IoT

MDPI

Article

An Edge-Based LWM2M Proxy for Device Management to Efficiently Support QoS-Aware IoT Services

Martina Pappalardo [1,2,*], Antonio Virdis [2] and Enzo Mingozzi [2]

[1] Department of Information Engineering, University of Firenze, 50139 Firenze, Italy
[2] Department of Information Engineering, University of Pisa, 56122 Pisa, Italy; antonio.virdis@unipi.it (A.V.); enzo.mingozzi@unipi.it (E.M.)
* Correspondence: martina.pappalardo@unifi.it

Abstract: The Internet of Things (IoT) brings Internet connectivity to devices and everyday objects. This huge volume of connected devices has to be managed taking into account the severe energy, memory, processing, and communication constraints of IoT devices and networks. In this context, the OMA LightweightM2M (LWM2M) protocol is designed for remote management of constrained devices, and related service enablement, through a management server usually deployed in a distant cloud data center. Following the Edge Computing paradigm, we propose in this work the introduction of a LWM2M Proxy that is deployed at the network edge, in between IoT devices and management servers. On one hand, the LWM2M Proxy improves various LWM2M management procedures whereas, on the other hand, it enables the support of QoS-aware services provided by IoT devices by allowing the implementation of advanced policies to efficiently use network, computing, and storage (i.e., cache) resources at the edge, thus providing benefits in terms of reduced and more predictable end-to-end latency. We evaluate the proposed solution both by simulation and experimentally, showing that it can strongly improve the LWM2M performance and the QoS of the system.

Keywords: LWM2M Proxy; IoT; OMA LWM2M; QoS

Citation: Pappalardo, M.; Virdis, A.; Mingozzi, E. An Edge-Based LWM2M Proxy for Device Management to Efficiently Support QoS-Aware IoT Services. *IoT* **2022**, *3*, 169–190. https://doi.org/10.3390/iot3010011

Academic Editor: Andrea Vitaletti

Received: 24 January 2022
Accepted: 24 February 2022
Published: 26 February 2022

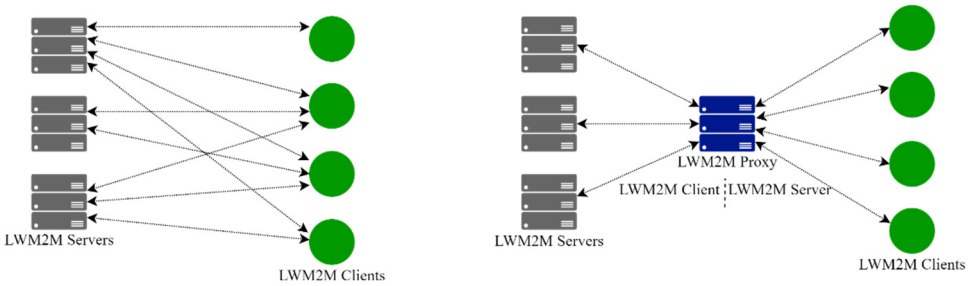
1. Introduction

The huge volume of IoT connected devices builds M2M systems that need to be managed considering the energy, memory, processing, and communication limitations of the devices themselves and the constraints imposed by the low power and lossy wireless networks they use to communicate. To simplify M2M deployments in these heterogeneous systems, there is the need for a common standard platform for managing devices.

Several standards responded to this demand. For example, the IETF CoRE WG proposed the introduction of an entity called Resource Directory (RD, [1]): CoAP servers register their resources with one or more RDs and CoAP clients look up resources in the RD. The TR-069 protocol [2] for remote device management was standardized by the Broadband Forum and it allows an Auto-Configuration Server to monitor, configure, and update a remote device. Another example is oneM2M [3]: this architecture defines a middleware between processing/communication hardware and the IoT applications, and it also supports device management. Finally, the Open Mobile Alliance (OMA) specified the LightweightM2M (LWM2M) protocol for device management and service enablement (see [4,5]). LWM2M defines a communication protocol between a LWM2M Server, i.e., the management application, and a LWM2M Client, i.e., the IoT device. The LWM2M protocol provides several benefits that make it suitable for constrained devices: (i) it is a simple and stateless protocol; (ii) it is appropriate for low powered battery devices because of its low client footprint; (iii) it has a small RAM requirement; (iv) it has a transport-agnostic design and an increased bandwidth efficiency based on CoAP bandwidth optimization; and, finally, (v) it can be used both for data plane and device management.

Although LWM2M was specifically created keeping in mind constrained devices and networks, enabling its functionalities in practice can be still challenging. The maximum frame size of some IoT network technologies can be in fact too small for a LWM2M message, moreover, multiple concurrent requests from different LWM2M Servers can easily overload device and network resources. In this work we thus propose an extension to the LWM2M standard architecture that aims at solving the limitations above, considering the characteristics of IoT devices and networks. Typically, IoT networks are accessed through IoT gateways or proxies that route application requests and device responses and that implement a number of extra functionalities to improve system performance [6]. We propose to introduce a LWM2M Proxy to be deployed at the network edge, between the LWM2M Clients and the LWM2M Servers. The LWM2M Proxy is aimed at overcoming the limits imposed by IoT devices and networks and to optimize the LWM2M device management procedures to avoid those resources from being overloaded. In the following, we refer to the LWM2M Client as Client, to the LWM2M Server as Server and to the LWM2M Proxy as Proxy.

LWM2M v1.2 [4] already introduced a LWM2M Gateway object that manages device(s) on behalf of the Server over LWM2M or non-LWM2M protocols. However, the LWM2M Gateway just forwards the management commands to the targeted device(s), and then, after it receives the responses, it notifies the Server about the command completion status. In this work, instead, we propose to introduce a Proxy that is designed to act smart by optimizing LWM2M management operations and by supporting QoS-aware services. In more detail, the Proxy breaks the communication between the Server and the Client into two segments acting as a Client towards the Servers and as a Server towards the Clients, as we show in Figure 1. Clients are resource-constrained, so it is possible they are able to serve only few entities and cannot communicate with all the required Servers; in our proposal, the Client communicates only with a single entity, i.e., the Proxy, deployed at the edge of the network; and the Servers are managed by the Proxy that can optimize the communication between them and the Clients. Indeed, many IoT scenarios involve multiple Clients and Servers. Consider, as an example, the environment monitoring in a smart city: the Servers in the cloud manage the city devices, i.e., the Clients, to monitor noise, air pollution, weather, etc. to keep citizens informed of air quality, conditions, and pollutants, etc. Thus, Servers communicate with a large number of Clients to manage them, e.g., they may need to reboot them, to retrieve their battery level or to update their firmware, and to collect their data, e.g., they may need to read humidity sensor values or to receive notifications of temperature sensors when their values change.



(**a**) (**b**)

Figure 1. LWM2M Architecture, (**a**) without LWM2M Proxy, (**b**) with LWM2M Proxy.

Since the Proxy sees all the exchanged messages between Client and Server, it can be used to implement an optimized version of LWM2M to reduce the number of exchanged messages and the number of operations to be executed on the Client, without affecting Server operations. Let us consider as an example, the device-registration procedure, a typical device management function that allows a device to register with a remote server and update registration information. In LWM2M this procedure is implemented with a

Register message and a periodic Update message sent from the Client to the Server. In the optimized version of the protocol the registration procedure is performed by the Proxy on behalf of the Client.

Leveraging its role of intermediate node, the Proxy can further improve system performance by implementing one or more additional extended functions that support QoS-aware services. In this work we propose the following three additional functions: (i) a request shaper that controls the flow of requests sent to the Clients over the constrained network, so that the Server experiments reduced delays and losses; (ii) a cache that reduces the number of exchanged messages in the IoT network, so that the Server experiments reduced response times; (iii) a compressor that compresses/decompresses the messages exchanged with the Client to be compliant with the limited packet payload length imposed by the energy-constrained networks. Our earlier work [7] proposed the introduction of a LWM2M-aware request shaper, deployed in between Clients and Servers. The request shaper controls the flow of LWM2M requests sent to the Clients. The paper demonstrates the effectiveness of the proposed approach in improving the performance of LWM2M in terms of service delay and packet loss as compared to a baseline without requests-shaping. The request shaper is assumed to be deployed in a proxy, without describing the architecture of the latter. In this paper, instead, we provide a complete specification of the proposed LWM2M Proxy, introducing the optimized LWM2M management procedures and the two additional functions, i.e., the cache and the compressor.

To evaluate the proposed extended architecture of LWM2M we set up two sample scenarios: in the first scenario we evaluate the request shaper and the cache. We consider a 6LoWPAN network [8], a technology that interconnects devices to communicate over a short distance. First, we conduct several experiments considering a scenario in which the Proxy implements the request shaper. Results show that the request shaper improves the performance in terms of service delay and packet loss as compared to the case with no shaper. Then, we conduct experiments considering a scenario in which the Proxy implements the cache. Results show that the cache reduces the number of exchanged messages and, at the same time, guarantees a given degree of data freshness. The data freshness is measured by the Age of Information (AoI) metrics [9] that define the timeliness of an IoT application's knowledge of the process running on an IoT device.

In the second scenario, we instantiate the Proxy in a LoRaWAN network [10], a technology that is suitable for sensors and applications that need to send data over long distances. We implement the compressor, and we use it for compressing/decompressing the LWM2M messages exchanged in the network. Results show that the compressed messages fit in a LoRaWAN packet.

The remainder of the paper is structured as follows: Section 2 gives a brief overview of the LWM2M protocol and describes related work. Section 3 illustrates our proposed architecture. Section 4 describes the QoS-aware functionalities of the Proxy and illustrates performance evaluation. Conclusions are drawn in the last section.

2. Related Work

2.1. Background on OMA LWM2M

OMA LWM2M ([4,5]) is an application layer communication protocol between a Server and a Client. It defines a set of REST-based interfaces and resources where each information made available by the Client is a Resource. This object model is easily extensible, and the Object and Resource registry is open to the industry.

Resources are organized into Objects. Objects and Resources can have multiple instances, and attributes, i.e., metadata, can be attached to them. A Resource which is instantiated within an Object Instance is a resource which can either: (i) contain a value, or (ii) be used by the Server to trigger an action in the Client. The Resource can be identified by its path, expressed in the following format: ObjectID/ObjectInstanceID/ResourceID. A Client must support three LWM2M mandatory Objects: (i) Security Object (ID: 0), that contains keying material enabling access to a specified Server; (ii) Server (ID: 1), that contains

information relating to the connection of a Server; (iii) Device (ID: 3), that contains device related information.

Four interfaces are designed between the Client and the Server: (i) Bootstrap and (ii) Client Registration that are used to make the Client accessible from the Server, (iii) Device Management and Service Enablement that is used to read/write a Resource or an Object, or to execute an action on the Client, (iv) Information Reporting that enables asynchronous information delivery. The main operations defined by these interfaces are illustrated in Table 1.

Table 1. LWM2M Operations.

Interface	Operation
Client Registration	Register: the Client registers with the Server
	Update: the Client updates registration information
	De-register: the Client de-registers with the Server
Device Management and Service Enablement	Read: the Server accesses the value of a Resource, a Resource Instance, an Object Instance or an Object
	Discover: the Server learns the attributes of an Object, Object Instances, and Resources, and discovers which Resources are instantiated in a given Object Instance
	Write: the Server changes the value of a Resource, a Resource Instance, and multiple Resources from an Object Instance
	Write-Attributes: the Server modifies the notification attributes
	Execute: the Server performs some action on individual Resources
	Create: the Server creates Object Instances within the Client
	Delete: the Server deletes an Object Instance or a Resource Instance within the Client
Information Reporting	Observe: the Server initiates an observation request for changes of a specific Resource, Resources within an Object Instance or for all the Object Instances of an Object within the Client
	Cancel-Observation: the Server ends an observation relationship
	Notify: the Client sends the new value of the Object Instance or Resource to the Server

We refer the reader to [4,5] for further details on LWM2M.

2.2. Related Work on IoT Proxies

In a typical IoT architecture, a gateway acts as an intermediary between devices in the IoT network and IoT applications running in data centers usually deployed in the cloud, because most IoT devices cannot connect directly with the application as they use near-range technology. IoT applications may require gateways to also provide quality of experience and better performance in terms of latency, throughput, energy consumption, response time, etc. [6].

Queue management and traffic shaping are typical functions implemented by IoT gateways. For example, in [11] the authors propose an IoT gateway that uses the Hierarchical Token Bucket algorithm to configure the output rate of a high priority queue for critical messages that need to be immediately forwarded to middleware and of a low priority queue for non-critical messages. Instead, since the IoT network can become easily congested, our proposed Proxy uses a request shaper based on the token bucket algorithm to control the flow of requests towards the Clients. As another example, in [12] authors implement a proxy virtualization framework to ensure scalability in a large IoT network and to support the implementation of custom functionalities. They implement a QoS prioritization policy

to differentiate the service offered to two groups of applications using a priority-based scheduler that buffers the requests.

Another typical function of an IoT gateway is reducing the number of exchanged messages for energy conservation and congestion avoidance. So, authors of [13] propose a CoAP proxy to provide support to applications that need to continuously retrieve data from the Wireless Sensor Network (WSN) using WebSocket. The CoAP proxy uses the observe protocol to receive status notifications from the device; so, the proxy is the only observer, reducing the WSN traffic. Moreover, the proxy implements a cache to reduce the number of interactions between the proxy and the WSN where the expiration time of the cached response is indicated using the Max-Age option. Similarly, authors of [14] propose a framework for proxies to regulate the transmission of notifications in a CoAP-based IoT network. It groups clients by their demands and computes an optimal observation period for each group such that the IoT device generates the minimum number of notifications; it finds an appropriate max-age value for each IoT device to cache notifications; and it combines notifications generated by IoT devices, so a proxy can forward fewer notifications to the client. However, in these solutions the proxies aim to reduce the traffic generated asynchronously by the IoT devices. Instead, we propose a solution that reduces the number of requests generated by the Servers and forwarded to the Clients. Moreover, we also propose that the Proxy implements a cache function, but the freshness of the cached items is measured by the Age of Information metrics [9] and the cached items are refreshed according to the IoT application freshness requirements.

IoT gateways can also improve the management of IoT devices. To address the heterogeneity of IoT devices, there is the need of a common standard platform for managing them. OMA LWM2M can be particularly suitable for this scope because it is a light protocol and uses an efficient resource data model. For these reasons, it is becoming widely used by both the industrial and research worlds. For example, authors of [15] also consider the LWM2M protocol and they propose a LWM2M Proxy that implements a group management extension for LWM2M. In this case, the proxy is introduced to minimize the traffic towards the LWM2M Server: it reduces the number of bytes sent to the server and it avoids retransmissions collecting the messages coming from the LWM2M Clients and aggregating them in one response to be sent to the LWM2M Server. In our work we instead try to minimize the traffic towards the LWM2M Client to cope with the constraints of IoT devices and networks. Authors of [16] propose a cloud based virtual network operator for Low Power Wide Area Networks that homogenizes deployments using OMA LWM2M and that provides a unified way to communicate over different technologies using Static Context Header Compression, SCHC. However, they implemented the header compression mechanism for the CoAP/UDP/IPv6 stack, leaving the LWM2M payload uncompressed. Moreover in [17] authors show the benefits that the use of the SCHC mechanism provides to constrained IoT networks: they consider a LPWAN network and, by applying the header reduction to the CoAP/UDP/IPv6 stack, they enable the transmission of packets that could not be transmitted previously. Moreover, in this case, the compression is applied only to the headers of the packets. In this work, we propose an extension of the SCHC mechanism for LWM2M that allows to also compress the LWM2M payload.

We propose the architecture of a Proxy that optimizes the LWM2M protocol and that also allows the implementation of one or more QoS-oriented customized functions that tackle the issues illustrated above. So, our proposed Proxy implements a freshness-aware cache, a shaper that buffers the requests to avoid network congestion, and a compressor that compresses the header and the payload of the packets to reduce their sizes. These functions can be applied singularly or together depending on the Server demands.

3. LWM2M Proxy

IoT devices possess limited computing power, memory, and energy and can use narrow-band protocols to communicate in energy-constrained networks. To further motivate our work, we estimate how these limitations may affect the LWM2M performance

in two exemplary network deployments. In the first example, we consider a sample scenario wherein a Client running on a wireless sensor node receives multiple concurrent requests from different Servers located outside of the sensor network. The sensor network is emulated using the COOJA network emulator [18] and is deployed using the 6LoWPAN protocol [8] on top of the IEEE 802.15.4 MAC [19] operating in the 2.4 GHz band. The Client is located three hops away from the 6LoWPAN Border Router, and RPL [20] is run as routing protocol. We model the generation of requests as a Poisson process with cumulative rate λ, and we measure the service delay, i.e., the time between when the request is sent by a Server and the time the response is received. First, we run a baseline experiment in which a Server sends 2000 requests back-to-back to the Client, i.e., a request is sent only when the response to the previous request is received. The resulting average service delay is 326.5 ms (95% CI [322.6, 330.4]). Then, we run experiments by varying the rate λ, and each experiment lasted 1500 s. As the rate increases, the service delay distribution curve quickly shifts to the right and a growing, non-negligible, fraction of packets starts experiencing a large delay (see Figure 2) and an increasing fraction of the requests starts not receiving a response (see Table 2 in Section 4). This happens because Clients can only allocate a small packet buffer to store incoming packets, since their memory is limited; moreover, they can take a non-negligible time to process and forward a packet because of their limited processing capabilities. Therefore, the buffer can quickly become overloaded and start dropping packets. Since requests are carried by CoAP CON messages, dropped requests are retransmitted and the experienced delay increases as the number of retransmissions increases.

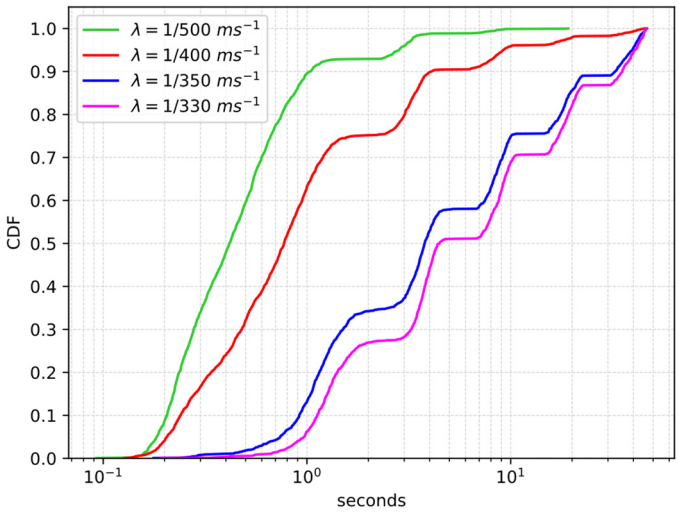


Figure 2. Service delay CDF of the sample scenario.

Table 2. Service Loss.

λ [ms^{-1}]	w/o Shaper	w/ Shaper
1/500	0	0
1/400	0.9%	0
1/350	30.6%	0
1/330	42.1%	0

As a second example, we consider a LoRaWAN network, one of the most adopted Low Power Wide Area Networks (LPWAN, [21]) technologies. LoRaWAN imposed strict limitations on packet payload length, with a minimum frame size of 59 bytes for the EU 868 MHz band and of 19 bytes for the US 915 MHz band [21]. Thus, it can happen that a LWM2M message does not fit in a LoRaWAN packet, especially if it contains an Object

or an Object Instance. As an example, consider the possible response to a request for the Device Object Instance (id: 3) of a Client (Read/3/0) illustrated in Section 7.4.5.1 of [4]. We can notice that the LWM2M payload may not fit in a LoRaWAN packet because its size is 121 bytes using the TLV format, or 399 bytes using the JSON format.

To adapt the LWM2M architecture to networks with limited bandwidth and computational resources, we propose to introduce a Proxy in between the Servers and the Clients that can optimize device management procedures and support QoS-aware services. The Proxy breaks the communication between Clients and Servers acting as a Client towards the Servers and as a Server towards the Clients. Therefore, the Client communicates only with the Proxy rather than exchanging messages with multiple Servers and maintaining the registration information and possibly the observation relationship for each of them; whereas Servers are managed by the Proxy that can optimize their communication with the Clients.

As illustrated in Figure 3, the Proxy is composed of a Core function and zero or more additional extended functions. The Core function improves network performance by implementing an optimized version of LWM2M and enabling the support of QoS-aware services by orchestrating the optional extended functions to efficiently use network, computing, and storage resources at the edge, thus providing a reduced and more predictable end-to-end latency. When a Server request arrives at the Proxy, the Core function first attempts to use the optimized version of LWM2M: in this case, it directly responds to the Server, as we show in Figure 4, Step 2. Then, if the optimized version of LWM2M cannot be applied, i.e., the request has to be relayed to the device, as we show in Figure 4, Step 4, the Core function determines the extended functions the request will go through, submits the request to them and finally responds to the Server. The rest of this Section will describe when and how the optimized LWM2M is applied, whereas Section 4 will describe and evaluate three exemplary QoS-aware extended functions.

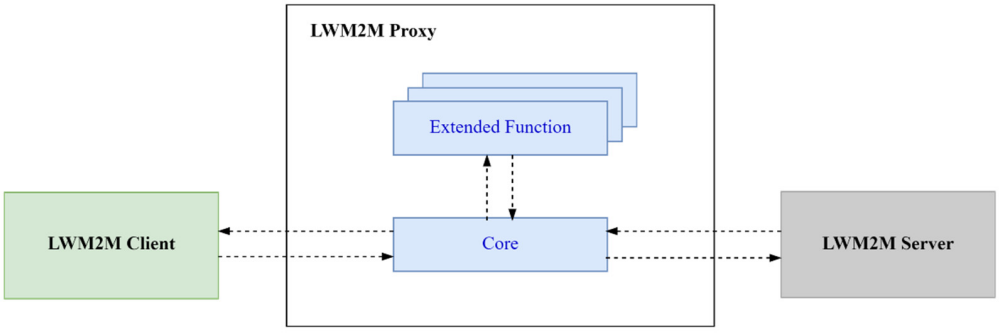


Figure 3. Architecture of the LWM2M Proxy and its relationship with other LWM2M components.

The optimized version of LWM2M implemented by the Core aims at reducing the number of exchanged messages and the amount of computation to be performed on the Client. To do so, the optimized LWM2M requires the Proxy to be aware of the Clients in the IoT network. The Proxy discovers the Clients gathering information from the lower layers, exploiting technology specific functions. If, for example, the underlying communication technology is LoRaWAN, the Proxy leverages the Join procedure to know which devices are connected; if instead a 6LoWPAN network is considered, the Proxy leverages the Neighbor Discovery procedure.

We provide here a description of the updated LWM2M operations, optimized through the proposed Proxy.

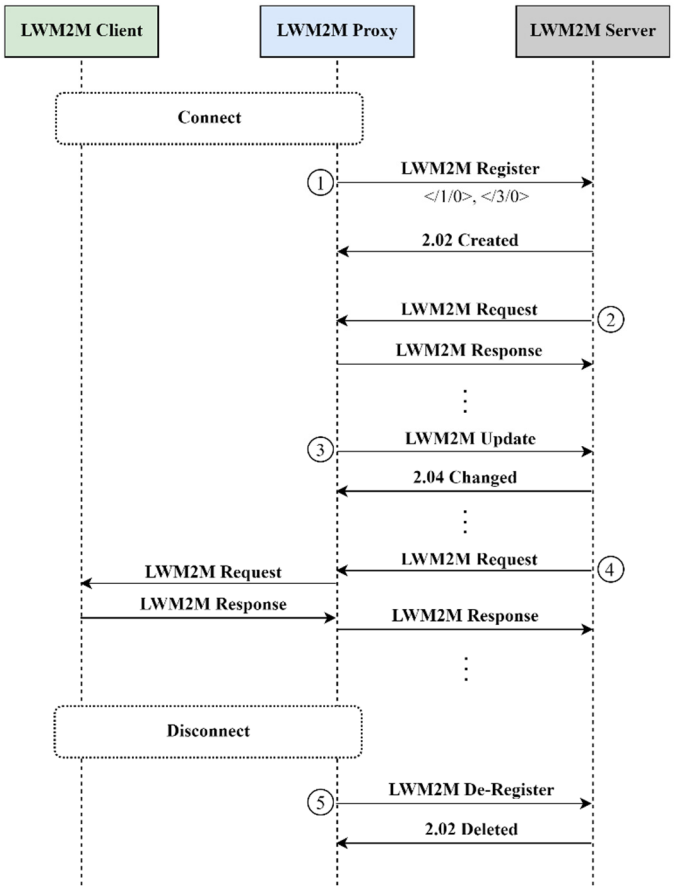


Figure 4. Optimized LWM2M.

3.1. Register, Update and De-Register

The registration procedure is performed by the Proxy on behalf of the Client. When a Client connects to the network, the Proxy sends a Register message to the Server (see Figure 4, step 1). Then, it also sends an Update message when a refresh of the registration is needed (see Figure 4, step 3). When the Client is no longer available, the Proxy sends a De-Register message (see Figure 4, step 5).

Let us consider as an example the scenario depicted in Figure 1a: we have that the first Client registers with the first Server; the second and third Clients register with all three Servers; the fourth Client registers with the first and the third Server. This means that the Clients send nine messages just to register with the Servers they need to communicate with. Then, suppose that the registration lifetime is 3600 s: this means that each Client has to send an Update message every 3600 s. So, during a day, i.e., 24 h, a Client sends 24 Update messages to each Server it is registered with. Therefore, during a day the Clients send 216 messages only to refresh their registrations. Instead, when we introduce the Proxy (Figure 1b), the Register and the Update messages are sent by the Proxy on behalf of the Clients, reducing the number of messages in the constrained network and saving energy on the device.

3.2. Write-Attributes

The Proxy rejects a Write-Attributes request if the conveyed Notification Condition Attributes are not consistent, i.e., when the "Change Value Conditions" attributes Less Than (lt), Greater Than (gt), Step (st), [4] are set in a single Write-Attributes operation, the Proxy does the required coherency checks: (i) $"lt"\ value < "gt"\ value$ and (ii) $"lt"\ value + 2 * "st"\ value < "gt"\ value$ on behalf of the Client (see Figure 4, Step 2).

3.3. Discover

The Proxy responds to a Discover request on behalf of the Client (see Figure 4, Step 2). To do so, the Proxy maintains a list of the Clients with their implemented Objects, Object Instances, Resources, Resource Instances, and Attributes. Being deployed at the edge, the Proxy is able to discover the Clients in the network gathering information from the lower layers; and whenever a Client connects to the network, the Proxy retrieves and stores the list of its Objects, Object Instances, Resources, and Resource Instances with their attached Attributes. The Proxy updates this list every time a Server creates or deletes an Object Instance or a Resource Instance and when it changes the notification Attributes of an Object, Object Instance, Resource, or Resource Instance, i.e., when it issues a successful Write-Attributes request or a successful Create, Delete, or Write request that modifies the list of Object Instances and Resource Instances.

3.4. Create

The Proxy sends an error message in response to a Create request when the latter is not well-formed, i.e., if all the mandatory resources are not present in the conveyed message payload, or if the message payload conveys an Object Instance ID in conflict with one already present (see Figure 4, step 2). In this latter case, the Proxy checks the correctness of the conveyed Object Instance ID using the list of Object Instances stored for each Client.

3.5. Write

The Proxy can reject a request without forwarding it to the Client if one of the following conditions is met: (i) the specified content format is not supported; (ii) the value of the incoming resource does not match the expected format; (iii) the value of the incoming resource is not in the range specified in the Object definition; (iv) the incoming resource is an Objlnk and its value is not valid; (v) the deletion or allocation of an instance of a multiple-instance resource is not allowed (see Figure 4, Step 2).

Moreover, the Proxy checks all the requests to reject them if they are not allowed, or if they are incorrect, i.e., if the required parameters are not present in the uri-path/uri-query of the message; or if the specified Object ID, Object Instance ID, or Resource does not exist or it is not supported by the Client, or if the specified format is not supported (see Figure 4, Step 2).

In addition to this, the Core function enables the support of QoS-aware services by orchestrating the extended functions. These functions are optional: the Proxy can choose not to implement them, so, in this case its only component is the Core; or it can choose to implement one, two, or all of them. The Proxy takes this decision according to the Server requirements. Indeed, before the Server starts sending messages to the Client, there is a configuration phase where the Server and the Proxy establishes a service level agreement: during this phase the Server specifies its QoS requirements in terms of service latency, data freshness, or both.

3.6. Proxy Deployment

As we highlighted in the introduction, we propose to deploy the LWM2M Proxy at the network edge in between Clients and Servers, so that it can take advantage of the proximity with the end devices. However, we highlight that the Proxy should not be deployed even closer to the end-devices and within the access network itself, e.g., within a node of the IoT network. As a matter of fact, in case of dynamic IoT deployment, e.g., scenarios involving frequent topology changes due to channel variation or node mobility, a Proxy deployed in an IoT node might fall in a sub-optimal placement with respect to the Client-Server path. If instead, an edge-placement is considered, topology changes can be easily managed by the Proxy, which need only to reconfigure few parameters of the extended functions that are influenced by the latency and the topology of the network, e.g., the request shaper and the cache.

As an example, let us consider the case of node mobility. When using an access network technology such as LoRaWAN or 6LoWPAN, node mobility is transparent to the Proxy. Indeed, when a node moves within the 6LoWPAN domain, the mobility is supported by the routing protocol used within the 6LoWPAN and it is transparent to the Proxy that is deployed at the edge on the 6LoWPAN Border Router or on a node in the backend near to the 6LoWPAN Border Router. In LoRaWAN, nodes are not associated with a specific gateway and data transmitted by a node can be received by multiple gateways. Each gateway will forward the packet to the network server, so complexity is transferred to the network server: if a node moves there is no handover needed from gateway to gateway. So, also in this case, node mobility is managed by the access network technology, and it is transparent to the Proxy that runs on the LoRaWAN Application Server.

If the access-network technology is a cellular network, e.g., Narrowband IoT, node mobility among the base stations of the network is possible and is managed by the handover function of the technology itself. In this case, the Proxy can still be deployed close to the serving base station, e.g., exploiting a Multi-Access Edge Computing (MEC) architecture.

4. QoS-Oriented Extended Functions

As we anticipated in the Introduction, the Proxy can provide extended functions to support QoS-aware services at the network edge. Motivated by the congestion issues showed by the experiments reported in Section 3, we propose to introduce a requests-shaping and a caching function. Motivated by the payload size limitations showed in Section 3, we propose to also introduce a compression function.

We propose here three relevant instances of said functions: (i) a request shaping function that guarantees a better QoS by controlling the delay experienced by the packets traveling in the constrained network and by reducing packet losses; (ii) a caching function that improves the QoS of the IoT system because it reduces the power consumption of the Client and the number of packets to be transmitted in the IoT network, saving network bandwidth and reducing response time, while guaranteeing information freshness; and, finally, (iii) a compression function that improves QoS by reducing the size of a packet, so that the number of transmitted packets and the overhead of a fragmentation mechanism are reduced. Indeed, if the compressed packet can fit in a network frame, fragmentation is not needed; if, instead, the compressed packet still cannot fit in a network frame, the number of transmitted packets is reduced because its length is smaller than its uncompressed version, and the fragmentation overhead is reduced because fragmentation is defined as a mechanism internal to the compression function.

In the following, we describe each proposed extended function highlighting its objectives and showing its possible implementation.

4.1. Requests-Shaping

The Proxy can alleviate network congestion by controlling the flow of requests to the Client [7]; consequently, it can guarantee a better QoS. To do so, the Proxy implements a congestion window that limits the number of outstanding requests over the constrained network up to a fixed maximum. When the Proxy receives a message request for a Client, it enqueues it in a buffer, and forwards it only if the window is open, i.e., if the number of requests forwarded to any Client and still waiting for a response is less than the maximum window size.

4.1.1. Requests Shaping Algorithm

The requests-shaping algorithm behaves similarly to a token-bucket algorithm wherein a token represents a packet. Hence, the bucket is filled with a number of tokens equal to the size of the congestion window. Then, when a packet arrives, it is enqueued in the buffer and, if there is at least one token in the bucket, the Proxy forwards the packet to the Client and it removes one token from the bucket. When the Proxy receives a response to a

previous transmitted request, it adds one token to the bucket, checks if a packet is queued for transmission, and possibly transmits it also removing one token from the bucket.

In an extreme case the maximum size of the congestion window is set to one, i.e., there is only one outstanding request. However, in some cases, it can be more efficient to exploit the network resources by sending concurrent requests, i.e., having a maximum congestion window size greater than one. Indeed, the size of the congestion window, i.e., the number of concurrent requests, depends on the number of requests that can be sent through the network without congesting it. This in turn, depends on several factors, including the following: the network/communication technology that is used; the network topology, i.e., the average number of hops to reach the destination; the memory and processing capabilities of the devices, i.e., the size of the buffer used to store incoming packets, and the time needed to process and forward a packet.

4.1.2. Performance Evaluation

To evaluate the advantages of introducing the proposed requests-shaping function, we consider a testbed scenario composed of a sensor network, a Server and a Proxy, as depicted in Figure 5. A Client runs on a wireless sensor node using the 6LoWPAN protocol [8] on top of the IEEE 802.15.4 MAC [19], operating in the 2.4 GHz band, and located three hops away from the 6LoWPAN Border Router. The system is emulated using the COOJA network emulator [18] and uses RPL [20] as routing protocol. The Server and the Proxy are developed using the Eclipse Leshan library [22]. We consider a Client with a sensor that collects information at a periodic sampling rate; indeed, periodic sampling is widely used by IoT devices because the simplicity of periodic sampling is well-fitted with devices which have limited computational resources. The onboard sensor of the Client is a temperature sensor that samples the temperature each 60 s.

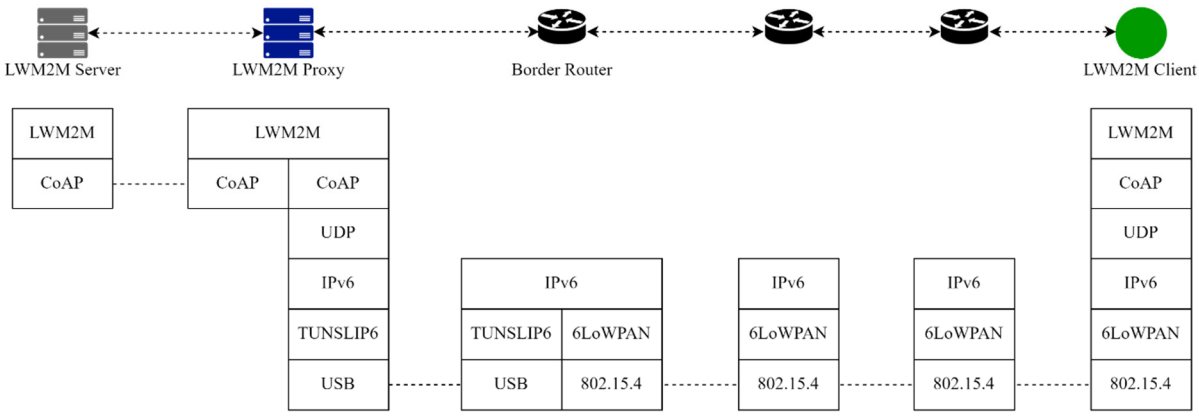


Figure 5. LWM2M architecture instantiated in a 6LoWPAN network.

We model the arrival of LWM2M requests as a Poisson process with cumulative rate λ, and we consider the following metrics: (i) service delay, defined as the time between the request is sent by the Server and the time the corresponding response is received, (ii) service loss, defined as the percentage of LWM2M requests that did not receive a response according to requests sent.

First, we evaluate the same scenario considered in Section 3, now taking into account the case when the Proxy is in place, and we compare the case where the Proxy does not implement the request shaper against the case where the Proxy implements the request shaper with the maximum size of the congestion window fixed to one. Figure 6 shows the cumulative distribution function of the service delay for different values of λ using a log scale on the x-axis. When the rate λ increases, if the Proxy does not implement the request shaper, the distribution shifts to the right and its tail is heavier; in addition to this also the packet loss increases, as reported in Table 2. Instead, when the request shaper is used, the maximum experienced service delay decreases significantly, the tail of the

distribution is shorter and there is no service loss. This happens because the Proxy that uses the requests-shaping function can control the traffic load in the network avoiding congestion. We can notice that the service delay experienced with the requests-shaping function using a window value equal to one is not always better than the service delay experienced without the requests-shaping function. So, a requests-shaping function using a larger window value could perform better.

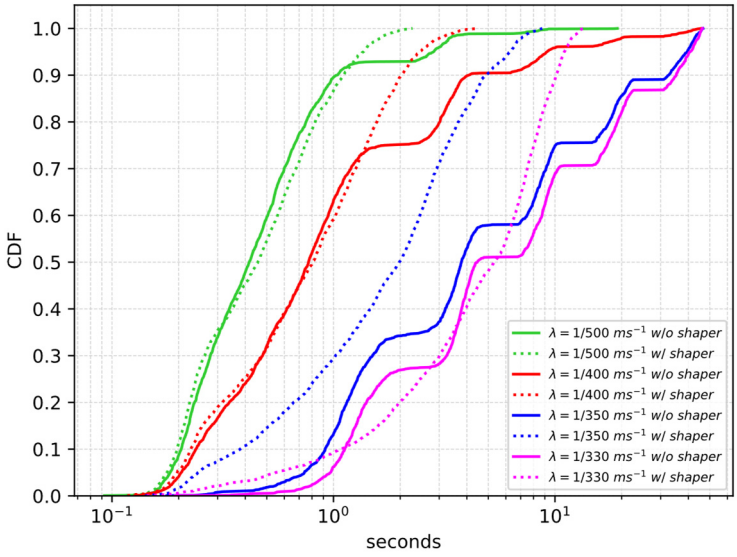


Figure 6. Service delay CDF w/ and w/o the request shaper.

Figure 7 compares the service delays of the scenarios that use the request shaper against the service delays of the scenarios that do not use the request shaper; the comparison is done using a box plot representation: in the box the 25th, the 50th, and the 75th percentiles are represented, respectively, the ends of the whiskers represent the 5th and 95th percentiles and the circles are the outliers. We can notice that especially for lower values of λ i.e., when $\lambda = 1/500$ ms^{-1} or when $\lambda = 1/400$ ms^{-1}, the 25th and 50th percentile values obtained with the request shaper and without the request shaper are almost the same; but we can also notice from the 75th and 95th percentile values that with the request shaper the maximum experienced service delay decreases significantly (please note the log scale on the *y*-axis).

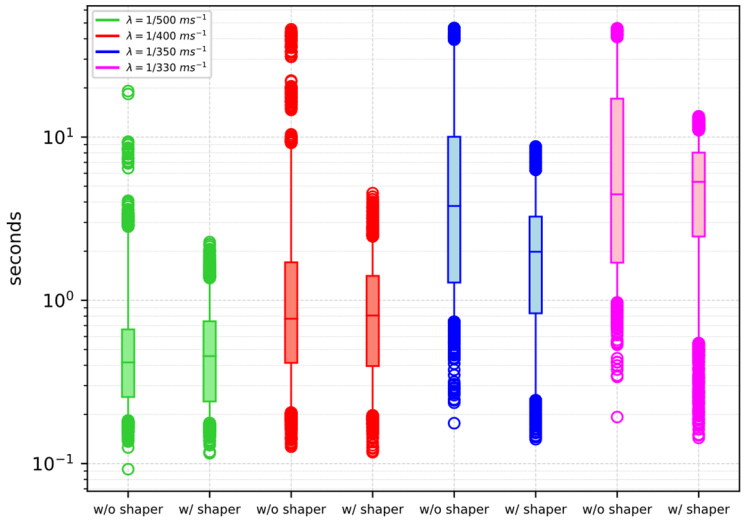


Figure 7. Service delay box plot w/ and w/o the request shaper.

In the second scenario under consideration, we tested different values for the maximum congestion window size. Figure 8 shows the cumulative distribution function of the service delay for different values of the congestion window size in the case of a rate λ equal to $1/330$ ms^{-1}. We can observe that when the congestion window size is fixed to three the service delay experienced with the request shaper is always better than the service delay experienced without the request shaper. But, of course, when there are concurrent requests in the network it is possible that some of them experience a larger service delay with respect to the case in which there is always only one pending request in the network. Indeed, we can observe that the maximum service delay experienced by the packets in the case of a window value equal to three is larger than the case of a window value equal to one.

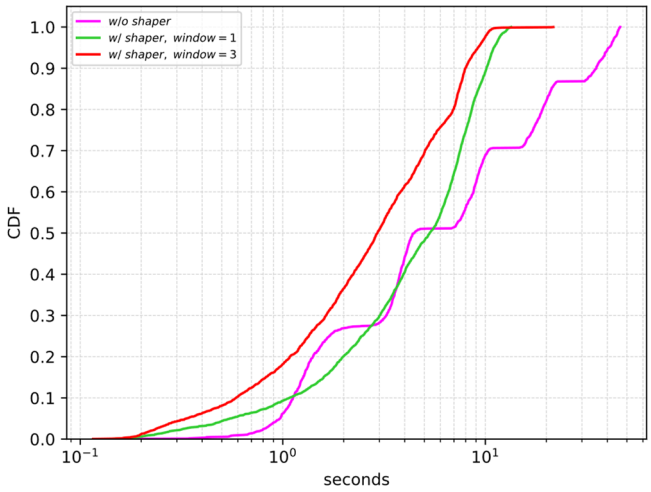


Figure 8. Service delay CDF for different values of the congestion window size, $\lambda = 1/330$ ms^{-1}.

4.2. Caching

The Proxy can improve QoS by caching data at the edge: it attempts to serve Server requests taking the responses from the cache, thus reducing network load. Since caching may lead to staleness of information, there is the need of a refreshing scheme to guarantee freshness of data, considering that each update of the cache requires additional messages to be sent on the IoT network that can degrade system performance. A typical cache-refreshing scheme associates a validity lifetime to each cached data item. When the data item expires, it is not useful anymore and it must be discarded.

4.2.1. Cache-Management Scheme

Several measures have been introduced and analyzed in order to measure the freshness of the data stored in the cache [23]. For example, the age of a cached item can be defined as the time difference between current time and the time the item became desynchronized with the source; or the age of a cached item can be defined by the Age of Information (AoI, [9]), i.e., the elapsed time between the current time and the time the item was generated at the source. In this case, for example, the LWM2M Object representing the Client sensor can support the LWM2M Resource Timestamp (ResourceID: 5518) that specifies the timestamp of when the measurement was performed.

To quantify the lifetime of a cached item we choose the AoI metrics. We assume that the AoI is the QoS parameter established by the service level agreement about data freshness between the Server and the Proxy. As an example, during the configuration phase the Server can require that at least a certain fraction of the requests, e.g., 90% of the requests, receive a data item whose AoI is not larger than a given target value [24].

So, a cached item is valid if its AoI is smaller than a value called refresh window (W). The Proxy responds to the Server request using its cached data item if it is valid; otherwise,

it relays the request message from the Server to the Client, then, it relays the response message back from the Client to the Server and updates the cache.

4.2.2. Performance Evaluation

To assess the performance of the caching function, we consider the 6LoWPAN network deployment of Section 4.1.2, and we let the Proxy implement the proposed cache-management scheme.

We model the arrival of LWM2M requests as a Poisson process with cumulative rate $\lambda = 1/10\ s^{-1}$, and we compare the case in which the Proxy does not implement the cache against the case in which the Proxy implements the cache. For this latter case we consider two different refresh window values: (i) W = 60 s, i.e., the window value is equal to the sampling period; and (ii) W = 240 s. Each scenario is run for 250,000 s. To evaluate the performance of the caching function we consider the following metrics: (i) the number of exchanged messages in the sensor network and (ii) the AoI of the data at the Server.

Figure 9 shows the number of exchanged messages, while Figure 10 shows the cumulative distribution function of the AoI. It is possible to notice that the case of no cache and the case of a cache with a window value equal to the sampling period guarantee the same AoI at the application, but the cache can reduce the number of exchanged messages with the Client. Moreover, results show the trade-off between the number of exchanged messages and the AoI of the data: when W increases, the number of exchanged messages decreases because the probability that a request gets a hit in the cache is higher, but the AoI increases. Minimizing the number of exchanged messages and minimizing the AoI of the data are opposite objectives, so, for example, we can choose the largest value of W that satisfies the AoI requirements of the Server (see [24]).

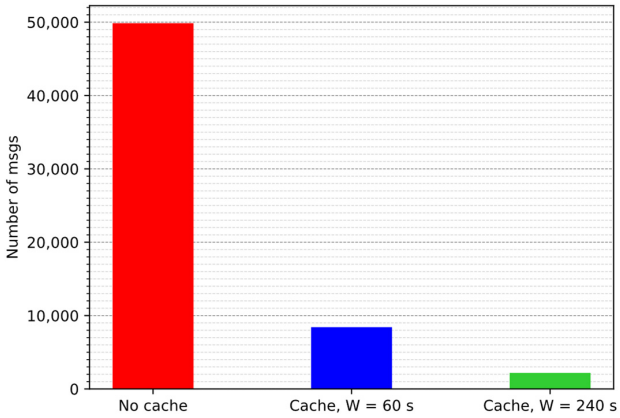


Figure 9. Number of exchanged messages w/ and w/o cache, $\lambda = 1/10\ s^{-1}$.

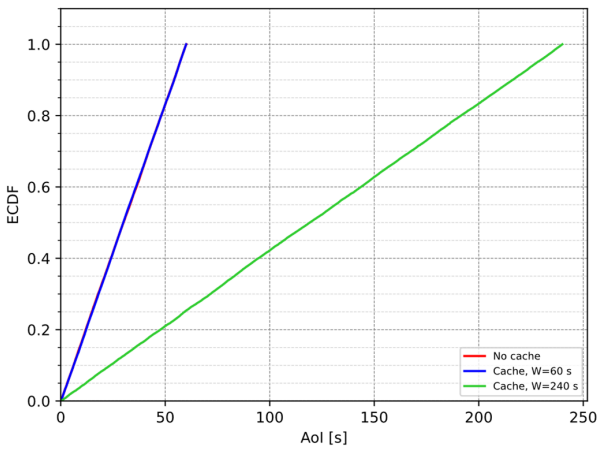


Figure 10. AoI empirical CDF w/ and w/o cache, $\lambda = 1/10\ s^{-1}$.

4.3. Compression

IoT networks can impose strict limitations on traffic. For example, LPWAN technologies such as LoRaWAN [10], are characterized by a reduced data rate and a limited payload length in order to support long range and low power operations.

The FUOTA Working Group of the LoRa Alliance addresses this limitation on packet size by defining a fragmentation mechanism. They propose an application layer messaging package [25] running over LoRaWAN that sends a fragmented block of data to the device. This method splits a data block into multiple fragments to fit into LoRaWAN packets; then, these fragments must be reassembled on the device, handling potential losses. To do so, first a session establishment mechanism must be run to inform the device that a fragmentation session will start. Fragmentation can affect the QoS of the system. Indeed, it introduces overhead because each fragment has to contain the L2 header, and it also introduces delay because a message can be reassembled only after all the fragments have been received. Moreover, LPWAN technologies are characterized by a high packet loss rate, so fragments can be lost causing retransmissions and therefore performance degradation.

We instead propose to reduce the packet size by applying a compression function, thus limiting the need for fragmentation. The compression function is based on the Static Context Header Compression (SCHC, [26,27]) framework and executed both on the Client and on the Proxy. SCHC compression relies on a common static context, i.e., a set of rules, stored both in the Proxy and in the Client. The Proxy can define a specific context for each Client to take into account their heterogeneity and the Servers can be unaware of these changes in the communication with the Clients. The Client stores a single context because it exchanges compressed packets only with the Proxy. The context does not change during packet transmission, so SCHC avoids the complexity of a synchronization mechanism. Moreover, SCHC supports fragmentation, thus if the compressed packet size still exceeds the maximum packet size, it is possible to fragment the compressed packet with a reduced overhead and with fewer fragments needed than in the case of the uncompressed packet fragmentation.

The main idea of the SCHC framework is to exploit the fact that the traffic of IoT applications is predictable, so it is possible to transmit the ID of a compression/decompression rule, i.e., rule ID, instead of sending known packet-field values. SCHC classifies fields in (i) static, i.e., well-known, and hence not sent on the link; (ii) dynamic, i.e., sent on the link; (iii) computed, i.e., rebuilt from other information. At the top of Figure 11 we make an example of a compression/decompression rule: as we can see from the figure, a rule contains a list of field descriptions composed of the following data: (i) Field Identifier (FID), it designates a protocol and a field; (ii) Field Length (FL), it represents the length of the field; (iii) Field Position (FP), it indicates which occurrence of the field this field description applies to; (iv) Direction Indicator (DI), it indicates the packet direction; (v) Target Value (TV), it is the value to match against the packet field; (vi) Matching Operator (MO), it is the operator used to match the field value and the target value; (vii) Compression/Decompression Action (CDA), it describes the compression and decompression actions applied on the field. If the field has a fixed length, then applying the CDA to compress it produces a fixed number of bits, which is called the compression residue of the field. If the CDA is "not sent", then the compression residue is empty. If, instead, the field has a variable length, e.g., the uri-path or the uri-query, then, applying the CDA may produce either a fixed-size or variable-size compression residue. The former case occurs when the field value is known and hence it is not sent, or when the field value belongs to a known list of values and hence the index of the list is sent. In the latter case, instead, the compression residue is the bits resulting from applying the CDA to the field, preceded by the size of the compression residue itself, encoded as specified in [27]. If a variable-length field is not present in the packet header being compressed, a size 0 must be specified to indicate its absence. We provide here a brief description of the compression and decompression algorithms of SCHC, whereas in Sections 4.3.1 and 4.3.2. we propose and evaluate an extension to SCHC for LWM2M.

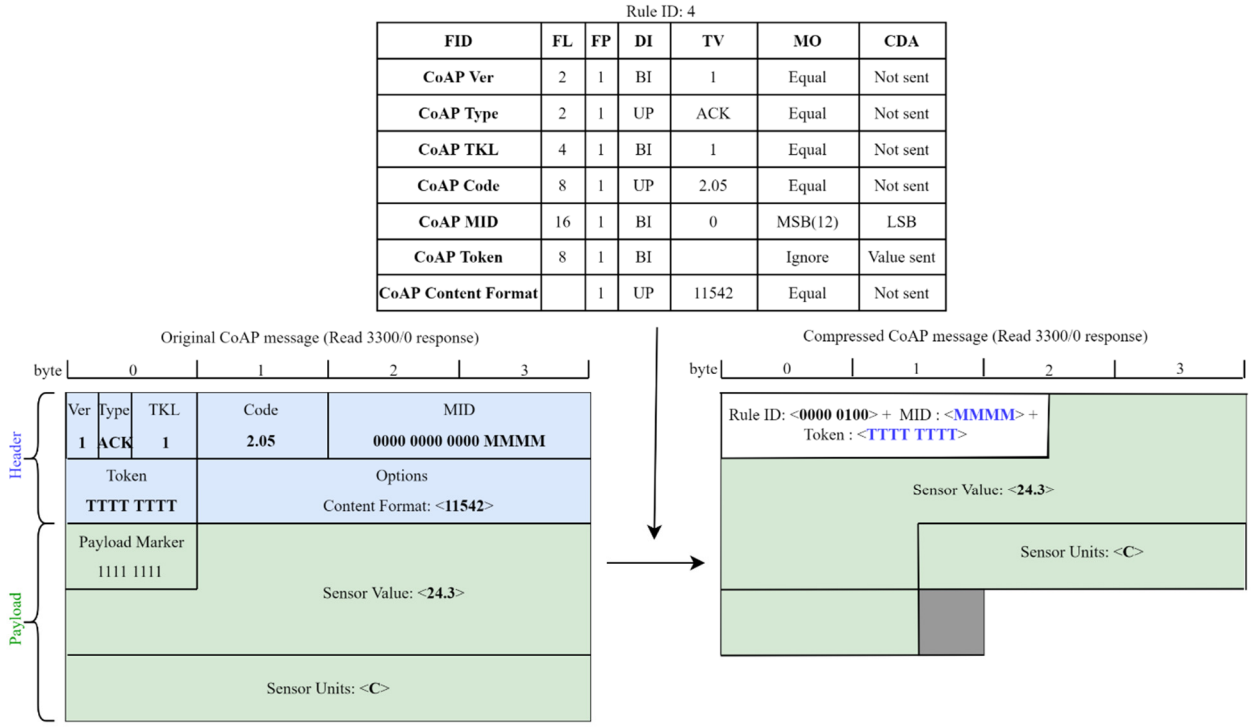


Figure 11. SCHC framework used in LoRaWAN.

The compression algorithm consists of three steps as follows: (i) rule selection: identify which rule will be used to compress the packet's headers. The rule will be selected by matching the field descriptions to the packet header; (ii) compression: compress the fields according to the compression actions. The fields are compressed in the order specified by the rule and the compression residue is the concatenation of the residues for each field; (iii) sending: the rule ID is sent followed by the compression residue. The decompression algorithm consists of a single step: (i) decompression: the receiver selects the appropriate rule from the rule ID. Then it applies the decompression actions to reconstruct the original fields.

SCHC allows one to specify a fixed number of fields in a rule. If, instead, a variable number of fields is needed, e.g., when the number of uri-path or uri-query varies, it is possible to either create multiple rules to cover all cases or to create one rule that defines multiple entries and send a compression residue with length zero to indicate that a field is empty. These two options introduce a trade-off between compression ratio and memory usage: indeed, if we define multiple rules that cover all possibilities, we increase the compression ratio, but we also need more memory on the device to store all the rules; if, instead, we define a single rule we obtain a lower compression ratio, but we also need less memory on the device.

To illustrate the SCHC framework, in Figure 11 we show an example of its possible use in a LWM2M architecture instantiated in a LoRaWAN network: on the left, we show the original uncompressed packet, while, on the right, we show the packet compressed using the rule listed at the top of the figure. Due to the limited bandwidth of LoRaWAN, the LWM2M standard defines the possibility of using CoAP over LoRaWAN (LoRaWAN binding, [5]) by placing the CoAP packet containing the LWM2M message directly in the LoRaWAN packet payload. Thus, in this case, a set of compression rules for the CoAP header can be defined according to the following criteria:

- The CoAP version, type, and token length fields have been elided, since their values are known.
- The CoAP code field has been reduced to the set of the used codes for each operation, defining a mapping list.

- The CoAP message id has been reduced to a four bits value, using the MSB (most significant bits) matching operator.
- The CoAP token field needs to be transmitted, but, since the token length is known, it is not necessary to send the size.
- The CoAP content format and accept fields have been elided when only a single value is possible; otherwise, they have been reduced to the set of used values using a mapping list.
- The CoAP uri path fields have been elided when only a single value is possible; otherwise, they have been reduced to the set of used values using a mapping list. Since the number of uri path elements may vary, it has been decided to define the length to variable and send a compression residue with a length of 0 when the uri path is empty.
- The CoAP uri query fields are used for setting the attributes, so they have been reduced to only their numeric values using the MSB matching operator. Since the number of uri query elements may vary, it has been decided to define the length to variable and send a compression residue with a length of 0 when the uri query is empty.

In the example of Figure 11, the receiver reconstructs the CoAP version, type, token length, code, and content format using the values stored in the corresponding target value entries; it rebuilds the message id value concatenating the most significant bits stored in the target value entry and the received bits, and it builds the token value from the received value. We can notice that SCHC reduces the size of the packet, from 8 bytes to 12 bits; however, the overall packet-size reduction is limited because the largest component of the packet, i.e., the payload, is still sent uncompressed. We can also notice that the payload marker used to indicate the end of options and the start of the payload is not present in the compressed packet because the length of the compression residue of the header is known; moreover, even if the compression is bitwise, the final size of the compressed packet is byte aligned to be sent on the network.

We refer the reader to [26–28] for further details on SCHC.

4.3.1. Extended SCHC for LWM2M

SCHC provides only header compression, so in the previous example the LWM2M payload must be sent uncompressed after the SCHC compression residue. In constrained technologies such as LoRaWAN, the header compression alone might not be enough to fit the maximum allowed packet size. In this work we thus extend the SCHC mechanism to also provide payload compression of LWM2M messages.

The LWM2M message structure can be efficiently compressed using the SCHC mechanism. In fact, a LWM2M Object Instance can be represented by an array of entries where each entry is a Resource, i.e., a ResourceID and its corresponding value, as, for example, a sensor-measurement value. The ResourceID can be considered equivalent to the SCHC FID and, hence, we can straightforwardly apply the SCHC compression algorithm also to the Resource field. When the value of the Resource is dynamic, it is also possible to compress it using a standard data compression algorithm; in this work, we just apply the SCHC compression mechanism.

If a LWM2M message contains multiple Instances of a given Object, the compression/decompression is applied for each Object Instance; the number of Object Instances present in the message is sent before the compression residue using the size encoding [27]. In more detail, the Object Instances and their Resources are ordered according to their IDs, so that it is possible to either: (i) elide the value of a Resource when it is known a priori, (ii) define a mapping list for a Resource when its value belongs to a defined set of elements, (iii) send the value of the Resource in all the other cases. Instead, when Read/Write operations involve one single Resource, the value of the Resource is sent uncompressed after the CoAP compressed header.

As illustrated in Section 3, whenever a Client connects to the network, the Proxy retrieves and stores its Objects and, for each of them, the list of supported Resources. We assume that this list does not change during the transaction between the Client and the

Proxy. After collecting the supported Resources, the Proxy can create the rules needed for the compression of the messages. In the Client, these rules are represented as LWM2M Objects, so the Proxy can write the context in the Client using LWM2M Write requests. As soon as this initialization phase has been completed, the Proxy and the Client can start exchanging compressed messages.

In Figure 12 we consider the same example of Figure 11, now applying the proposed extended SCHC framework. In the figure, the additional field descriptions of the rule that allow the compression/decompression of the LWM2M message are highlighted in red. The compression residue of the CoAP header is the same as that obtained in the example of Figure 11, now preceded by the number of Object Instances present in the message, i.e., 1, and followed by the compression residue of the LWM2M payload. The LWM2M field descriptions of the rule specify that the Instance ID of the Object has a variable length and its value is sent; in the example, the Instance ID is not present in the LWM2M payload because the request is for a specific Instance of the Object, i.e., for the Instance 0, and not for the Object, so a compression residue with a length of 0 is sent; the value of the Sensor Value Resource is sent; the Sensor Units Resource can take only the three values listed in the TV, so the index of the value in the target value list is sent. We can notice that the packet size is further reduced because the LWM2M message is also compressed.

Rule ID: 4

FID	FL	FP	DI	TV	MO	CDA
CoAP Ver	2	1	BI	1	Equal	Not sent
CoAP Type	2	1	UP	ACK	Equal	Not sent
CoAP TKL	4	1	BI	1	Equal	Not sent
CoAP Code	8	1	UP	2.05	Equal	Not sent
CoAP MID	16	1	BI	0	MSB(12)	LSB
CoAP Token	8	1	BI		Ignore	Value sent
CoAP Content Format		1	UP	11542	Equal	Not sent
Instance ID		1	UP		Ignore	Value sent
Sensor Value	32	1	UP		Ignore	Value sent
Sensor Units	8	1	UP	K, F, C	Match mapping	Mapping sent

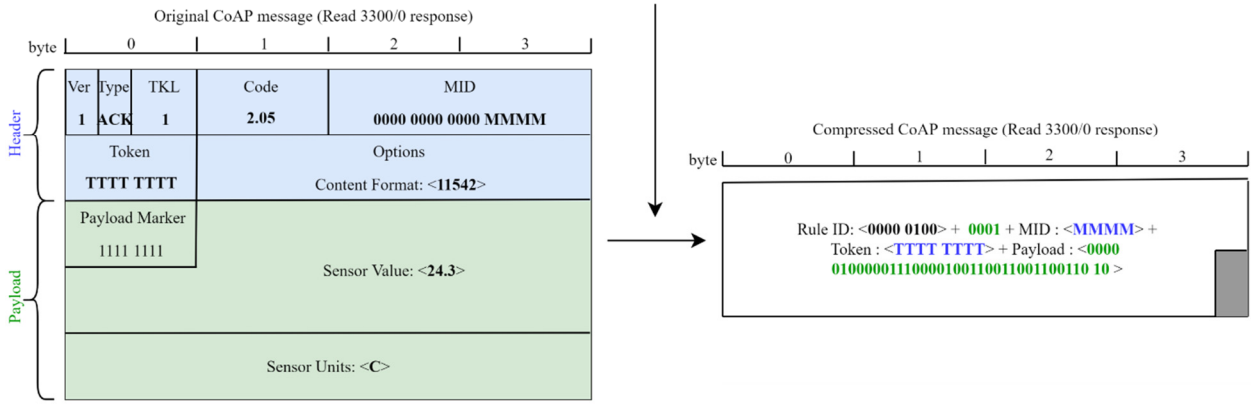


Figure 12. Extended SCHC used in LoRaWAN.

4.3.2. Performance Evaluation

To illustrate the benefits of introducing a Proxy that implements the compression function, motivated by the payload size issues illustrated in Section 3, we instantiated the proposed architecture in a LoRaWAN network [10].

To cope with such constrained underlying technology, we implemented the LoRaWAN binding in addition to the compression function. We assume that the device that runs the Client implements only the LWM2M application, so the communication is end-to-end with the Proxy and the IP and UDP layers becoming overhead and able to be easily removed. As a consequence, the packet size is reduced and the number of computations

and exchanged messages are reduced as well, because the Client does not have to manage the IP protocol stack.

The packet size is further reduced by applying the proposed compression mechanism based on the SCHC framework both on the Client and on the Proxy.

We implement a simple LoRaWAN system architecture composed of (see Figure 13):

1. a Client. The device that runs the Client is an Heltec WIFI LoRa 32 (V2) and implements a Server Object Instance (ObjectID: 1), a Device Object Instance (ObjectID: 3), and a Temperature Object Instance (ObjectID: 3303). Each of these Objects implements the mandatory resources. The Read, Write, and Execute operations are implemented for Server and Device Objects. The Observe, Notify, and Cancel Observation are implemented for Temperature Object.
2. a LoRaWAN Gateway. The gateway is a Laird Sentrius RG186. The gateway relays messages between devices and a network server in the backend using the Gateway Message Protocol (GWMP) [29]. Thus, communication between gateways and network server is via JSON/GWMP/UDP/IP.
3. a LoRaWAN Network Server. The network server routes the packets to the application server.
4. a LWM2M Proxy/LoRaWAN Application Server. The communication between the application server and the network server is via JSON over TCP over IP [30].
5. a Server.

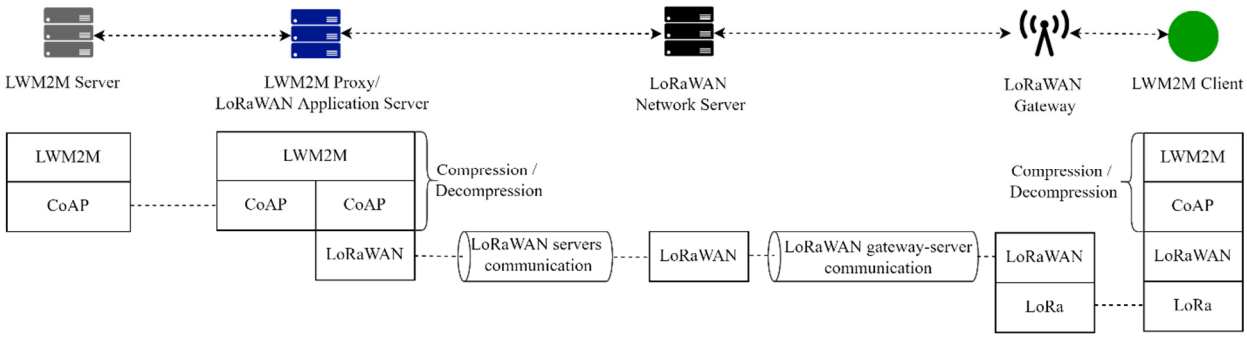


Figure 13. LWM2M architecture instantiated in a LoRaWAN network.

All the servers are Java servers and the Proxy and the Server use the Eclipse Leshan library [22] to implement the LWM2M protocol.

Table 3 shows compression results. Compression is applied both to Object Instances, i.e., when reading the Device Object Instance, the Server Object Instance, and the Temperature Object Instance, and when writing the Server Object Instance, and to Resources or to a set of Resources, i.e., when reading a single Resource of the Device or Server Object Instance and when writing one or more Resources of the Server Object Instance. When considering operations involving Resources, we obtain different compression ratios depending on the number of bytes used to represent the resource itself, e.g., the Lifetime Resource (ID: 1) of the Server Object is an integer, while the Binding Resource (ID: 7) of the Server Object is a string.

The compression function is effective because the size of the compressed packet is always smaller than that of the uncompressed one. This compression function can greatly reduce the packet size, especially when it conveys an Object Instance or an Object; indeed, from Table 3 we can observe that the maximum percentage reduction of the packet is 75% and is obtained when considering the response to a Read request for the Server Object Instance of the Client; and the mean percentage reduction of packets containing an Object Instance is 58.6%. However, the compression function can bring remarkable benefits also when the packets convey just one or more Resources: results show that for these packets

the mean percentage reduction is 46.1%. Moreover, the compressed message always fits in a LoRaWAN packet.

Table 3. Compression Results.

Type of Message	Read Object		Notify Object			Read Resource				Write Object	Write Resources			Write Resource		
	3/0	**1/0**	**3303/0**	**3/0/11**	**3/0/16**	**1/0/0**	**1/0/1**	**1/0/6**	**1/0/7**	**1/0**	**1/0/1 and 1/0/6**	**1/0/1 and 1/0/7**	**1/0/6 and 1/0/7**	**1/0/1**	**1/0/6**	**1/0/7**
Percentage reduction	58.8%	75%	44.4%	35.7%	37.5%	37.5%	21.4%	37.5%	37.5%	56%	54.5%	54.5%	63.2%	45%	64.3%	64.3%
Compression ratio	2.43	6	1.8	1.56	1.6	1.6	1.27	1.6	1.6	2.27	2.2	2.2	2.71	1.81	2.8	2.8
Fit in a LoRaWAN packet	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

5. Conclusions

In this work we considered LWM2M-based networks and we proposed the introduction of a LWM2M Proxy at the network edge, in between LWM2M Clients and LWM2M Servers. The Proxy is composed of a Core function that implements an optimized version of LWM2M and of one or more optional extended functions that support QoS-aware services. The optimized version of LWM2M allows to reduce the number of messages exchanged in the IoT network and to reduce the computations executed on the Client by decreasing the number of requests forwarded to the Clients. The extended functions are managed by the Core function to improve network performance and they tackle the issues of avoiding network congestion, of further reducing the number of requests forwarded to the Clients, and of reducing the packet size to deal with the limitations on packet payload size imposed by energy-constrained networks. Thus, we proposed that the Proxy implements the following extended functions: a (i) requests-shaping function that regulates the flow of requests sent over the IoT network, so that the Server experiments reduced delays and packet losses; a (ii) cache function that reduces the number of messages exchanged in the constrained network while guaranteeing information freshness measured by the Age of Information metrics, so that the Server experiments reduced response times and the power consumption of the Client is reduced; and a (iii) compressing function that compresses/decompresses the messages exchanged with the Client and that applies a SCHC-based compression also to the LWM2M payload. These functions can be applied singularly or together depending on the LWM2M Server demands and the LWM2M Proxy design allows to easily define new extended functions.

Experiments considering a Proxy that implements each of the proposed functions demonstrate that it can alleviate the traffic load of the IoT network, improving the QoS of the system in terms of a reduced latency and a reduced packet loss, and that it can reduce the size of the exchanged packets.

As future work we aim at analyzing the case of cellular networks where end-devices have high-mobility characteristics. Indeed, the Proxy will be required to be moved accordingly, to preserve proximity. To this aim, the Proxy can be seen as a network service that is migrated with the support of the underlying edge platform. It is worth noting that, since some functions of the Proxy are Client-dependent, i.e., the Core, the cache, and the compression functions, the migration should be stateful. Moreover, we also aim at investigating the scalability of our proposed solution in case of massive IoT deployments.

Author Contributions: Conceptualization: M.P., A.V. and E.M.; software: M.P.; writing—original draft: M.P., A.V. and E.M.; writing—review and editing: M.P., A.V. and E.M. All authors have read and agreed to the published version of the manuscript.

Funding: Work partially supported by the Italian Ministry of Education and Research (MIUR) in the framework of the CrossLab project (Departments of Excellence).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Shelby, Z.; Koster, M.; Bormann, C.; van der Stok, P.; Amsüss, C. *CoRE Resource Directory, Internet-Draft Draft-Ietf-Core-Resource-Directory-28*; IETF: Fremont, CA, USA, 2021.
2. Broadband Forum TR-069: "CPE WAN Management Protocol v1.4". Available online: https://www.broadband-forum.org/download/TR-069_Amendment-6.pdf (accessed on 24 January 2022).
3. Available online: http://onem2m.org/ (accessed on 24 January 2022).
4. Open Mobile Alliance. *Lightweight Machine to Machine Technical Specification: Core*; Open Mobile Alliance: San Diego, CA, USA, 2020.
5. Open Mobile Alliance. *Lightweight Machine to Machine Technical Specification: Transport Bindings*; Open Mobile Alliance: San Diego, CA, USA, 2020.
6. Beniwal, G.; Singhrova, A. A systematic literature review on IoT gateways. *J. King Saud Univ. Comput. Inf. Sci.* **2021**, *in press*. [CrossRef]
7. Pappalardo, M.; Tanganelli, G.; Mingozzi, E. Enhanced Support of LWM2M in Low Power and Lossy Networks. In Proceedings of the IEEE International Conference on Smart Computing (SMARTCOMP), Bologna, Italy, 14–17 September 2020.
8. Available online: https://datatracker.ietf.org/wg/6lowpan/documents/ (accessed on 24 January 2022).
9. Yates, R.D.; Sun, Y.; Brown, D.R.; Kaul, S.K.; Modiano, E.; Ulukus, S. Age of information: An introduction and survey. *IEEE J. Sel. Areas Commun.* **2021**, *39*, 1183–1210. [CrossRef]
10. LoRa Alliance. LoRaWAN 1.1 Specification. 2017. Available online: https://resources.lora-alliance.org/technical-specifications/lorawan-specification-v1-1 (accessed on 24 January 2022).
11. Filho, F.L.; Rocha, R.L.; Abbas, C.J.B.; Martins, L.M.C.E.; Canedo, E.D.; de Sousa, R.T. QoS Scheduling Algorithm for a Fog IoT Gateway. In Proceedings of the Workshop on Communication Networks and Power Systems (WCNPS), Brasilia, Brazil, 3–4 October 2019.
12. Mingozzi, E.; Tanganelli, G.; Vallati, C. CoAP Proxy Virtualization for the Web of Things. In Proceedings of the IEEE 6th International Conference on Cloud Computing Technology and Science, Singapore, 15–18 December 2014.
13. Ludovici, A.; Calveras, A. A proxy design to leverage the interconnection of coap wireless sensor networks with web applications. *Sensors* **2015**, *15*, 1217–1244. [CrossRef] [PubMed]
14. Lai, W.K.; Wang, Y.C.; Lin, S.Y. Efficient Scheduling, Caching, and Merging of Notifications to Save Message Costs in IoT Networks Using CoAP. *IEEE Internet Things J.* **2020**, *8*, 1016–1029. [CrossRef]
15. Robles, M.I.; D'Ambrosio, D.; Bolonio, J.J.; Komu, M. Device Group Management in Constrained Networks. In Proceedings of the IEEE International Conference on Pervasive Computing and Communication Workshops, Sydney, Australia, 14–18 March 2016.
16. Hoebeke, J.; Haxhibeqiri, J.; Moons, B.; van Eeghem, M.; Rossey, J.; Karagaac, A.; Famaey, J. A Cloud-based Virtual Network Operator for Managing Multimodal LPWA Networks and Devices. In Proceedings of the 2018 3rd Cloudification of the Internet of Things (CIoT), Paris, France, 2–4 July 2018.
17. Sanchez-Gomez, J.; Gallego-Madrid, J.; Sanchez-Iborra, R.; Santa, J.; Skarmeta, A.F. Impact of SCHC compression and fragmentation in LPWAN: A case study with LoRaWAN. *Sensors* **2020**, *20*, 280. [CrossRef] [PubMed]
18. Available online: https://github.com/contiki-ng/cooja (accessed on 24 January 2022).
19. *IEEE Std 802.15.4-2015*; IEEE Standard for Low-Rate Wireless Networks. IEEE: Piscataway Township, NJ, USA, 2016.
20. Winter, T.; Thubert, P.; Brandt, A.; Hui, J.; Kelsey, R.; Levis, P.; Pister, K.; Struik, R.; Vasseur, J.P.; Alexander, R. *RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks*; RFC 6550; IETF: Fremont, CA, USA, 2012.
21. Farrell, S. *Low-Power Wide Area Network (Lpwan) Overview*; RFC 8376; IETF: Fremont, CA, USA, 2018.
22. Available online: https://github.com/eclipse/leshan (accessed on 24 January 2022).
23. Zhong, J.; Yates, R.D.; Soljanin, E. Two Freshness Metrics for Local Cache Refresh. In Proceedings of the IEEE International Symposium on Information Theory (ISIT), Vail, CO, USA, 17–22 June 2018.
24. Pappalardo, M.; Mingozzi, E.; Virdis, A. A Model-Driven Approach to AoI-Based Cache Management in IoT. In Proceedings of the IEEE International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD), Porto, Portugal, 25–27 October 2021.
25. FUOTA Working Group of the LoRa Alliance Technical Committee. LoRaWAN Fragmented Data Block Transport Specification. 2018. Available online: https://lora-alliance.org/wp-content/uploads/2020/11/fragmented_data_block_transport_v1.0.0.pdf (accessed on 24 January 2022).
26. Minaburo, A.; Toutain, L.; Gomez, C.; Barthel, D.; Zúñiga, J.C. *SCHC: Generic Framework for Static Context Header Compression and Fragmentation*; Technical Report RFC 8724; IETF: Fremont, CA, USA, 2020.

27. Gomez, C.; Minaburo, A.; Toutain, L.; Barthel, D.; Zuniga, J.C. IPv6 over LPWANs: Connecting Low Power Wide Area Networks to the Internet (of Things). *IEEE Wirel. Commun.* **2020**, *27*, 206–213. [CrossRef]
28. Minaburo, A.; Toutain, L.; Andreasen, R. *Static Context Header Compression (SCHC) for the Constrained Application Protocol (CoAP)*; RFC 8824; IETF: Fremont, CA, USA, 2021.
29. Semtech Ltd. *LoRaWAN Network Server Demonstration: Gateway to Server Interface Definition*. 2015. Available online: https://www.thethingsnetwork.org/forum/uploads/default/original/1X/4fbda86583605f4aa24dcedaab874ca5a1572825.pdf (accessed on 24 January 2022).
30. Semtech Ltd. *LoRaWAN Network Server Demonstration: Inter-Server Interface Definition*. 2015. Available online: https://www.thethingsnetwork.org/forum/uploads/default/original/1X/555030509bdcdee51a0d3d87382a17dd6211b11c.pdf (accessed on 24 January 2022).