electronics

MDPI

Article

A LoRaWAN Network Architecture with MQTT2MULTICAST

Jorge Navarro-Ortiz [1,2,*], Natalia Chinchilla-Romero [1,2], Felix Delgado-Ferro [1,2] and Juan Jose Ramos-Munoz [1,2]

1 Department of Signal Theory, Telematics and Communications, University of Granada, 18071 Granada, Spain; nataliachr@ugr.es (N.C.-R.); felixdelgado@ugr.es (F.D.-F.); jjramos@ugr.es (J.J.R.-M.)
2 Research Center on Information and Communication Technologies, University of Granada, 18014 Granada, Spain
* Correspondence: jorgenavarro@ugr.es

Abstract: In this work, an architecture for IoT networks oriented towards environmental sustainability is presented. Due to the suitability of its characteristics in terms of coverage, power and support of a large number of devices, an enhanced LoRaWAN network has been chosen as the basis for this proposal. The architecture is completed with the virtualization of the different LoRaWAN network entities and the usage of a software-defined network for their interconnection. The publication and subscription to environmental data is carried out by using the MQTT protocol. MQTT has been optimized thanks to the use of the SDN network and the use of edge computing resources, which allows multicasting of published data. Thanks to our developed MQTT2MULTICAST protocol, latency is improved by approx. 90% and the traffic load within the SDN network is reduced by 55%. An scalability analysis shows that this solution is able to support tens of thousands of LoRaWAN gateways. The proposed architecture has been implemented using commercial equipment as a proof of concept.

Keywords: IoT; LPWAN; LoRaWAN; MQTT; SDN

check for updates

Citation: Navarro-Ortiz, J.; Chinchilla-Romero, N.; Delgado-Ferro, F.; Ramos-Munoz, J.J. A LoRaWAN Network Architecture with MQTT2MULTICAST. *Electronics* **2022**, *11*, 872. https://doi.org/10.3390/electronics11060872

Academic Editor: Cheng-Chi Lee

Received: 30 January 2022
Accepted: 6 March 2022
Published: 10 March 2022

1. Introduction

The objective of this work is the design of an IoT (Internet of Things) network architecture for the collection, processing and distribution of environmental information. The proposed architecture is intended to be flexible and powerful enough to integrate, in the future, the different solutions developed by the research group in this field.

The network is made up of several parts. On the one hand, a radio access network based on an LPWAN (Low Power Wide Area Network) suitable for mass sensor communications. More specifically, a LoRaWAN (Long Range Wide Area Network) [1] network has been chosen due to its adequate characteristics of low power consumption, high coverage, easy scalability, and its license free operation, which facilitates its development and reduces costs.

On the other hand, a backbone network that will integrate the different entities necessary for this type of networks and for the processing and distribution of data. Since a LoRaWAN network is being used for the radio part, the backbone network will include the required network and application servers. These types of servers usually utilizes the MQTT (Message Queuing Telemetry Transport) [2] protocol to exchange information with both internal and external entities. Many of these entities receive the same information, which makes multicast suitable and more efficient for this type of networks. For this reason, as will be explained in the following sections, we will introduced a multicast-based solution that will reduce both MQTT generated traffic and its latency. An Artificial Intelligence (AI) platform will also be included for data processing. All entities will be implemented as Cloud-Native Network Functions (CNFs) to facilitate their orchestration, deployment and execution in local clouds.

A Software Defined Network (SDN) will be used for the communication between the radio access network and the backbone network. SDN provides great flexibility and ease of development to include, e.g., new protocols or optimizations over existing ones.

Main Contributions

The main contributions of this paper are enumerated as follows:

- This paper presents an IoT network architecture for the collection, processing and distribution of environmental information.
- This architecture is based on LoRaWAN, with the addition of SDN and edge computing paradigms. This allows us to improve the scalability and the innovation capabilities of the proposed architecture.
- Based on the SDN network and the edge computing nodes, we propose an optimization of the MQTT protocol, protocol used to publish/subscribe data. For that purpose, MQTT proxies are deployed in edge nodes which multicast MQTT data to other edge nodes. Our solution does not require any modification to end nodes.
- Since multicast requires using UDP (User Datagragram Protocol) instead of TCP (Transport Control Protocol), which does not establish a connection, MQTT latency within the SDN is reduced.
- Although another multicast routing strategies could be employed, in our proposal the SDN controller creates source-based trees using the shortest paths, following an approach similar to PIM-SSM (Protocol Independent Multicast—Source Specific Mode). This further reduces the MQTT latency compared to other non-optimal multicast routing based on, e.g., shared trees. An scalability analysis shows that the computation time for multicast paths is reasonable for large networks (up to a few thousand of edge nodes or, equivalently, tens of thousands of LoRaWAN networks).
- We have created a control protocol, *MQTT2MULTICAST*, which maps MQTT topics to multicast IP addresses. MQTT2MULTICAST also includes the required signaling to join/leave a multicast group (i.e., an MQTT topic), thus removing the need for IGMP or similar protocols. This further reduces the traffic within the SDN network.
- A prototype has been developed as Proof of Concept. Our prototype includes real LoRaWAN equipment connected to an SDN network based on *Mininet* and the RYU SDN controller. Different LoRaWAN nodes send data through the SDN network over MQTT, received by an MQTT subscriber connected to an edge node, and finally visualized using Grafana.
- Our testbed environment is released as an open-source project [3], allowing for reproducible research.

This article is organized into the following sections. The current section introduces the objectives of the work and its context. Section 2 conducts a review of the State of the Art. Section 3 gives an overview of LoRaWAN. Section 4 describes the design of the IoT network architecture, whereas the MQTT enhancements are explained in Section 5. Section 6 shows our network prototype, carried out as a proof of concept. Finally, Section 7 presents the performance evaluation results, concluding the article in Section 8.

2. State of the Art

In this section we will first present a brief summary of the different possibilities to deploy a LoRaWAN network. Other aspects such as platforms for IoT, more focused on data storage, distribution, processing and visualization (such as Google Cloud Platform, IBM Watson IoT, Amazon AWS IoT Core, Microsoft Azure, among others) are outside the scope of this work.

An example of a large LoRaWAN network architecture is The Things Networks (TTN), an open, collaborative network with more than 20,000 gateways around the world. Its architecture [4] is made up of bridges that interact with gateways, connected to routers that route to the corresponding brokers (depending on the geographical location). These interact with network servers and handlers that manage communication with application servers.

There is no detailed information about the real elements composing the architecture or their organization.

Regarding LoRaWAN platforms, it is worth highlighting Chirpstack [5], a free and open source platform that allows to deploy private networks, and LORIOT [6], free for small networks (up to 30 nodes) but without the possibility of setting up private networks. Chirpstack provides the different entities (LoRaWAN network and application servers, and a Gateway Bridge) so it is up to the developer which network topology to use or other deployment details. LORIOT includes 13 free community network servers, a professional LoRaWAN network server with a 99.9% SLA, built-in redundancy and new accessible features to manage and scale a secure LoRaWAN network, and a carrier-grade private LoRaWAN network server solution for production services at any scale. Apart from this, no further details about their architecture are given.

It shall be noticed that, since the Chirpstack platform freely provides all the entities required to deploy a private network, we will use Chirpstack for our deployment.

Additionally, since this work will enhance MQTT in our platform, we also include here some relevant works found in the literature that optimize the MQTT protocol for IoT deployments.

One of the most relevant modifications of the MQTT protocol for IoT is MQTT-SN (MQTT for Sensor Networks) [7]. MQTT-SN is designed for wireless networks, with low power battery operated sensors with very limited processing power and storage, with a limited payload size and not always connected (i.e., sleeping). With this objective in mind, it uses UDP as transport protocol and topic identifiers instead of topic names. The architecture also changes with respect to MQTT, with MQTT-SN clients, gateways and forwarders. The gateways act as MQTT-SN brokers and also translate to MQTT to forward messages to MQTT brokers (aggregating clients or not). The forwarders just retransmit MQTT-SN packets from clients to gateways. MQTT-SN utilizes multicast for discovery, but data is sent using unicast. The main differences with our solution is that nodes do not use standard MQTT and that messages are not multicasted.

Another work, MQTT-ST (MQTT Spanning Tree), focuses on the communication between distributed MQTT brokers. Its main objective is to avoid potentially harmful message loops between brokers. After an initial phase, at runtime, the MQTT-SN brokers work exactly as MQTT brokers. Although this optimization may be very useful, it does not contemplate multicasting or any other feature to reduce latency or traffic generation.

Direct Multicast-MQTT (DM-MQTT) [8] follows a similar approach to our solution, since it multicasts MQTT messages within an SDN network. However, for QoS (Quality of Service) level 0, multicast is performed directly between publishers and subscribers without brokers. This may create scalability issues in the SDN controller if we have a large number of MQTT clients. In addition, they use a CBT (core-based tree) with a rendezvous point, which may not be optimal from a delay point of view. Although a CBT is more scalable than an SBT (source-based tree, which uses the shortest paths), our proposal aggregates a large number of MQTT clients in a few number of MQTT proxies, so using a source-based tree does not represent a scalability problem and additionally optimizes delay. Furthermore, the authors do not explain how a TCP-based protocol such as MQTT can be sent using multicast. In our work, MQTT proxies forward MQTT messages over UDP, which allows multicasting. The mapping between MQTT topics and multicast IP addresses is done thanks to a new MQTT2MULTICAST protocol. We also propose an aggregation scheme for MQTT topics used by LoRaWAN in order to reduce the number of multicast groups. Another difference is that DM-MQTT requires a hierarchical structure for the MQTT brokers, with one master broker located on the SDN controller (although logically separated). In our solution, all MQTT proxies are equal which also reduces the possibility of bottlenecks.

Finally, MQTT multicast [9] is used to allow a single publisher to reach a broad audience of subscribers without the intervention of an MQTT broker. This software is intended for GNUnet [10], a free software for decentrilized P2P (Peer-to-Peer) networks. In

MQTT multicast, a publisher creates a GNUnet-MULTICAST channel for each topic to start a group. The main differences with our solution is that (1) it requires a P2P network, (2) the multicasting is done by the publishers, i.e., messages and topics are not aggregated by brokers or proxies, and (3) it is not compatible with already existing MQTT clients.

3. Technical Overview of LoRaWAN

LoRaWAN is an open standard which specifies the network architecture and the communication protocol to connect between each entity of the network. The design of the network architecture and the relates protocols are determined by the network capacity, the battery lifetime of the nodes, the security, and the applications that can run over the network [11]. To illustrate a typical LoRaWAN network, Figure 1 shows a star-of-stars topology where one or more gateways (concentrators) relay download and upload traffic among end-nodes (mobile or fixed in a location) and a central network server respectively. The network server steers the traffic between each end-device and the associated application server. The communications are bi-directional. Communications between gateways and end-devices are a single-hop LoRa (the physical layer of a LoRaWAN network) or FSK (Frequency Shift Keying) connections. Communcations between gateways and the network servers use standard IP.

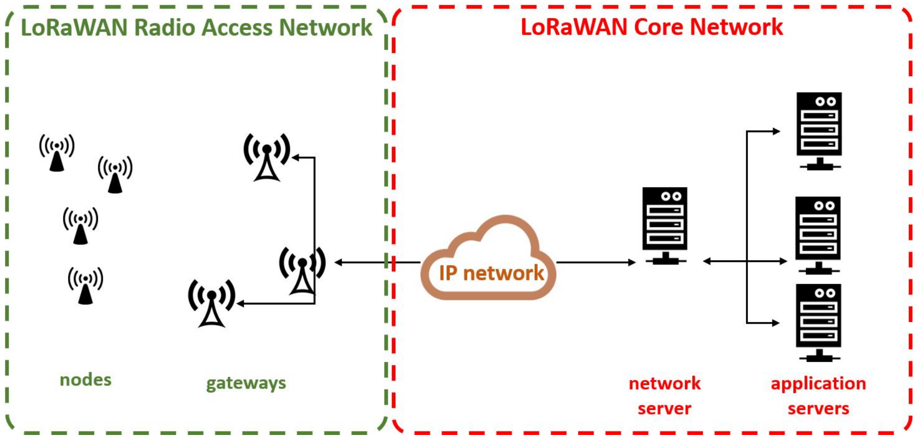


Figure 1. LoRaWAN network architecture.

LoRa is a chirp spread spectrum based modulation developed by Cycleo in 2009 and acquired by Semtech in 2012. LoRa modulation complexity is low thanks to the equivalent offsets in timing and frequency between the transmitter and the receiver. The data signal is modulated onto a chirp signal that changes its frequency with time. The chirp rate defines the spectral bandwidth (BW) of a LoRa signal. For instance, a spectral BW of 125 kHz corresponds to a chip rate of 125,000 chips/sec. The data rate can change depending on the employed spreading factor (SF), if we assume a fixed BW. The number of raw bits carried per symbol is expressed as SF, and varies between 7 and 12. LoRa modulation includes a FEC (Forward Error Correction) scheme which adds redudancy in the code to find and correct erroneous bits. This improves the robustness of the transmitted signal [12]. CR is the Code Rate and its values can vary between 1 and 4.

The data rate R_b is calculated as follows:

$$R_b = SF \times \left(\frac{BW}{2^{SF}}\right) \times \left(\frac{4}{4+CR}\right) \tag{1}$$

where SF is the spreading factor, the second term corresponds to R_s or symbol rate (symbols/sec) and the third term depends on the coding rate (CR). Thus, assuming fixed BW and coding rate, the data rate decreases as the SF increases.

Table 1 presents a data rate (DR) configuration assuming a 125 kHz bandwidth and the default LoRaWAN Code Rate (4/5). The bit rate increases and the ToA (Time on Air) decreases as the SF is decremented.

Table 1. Data Rate configuration for a 125 kHz bandwidth in the EU868 ISM band.

Data Rate	Spreading Factor	Bit Rate (bps)	ToA for a 24B Packet (ms)
0	12	293	1482.75
1	11	540	823.30
2	10	980	411.65
3	9	1757	205.82
4	8	3125	113.15
5	7	5470	61.70

The communication between end-nodes and gateways uses different spreading factors and channels. In every transmission, the end-device selects the transmission channel in a pseudo-random way. The SF (or DR) is selected considering the message duration and communication range by using an Adaptive Data Rate (ADR) scheme [1]. In each transmission, the DR is selected considering the link budget (the higher the link budget, the faster DR or SF selected). Parallel transmissions in the same channel but in different SFs will not collide due to the orthogonality of the SFs. This can be used to improve the network capacity.

Three classes of LoRaWAN end-devices are considered in the LoRaWAN standard (classes A, B and C), depicted in Figure 2. Class A is mandatory and the most employed, typical for battery-operated sensors. In Class A devices, two receiving windows are opened after an uplink transmission in order to receive a downlink transmission. Class B devices are typically battery-operated actuators. These devices behave similar, but use extra receive windows scheduled by beacon frames. Class C devices are typically mains powered. They are continuously listening except when transmitting.

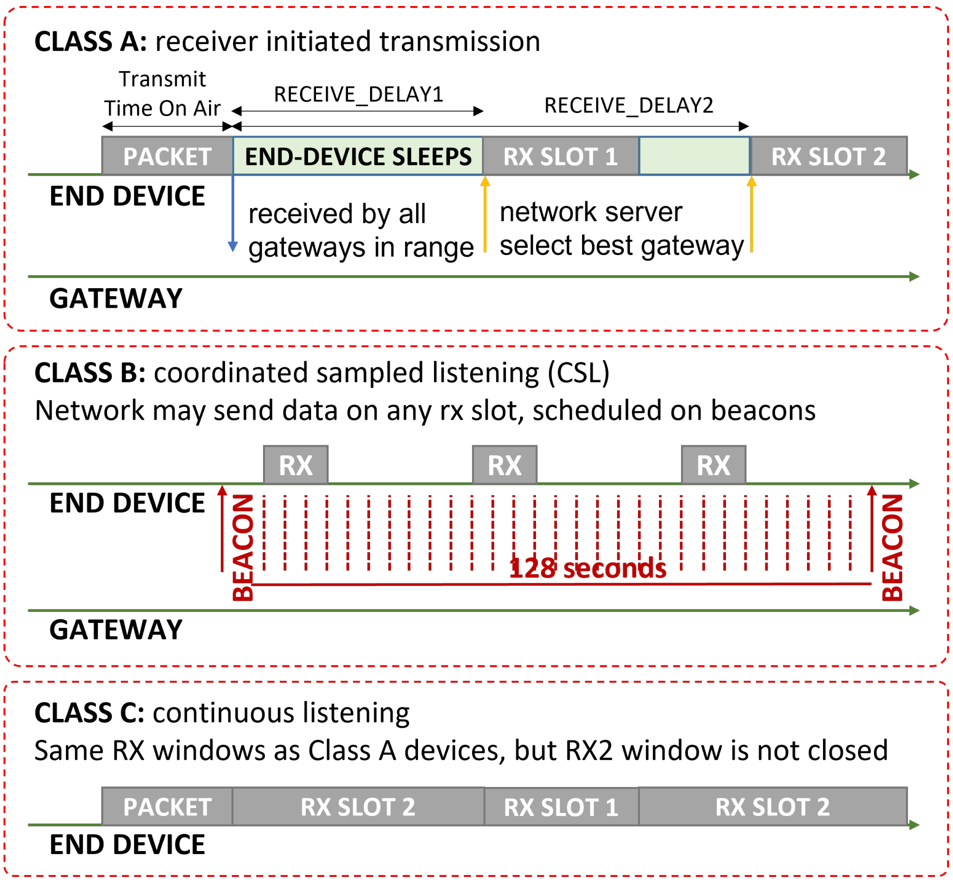

CLASS A: receiver initiated transmission
Transmit Time On Air
RECEIVE_DELAY1
RECEIVE_DELAY2
PACKET
END-DEVICE SLEEPS
RX SLOT 1
RX SLOT 2
END DEVICE
received by all gateways in range
network server select best gateway
GATEWAY
CLASS B: coordinated sampled listening (CSL)
Network may send data on any rx slot, scheduled on beacons
RX
RX
RX
END DEVICE
BEACON
128 seconds
BEACON
GATEWAY
CLASS C: continuous listening
Same RX windows as Class A devices, but RX2 window is not closed
PACKET
RX SLOT 2
RX SLOT 1
RX SLOT 2
END DEVICE

Figure 2. LoRaWAN device classes.

Regarding frequency spectrum usage, LoRaWAN uses license-free sub-gigahertz radio frequency bands like 433 MHz, 868 MHz, and 915 MHz depending on the country. In Europe, LoRaWAN networks operates in the EU863-870 ISM band which allows up to eight channels with a 125 kHz bandwidth each, being three of them mandatory (868.1, 868.3, and 868.5 MHz). In order to respect the different regulations for these frequency bands, LoRaWAN devices are limited to a duty-cycle, being 1% in Europe due to ETSI (European Telecommunications Standards Institute) regulations.

4. SDN-Based IoT Network Architecture Design

Figure 3 shows the proposed architecture, which consists of an SDN network connecting edge nodes. These nodes will implement MQTT proxy functionality and/or AI processing. These edge nodes will connect with a node or cluster that will implement the different entities of a LoRaWAN network, that is, the network server, the application server and a bridge that allows the connection between the network server and the gateway, as well as the necessary databases. The gateway allows the radio connection with the LoRaWAN nodes, which will be the sensors and actuators that send environmental information. A typical LoRaWAN network would also include an MQTT broker, which in this case is replaced by the MQTT proxy.

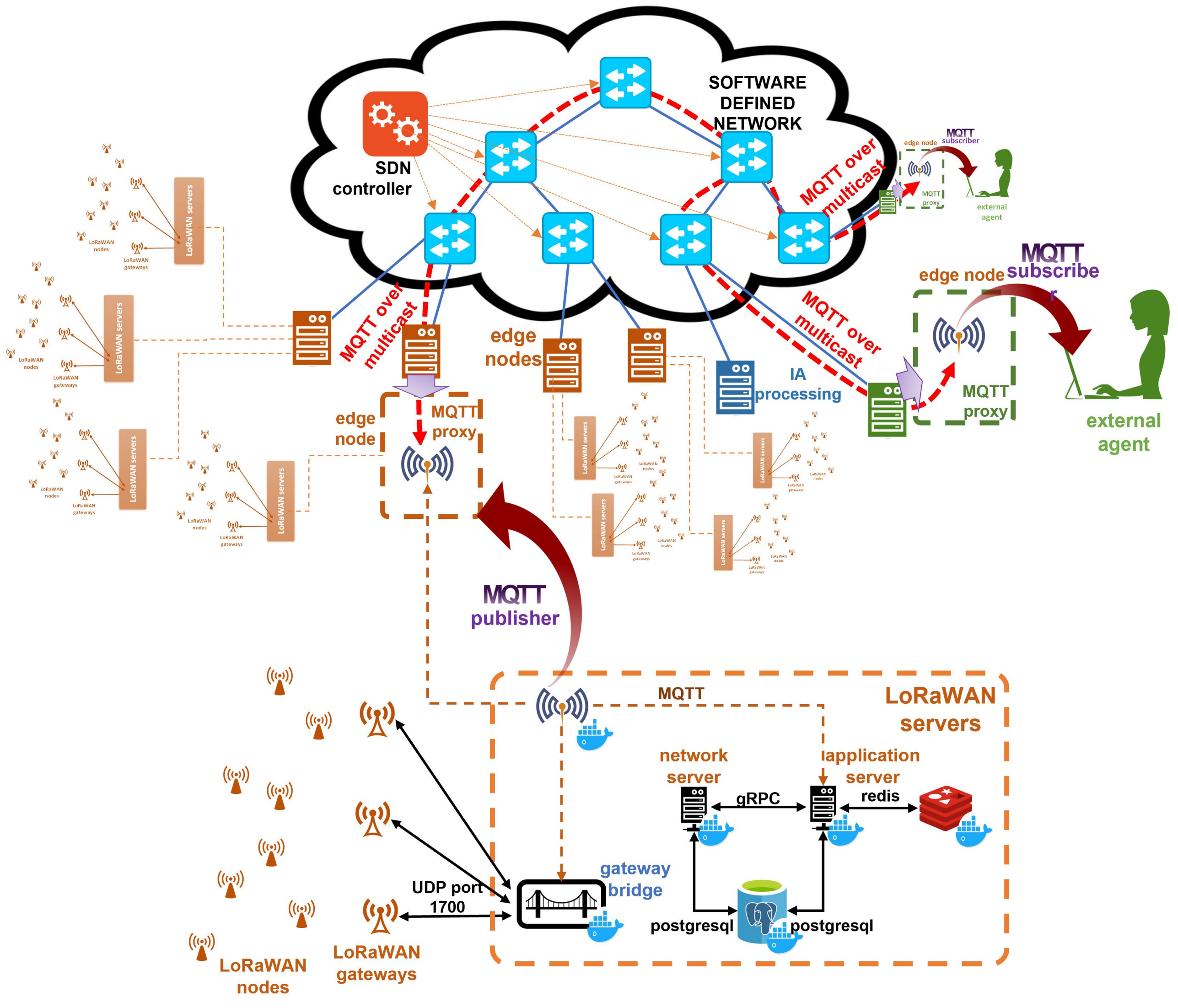


Figure 3. Proposal for an IoT network architecture for the collection, processing and use of environmental variables.

These proxies forward MQTT messages using UDP between them, which reduces the required Round-Trip Times (RTTs)—and thus latency—and allows the multicasting of these messages, thus reducing the amount of traffic within the SDN network. These enhancements are explained in Section 5.

In order to reduce the number of MQTT topics between proxies, which will be mapped to multicast IP addresses (see Section 5.3), we propose to follow the TTN approach with "`application/[ApplicationID]/device/[DevEUI]/[EventType]`" as topic. By aggregating them based on their `ApplicationID`, all the MQTT messages corresponding to the same application will be mapped to the same multicast IP address.

Finally, we propose that the SDN network follows a tree topology, which will (1) allows to have hierarchical levels (e.g., local, regional and national), and (2) will simplify the routing for multicast (which are based on trees).

5. MQTT Enhancements in the SDN Segment

This section explains the MQTT enhancements that have been accomplished thanks to the use of the SDN network and the MQTT proxies in our architecture. These improvements are based on the usage of UDP for MQTT and the utilization of multicast for the publication of the messages.

5.1. MQTT

In order to have a better understanding of the MQTT protocol, Figure 4 depicts the messages exchanged between the publisher and the subscriber through the MQTT broker. All these messages are transmitted over TCP (port 1883 by default, or port 8883 for secure MQTT, i.e., over TLS).

First, both the publisher and the subscriber send a connection request to the broker. Additionally, the subscriber sends a subscription request for a specific topic. MQTT has 3 Quality of Service (QoS) levels. When the publisher publish some data with an MQTT Publish message, there may be different messages interchanges between the MQTT entities, depending on the QoS that it uses. As Figure 4 illustrates, in the lowest QoS level 0, the sender (publisher or broker) sends the message and no longer cares whether it has been received by the other party or not. With QoS 1, the message is guaranteed to be received at least once. For that purpose, it employs the Publish Acknowledgment message. Finally, QoS 2 implies that the message is published exactly once. For that purpose, it employs the Publish Received, Publish Release, and Publish Complete messages. Finally, the publisher disconnects. The subscriber may continue with the MQTT connection, so it can receive new Publish messages from other publishers, or it may opt for also finishing the connection.

5.2. MQTT Proxies

As described in Section 4, we introduced MQTT proxies in the edge nodes of the SDN network in our architecture. MQTT proxies reduce the amount of MQTT traffic within the SDN network and its latency.

For this, MQTT proxies act as MQTT brokers with respect to the clients directly connected, implementing the same security mechanisms, e.g., using MQTT over TLS/SSL. In this architecture, several entities of the LoRaWAN network will use the proxies to replace the broker: the gateway bridge, the network server and the application server. In addition, the data published through MQTT can be queried by external entities ("external agent" in Figure 3). To do this, these MQTT subscribers will in turn connect to another MQTT proxy that will act as their broker and exchange the necessary information with the other proxy using UDP. The advantage of using UDP is that latency is reduced (several Round-Trip Times (RTT) in case of using TCP) and it will allow us to multicast the messages. Furthermore, although it has not been included in our proof of concept, the solution given in [13] can be used to guarantee the reliability of UDP over SDN. Moreover, MQTT over UDP may utilize DTLS (Datagram Transport Layer Security) in order to secure the communications, as done for MQTT-SN [14].

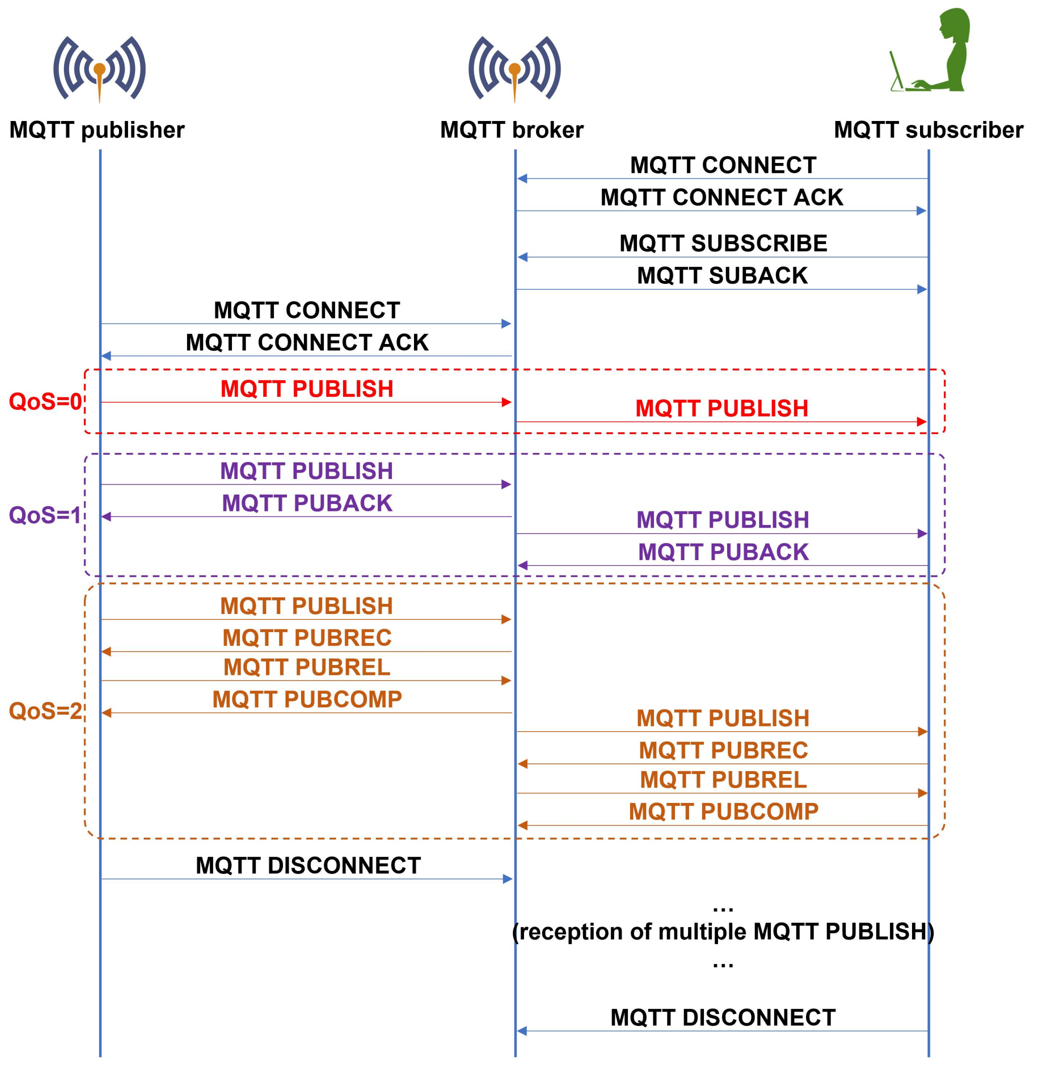


Figure 4. MQTT messages between publisher and subscriber through the broker.

Specifically, the proxies will forward subscription messages (*Subscribe*)—when a new topic appears that did not have previous subscribers in that proxy; publication (*Publish*)—for all messages published in topics that have a subscriber in the destination proxy; and *Disconnect*—when the last subscriber to that topic in that proxy is disconnected. In this way, the proxies will have the necessary information to forward the messages of active topics.

Additionally, they will employ our MQTT2MULTICAST protocol (see Section 5.3), which allows the proxies to map the MQTT topics to their corresponding multicast IP addresses.

5.3. *MQTT2MULTICAST Protocol*

We have developed MQTT2MULTICAST with two objectives. Its main purpose is to provide a mapping between MQTT topics and the IP address of the multicast group. Furthermore, taking advantage of these messages, MQTT2MULTICAST also allows adding or removing members to the group. In that sense, it eliminates the need of IGMP and thus it further reduces the amount of traffic on the network.

MQTT2MULTICAST is composed of two different messages, *MQTT2MULTICAST Request* and *MQTT2MULTICAST Reply*, which are exchanged between the MQTT proxies and the MQTT2MULTICAST server (implemented as an application at the SDN controller), see Figure 5.

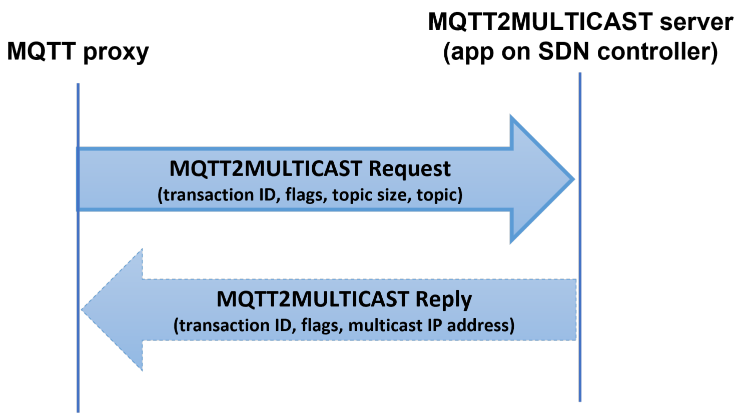


Figure 5. MQTT2MULTICAST message exchange.

Request are sent by MQTT proxies in order to ask for the multicast IP address assigned to a specific MQTT topic. The message contains four different fields: a *transaction ID* (2 bytes, used to match request and reply messsages), *flags* (1 byte, which indicates the reason for the request), *topic size* (2 bytes) and *topic* (MQTT topic whose size is determined by the *topic size* field).

The first bit (*subscribe/publish*) of the *flags* field is set to 0 if the reason for the *Request* message is that an MQTT publisher is sending an *MQTT Publish* message through the proxy, or 1 if there is an MQTT subscriber sending an *MQTT Subscribe* message for a new topic. The MQTT proxy will not send *Requests* for new subscribers to topics that have been already announced from the MQTT proxy to the MQTT2MULTICAST server. The second bit (*unsubscribe*) of the *flags* field is set to 1 if the last subscriber to a specific topic is sending an *MQTT Unsubscribe* message to the MQTT proxy.

The response to the previous *MQTT2MULTICAST Request* is an *MQTT2MULTICAST Reply*, which contains a *transaction ID* (which shall match the value from the corresponding request), *flags* (set to 0, for future use) and the multicast IP address (assigned to the MQTT topic included in the request).

These messages allow us to achieve the following functionalities:

- The first time an MQTT publisher or subscriber sends an MQTT message about a new topic, the MQTT2MULTICAST server assigns the next available multicast IP address from its addresses pool for that topic. This mapping is saved onto an internal database, and will be used for all future messages related to that topic.
- During the previous mapping, the SDN controller will also save the publishers (sources) and subscribers (destinations) for that topic. With this data and the network topology information (obtained from LLDP and ARP messages), it will generate the required multicast routing trees. Since MQTT proxies aggregate MQTT messages from a large amount of publishers and subscribers, we have opted for source-based trees (SBT), in order to optimize the delay between MQTT proxies. These trees are then transferred to the SDN switches by using one flow entry that redirects to a group table entry. This group table entry with type *ALL* in which the buckets represent the switch ports belonging to the tree, i.e., which shall forward the multicast message.
- When new devices subscribe to a topic that have been already processed by the MQTT proxy (i.e., there are already subscribers for that topic connected to that proxy), there is no need to send an MQTT2MULTICAST request to the SDN controller, since the MQTT proxy is already included in the corresponding multicast trees.
- When the last subscriber for a given topic unsubscribes, the MQTT proxy shall inform the SDN controller in order to be removed from the corresponding multicast trees.

Figure 6 depicts the aforementioned behavior.

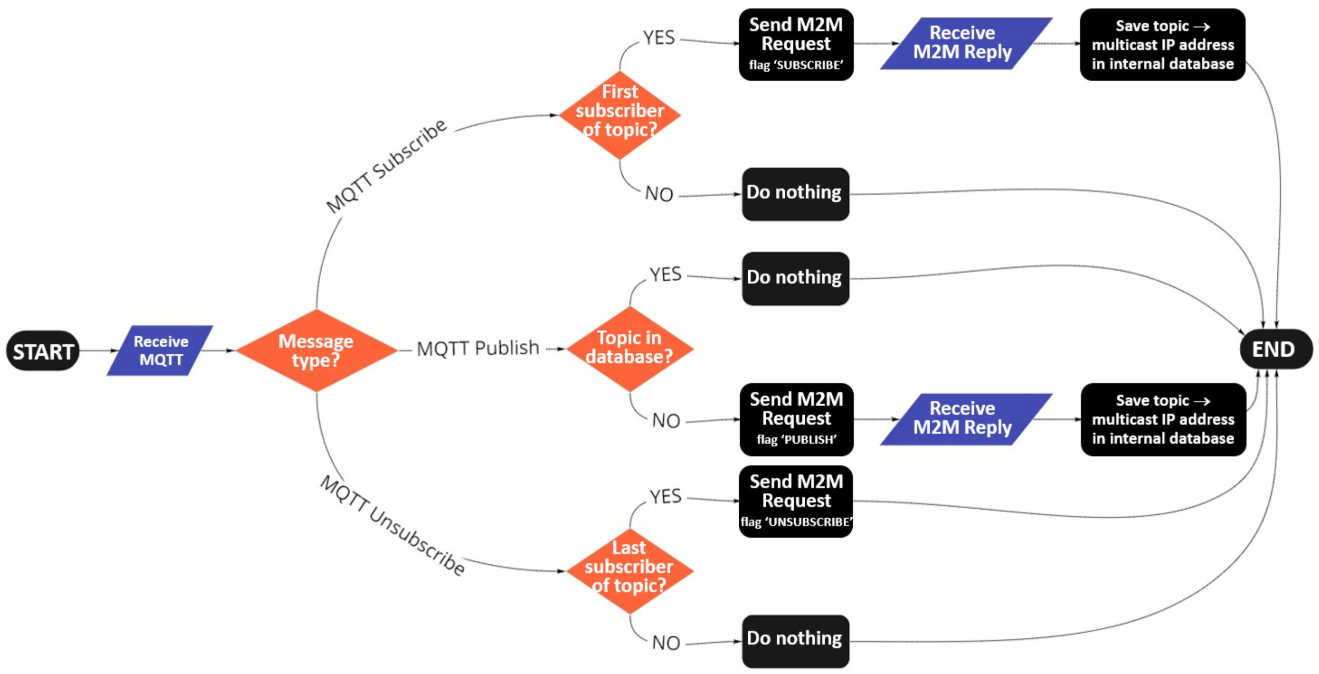

START
Receive MQTT
Message type?
MQTT Subscribe
MQTT Publish
MQTT Unsubscribe
First subscriber of topic?
YES
Send M2M Request flag 'SUBSCRIBE'
Receive M2M Reply
Save topic → multicast IP address in internal database
NO
Do nothing
Topic in database?
YES
Do nothing
NO
Send M2M Request flag 'PUBLISH'
Receive M2M Reply
Save topic → multicast IP address in internal database
Last subscriber of topic?
YES
Send M2M Request flag 'UNSUBSCRIBE'
NO
Do nothing
END

Figure 6. MQTT2MULTICAST flowchart from the MQTT proxy point of view.

5.4. Global Operation

Figure 7 includes the messages exchanged between the different entities (MQTT, MQTT2MULTICAST and MQTT over UDP (multicasting)), providing a global overview of how all the elements in our architecture interact.

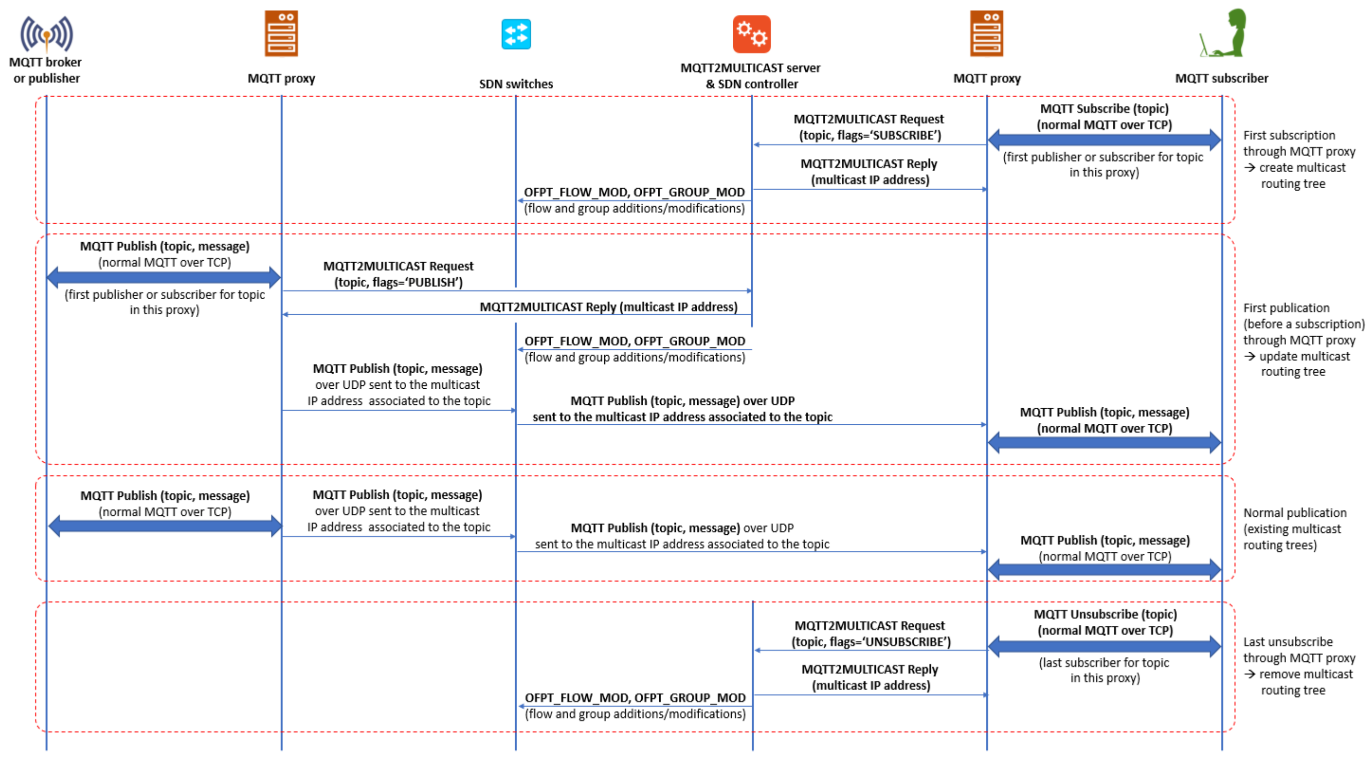

MQTT broker or publisher
MQTT proxy
SDN switches
MQTT2MULTICAST server & SDN controller
MQTT proxy
MQTT subscriber
MQTT Subscribe (topic) (normal MQTT over TCP)
(first publisher or subscriber for topic in this proxy)
MQTT2MULTICAST Request (topic, flags='SUBSCRIBE')
MQTT2MULTICAST Reply (multicast IP address)
OFPT_FLOW_MOD, OFPT_GROUP_MOD (flow and group additions/modifications)
First subscription through MQTT proxy → create multicast routing tree
MQTT Publish (topic, message) (normal MQTT over TCP)
(first publisher or subscriber for topic in this proxy)
MQTT2MULTICAST Request (topic, flags='PUBLISH')
MQTT2MULTICAST Reply (multicast IP address)
OFPT_FLOW_MOD, OFPT_GROUP_MOD (flow and group additions/modifications)
MQTT Publish (topic, message) over UDP sent to the multicast IP address associated to the topic
MQTT Publish (topic, message) over UDP sent to the multicast IP address associated to the topic
MQTT Publish (topic, message) (normal MQTT over TCP)
First publication (before a subscription) through MQTT proxy → update multicast routing tree
MQTT Publish (topic, message) (normal MQTT over TCP)
MQTT Publish (topic, message) over UDP sent to the multicast IP address associated to the topic
MQTT Publish (topic, message) over UDP sent to the multicast IP address associated to the topic
MQTT Publish (topic, message) (normal MQTT over TCP)
Normal publication (existing multicast routing trees)
MQTT Unsubscribe (topic) (normal MQTT over TCP)
(last subscriber for topic in this proxy)
MQTT2MULTICAST Request (topic, flags='UNSUBSCRIBE')
MQTT2MULTICAST Reply (multicast IP address)
OFPT_FLOW_MOD, OFPT_GROUP_MOD (flow and group additions/modifications)
Last unsubscribe through MQTT proxy → remove multicast routing tree

Figure 7. Message exchange between the different entities in the proposed LoRaWAN network architecture.

6. Proof of Concept

As proof of concept, an SDN network has been implemented using Mininet [15] which will create a tree topology with the required number of levels with their corresponding fanouts. For initial testing, we employed a tree topology with 3 switches, i.e., a root switch (s1) and two leaf switches (s2 and s3), which connect 4 edge nodes (h1 and h2 connected to s2, and h3 and h4 connected to s3), see Figure 8.

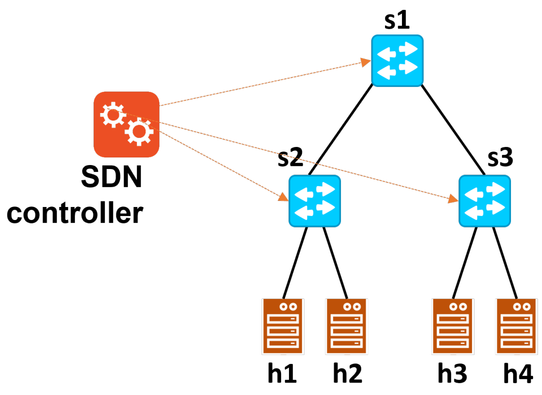


Figure 8. Simple tree topology for the SDN network for initial testing.

Nodes h1 and h4 run MQTT proxies that have been programmed using Scapy [16]. Our implementation supports MQTT QoS 0, although it can be easily extended to support QoS levels 1 and 2. One of the nodes (tipically h1) uses a real network interface of the PC it is running on, allowing it to connect directly to the LoRaWAN backbone. In this case, for simplicity, we choose to use containers, orchestrated using Kubernetes [17], which execute the different entities of the Chirpstack platform (gateway bridge, network and application servers) in the LoRaWAN gateway itself, as done in [18]. This gateway is a Pygate from Pycom [19]. FiPy [20] nodes are available with PySense expansion boards, in addition to other LoRaWAN nodes such as TTGO T-Hygrow [21] with its LoRa extension module. RYU [22] has been selected as the SDN controller, which supports OpenFlow v1.3 [23]. We implement the MQTT2MULTICAST server and the multicast routing algorithm (based on the shortest path) on top of RYU. The data sent by MQTT are stored in an InfluxDB database [24] by a Python script. These data are finally visualized using Grafana [25]. Figure 9 shows our prototype.

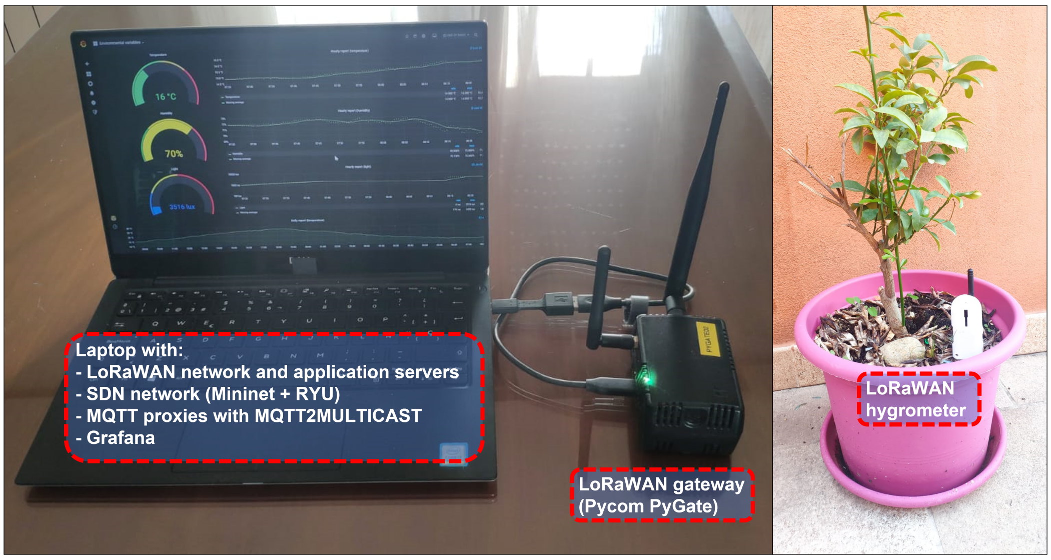
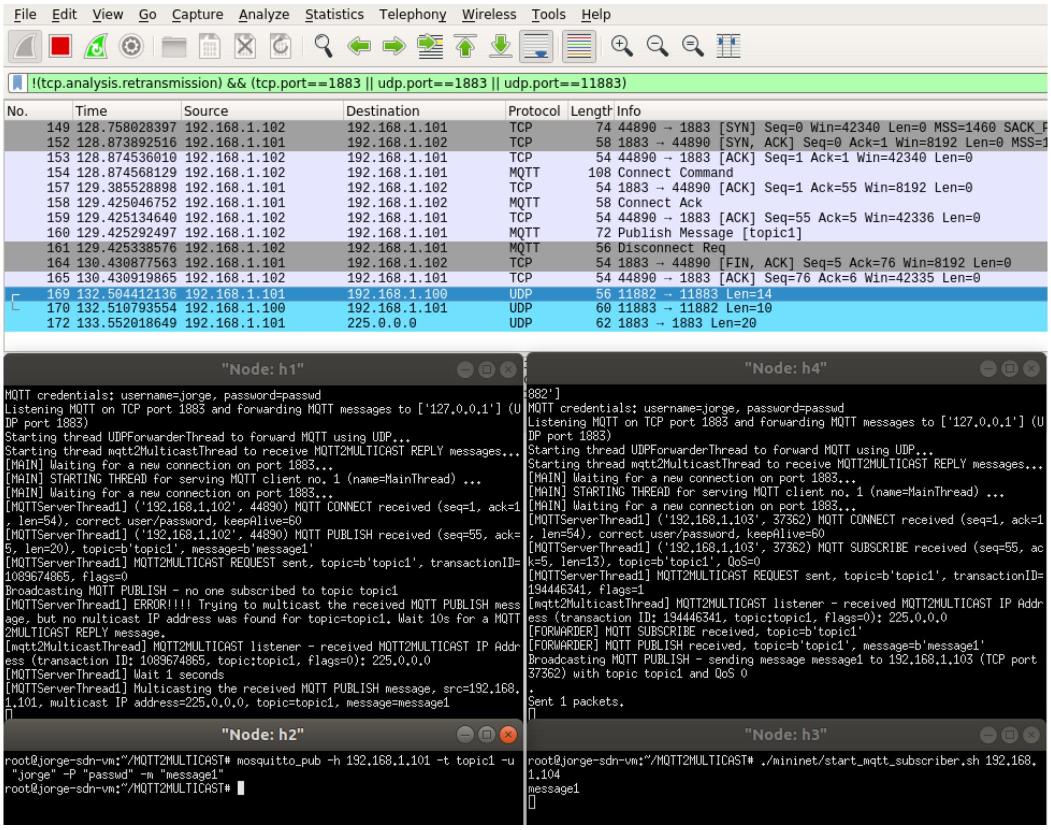

Laptop with:
- LoRaWAN network and application servers
- SDN network (Mininet + RYU)
- MQTT proxies with MQTT2MULTICAST
- Grafana
LoRaWAN gateway (Pycom PyGate)
LoRaWAN hygrometer

Figure 9. Proof of concept implementation.

As a demonstration of its operation, Figure 10 shows the traces of all the devices when h2 connects to h1 as a subscriber (using the `mosquitto_sub` command) and h4 sends a message "message1" as publisher to the topic "topic1" (using the `mosquitto_pub` command). As shown in the Wireshark trace, taken at h1, h1 responds to h2's subscription request behaving as a broker would, with the MQTT messages Connect, ConnAck, Subscribe and SubAck. Additionally, h1 asks to the MQTT2MULTICAST server (implemented in the SDN controller, with IP address 192.168.1.100) for a multicast IP address associated to the topic using an MQTT2MULTICAST Request (UDP port 11,883 in Wireshark). In response, h1 receives an MQTT2MULTICAST Reply assigning the multicast IP address 225.0.0.0. Finally, we can observe the last UDP packet at port 1883 sent to the address 225.0.0.0, which is the MQTT Publish message being forwarded using multicast, in this example, to the MQTT proxy at h4. Finally, h4 forwards the MQTT Publish message to h3, which receives it correctly, as shown in the corresponding console.


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
"Node: h1"
"Node: h4"
"Node: h2"
"Node: h3"

```
!(tcp.analysis.retransmission) && (tcp.port==1883 || udp.port==1883 || udp.port==11883)
```

```
No.  Time           Source         Destination    Protocol Length Info
 149 128.758028397 192.168.1.102  192.168.1.101  TCP       74 44890 → 1883 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_P
 152 128.873892516 192.168.1.101  192.168.1.102  TCP       58 1883 → 44890 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1
 153 128.874536010 192.168.1.102  192.168.1.101  TCP       54 44890 → 1883 [ACK] Seq=1 Ack=1 Win=42340 Len=0
 154 128.874568129 192.168.1.102  192.168.1.101  MQTT     108 Connect Command
 157 129.385528898 192.168.1.101  192.168.1.102  TCP       54 1883 → 44890 [ACK] Seq=1 Ack=55 Win=8192 Len=0
 158 129.425046752 192.168.1.101  192.168.1.102  MQTT      58 Connect Ack
 159 129.425134640 192.168.1.102  192.168.1.101  TCP       54 44890 → 1883 [ACK] Seq=55 Ack=5 Win=42336 Len=0
 160 129.425292497 192.168.1.102  192.168.1.101  MQTT      72 Publish Message [topic1]
 161 129.425338576 192.168.1.102  192.168.1.101  MQTT      56 Disconnect Req
 164 130.430877563 192.168.1.101  192.168.1.102  TCP       54 1883 → 44890 [FIN, ACK] Seq=5 Ack=76 Win=8192 Len=0
 165 130.430919865 192.168.1.102  192.168.1.101  TCP       54 44890 → 1883 [ACK] Seq=76 Ack=6 Win=42335 Len=0
 169 132.504412136 192.168.1.101  192.168.1.100  UDP       56 11882 → 11883 Len=14
 170 132.510793554 192.168.1.100  192.168.1.101  UDP       60 11883 → 11882 Len=10
 172 133.552018649 192.168.1.101  225.0.0.0      UDP       62 1883 → 1883 Len=20
```

```
MQTT credentials: username=jorge, password=passwd
Listening MQTT on TCP port 1883 and forwarding MQTT messages to ['127.0.0.1'] (U
DP port 1883)
Starting thread UDPForwarderThread to forward MQTT using UDP...
Starting thread mqtt2MulticastThread to receive MQTT2MULTICAST REPLY messages...
[MAIN] Waiting for a new connection on port 1883...
[MAIN] STARTING THREAD for serving MQTT client no. 1 (name=MainThread) ...
[MAIN] Waiting for a new connection on port 1883...
[MQTTServerThread1] ('192.168.1.102', 44890) MQTT CONNECT received (seq=1, ack=1
, len=54), correct user/password, keepAlive=60
[MQTTServerThread1] ('192.168.1.102', 44890) MQTT PUBLISH received (seq=55, ack=
5, len=20), topic=b'topic1', message=b'message1'
[MQTTServerThread1] MQTT2MULTICAST REQUEST sent, topic=b'topic1', transactionID=
1089674865, flags=0
Broadcasting MQTT PUBLISH - no one subscribed to topic topic1
[MQTTServerThread1] ERROR!!!! Trying to multicast the received MQTT PUBLISH mess
age, but no nulticast IP address was found for topic=topic1. Wait 10s for a MQTT
2MULTICAST REPLY message.
[mqtt2MulticastThread] MQTT2MULTICAST listener - received MQTT2MULTICAST IP Addr
ess (transaction ID: 1089674865, topic:topic1, flags=0): 225.0.0.0
[MQTTServerThread1] Wait 1 seconds
[MQTTServerThread1] Multicasting the received MQTT PUBLISH message, src=192.168.
1.101, multicast IP address=225.0.0.0, topic=topic1, message=message1
```

```
882']
MQTT credentials: username=jorge, password=passwd
Listening MQTT on TCP port 1883 and forwarding MQTT messages to ['127.0.0.1'] (U
DP port 1883)
Starting thread UDPForwarderThread to forward MQTT using UDP...
Starting thread mqtt2MulticastThread to receive MQTT2MULTICAST REPLY messages...
[MAIN] Waiting for a new connection on port 1883...
[MAIN] STARTING THREAD for serving MQTT client no. 1 (name=MainThread) ...
[MAIN] Waiting for a new connection on port 1883...
[MQTTServerThread1] ('192.168.1.103', 37362) MQTT CONNECT received (seq=1, ack=1
, len=54), correct user/password, keepAlive=60
[MQTTServerThread1] ('192.168.1.103', 37362) MQTT SUBSCRIBE received (seq=55, ac
k=5, len=13), topic=b'topic1', QoS=0
[MQTTServerThread1] MQTT2MULTICAST REQUEST sent, topic=b'topic1', transactionID=
194446341, flags=1
[mqtt2MulticastThread] MQTT2MULTICAST listener - received MQTT2MULTICAST IP Addr
ess (transaction ID: 194446341, topic:topic1, flags=0): 225.0.0.0
[FORWARDER] MQTT SUBSCRIBE received, topic=b'topic1'
[FORWARDER] MQTT PUBLISH received, topic=b'topic1', message=b'message1'
Broadcasting MQTT PUBLISH - sending message message1 to 192.168.1.103 (TCP port
37362) with topic topic1 and QoS 0
.
Sent 1 packets.
```

```
root@jorge-sdn-vm:~/MQTT2MULTICAST# mosquitto_pub -h 192.168.1.101 -t topic1 -u
 "jorge" -P "passwd" -m "message1"
root@jorge-sdn-vm:~/MQTT2MULTICAST#
```

```
root@jorge-sdn-vm:~/MQTT2MULTICAST# ./mininet/start_mqtt_subscriber.sh 192.168.
1.104
message1
```

Figure 10. Operation of MQTT proxies on edge switches.

Figure 11 shows part of the data displayed in Grafana in our prototype. In this example, a LoRaWAN device with several sensors measures temperature, humidity and light. The device sends the data via LoRaWAN to a gateway, which is connected to the corresponding LoRaWAN network and application servers. The application server is connected to one of our MQTT proxies, which sends the information over MQTT to other proxies but also to our Grafana platform. As it can be seen, as an example, instantaneous temperature, humidity and light values and graphs with the values of the last day are shown. In the implemented dashboards, graphs of the last hour and with the minimum, average and maximum values of these metrics are also displayed for the last day per hour, and for the last week and last month per day.

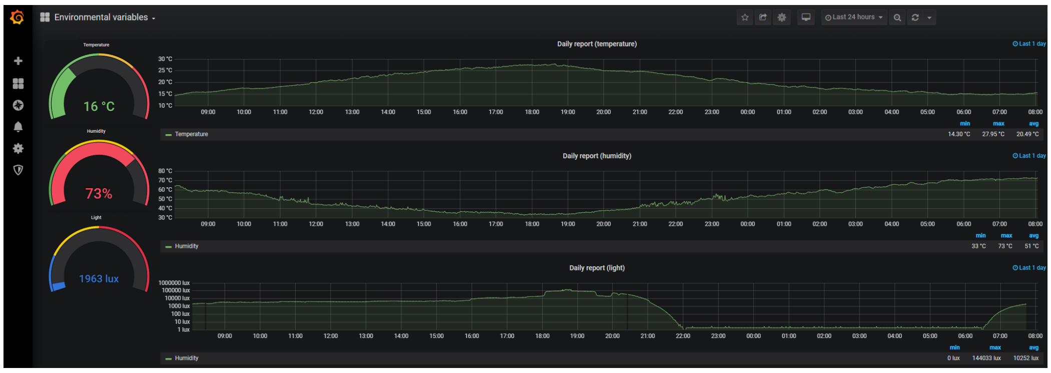


Figure 11. Example of visualization of environmental variables.

7. Results

We have conducted three experiments to evaluate the performance of our solution. In particular, we compare (1) the delay between one data publication and its reception by a subscriber using MQTT and MQTT over multicast, (2) the amount of traffic in the SDN network also for both MQTT and MQTT over multicast, and (3) the amount of signaling traffic for a typical multicast routing protocol (PIM-SSM, Protocol Independent Multicast—Source Specific Mode) and MQTT2MULTICAST.

First we tested the MQTT latency reduction due to the usage of MQTT over UDP/multicast in the SDN network instead of MQTT over TCP. Using MQTT over TCP, a publication requires 12 messages (TCP connection establishment, Connect/ConnAck, Publish, Disconnect, connection termination, and TCP ACKs). Using our solution, only the Publish message would traverse the SDN network. This reduction in messages implies a lower latency (from data publication until reception), being a more noticeable effect for high RTT values, as shown in Figure 12. In order to achieve a fair comparison, in this experiment the broker is a modification of our proxy implemented with Python/Scapy. For this experiment, we capture packet traces using `tshark` [26] at the publisher and the subscribers, so we can obtain the time between the publication and its reception by post-processing these traces. On average, the latency is reduced by approx. 90% (between 84% for RTT = 10 ms and 95% for RTT = 250 ms).

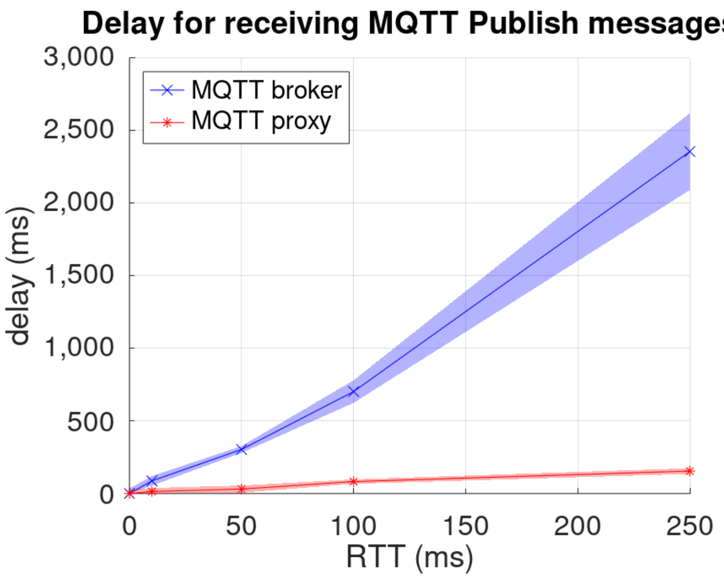


Figure 12. Delay between sending MQTT Publish messages (publisher) and their reception (subscriber).

Next, we describe the reduction of packets transmitted over the SDN network when using multicast instead of unicast for the proposed tree topology. In the case of unicast, e.g., when using normal MQTT brokers which transmit over TCP, the number of packets is depicted in Figure 13. We assume that one of the brokers connected to a leaf switch will be sending one packet (e.g., an MQTT Publish message) to the brokers connected to the remaining leaf switches, i.e., a total of $F_1 \times F_2 - 1$, where F_2 is the fanout in the last level and F_1 is the fanout in the first level ($F_1 \times F_2$ is the number of leaf switches). Its corresponding leaf switch will forward these messages to its parent switch. The parent of that leaf switch will receive the packets and will (1) send one packet to its remaining $F_2 - 1$ child switches and (2) forward $F_1 \times F_2 - F_2 = (F_1 - 1) \times F_2$ messages for the rest of the leaf switches, messages which are sent to the root switch. The root switch will forward these messages to its child switches, except to the parent of the source leaf switch. The number of messages in this case is F_2 to each switch, i.e., a total of $F_2 \times (F_1 - 1)$ messages, matching the number of packets received by the root switch. Each of the switches in the first level (except the switch corresponding to the source) will then forward one message to each leaf switch, i.e., a total of $(F_1 - 1) \times F_2$ messages. Finally, each of these $(F_1 - 1) \times F_2$ leaf switches will forward one message to their corresponding brokers, i.e., $(F_1 - 1) \times F_2$ messages. All these messages total $6F_1F_2 - 3F_2 - 1$. If we consider the TCP ACK messages in the opposite direction, the number of packets would double.

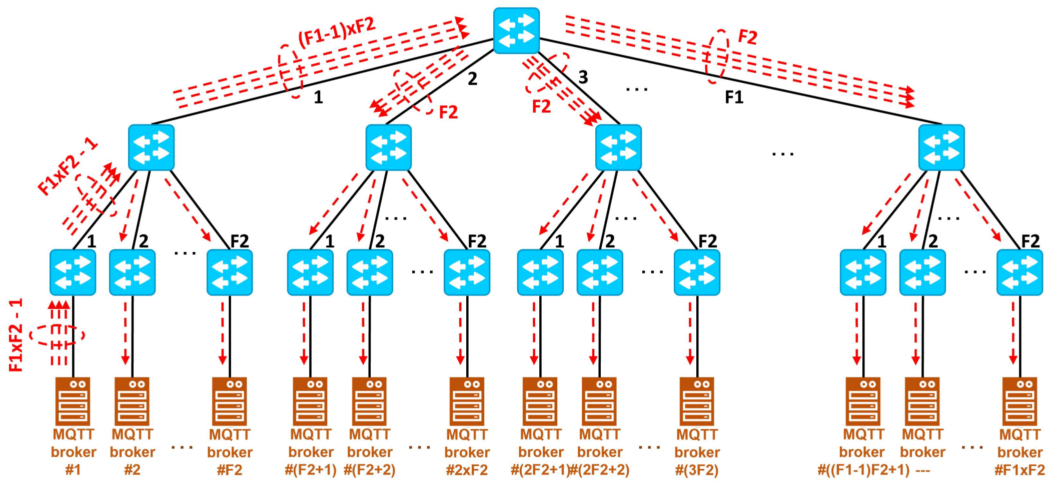


Figure 13. Packets retransmitted within the SDN network due to one incoming packet using unicast, e.g., with typical MQTT brokers.

In the case of multicast, e.g., when using our MQTT proxies which transmit using multicast over UDP, the number of messages is depicted in Figure 14. In this case, the source will send only one message, which is forwarded by its leaf switch to its parent switch. This switch will forward one message to the remaining child switches, i.e., $F_2 - 1$, which are in turn forwarded to their respective proxies, and one message to the root tree. The root tree will forward the message to the remaining $F_1 - 1$ switches in the first level, which will then transmit the message to their respective F_2 leaf switches. These leaf switches will also forward the message to their respective proxies. All these messages total $2F_1F_2 + F_1$. There are no ACKs in this scheme.

As an illustrative example, let us consider some numbers from The Things Network [4]. Currently, TTN is composed of 20.6K LoRaWAN gateways in 547 communities (coinciding with cities) across 151 countries, with a total of 44.3M messages/day. This means that, on average, 1 message is sent every 40.2 s per gateway. The top 5 largest communities own between 218 and 167 gateways. Thus, we can assume that a large city could deploy around 200 gateways.

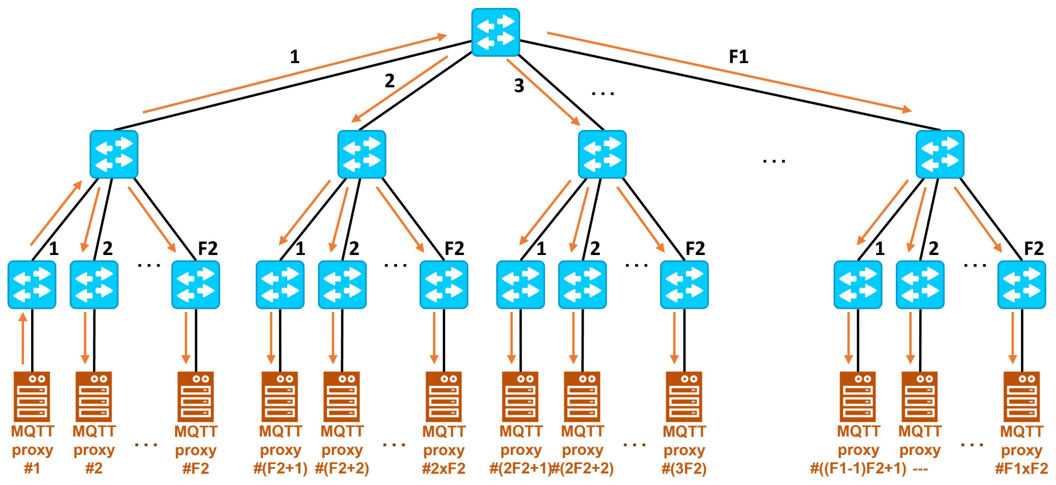


Figure 14. Packets retransmitted within the SDN network due to one incoming packet using multicast, e.g., with our MQTT proxies.

Let us further consider that the LoRaWAN servers and the MQTT proxies are deployed as containers in the edge nodes. For load balancing and redundancy purposes, we should deploy at least two LoRaWAN servers (with their corresponding MQTT proxies) per city, i.e., one every 100 gateways. In the case of a large region such as Andalusia, with 8 provinces, we could employ an SDN network with a tree topology with fanouts $F_2 = 2$ and $F_1 = 8$.

With these numbers, one MQTT message (e.g., when a node sends one LoRaWAN frame) will generate 89 messages ($6 \times 8 \times 2 - 3 \times 2 - 1 = 89$) within the SDN network (plus their corresponding TCP ACKs) if normal MQTT brokers are employed. If we employ our MQTT proxies with MQTT2MULTICAST, only 40 messages ($2 \times 8 \times 2 + 8 = 40$) are required (without TCP ACKs).

Since we are considering in this example 8 cities with 200 gateways per city, assuming 1 message every 40 s (as in TTN), Figure 15 shows the evolution of the number of packets within the SDN network. The reduction in the number of packets is 55% if we only consider data packets, and 77.5% if we also take into account TCP ACKs.

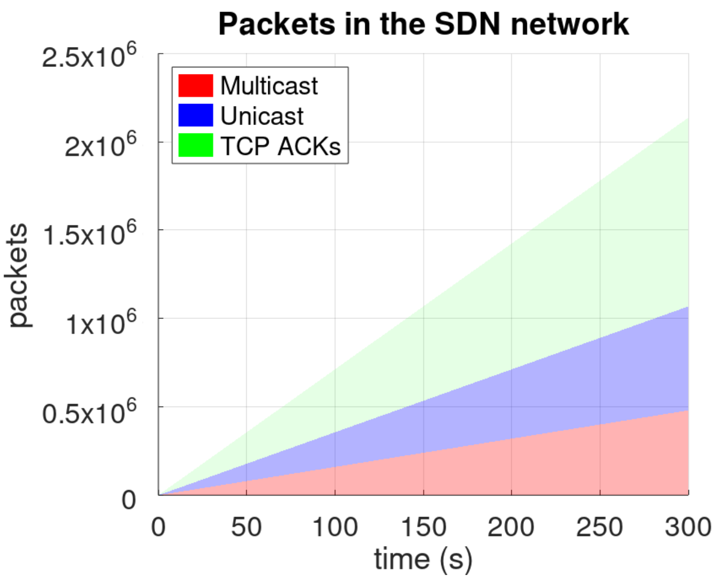


Figure 15. Example of packets retransmitted within the SDN network.

We also tested the reduction of signalling traffic comparing our MQTT2MULTICAST scheme to PIM-SSM, a typical source-based multicast routing protocol. Using PIM-SSM, the nodes and the edge routers require to also send IGMP packets to join multicast groups. This is not required using MQTT2MULTICAST, because MQTT2MULTICAST already supports the mapping between publishers, subscribers, their topics, and their corresponding multicast addresses. In this experiment, we again used `tshark` in all the routers/switches and obtained the number of packets for each protocol by using the appropriate packet filters.

Since MQTT proxies aggregate many subscribers, we assume that most likely there is at least one subscriber for a specific topic (a specific ApplicationID in our LoRaWAN network, which aggregates several MQTT topics, as discussed in Section 4). This means that, using MQTT2MULTICAST, only the initial MQTT2MULTICAST requests and replies are required. Assuming the previous scenario ($F_1 = 8, F_2 = 2$) and one topic (i.e., one ApplicationID mapped to one multicast IP address), the number of messages is very low (only 32 packets). However, the number of messages for joining/maintaining IGMP groups and the PIM-SSM messages to create the multicast routing trees are much higher. We create specific experiments using our network prototype (based on Mininet and RYU) with both solutions (PIM-SSM and MQTT2MULTICAST). Figure 16 compares both situations, being near 15,000 messages after 5 min in the second case. Thus, our solution highly decreases the required signaling for multicast routing.

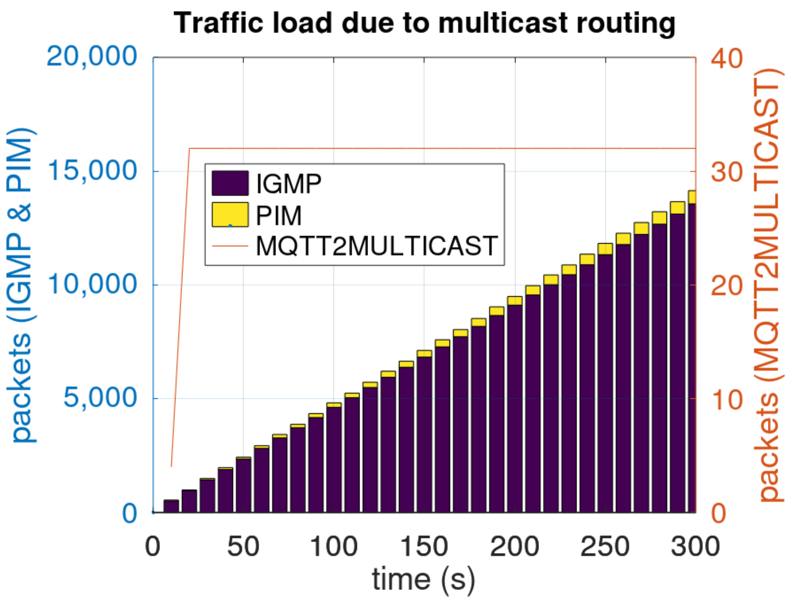


Figure 16. Comparison of traffic due to multicast routing between PIM & IGMP and MQTT2MULTICAST.

Finally, we have also tested the scalability of using multicast in our architecture. Since the number of switches in the Mininet is limited by the computational cost of their emulation, we have chosen to calculate the time to generate the multicast routes using graphs instead. For this, the NetworkX [27] library has been used, following the same approach implemented in our testbed.

The tree topology consists of a root node and two levels, similar to those in Figures 13 and 14. For simplicity, we assume that the fanouts of each level are equal. We also assume the worst case scenario, in which all leaf nodes subscribe to all other nodes. Multicast trees have been calculated using a virtual machine with 1 core and 8 GB of RAM in VirtualBox. Ten repetitions have been performed for each number of edge nodes in order to improve the reliability of the results. Figure 17 shows the time required to compute all the possible paths between edge nodes.

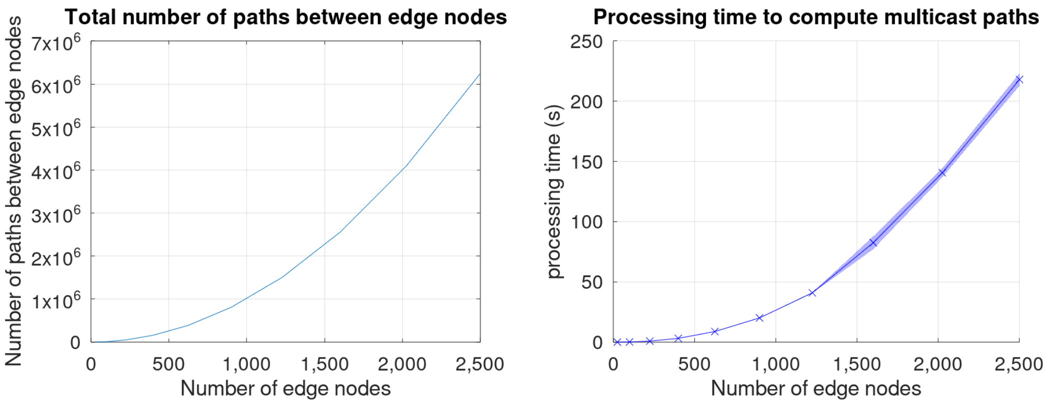


Figure 17. Total number of paths and Time to generate all possible multicast paths.

As shown, the total number of possible paths increases exponentially with the number of edge nodes. The behaviour is also similar for the time to compute those paths. This time is lower than four minutes for up to 2500 nodes, being lower than 30 s for 1000 nodes. These values are tolerable for large networks, especially considering that (1) this is the worst case scenario (all possible paths are created), (2) the executing machine may have better performance than the one used in our experiments (e.g., more CPU cores, possibly using parallel execution of the shortest path algorithm), (3) the shortest path algorithm may be optimized for our tree topology and (4) in a real network, all the paths are not required to be computed simultaneously but progressively, based on publishers and subscribers' behaviour. These calculations are performed by an application at the SDN controller. Thus, the SDN controller will be able to accommodate large networks (up to a few thousands of edge nodes) without scalability problems.

Finally, since each edge node can manage tens of LoRaWAN gateways, our architecture will be able to support tens of thousands of them.

8. Conclusions

In this article, we present the design of an IoT network architecture oriented to the collection, processing and visualization of environmental variables. This network is composed of a radio access network and a core network. The RAN is based on LoRaWAN due to its characteristics of low power consumption, high converage, easy scalability and license free operation, which makes it adequate for massive IoT communications. The core network is based on Software Defined Networking, which provides great flexibility and easy of development. This SDN network provides connectivity between the LoRaWAN networks and other entities required for data distribution, processing and visualization. Most of these functionalities are expected to be deployed in remote clouds or edge resources.

One of the most relevant protocols for data distribution in IoT applications is MQTT, which is based on the publish/subscribe paradigm. Leveraging SDN for enhancing MQTT, we propose MQTT2MULTICAST. Our solution employs MQTT brokers at the edge nodes, which act as MQTT brokers to local clients and multicast MQTT publications (using UDP instead of TCP) to other proxies that have subscribers for the specific topic. For that purpose, MQTT proxies utilize the MQTT2MULTICAST protocol to ask a server for a mapping between MQTT topics and their corresponding multicast IP addresses. The MQTT2MULTICAST server is implemented as an SDN controller application, which creates the required multicast routing trees within the SDN. These multicast routing trees are source-based, so the shortest paths between the publishers and the proxies are created by enforcing flow and group entries in the required SDN switches. Once the routing trees are created, the MQTT proxies can start multicasting the Publish messages.

We implemented this network architecture using Mininet for the SDN network, RYU as the SDN controller, and implementing (1) the MQTT proxies in Python with the Scapy library, (2) the MQTT2MULTICAST server as a RYU application, and (3) source-based multicast routing also as part of the RYU application. We tested its proper behaviour using real applications (`mosquitto_sub` and `mosquitto_pub`) and also by implementing a proof of concept with real equipment, i.e., a LoRaWAN network with nodes, gateways, LoRaWAN servers (using the Chirpstack platform) and data visualization (using InfluxDB and Grafana). We released our testbed environment as an open-source project [3], allowing for reproducible research.

Finally, we conducted experiments to assess the performance of our prototype. We improved the publication delay approx. by 90% by eliminating non-required signaling and their corresponding round-trip times. In the scenario under evaluation (a tree topology with 16 edge nodes), we reduced the number of packets within the SDN network by 55% (or 77.5% if we consider TCP ACKs) and we almost eliminated the signaling related to multicast routing (compared to other multicast routing protocols such as PIM-SSM). Moreover, we conducted experiments to analyze the scalability of our proposal, showing that the architecture is able to accommodate tens of thousands of LoRaWAN gateways.

Author Contributions: Conceptualization, J.N.-O., N.C.-R., F.D.-F. and J.J.R.-M.; methodology, J.N.-O. and J.J.R.-M.; software, J.N.-O. and N.C.-R.; validation, J.N.-O., N.C.-R. and J.J.R.-M.; formal analysis, J.N.-O. and N.C.-R.; investigation, J.N.-O., N.C.-R., F.D.-F. and J.J.R.-M.; resources, J.N.-O. and N.C.-R.; writing—review and editing, J.N.-O., N.C.-R., F.D.-F. and J.J.R.-M. All authors have read and agreed to the published version of the manuscript.

Funding: This research was partially funded by the Andalusian Knowledge Agency (projects A-TIC-241-UGR18 and B-TIC-568-UGR20), the Spanish Ministry of Science and Innovation (project PID2019-108713RB-C53), the Spanish Ministry of Economic Affairs and Digital Transformation (project TSI-063000-2021-28) and the Spanish Ministry of Universities (FPU Grant Number: 20/02621).

Data Availability Statement: The source code of the MQTT proxy (Python application using the Scapy library) and the MQTT2MULTICAST protocol (as an application for the RYU controller), including the multicast routing tree generation in the SDN network (also implemented in the RYU app) is available at https://github.com/jorgenavarroortiz/MQTT2MULTICAST (accessed on 29 January 2022). Besides, this repository contains the Mininet topology and a detailed description of the steps to reproduce the experiments within this paper.

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

Abbreviations

The following abbreviations are used in this manuscript:

ADR	Adaptive Data Rate
AI	Artificial Intelligence
CNF	Cloud-Native Network Function
DM-MQTT	Direct Multicast-MQTT
DR	Data Rate
ETSI	European Telecommunications Standards Institute
IoT	Internet of Things
LoRa	Long Range modulation
LoRaWAN	Long Range Wide Area Network
LPWAN	Low Power Wide Area Network
MQTT	Message Queuing Telemetry Transport
MQTT-SN	MQTT for Sensor Networks
MQTT-ST	MQTT-Spanning Tree
P2P	Peer-to-Peer
SDN	Software Defined Network
SF	Spreading Factor
TCP	Transport Control Protocol
UDP	User Datagram Protocol

References

1. LoRa Alliance. LoRaWAN Specification v1.1. Available online: https://lora-alliance.org/resource_hub/lorawan-specification-v1-1/ (accessed on 29 January 2022).
2. IBM and Eurotech. MQTT v3.1 Protocol Specification. Available online: https://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html (accessed on 1 June 2021).
3. Navarro-Ortiz, J. MQTT to Multicast GitHub Repository. Available online: https://github.com/jorgenavarroortiz/MQTT2MULTICAST (accessed on 29 January 2022).
4. The Things Network—Network Architecture. Available online: https://www.thethingsnetwork.org/docs/network/architecture/ (accessed on 29 January 2022).
5. Chirpstack, Open-Source LoRaWAN Network Server Stack. Available online: https://www.chirpstack.io/ (accessed on 29 January 2022).
6. LORIOT—Connecting the Internet of Things. Available online: https://www.loriot.io/ (accessed on 29 January 2022).
7. Stanford-Clark, A.; Truong, H.L. *MQTT For Sensor Networks (MQTT-SN), Protocol Specification, Version 1.2*; Protocol specification; IBM: Armonk, NY, USA, 2013.
8. Park, J.H.; Kim, H.S.; Kim, W.T. DM-MQTT: An Efficient MQTT Based on SDN Multicast for Massive IoT Communications. *Sensors* **2018**, *18*, 3071. [CrossRef] [PubMed]
9. Sawadski, V. MQTT Multicast. Available online: https://github.com/vsaw/gnunet-mqtt/wiki/MQTT-multicast (accessed on 29 January 2022).
10. GNUnet, Secure Peer-to-Peer Networking Framework. Available online: https://www.gnunet.org/en/index.html (accessed on 29 January 2022).
11. Alliance, L. LoRaWAN, What is it? A Technical Overview of LoRa and LoRaWAN. Available online: https://lora-alliance.org/resource_hub/what-is-lorawan/ (accessed on 29 January 2022).
12. *SX1272/3/6/7/8: LoRa Modem, Designer's Guide, AN1200.13, Revision 1.0*; Technical Report; Semtech: Camarillo, CA, USA, 2013.
13. Wang, M.H.; Chen, L.W.; Chi, P.W.; Lei, C.L. SDUDP: A Reliable UDP-Based Transmission Protocol Over SDN. *IEEE Access* **2017**, *5*, 5904–5916. [CrossRef]
14. Park, C.S.; Nam, H.M. Security Architecture and Protocols for Secure MQTT-SN. *IEEE Access* **2020**, *8*, 226422–226436. [CrossRef]
15. Mininet—An Instant Virtual Network on Your Laptop. Available online: http://mininet.org (accessed on 29 January 2022).
16. Scapy—Packet Crafting for Python2 and Python3. Available online: https://scapy.net (accessed on 29 January 2022).
17. Kubernetes—Production-Grade Container Orchestration. Available online: https://kubernetes.io (accessed on 29 January 2022).
18. Navarro-Ortiz, J.; Ramos-Munoz, J.J.; Lopez-Soler, J.M.; Cervello-Pastor, C.; Catalan, M. A LoRaWAN Testbed Design for Supporting Critical Situations: Prototype and Evaluation. *Wirel. Commun. Mob. Comput.* **2019**, *2019*, e1684906. [CrossRef]
19. Pycom. Pygate LoRaWAN Gateway. Available online: https://pycom.io/product/pygate/ (accessed on 29 January 2022).
20. Pycom. FiPy Development Board. Available online: https://pycom.io/product/fipy/ (accessed on 29 January 2022).
21. LilyGo. TTGO T-Higrow ESP32. Available online: http://www.lilygo.cn/prod_view.aspx?TypeId=50033&Id=1172 (accessed on 29 January 2022).
22. RYU—Component-Based Software Defined Networking Framework. Available online: https://ryu-sdn.org/ (accessed on 29 January 2022).
23. *OpenFlow Switch Specification, Version 1.3.0*; Open Networking Foundation: Menlo Park, CA, USA, 2014.
24. InfluxDB: Open Source Time Series Database | InfluxData. Available online: https://www.influxdata.com/ (accessed on 29 January 2022).
25. GrafanaLabs. Grafana—Your Observability Wherever You Need It. Available online: https://grafana.com/ (accessed on 29 January 2022).
26. tshark Manual Page. Available online: https://www.wireshark.org/docs/man-pages/tshark.html (accessed on 29 January 2022).
27. Hagberg, A.; Schult, D.; Swart, P. NetworkX, Network Analysis in Python. Available online: https://networkx.org/ (accessed on 2 March 2022).