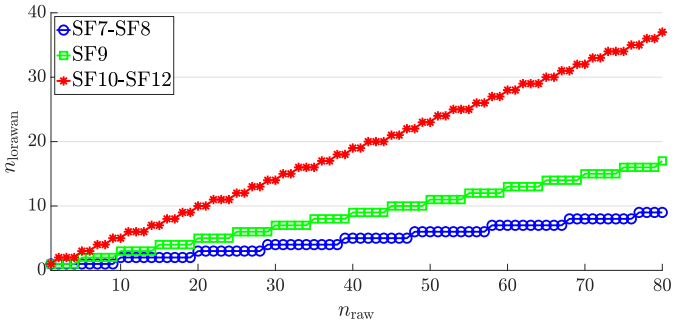

Fig. 13: Number of XML-encoded LoRaWAN packets as a function of the number of raw packets to be sent, for various values of the SF.

6 Conclusions

In this work, a container-based virtualization architecture for IoT scenarios with LoRaWAN nodes has been presented. Each LoRaWAN end node is virtualized as a CoAP server, denoted as vEN, in a central Virtualizer entity interacting with the LoRaWAN NS. The sensors equipping each LoRaWAN end node can be *discovered* by analyzing its packets' payload (encoded according to CLPP format) and *emulated* as CoAP resources associated with the vEN representing the virtualized version of the LoRaWAN end node. In this way, following a digital twin-like strategy, an IP-based external CoAP client can virtually interact

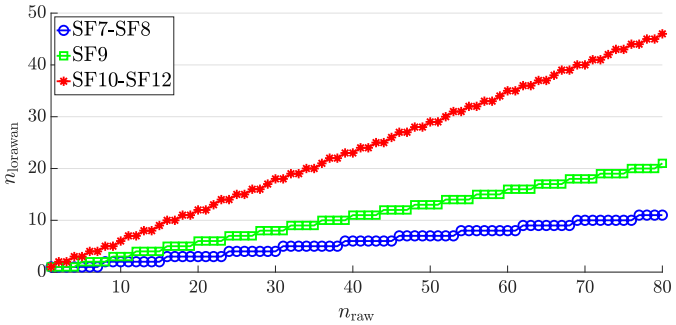

Fig. 14: Number of CoRE Link Format-encoded LoRaWAN packets as a function of the number of raw packets to be sent, for various values of the SF.

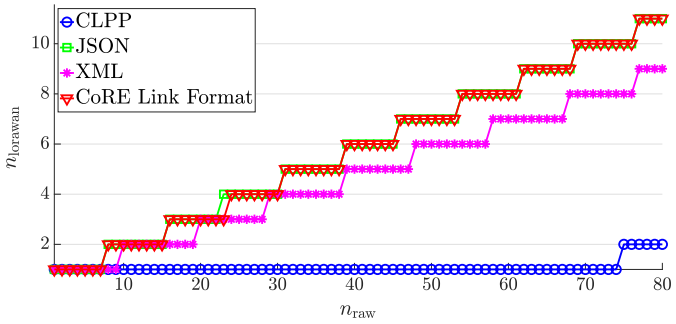

Fig. 15: Number of LoRaWAN packets as a function of the number of raw packets to be sent adopting a LoRaWAN SF equal to SF7-SF8 (sharing the same maximum payload length $\ell_{\text{SF7}} = \ell_{\text{SF8}} = 222$ bytes), shown for each considered data representation format.

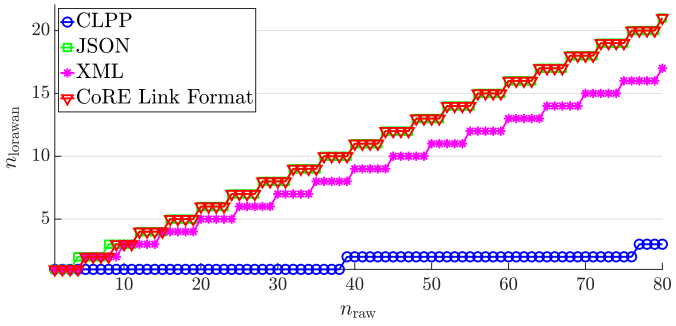

Fig. 16: Number of LoRaWAN packets as a function of the number of raw packets to be sent adopting a LoRaWAN SF9 (having a maximum payload length $\ell_{\text{SF9}} = 115$ bytes), shown for each considered data representation format.

with a LoRaWAN end node in both uplink and downlink directions.

The proposed virtualization approach allows both communication and routing among LoRaWAN nodes, thus overcoming the native impossibility of inter-node communication (imposed by the LoRaWAN's specifications) and simplifying H2M and M2M approaches. In fact, the proposed LoRaWAN-to-IP conversion does not affect the original LoRaWAN stack and has no impact on the performance of the LoRaWAN end nodes, in terms of power

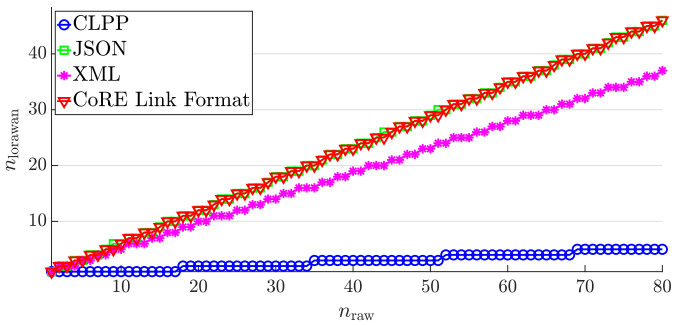

Fig. 17: Number of LoRaWAN packets as a function of the number of raw packets to be sent adopting a LoRaWAN SF equal to SF10-SF12 (sharing the same maximum payload length $\ell_{\text{SF10}} = \ell_{\text{SF11}} = \ell_{\text{SF12}} = 51$ bytes), shown for each considered data representation format.

and operational requirements (e.g., LoRaWAN's Class A, B, or C, as well as SF value). To this end, the LoRaWAN network's behavior is completely transparent to external CoAP clients, which are unaware of the real nature of the entities hosting sensors and sending updated values to their neighboring LoRaWAN GWs. The proposed approach can be further extended to other types of networks and application layer protocols, thus making the integration of heterogeneous IoT devices feasible. For example, it could be possible to define proper adaptation and protocol translation mechanisms to enable HTTP-based clients to interact with the LoRaWAN environment: this could be the case of a control system in an industrial environment.

We remark that the focus of the proposed virtualization architecture is not on the specific adopted virtualization software, but, rather, on the virtualization of the functionalities of each LoRaWAN end node, thus exploiting a Virtual Network Function (VNF)-like strategy. Our experimental evaluation has highlighted the feasibility of the proposed virtualization architecture on a RPi3, a widely adopted IoT SBC. In general, two virtualization approaches have been compared: (i) running one vEN per container and (ii) running multiple vENs per container. The main finding is that the best solution is hybrid: there should be several containers, each running more than one vEN. Finally, the proposed container-based architecture has the advantage of being compliant with standard protocols, such as CoAP, which is relevant in IoT-oriented scenarios. To this end, further research activities (besides those already mentioned in the manuscript) will focus on comparisons between the current architecture and an equivalent implementation where CoAP is replaced by MQTT, another widely adopted communication protocol in IoT scenarios.

Acknowledgments

The work of L. Davoli and G. Ferrari received funding from the European Union's Horizon 2020 research and innovation program ECSEL Joint Undertaking (JU) under grant agreements: No. 876019, ADACORSA project -

"Airborne Data Collection on Resilient System Architectures;" No. 876038, InSecTT project - "Intelligent Secure Trustable Things." It has also received funding from the European Union's Horizon Europe research and innovation program Key Digital Technology (KDT) JU under grant agreement No. 101097267, OPEVA project - "OPtimization of Electric Vehicle Autonomy." Finally, we acknowledge also partial support from the Agritech project - "National Research Centre for Agricultural Technologies," project code CN00000022, funded under the National Recovery and Resilience Plan (NRRP), Mission 4 Component 2 Investment 1.4 - Call for tender no. 3138 of 16/12/2021 of Italian Ministry of University and Research funded by the European Union – NextGenerationEU, Concession Decree no. 1032 of 17/06/2022 adopted by the Italian Ministry of University and Research. The ECSEL/KDT JUs received support from the European Union's Horizon 2020/Horizon Europe research and innovation programme and the nations involved in the mentioned projects. The work reflects only the authors' views; the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] M. Mangia *et al.*, "Rakeness-based compressed sensing and hub spreading to administer short/long-range communication tradeoff in iot settings," *IEEE Internet of Things Journal*, vol. 5, no. 3, pp. 2220–2233, 2018, doi:10.1109/JIOT.2018.2828647.

[2] LoRa Alliance, "LoRaWAN Specification," Accessed on August 16, 2023. [Online]. Available: https://tinyurl.com/lwspec11

[3] ETSI Technical Committee Electromagnetic compatibility and Radio spectrum Matters (ERM), "Data Transmission Systems using Wide Band technologies in the 2,4 GHz band," Technical Report, ETSI, SRD ETSI TR 103 665, May 2021. [Online]. Available: https://tinyurl.com/etsiism

[4] M. Centenaro *et al.*, "Long-Range Communications in Unlicensed Bands: The Rising Stars in the IoT and Smart City Scenarios," *IEEE Wireless Communications*, vol. 23, no. 5, pp. 60–67, Oct 2016, doi:10.1109/MWC.2016.7721743.

[5] D. Magrin, M. Centenaro, and L. Vangelista, "Performance Evaluation of LoRa Networks in a Smart City Scenario," in *2017 IEEE International Conference on Communications (ICC)*, Paris, France, May 2017, pp. 1–7, doi:10.1109/ICC.2017.7996384.

[6] F. Mason, M. Capuzzo, D. Magrin, F. Chiariotti, A. Zanella, and M. Zorzi, "Remote Tracking of UAV Swarms via 3D Mobility Models and LoRaWAN Communications," *IEEE Transactions on Wireless Communications*, vol. 21, no. 5, pp. 2953–2968, 2022, doi:10.1109/TWC.2021.3117142.

[7] L. Davoli, E. Pagliari, and G. Ferrari, "Hybrid LoRa-IEEE 802.11s Opportunistic Mesh Networking for Flexible UAV Swarming," *Drones*, vol. 5, no. 2, 2021, doi:10.3390/drones5020026.

[8] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," Internet Requests for Comments, IETF, RFC 7252, Jun 2014. [Online]. Available: https://tools.ietf.org/html/rfc7252

[9] R. Morabito, "Virtualization on Internet of Things Edge Devices With Container Technologies: A Performance Evaluation," *IEEE Access*, vol. 5, pp. 8835–8850, May 2017, doi:10.1109/ACCESS.2017.2704444.

[10] S. Mihai *et al.*, "Digital Twins: A Survey on Enabling Technologies, Challenges, Trends and Future Prospects," *IEEE Communications Surveys & Tutorials*, vol. 24, no. 4, pp. 2255–2291, 2022, doi:10.1109/COMST.2022.3208773.

[11] R. Marini *et al.*, "Low-Power Wide-Area Networks: Comparison of LoRaWAN and NB-IoT Performance," *IEEE Internet of Things Journal*, pp. 1–1, 2022, doi:10.1109/JIOT.2022.3176394.

[12] A. Lombardo *et al.*, "LoRaWAN Versus NB-IoT: Transmission Performance Analysis Within Critical Environments," *IEEE Internet of Things Journal*, vol. 9, no. 2, pp. 1068–1081, 2022, doi:10.1109/JIOT.2021.3079567.

[13] M. Stusek *et al.*, "LPWAN Coverage Assessment Planning Without Explicit Knowledge of Base Station Locations," *IEEE Internet of Things Journal*, vol. 9, no. 6, pp. 4031–4050, 2022, doi:10.1109/JIOT.2021.3102694.

[14] Z. Zhang *et al.*, "ZCNET: Achieving High Capacity in Low Power Wide Area Networks," *IEEE/ACM Transactions on Networking*, pp. 1–14, 2022, doi:10.1109/TNET.2022.3158482.

[15] K. Staniec and M. Kowal, "LoRa Performance under Variable Interference and Heavy-Multipath Conditions," *Wireless Communications and Mobile Computing*, vol. 2018, pp. 1–9, 2018, doi:10.1155/2018/6931083.

[16] Q. M. Qadir, "Analysis of the Reliability of LoRa," *IEEE Communications Letters*, vol. 25, no. 3, pp. 1037–1040, 2021, doi:10.1109/LCOMM.2020.3034865.

[17] "LoRa Alliance," Accessed on August 16, 2023. [Online]. Available: https://lora-alliance.org/

[18] "Semtech Corporation," Accessed on August 16, 2023. [Online]. Available: https://www.semtech.com/

[19] "European Telecommunications Standards Institute (ETSI)," Accessed on August 16, 2023. [Online]. Available: https://www.etsi.org/

[20] Cayenne, "How Cayenne LPP works," . [Online]. Available: https://developers.mydevices.com/cayenne/docs/intro/

[21] J. Jimenez, M. Koster, and H. Tschofenig, "IPSO Smart Objects," in *IoT Semantic Interoperability Workshop 2016*, San Jose, California, Mar 2016, pp. 1–7. [Online]. Available: https://www.iab.org/wp-content/IAB-uploads/2016/03/ipso-paper.pdf

[22] A. Cilfone, L. Davoli, and G. Ferrari, "Virtualizing LoRaWAN Nodes: a CoAP-based Approach," in *2019 International Symposium on Advanced Electrical and Communication Technologies (ISAECT)*, Rome, Italy, Nov 2019, pp. 1–6, doi:10.1109/ISAECT47714.2019.9069691.

[23] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," Internet Requests for Comments, IETF, RFC 8259, Dec 2017. [Online]. Available: https://tools.ietf.org/html/rfc8259

[24] World Wide Web Consortium (W3C), "Extensible Markup Language (XML) 1.0," Accessed on August 16, 2023. [Online]. Available: https://www.w3.org/TR/xml/

[25] M. Nottingham, "Web Linking," Internet Requests for Comments, IETF, RFC 8288, Oct 2017. [Online]. Available: https://tools.ietf.org/html/rfc8288

[26] Z. Shelby, "Constrained RESTful Environments (CoRE) Link Format," Internet Requests for Comments, IETF, RFC 6690, Aug 2012. [Online]. Available: https://tools.ietf.org/html/rfc6690

[27] E. Kristiani *et al.*, "The Implementation of an Edge Computing Architecture with LoRaWAN for Air Quality Monitoring Applications," in *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*. Springer International Publishing, 2020, pp. 210–219, doi:10.1007/978-3-030-52988-8_19.

[28] G. Dandachi and Y. Hadjadj-Aoul, "A Frequency-Based Intelligent Slicing in LoRaWAN with Admission Control Aspects," in *Proceedings of the International Conference on Modeling Analysis and Simulation of Wireless and Mobile Systems on International Conference on Modeling Analysis and Simulation of Wireless and Mobile Systems*, Montreal, QC, Canada, 2022, doi:10.1145/3551659.3559055.

[29] R. Yasmin *et al.*, "On the integration of lorawan with the 5g test network," in *2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, Montreal, QC, Canada, 2017, pp. 1–6, doi:10.1109/PIMRC.2017.8292557.

[30] J. Navarro-Ortiz *et al.*, "Integration of LoRaWAN and 4G/5G for the Industrial Internet of Things," *IEEE Communications Magazine*, vol. 56, no. 2, pp. 60–67, 2018, doi:10.1109/MCOM.2018.1700625.

[31] F. Flammini *et al.*, "Virtualization Technology for LoRaWAN Roaming Simulation in Smart Cities," in *Studies in Computa-*

tional Intelligence. Springer International Publishing, 2021, pp. 251–265, doi:10.1007/978-3-030-72065-0_14.
[32] G. Tanganelli, C. Vallati, and E. Mingozzi, "Edge-Centric Distributed Discovery and Access in the Internet of Things," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 425–438, 2018, doi:10.1109/JIOT.2017.2767381.
[33] S. Cirani *et al.*, "A scalable and self-configuring architecture for service discovery in the internet of things," *IEEE Internet of Things Journal*, vol. 1, no. 5, pp. 508–521, Oct 2014, doi:10.1109/JIOT.2014.2358296.
[34] L. Rodrigues, J. Guerreiro, and N. Correia, "Resource design in federated sensor networks using RELOAD/CoAP overlay architectures," *Computer Communications*, vol. 179, pp. 11–21, 2021, doi:10.1016/j.comcom.2021.07.019.
[35] L. Belli *et al.*, "A Novel Smart Object-Driven UI Generation Approach for Mobile Devices in the Internet of Things," in *Proceedings of the 1st International Workshop on Experiences with the Design and Implementation of Smart Objects*, ser. SmartObjects '15, Paris, France, 2015, pp. 1—6, doi:10.1145/2797044.2797046.
[36] OASIS Open, "MQ Telemetry Transport (MQTT) Specifications," Accessed on August 16, 2023. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html
[37] "Docker," Accessed on August 16, 2023. [Online]. Available: https://www.docker.com/
[38] M. D. Day, C. E. Perkins, J. Veizades, and E. Guttman, "Service Location Protocol, Version 2," Internet Requests for Comments, IETF, RFC 2608, Jun 1999. [Online]. Available: https://tools.ietf.org/html/rfc2608
[39] Zero Configuration Networking (Zeroconf), "http://www.zeroconf.org/," 1999.
[40] S. Cheshire and M. Krochmal, "Multicast DNS," Internet Requests for Comments, IETF, RFC 6762, Feb 2013. [Online]. Available: https://tools.ietf.org/html/rfc6762
[41] ——, "DNS-Based Service Discovery," Internet Requests for Comments, IETF, RFC 6763, Feb 2013. [Online]. Available: https://tools.ietf.org/html/rfc6763
[42] I. Fette and A. Melnikov, "The WebSocket Protocol," Internet Requests for Comments, IETF, RFC 6455, Dec 2011. [Online]. Available: https://tools.ietf.org/html/rfc6455
[43] L. Davoli *et al.*, "Integration of Wi-Fi mobile nodes in a Web of Things Testbed," *ICT Express*, vol. 2, no. 3, pp. 95–99, 2016, special Issue on ICT Convergence in the Internet of Things (IoT).
[44] Eclipse Foundation, "Eclipse Californium (Cf) CoAP Framework," Accessed on August 16, 2023. [Online]. Available: https://eclipse.dev/californium/
[45] ——, "MQTT Paho," Accessed on August 16, 2023. [Online]. Available: https://www.eclipse.org/paho/clients/java/
[46] G. Fu, Y. Zhang, and G. Yu, "A Fair Comparison of Message Queuing Systems," *IEEE Access*, vol. 9, pp. 421–432, 2021, doi:10.1109/ACCESS.2020.3046503.
[47] A. Stratikopoulos *et al.*, "Transparent Acceleration of Java-Based Deep Learning Engines," in *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes*, ser. MPLR '20, Virtual, UK, 2020, pp. 73–79, doi:10.1145/3426182.3426188.
[48] STMicroelectronics, "STEVAL-STRKT01 LoRa IoT Tracker," Accessed on August 16, 2023. [Online]. Available: https://www.st.com/en/evaluation-tools/steval-strkt01.html
[49] Petr Gotthard, "Compact LoRaWAN Network Server for private LoRaWAN networks," Accessed on August 16, 2023. [Online]. Available: https://github.com/gotthardp/lorawan-server
[50] The Things Industries, "The Things Network (TTN)," Accessed on August 16, 2023. [Online]. Available: https://www.thethingsnetwork.org/
[51] E. Guttman and N. Brownlee, "Expectations for Computer Security Incident Response," Internet Requests for Comments, IETF, RFC 2350, Jun 1998. [Online]. Available: https://tools.ietf.org/html/rfc2350
[52] M. Eldefrawy *et al.*, "Formal Security Analysis of LoRaWAN," *Computer Networks*, vol. 148, pp. 328–339, 2019, doi:10.1016/j.comnet.2018.11.017.
[53] R. Kloibhofer, E. Kristen, and L. Davoli, "LoRaWAN with HSM as a Security Improvement for Agriculture Applications," in *Computer Safety, Reliability, and Security. SAFECOMP 2020 Workshops*, A. Casimiro, F. Ortmeier, E. Schoitsch, F. Bitsch, and P. Ferreira, Eds., 2020, pp. 176–188, doi:10.1007/978-3-030-55583-2_13.
[54] A. Martin *et al.*, "Docker ecosystem – Vulnerability Analysis," *Computer Communications*, vol. 122, pp. 30–43, 2018, doi:10.1016/j.comcom.2018.03.011.
[55] I. Butun, P. Österberg, and H. Song, "Security of the Internet of Things: Vulnerabilities, Attacks, and Countermeasures," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 1, pp. 616–644, 2020, doi:10.1109/COMST.2019.2953364.

Antonio Cilfone received his M.Sc. in Communication Engineering and his Ph.D. in Information Technologies from the University of Parma, Parma, Italy, in 2016 and 2019, respectively. He has been member of the Internet of Things (IoT) Lab at the Department of Engineering and Architecture of the University of Parma from 2016 until 2020, working on heterogeneous networking, signal processing, and smart systems topics. He is currently working as R&D software engineer at Tesmec Automation s.r.l., Italy.

Luca Davoli (GS'15, M'17) is a non-tenured assistant professor at the Internet of Things (IoT) Laboratory, Department of Engineering and Architecture, University of Parma, Parma, Italy. He received his Dr. Ing. degree in Computer Engineering and his Ph.D. in Information Technologies at the Department of Information Engineering of the same university, in 2013 and 2017, respectively. His research interests focus on IoT, Pervasive Computing, Big Stream and Software-Defined Networking.

Gianluigi Ferrari (S'96–M'98–SM'12) received the Laurea (*summa cum laude*) and Ph.D. degrees in electrical engineering from the University of Parma, Parma, Italy, in 1998 and 2002, respectively. Since 2002, he has been with the University of Parma, where he is currently a full professor of telecommunications and also the coordinator of the Internet of Things (IoT) Laboratory, Department of Engineering and Architecture. His current research interests include signal processing, advanced communication and networking, and IoT and smart systems.