Digital Communications and Networks 8 (2022) 257–266

Contents lists available at ScienceDirect

Digital Communications and Networks

journal homepage: www.keaipublishing.com/dcan

A LoRa-based protocol for connecting IoT edge computing nodes to provide small-data-based services

Kiyoshy Nakamura [a], Pietro Manzoni [a,*], Alessandro Redondi [c], Edoardo Longo [c], Marco Zennaro [b], Juan-Carlos Cano [a], Carlos T. Calafate [a]

[a] Universitat Politécnica de Valéncia, Valencia, Spain
[b] Abdus Salam International Centre for Theoretical Physics (ICTP), Trieste, Italy
[c] Politecnico di Milano, Milan, Italy

ARTICLE INFO

Keywords:
Small data
Edge computing
LoRa
IoT

ABSTRACT

Data is becoming increasingly personal. Individuals regularly interact with a variety of structured data, ranging from SQLite databases on the phone to personal sensors and open government data. The "digital traces left by individuals through these interactions" are sometimes referred to as "small data". Examples of "small data" include driving records, biometric measurements, search histories, weather forecasts and usage alerts. In this paper, we present a flexible protocol called LoRaCTP, which is based on LoRa technology that allows data "chunks" to be transferred over large distances with very low energy expenditure. LoRaCTP provides all the mechanisms necessary to make LoRa transfer reliable by introducing a lightweight connection setup and allowing the ideal sending of an as-long-as necessary data message. We designed this protocol as communication support for small-data edge-based IoT solutions, given its stability, low power usage, and the possibility to cover long distances. We evaluated our protocol using various data content sizes and communication distances to demonstrate its performance and reliability.

1. Introduction

Big data is defined by the three big Vs of its data: volume, variety and processing velocity. By contrast, small data consists of analyzing a dataset comprising small volumes and formats to make them accessible, processable, and understandable. Small data is defined in different terms; for example, D. Estrin [1] described it as "digital traces around individuals" and argued that it "is going to change the way we think about and practice health." Another characterization of small data emphasizes its focus on the individual as the location data is collected, analyzed, and utilized [2]. This enables individuals to improve their own abilities and grants them the freedom to choose what they pursue.

Smart devices and other low-cost computing platforms provide data personalization [3], and bring intelligence closer to the edge [4]. TinyML [5] makes it easier to solve complex problems. The combination of these two trends creates new opportunities to use AI to solve problems in a wider set of scenarios with systems that can use a small amount of data to learn about and resolve tasks quickly.

In this paper, we present a flexible protocol based on LoRa technology [6] that allows data "chunks" (i.e., self-contained pieces of data with highly variable lengths, such as JavaScript Open Notation (JSON)-encoded messages) to be transferred over long distances with low energy expenditure. The proposed protocol, named Long-Range Content Transfer Protocol (LoRaCTP), provides all the mechanisms necessary to make LoRa data transfers reliable by introducing a lightweight connection that permits data messages to be of any ideal length. To test such message lengths, our protocol evaluation used a library with data message lengths of up to 150 KB to obtain stable and reliable behavior.

The LoRa spread spectrum modulation technique derived from the Chirp Spread Spectrum (CSS) technology to provide long-range, low-power wireless platforms that have been widely adopted for the Internet of Things (IoT) networks worldwide. LoRa devices and the open LoRaWAN protocol enable smart IoT applications in varying contexts, such as energy management, natural resource reduction, pollution control, infrastructure efficiency, and disaster prevention.

In order to provide small data oriented computing services in poor-connection and resource-limited scenarios, LoRaCTP has been used to design a generic FrUgal eDGE (FUDGE)/fog architecture [7]. Given this

* Corresponding author.
E-mail address: pmanzoni@disca.upv.es (P. Manzoni).

https://doi.org/10.1016/j.dcan.2021.08.007
Received 21 January 2021; Received in revised form 13 August 2021; Accepted 21 August 2021
Available online 26 August 2021

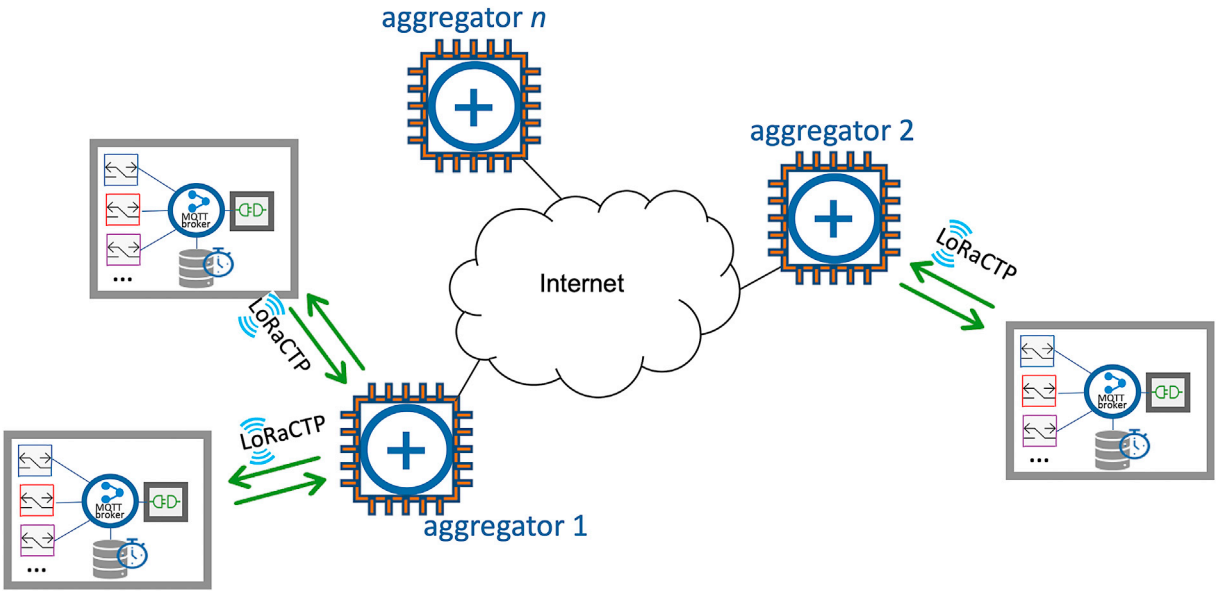

Fig. 1. FUDGE system's overall architecture.

architecture, we introduce the concept of "aggregator", i.e., a process that coordinates the data flow between the edge nodes and cloud services. These FUDGE nodes interchange data using an Application Programming Interface (API) based on the Message Queuing Telemetry Transport (MQTT) pub/sub system. Data enters into a FUDGE node locally through a so-called content proxy; using a dedicated channel with an aggregator based on LoRaCTP, and data transfers adapt to channel quality.

The paper is organized as follows. Section 2 describes related work about system designs equivalent to ours, whereas Section 3 describes the overall structure of our proposal with some details on FUDGE nodes. Section 4 presents our devised solution for connecting varying content aggregators autonomously. Section 5 details our proposal, which is the core of this paper, and Section 6 provides the performance evaluation of our LoRaCTP protocol. The paper ends with Section 7, where some final considerations are described.

2. Related works

The research area of message dissemination in distributed generic pub/sub system has been notably active over the last 20 years. Most works have focused on developing efficient and scalable routing algorithms to create topic-based dissemination trees (in the form of multicast groups) that cover only subscribers matching a particular topic [8–12]. Such works do not consider a specific broker implementation; instead, an overall broker topology is assumed. It is only recently that the popularity of protocols has drawn attention to the problem of interconnecting MQTT-specific brokers [13]. Some works have focused primarily on vertical clustering, where single brokers are replaced by multiple virtualised broker instances running behind a single endpoint, typically a load balancer [14,15]. These approaches introduced the concept of multiple brokers cooperating with each other, although broker clusters were viewed as single centralised entities from the perspective of clients.

Banno et al. in Ref. [16] introduced pure MQTT by way of an Internetworking Layer of Distributed MQTT (ILDM) for brokers where specific nodes were placed between clients and heterogeneous brokers to connect both groups and the brokers themselves. Similar to our work, message distribution was obtained with publication flooding, but the underlying network of ILDM nodes was assumed to be loop-free, and there were no automatic mechanisms present for broker failure recovery. The subsequent study brought proposals of interconnected MQTT brokers that could dynamically change their topology configuration at run time using specific MQTT messages transmitted by a trusted centralized entity [17,18].

A parallel study suggested creating a broker network and used an external monitoring agent to check individual broker status [19]. Clients were connected to brokers through local gateways; upon any broker configuration changes (e.g., broker failure and increase in latency), the gateway would reconnect clients to new brokers using the information retrieved by the monitoring agent. Such an approach enabled client mobility, dynamic broker provisioning, and broker load balancing. Finally, an example of tree-based MQTT broker topology was provided by Ref. [20] with a proposal of Software Defined Networking (SDN) to minimize data transfer delay by creating per-topic multicast groups. The proposed SDN controller gathered client and relative pub/sub topic information from all the edge brokers through a master broker, which acts as the root of the multicast tree. However, the paper assumed a static topology with no indication of how to elect a master broker.

Satyanarayanan et al. [21] introduced the concept of "edge-native applications" that fully exploited edge computing potential while maintaining a strong relationship with it. Such custom-designed applications would take advantage of one or more unique attributes of edge computing such as (a) bandwidth scalability, (b) low-latency offload, (c) privacy-preserving, and (d) WAN-failure resiliency. In Ref. [22], the authors investigated redundant IoT edge node relocations to provide a timely "Confident Information Coverage" service that would extend network lifetimes using an offloading-assisted energy-balanced manner. In Ref. [23], the authors propose a "Verifiable and Flexible Data Sharing" mechanism for an information-centric IoT that exploited ciphertext-policy attribute-based encryption for authorization and identity-based signatures for distributed identity verification. In Ref. [24], the authors defined two important concepts among others: the paradigm of edge/cloud computing transparency and IoT computing topology management. The former permits computation nodes to change dynamically without administrator intervention, while the latter provides a global IoT system view ranging from hardware and communication infrastructures to the software they deploy. In Ref. [25], the authors describe the design of an IoT-based platform that aimed to manage energy consumption in real-time for water resource recovery facilities, integrating them into a future demand-side management environment. LoRa and MQTT were the basic technologies employed in an interesting proposal for a low-cost drone-based remote monitoring system for dangerous areas [26]. Also, in Ref. [27] an open-source earthquake and weather monitoring system is presented based on a Long Range (LoRa)-based star topology with a fully energy-autonomous sensor node.

With all this, we consider that IoT and edge computing are growing together, and for this reason, our work aims to provide a reliable content

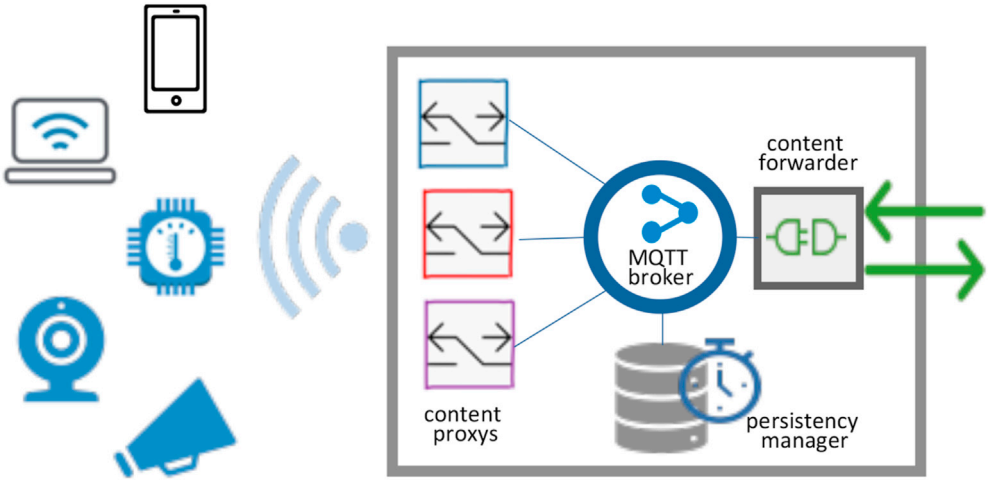

Fig. 2. FUDGE node's structure.

transfer protocol based on a widely used technology to simplify the connection from remote areas to edge based solution, thus simplifying the introduction of this approach in IoT scenarios.

3. FUDGE architecture

Fig. 1 shows the overall architecture of a FUDGE system. Its basic structure is based on various data sharing nodes called "aggregators" and several edge stations (FUDGE nodes). The stations have a dedicated channel with a specific LoRa-based aggregator to enable long-range transmissions with low power consumption. In addition, the LoRaCTP[1] is in place to transfer data chunks (content in this context) reliably and bidirectionally by adapting dynamically to channel quality; Protocol details are described further in Section 5.

Transferred content is encrypted using the associated aggregator's certificate, which must be preinstalled manually in the edge nodes. This approach guarantees both communication privacy and that only the authorized aggregator can handle the incoming data. We understand that this approach uses minimal security, but considering the context is enough to define system reliability.

The aggregator uses a polling approach to coordinate data flow with the edge nodes. Polling was chosen out of necessity to address LoRa difficulties when detecting and handling collisions and given the low bandwidth offered by LoRa, to guarantee the highest possible speed of reliable transmission. Moreover, polling does not require more complex hardware/software infrastructure to manage e-link establishment. Overall, this approach allows for greater channel usage flexibility, sleep mode in edge nodes, and easy processing of disconnected cycles.

Once an edge node is selected during polling, a push/pull sequence occurs where the node pushes data to the aggregator and then pulls the previously polled data stored in the aggregator.

The content forwarder is in charge of this task, and during the push stage, it sends all the content stored by the consistency manager tagged as Global. During the pull stage, in addition to previously stored polling content, aggregators return content from other aggregators on topics registered from any service in the edge node. In the pull stage, in addition to previously stored polling content, aggregators return content from other aggregators on topics registered from any service in the edge node. This allows local processes to receive replies to local requests and data from other cloud services.

A FUDGE node

The structure and organization of a "FUDGE node" is depicted in Fig. 2. FUDGE architecture is based on microservices and allows easier scaling and faster development of applications. In a FUDGE node, several microservices can coexist and interchange data with each other using an API based on the MQTT pub/sub system.

Three basic elements constitute a FUDGE node: an MQTT broker, a persistency manager, and a content forwarder. Beyond these elements, a FUDGE node handles as many content sources as it does content proxies. The MQTT broker is the core of a FUDGE node, handling content flow inside the node. The persistency manager element takes care of storing content labeled as persistent in a time-series database. This allows the system to maintain its temporal evolution while retaining all the data required to handle asynchronous operations or possible disconnections. Moreover, data analytics tools can be used to visualize and extract metrics from collected data or to implement custom monitoring dashboards. The last main FUDGE element, the content forwarder, communicates with aggregators with LoRaCTP for all required content interchanges.

Since MQTT is at the core of the system, we define a standard format for topics. The structure used in the system is as follows:

<device id><service id>/scope/persistency/ …others …

where:

- <device id> identifies the specific FUDGE node device;
- <service id> identifies the service providing the content, which can be anything from a simple temperature sensor or camera to a messaging system;
- scope indicates whether content is designated for use by other local services within the FUDGE node (L) or forwarded to the aggregator (G);
- persistency indicates whether content requires a persistency manager, with P, N, and X indicating necessary action taken, no action required, and a search performed in the persistency repository for data retrieval, respectively. The last uses other fields to indicate whether the search is a *request* or a *response* for a search, or other details; and
- others indicate any additional tags required by a specific service.

The content itself has to be structured using the JSON data-interchange format according to the format in Listing 1.

[1] The LoRaCTP code can be found here: https://github.com/pmanzoni/loractp.

```
payload = {
    "measurement": <measurement_id>,
    "tags": {
        "tag1": <tag1_value>,
        ...,
        "tagn": <tagn_value>
    },
    "fields": {
        "field1": <field1_value>,
        ...,
        "fieldn": <fieldn_value>
    }
}
```

Listing 1. JSON content structure.

The <measurement_id> shows a specific set of values that could be generated from multiple sources on the same set, e.g., several weather stations providing weather values on an area close to the edge node. The difference between <tag_value> and <field_value> is given by the underlying time-series database. Basically, a measurement that changes over time should be a field, and the metadata about the measurement should be a tag. To continue the previous example, pressure and temperature values are fields, and weather station names are tags.

4. Aggregator overview

Aggregators are used to distribute data among the various FUDGE nodes. These aggregators are based on MQTT standard brokers augmented with the capacity to integrate content coming from various end-points. Some existing MQTT broker implementations (e.g., Mosquitto, CloudMQTT, HiveMQ) allow the use of bridging, i.e., a direct connection between brokers. Such solutions are prone to message loops among brokers. Indeed, the existence of a cycle where a message is continuously republished by the participating brokers can quickly deplete a broker's resources, ultimately making it unable to deliver meaningful traffic. As implementing duplicate detection (i.e., tracking all producers of original messages received and forwarded by any broker) in a distributed scenario would create enormous complexity, existing solutions require the manual configuring of inter-broker connections using a loop-free (tree) topology. However, configuring MQTT bridges manually has two significant drawbacks: 1) similar to wiring switches for small- or medium-sized enterprises, the potential for confusion and accidental duplicate connection creation is significant, especially for larger topologies; and 2) enforcing a static loop-free inter-broker topology completely sacrifices adaptivity and robustness.

Our proposal relies on an automatic algorithm, the classical Spanning Tree Protocol (STP) used in switched networks, to set up the brokers' tree in MQTT, thus allowing for the creation of a loop-free, dynamic inter-broker network. STP (standardised as IEEE 802.1D) is a distributed protocol that creates a logical spanning tree over a meshed network of layer 2 switches. Such a spanning tree is obtained by electing a root switch and blocking some output ports of the other switches: blocked ports do not forward data frames, thus preventing broadcast storms. To determine the root node and ports to be blocked, switches exchange control packets known as Bridge Protocol Data Units (BPDUs). General scheme enhancements have been proposed and standardized; these include the Rapid STP (RSTP, IEEE802.1D-2004), which provides significantly faster spanning tree convergence, and the Multiple Spanning Tree Protocol (MSTP, IEEE802.1Q-2005), which allows the creation of multiple spanning trees for different VLANs or groups of VLANs, and the recent shortest path bridging protocol (IEEE 802.1aq), which allows redundant links between switches to be active simultaneously, thus increasing bandwidth.

Given these available protocols, we developed MQTT-ST [28], starting from the latest MQTT protocol specifications [29]. In MQTT-ST, when a broker is willing to create a bridge with another broker or group of brokers, it transmits an MQTT CONNECT message upon startup. The broker (brokers) targeted for connection have their addresses and ports specified in a configuration file. To inform a broker that a connection request comes from another broker and not from a client, the most significant bit of the protocol version byte is set in the CONNECT header.

Upon receiving a CONNECT message with an appropriate bit set, a targeted broker will immediately reply with a modified CONNECT message to the originating node, using the standard MQTT port 1883, to allow bidirectional communication. This also allows a node with no configuration file set to be part of the broker network if contacted by a networked broker. The targeted broker then stores the IP addresses of all directly connected brokers in a local table, which is used to keep track of the state of each connection marked as root, designated, or blocked. For each connection, the table also stores the average Round Trip Time (RTT) and a value C, which summarizes the resource capability of the endpoint broker as detailed later. Finally, the broker sets itself as root and starts transmitting signaling messages to all connected brokers. For this task, instead of creating a new specific message, we reuse MQTT PINGREQ messages.

Standard MQTT specifies a keep-alive parameter, which defines the maximum time interval permitted to elapse after the last client transmission. In case the timer expires, the broker closes the connection with the client. Therefore, to maintain a live broker connection, clients periodically transmit PINGREQ messages. MQTT-ST reuses such messages to play the role of STP BPDUs. The values of the current root broker's IP address for the transmitter, the broker capability value C, and a root path cost P are appended to PINGREQ messages. The latter two fields (C and P) are used for root selection and path computation.

Root brokers play a crucial role in the broker trees, acting as relay nodes for all traffic and enduring increased computational loads as a result. Indeed, selecting a broker with poor or overloaded resources may result in poor overall performance. While STP selects a root based on identifier alone, which does not suit the scenario under consideration well. In MQTT-ST, instead, the root broker is selected according to the capability value C, defined as

$$C = \alpha L + \beta M \tag{1}$$

where L is the broker CPU speed, R is the amount of RAM, and α, β are tuneable conversion parameters.[2] In the case of a tie, the broker with the lowest IP address is selected as the root.

In STP, each node selects the best path to the root according to bandwidth-related criteria, preventing the use of reduced capacity links in the tree that may slow down an entire network. For MQTT-ST, we observe that latency, rather than bandwidth, plays a critical role. Each broker, therefore, continuously monitors the RTT to other brokers and uses that value for updating the root path cost P. In order to do this, we leverage the request/response mechanism already present in MQTT through the PINGREQ/PINGRESP. A timer is started when a client transmits a PINGREQ message that stops when the corresponding PINGRESP message is received by the broker, providing an estimate of the current RTT. Upon receiving the PINGREQ message, the connection that provides the lowest latency path towards the root is marked as the root connection. In the case of ties in cumulative latency, the connection passing through the broker with the highest amount of resources is selected as the root connection. All connections are then labeled following the same logic as the STP: (i) all the root broker connections are marked as designated, (ii) non-root connections of other brokers are marked as designated if the broker has a better path cost (or a better value of C in the case of a tie) compared to a neighboring broker, and (iii) all other ports are labeled as blocked.

While executing, an MQTT-SN broker works exactly like an MQTT broker from a connected client's perspective. Moreover, the broker

[2] Values L and R are read by the broker at startup using the respective/proc/cpuinfo and/proc/meminfo system files available on Linux.

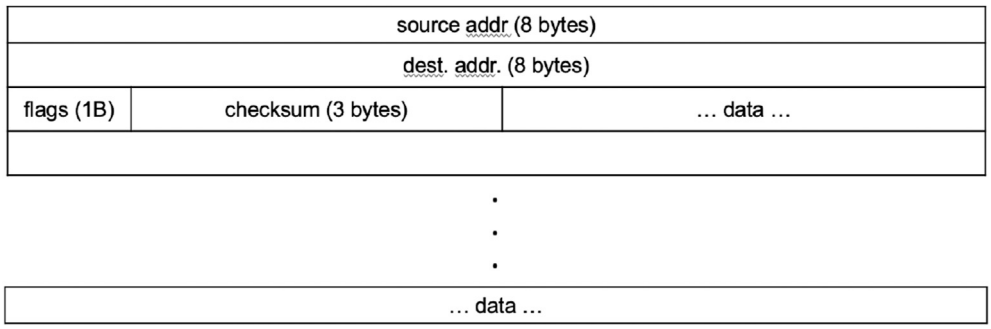

Fig. 3. Packet structure used by the stop-and-wait ARQ.

forwards all published messages (regardless of topic) along with non-blocked connections (root or designated) while discarding messages coming from blocked connections. The forwarding is performed like a standard MQTT PUBLISH message. It should be noted that in order to allow full replication of messages published at one broker to all other brokers, message forwarding is performed with the highest MQTT Quality of Service (QoS) available (QoS = 2). On the one hand, this guarantees that each message will be received only once by its intended recipients. On the other hand, it requires a four-part handshake with a non-negligible associated overhead. Since forwarding is implemented through a standard MQTT PUBLISH, all other features of the latest MQTT specification are conserved (e.g., retain, topic alias, message expiry interval).

Upon broker failure, MQTT-SN handles the corresponding socket error to re-establish the forwarding tree. More specifically, a broker detecting the socket error transmits a special PINGREQ message used to restart the tree construction from scratch. The broker sets itself as root and appends an additional Topology Change (TC) field set to the message, similar to STP. Any broker receiving such a message will restart its own root selection procedure, eventually leading to the converging of a new tree.

5. LoRaCTP details

This section presents details on LoRaCTP. Our protocol works directly at LoRa to create a supporting long-range communication link. LoRa is based on CSS modulation that maintains low power characteristics as FSK modulation. The advantage of LoRa is in the technology's long-range capability. As range depends highly on the location environment and obstructions, LoRa technology (e.g., LoRaWAN) enjoys a distinct capability advantage with a link budget greater than any other standardized communication technology.

LoRaCTP is used to transfer data chunks (content) or self-contained pieces of data, e.g., a JSON-encoded message with a potentially unlimited length. We tested out a library with messages of up to 150 KB (Section 6). LoRaCTP is based on a unicast protocol that adopts a classical stop-and-wait Automatic Repeat Request (ARQ) approach with a dynamic and adaptive value for the retransmission delay. The protocol ensures that information is not lost due to dropped packets and that packets are received in the correct order.

Once the content is designated for sending, it is split into as many blocks as necessary, given the maximum application payload size, which value depends on the selected data rate. For example, assuming the European 863–870 MHz band with the worst propagation conditions, the lowest data rate should also be assumed, preventing any node from sending more than roughly 51 bytes per packet given by Spreading Factor (SF) 12); under SF7, packets can contain provided by SF12, where the node cannot send more than about 51 bytes per packet; with SF7, the packet can be up to 222 bytes. The packet structure is shown in Fig. 3.

Each packet has a fixed 20-byte header with source address and destination address shortened to the rightmost 8 bytes of the LoRa

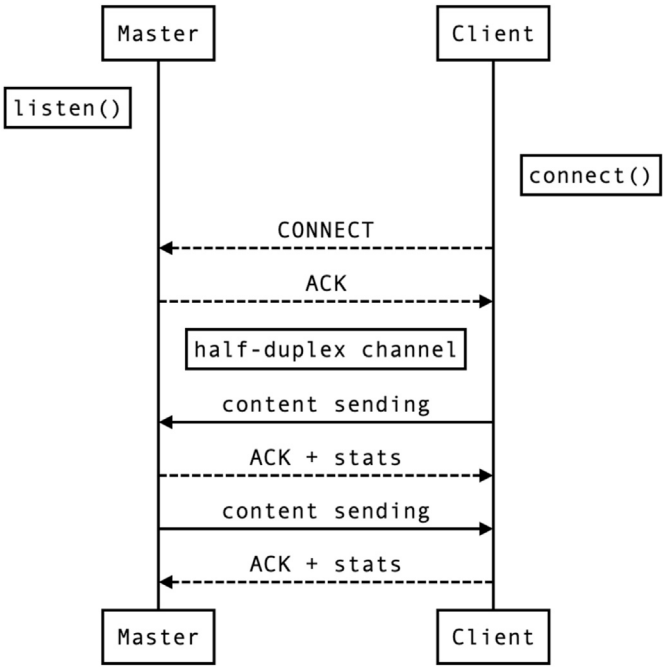

Fig. 4. Flow of the establishment and interchange of data.

adapter address. The flag field contains four values: the packet sequence numbers (encoded as an alternating 0/1 sequence), the ACK sequence numbers (encoded as an alternating 0/1 sequence, too), a flag that indicates if this is the last packet of the content sent, and a flag that indicates whether. Three bytes more are used to store the checksum of the packet using the rightmost 3 bytes of the SHA-256 hash digest, which is computed over the data to be sent.

Each packet is sent using a three attempts scheme. That is, if an ACK is not received after three attempts, the channel is deemed busy or too noisy for the content to be sent, and the sending is dropped. Standard situations set the retransmission delay (the time a system waits for an ACK to be received) to 5 s, which is deemed reasonable for preventing any unnecessary packet retransmissions. When more than one packet is necessary to send content, retransmission delay is recomputed by continuously evaluating the RTT and adapting the delay as follows:

```
srtt = rtime - stime
if ertt == -1:
    ertt = srtt
else:
    ertt = ertt * 0.875 + srtt * 0.125
urtt = 0.75 * urtt + 0.25 * |srtt - ertt|,
```

where *rtime* is the time the ACK was received, *stime* is the time the packet was sent, *ertt* is the estimated RTT delay, and *urtt* is the used RTT delay.

```
import ...

ctpc = loractp.CTPendpoint()

mya, rcv, qua, res = ctpc.connect()
if (res == 0):
  print("conn to ``, rcv, ``myadr: ``, mya,
        ``quality: ``, qua)
else:
  print("fail to ``, rcv, ``myadr: ``, mya,
        ``quality: ``, qua)

while True:
  tbs_ = {"type": ``PING",
      ``value": random.randint(1,100),
      ``time": time.time()}
  tbsb = str.encode(json.dumps(tbs_))
  try:
    add, qua, res = ctpc.sendit(rcv, tbsb)
    print("ACK from ``, add,
          ``quality: ``, qua, ``result:", res)
  except Exception as e:
    print ("EXCEPTION  -> ``, e)
    break

  print(waiting pong from: ', rcv)
  try:
    r_d, ad = ctpc.recvit(rcv)
    print("pong", r_d, ``from", ad)
  except Exception as e:
    print ("EXCEPTION  ->", e)
    break

  if input("Q to exit: ``) == ``Q": break
```

Fig. 5. Code of the ping.py example.

```
import ...

ctpc = loractp.CTPendpoint()

myaddr, rad, status = ctpc.listen()
if (status == 0):
    print("conn from:", rad,
          `` to me: ``, myaddr)
else:
    print("fail conn from:", rad,
          `` to me: ``, myaddr)

while True:
    print('waiting data from ', rad)
    try:
        rcvd_data, addr = ctpc.recvit(rad)
        print("got ``, rcvd_data, addr)
    except Exception as e:
        print ("EXCEPTION!! ``, e)
        break

    tbs = {"type": ``PONG",
           ``value": random_in_range(),
           ``time": time.time()}
    tbsb = str.encode(json.dumps(tbs))
    try:
        add,qual,res=ctpc.sendit(rad, tbsb)
        print("ACK from ``, add,
              ``quality: ``, qual,
              ``result: ``, res)
    except Exception as e:
        print ("EXCEPTION!! ``, e)
        break
```

Fig. 6. Code of the pong.py example.

We give more weight (0.875) to historical values than the last measured values. The sending operation returns statistics regarding the total packets sent, the total number of retransmissions, and if the transmission had to be aborted, which is indicated with a FAILED flag.

On top of this flow-control protocol, a lightweight transport protocol is used to establish a connection between a master node and a client node. Fig. 4 shows a simple sequence example. The master node monitors incoming connections, which are established in the unicast manner (i.e., providing the master node's address) or the anycast manner (i.e., sending to the generic "00000000″ address) to receive replies from nearby listening devices. This possibility allows for greater flexibility, such as establishing dynamic topologies in rural areas.

The evaluation code (see Section 6) is a simple adaptation of the "ping-pong" scheme, where applications send messages (pings, see Fig. 5) to listening nodes and obtain replies (pongs, see Fig. 6) once a node receives a ping. We adapted our evaluation code to allow for variable-sized content. These code snippets (Figs. 5 and 6) illustrate the LoRaCTP's API and the four primitives that permit easy control of content transfer: connect(), listen(), sendit(), recvit().

6. Evaluation

This section presents the performance results of LoRaCTP when it is used to transfer content over varying distances between nodes with varying data sizes; we also compared performance with two differing SFs (SF7 and SF12).

The devices we used as nodes were LoPy4 by Pycom.[3] The LoPy4 (see Fig. 7) is a quadruple bearer MicroPython-enabled development board with IEEE 802.11 b/g/n, Bluetooth v4.2 BR/EDR and BLE, LoRa (Semtech SX1276), Sigfox with an Espressif ESP32 chipset (Xtensa dual–core 32–bit LX6), 520 KB + 4 MB of RAM, and 8 MB of an external flash. It has a dual processor with a network processor that handles its WiFi connectivity and IP stack, leaving the main processor free to run user applications.

In addition to the LoRaCTP code used with each LoPys, an extension was employed to allow each device to be used as a LoRa adapter due to connections going through a serial/USB port.

We considered the following distances between any two nodes: 1, 100, 750, and 6000 m. The 1- and 100-m tests were performed using the facilities of the Universitat Politécnica de Valéncia. The 750-m tests were performed in the Ciudad de las Artes y las Ciencias area of Valencia, Spain, while 6000 m testing was performed between two viewpoints in Chiapas, Mexico. The latter locations were in areas high enough to preclude intervening obstacles.

We measured the system performance using a metric called "Successful Transfer Time (STT)". This metric measures message transfer times from the sender's perspective and is computed from the moment the first content of a message is sent to the moment the ACK of the final message content is received. All tests were performed using SF7 and SF12.

To determine the system stability, 10 message bursts were sent. We rarely observed retransmissions, even over the longest distances, indicating a negligible impact on SST. As we had to consider delays on the order of hundreds of seconds, a few additional seconds did not affect system usability significantly, and no effect was detected on message delivery.

Fig. 8 provides a clear illustration of STT evolution as a direct function of message size given SF7 (i.e., STT grows with message size) while the distance between nodes had a negligible effect. As expected, the overall throughput offered by LoRa is quite low, in the order of 250 bps (see Fig. 9).

Figs. 10 and 11 show the average STT values of various-sized messages (1, 10, 50, and 100 KB) sent at SF7 and SF12, respectively, measured against the distances between two nodes. Almost constant behavior can be observed in the results, although the STT clearly grows as the message size increases. The system is quite stable at an increasing

[3] https://pycom.io/.

Fig. 7. Node example based on a LoPy device.

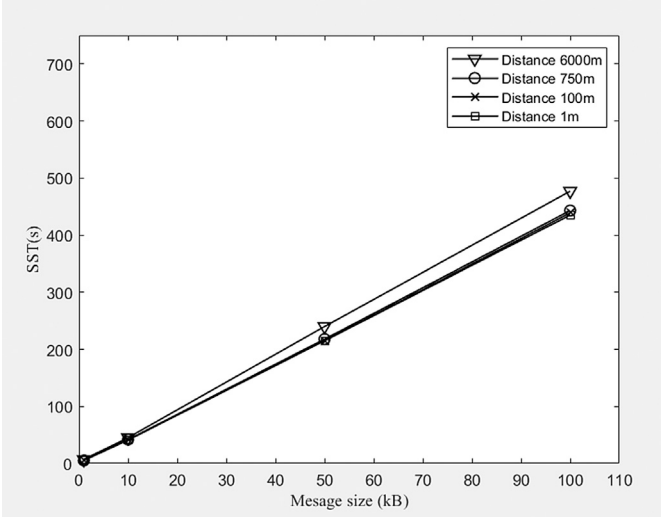

Fig. 8. STT behavior with varying message size (median values) using SF7.

distance, and very few retransmissions were required during the experiments.

From the above results, we can conclude that our solution is an effective and stable solution to integrate data from long distances using LoRa. For example, 100 KB messages transmitted under SF7 obtained a maximum delay of 457.56 s and a minimum of 451.49 s. Under SF12, the maximum and minimum delays increased to 82.545 and 82.513 min, respectively. While it is clear that the worst aspect of LoRa usage is low throughput, that aspect is compensated for by the distance covered and the low energetic cost required by these devices, providing a frugal solution to a notable problem. We are considering future work that adds a smart algorithm to our protocol, enabling switching between SF7 and SF12 according to specific scenarios, message sizes, and other aspects.

Preliminary evaluations were also conducted to determine data transfer energy costs. Fig. 12 shows how much energy was required by a device that was operating. Peaks were due to LED blinking, and the average power usage was approximately 176 μWh.

Fig. 13 shows the energy required to send a single packet message, with the segment between the two red arrows representing all phases involved in session establishment, sending the packet, and receiving the

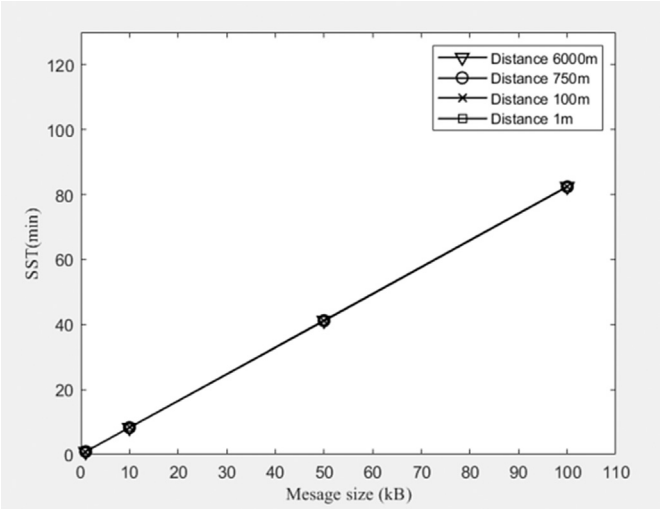

Fig. 9. STT behavior with varying message size (median values) using SF12.

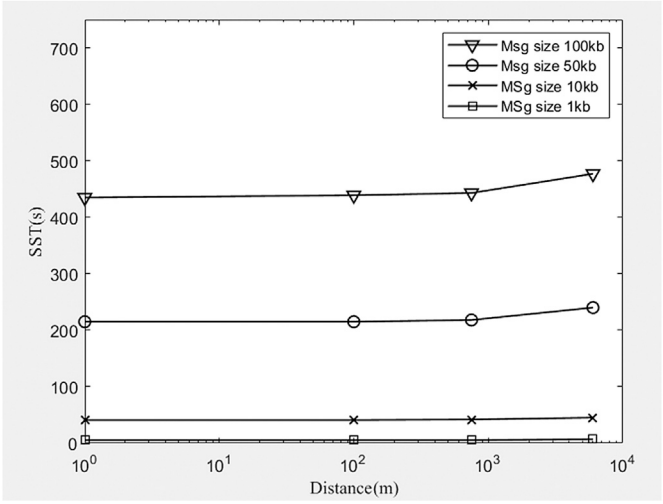

Fig. 10. STT versus distance between two nodes using SF7.

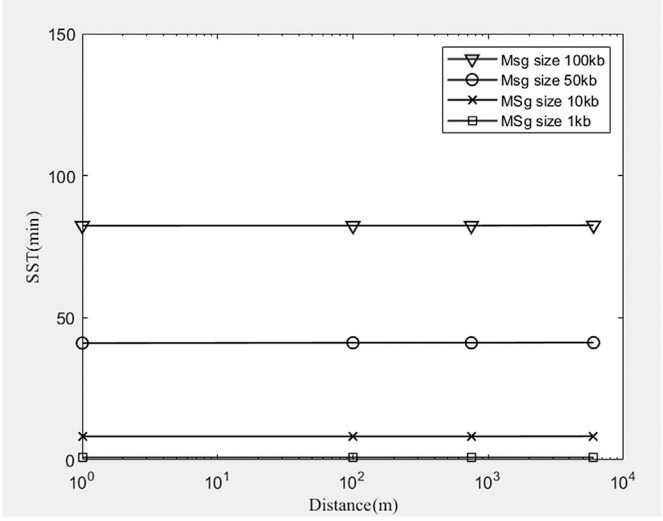

Fig. 11. STT versus distance between two nodes using SF12.

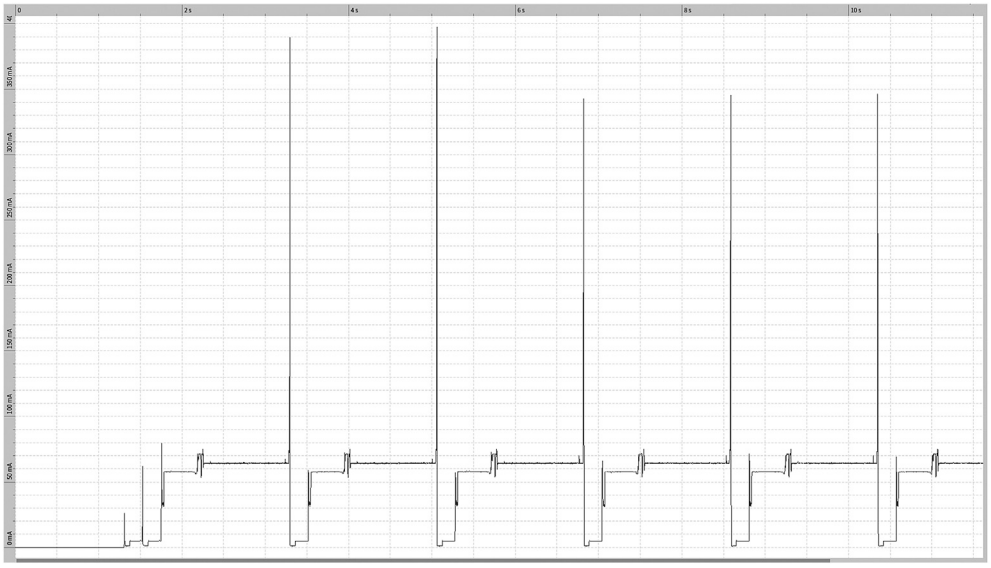

Fig. 12. Representation of energy consumed by the device when no operations are performed.

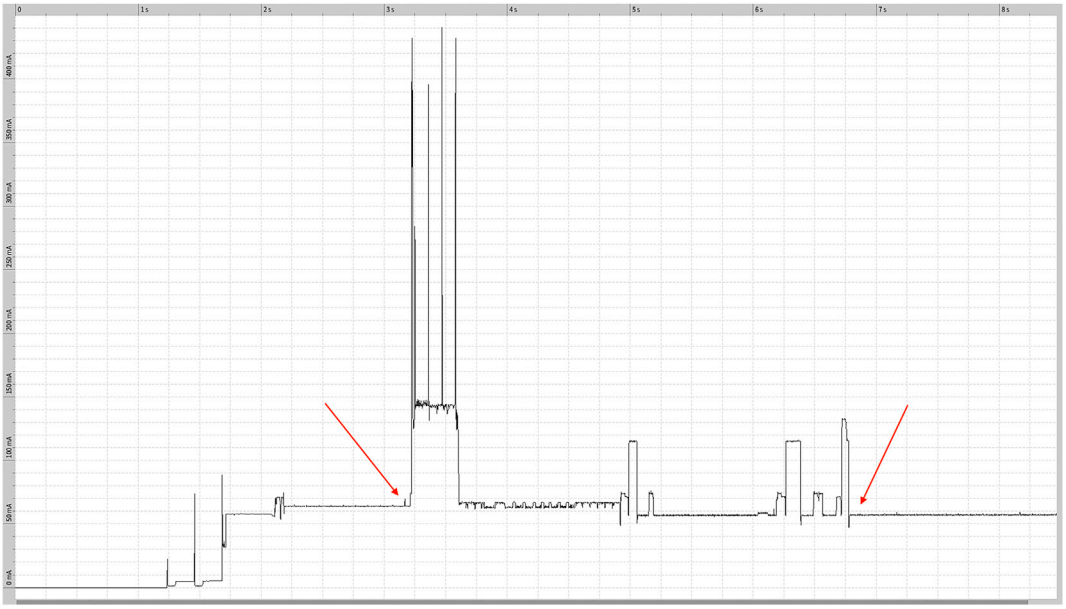

Fig. 13. Representation of energy consumed by the device when a single packet message is sent.

ACK. The block required 287 μWh, indicating an expenditure of 110 μWh to send the packet.

Fig. 14 shows the energy required by the device to send five packets and the corresponding process. This figure clearly indicates the sending of each packet, the waiting interval for the ACK to arrive, and the arrival of the ACM message (for example, the sequence indicated between the two green arrows). Each of these block sequences has an energetic cost of 60 μWh

We can therefore devise a function to compute the energetic cost e_c of sending a packet. Under SF7, a given packet size p in bytes gives

$$e_c[\mu Wh] = 287 + (\lceil p / 222 \rceil - 1)*60$$

Since the LoPy was powered with 3.3V, the energetic cost in amperes is

$$e_c[\mu Ah] = e_c[\mu Wh]/3.3$$

7. Conclusions

In this paper, we presented a flexible protocol based on LoRa technology that allows content transfers across long distances with low energy expenditure, provided all necessary mechanisms for LoRa reliability, and introduced a lightweight connection that permitted the ideal sending of any data message length. We evaluated our protocol's performance and reliability using the content of varying sizes transmitted across various distances.

We designed this protocol as communication support for small-data edge-based IoT solutions, given its stability, low power usage, and distance capacity.

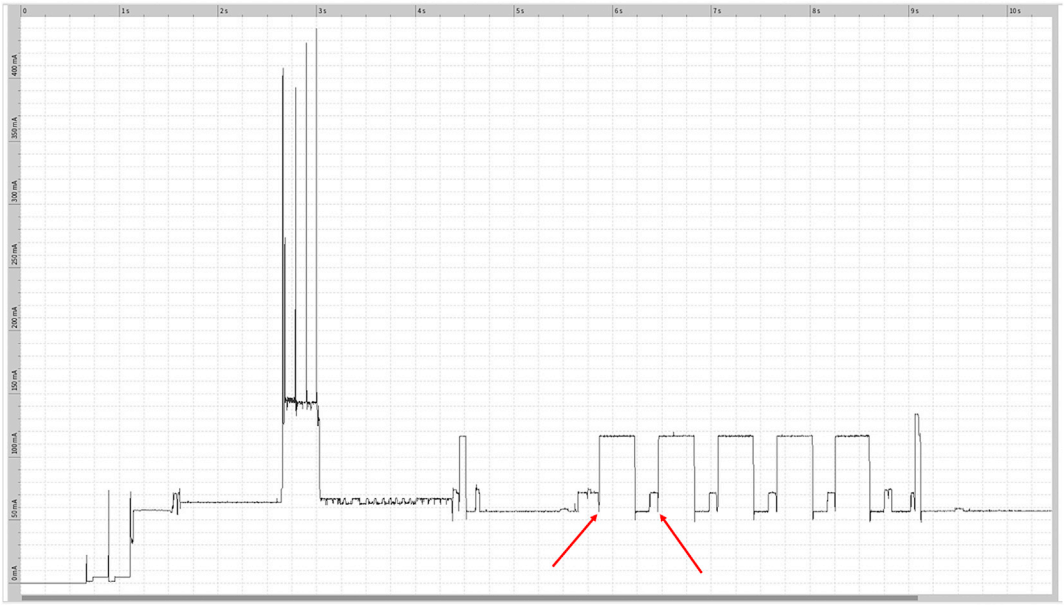

Fig. 14. Representation of energy consumed by the device when five packets are sent.

Declaration of competing interest

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

Acknowledgment

This work was partially supported by the "Conselleria de Innovación, Universidades, Ciencia y Sociedad Digital", Proyectos AICO/2020, Spain, under Grant AICO/2020/302 and "Ministerio de Ciencia, Innovación y Universidades, Programa Estatal de Investigación, Desarrollo e Innovación Orientada a los Retos de la Sociedad, Proyectos I+D+I 2018", Spain, under Grant RTI2018-096384-B-I00.

References

[1] D. Estrin, Small data, where n = me, Commun, ACM 57 (4) (2014) 32–34, https://doi.org/10.1145/2580944.
[2] M. Thinyane, Small data and sustainable development — individuals at the center of data-driven societies, in: 2017 ITU Kaleidoscope: Challenges for a Data-Driven Society, ITU K, 2017, pp. 1–8, https://doi.org/10.23919/ITU-WT.2017.8246991.
[3] O. Kennedy, D.R. Hipp, S. Idreos, A. Marian, A. Nandi, C. Troncoso, E. Wu, Small data, in: 2017 IEEE 33rd International Conference on Data Engineering, ICDE, 2017, pp. 1475–1476, https://doi.org/10.1109/ICDE.2017.216.
[4] Y.W. Teh, On big data learning for small data problems, in: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 3, https://doi.org/10.1145/3219819.3219941.
[5] C.R. Banbury, et al., Benchmarking TinyML Systems: Challenges and Direction, 2020, 04821 arXiv 2003.
[6] B. S. Chaudhari Bs, Zennaro M, LPWAN technologies: emerging application characteristics, requirements, and design considerations, Future Internet 12 (3).
[7] K. Nakamura, P. Manzoni, M. Zennaro, J.-C. Cano, C.T. Calafate, J.M. Cecilia, FUDGE: A Frugal Edge Node for Advanced IoT Solutions in Contexts with Limited Resources, Association for Computing Machinery, New York, NY, USA, 2020, pp. 30–35, https://doi.org/10.1145/3410670.3410857. URL.
[8] R. Baldoni, R. Beraldi, L. Querzoni, A. Virgillito, Efficient publish/subscribe through a self-organizing broker overlay and its application to SIENA, Comput. J. 50 (4) (2007) 444–459, https://doi.org/10.1093/comjnl/bxm002. https://academic.oup.com/comjnl/article-pdf/50/4/444/1179684/bxm002.pdf.
[9] A. Majumder, N. Shrivastava, R. Rastogi, A. Srinivasan, Scalable content-based routing in pub/sub systems, in: IEEE INFOCOM 2009, IEEE, 2009, pp. 567–575.
[10] J.L. Martins, S. Duarte, Routing algorithms for content-based publish/subscribe systems, IEEE Communications Surveys & Tutorials 12 (1) (2010) 39–58.
[11] G. Siegemund, V. Turau, K. Maâmra, A self-stabilizing publish/subscribe middleware for wireless sensor networks, in: 2015 International Conference and Workshops on Networked Systems (NetSys), IEEE, 2015, pp. 1–8.
[12] V. Turau, G. Siegemund, Scalable routing for topic-based publish/subscribe systems under fluctuations, in: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), IEEE, 2017, pp. 1608–1617.
[13] A. Al-Fuqaha, A. Khreishah, M. Guizani, A. Rayes, M. Mohammadi, Toward better horizontal integration among iot services, IEEE Commun. Mag. 53 (9) (2015) 72–79.
[14] P. Jutadhamakorn, T. Pillavas, V. Visoottiviseth, R. Takano, J. Haga, D. Kobayashi, A scalable and low-cost mqtt broker clustering system, in: 2017 2nd International Conference on Information Technology (INCIT), IEEE, 2017, pp. 1–5.
[15] S. Sen, A. Balasubramanian, A highly resilient and scalable broker architecture for iot applications, in: 2018 10th International Conference on Communication Systems & Networks (COMSNETS), IEEE, 2018, pp. 336–341.
[16] R. Banno, J. Sun, M. Fujita, S. Takeuchi, K. Shudo, Dissemination of edge-heavy data on heterogeneous mqtt brokers, in: 2017 IEEE 6th International Conference on Cloud Networking (CloudNet), IEEE, 2017, pp. 1–7.
[17] A. Schmitt, F. Carlier, V. Renault, Dynamic bridge generation for iot data exchange via the mqtt protocol, Procedia computer science 130 (2018) 90–97.
[18] A. Schmitt, F. Carlier, V. Renault, Data exchange with the mqtt protocol: dynamic bridge approach, in: 2019 IEEE 89th Vehicular Technology Conference (VTC2019-Spring), IEEE, 2019, pp. 1–5.
[19] T. Rausch, S. Nastic, S. Dustdar, Emma: distributed qos-aware mqtt middleware for edge computing applications, in: 2018 IEEE International Conference on Cloud Engineering (IC2E), IEEE, 2018, pp. 191–197.
[20] J.-H. Park, H.-S. Kim, W.-T. Kim, Dm-mqtt, An efficient mqtt based on sdn multicast for massive iot communications, Sensors 18 (9) (2018) 3071.
[21] M. Satyanarayanan, G. Klas, M. Silva, S. Mangiante, The seminal role of edge-native applications, in: 2019 IEEE International Conference on Edge Computing, EDGE, 2019, pp. 33–40.
[22] M. Wang, L. Zhu, L.T. Yang, M. Lin, X. Deng, L. Yi, Offloading-assisted energy-balanced iot edge node relocation for confident information coverage, IEEE Internet of Things Journal 6 (3) (2019) 4482–4490.
[23] R. Li, H. Asaeda, J. Li, X. Fu, A verifiable and flexible data sharing mechanism for information-centric iot, in: 2017 IEEE International Conference on Communications, ICC, 2017, pp. 1–7.
[24] M.A. López Peña, I. Muñoz Fernández, Sat-iot: an architectural model for a high-performance fog/edge/cloud iot platform, in: 2019 IEEE 5th World Forum on Internet of Things, WF-IoT, 2019, pp. 633–638.
[25] M. Nunes, R. Alves, A. Casaca, P. Póvoa, J. Botelho, An internet of things based platform for real-time management of energy consumption in water resource recovery facilities, in: L. Strous, V.G. Cerf (Eds.), Internet of Things. Information Processing in an Increasingly Connected World, Springer International Publishing, Cham, 2019, pp. 121–132.
[26] L. Angrisani, A. Amodio, P. Arpaia, M. Asciolla, A. Bellizzi, F. Bonavolontà, R. Carbone, E. Caputo, G. Karamanolis, V. Martire, M. Marvaso, R. Peirce,

A. Picardi, G. Termo, A.M. Toni, G. Viola, A. Zimmaro, An innovative air quality monitoring system based on drone and IoT enabling technologies, in: 2019 IEEE International Workshop on Metrology for Agriculture and Forestry, MetroAgriFor, 2019, pp. 207–211.

[27] P. Boccadoro, B. Montaruli, L.A. Grieco, Quakesense, a lora-compliant earthquake monitoring open system, in: 2019 IEEE/ACM 23rd International Symposium on Distributed Simulation and Real Time Applications, DS-RT, 2019, pp. 1–8.

[28] E. Longo, A.E.C. Redondi, M. Cesana, A. Arcia-Moret, P. Manzoni, Mqtt-st: a spanning tree protocol for distributed mqtt brokers, in: ICC 2020 - 2020 IEEE International Conference on Communications, ICC, 2020, pp. 1–6, https://doi.org/10.1109/ICC40277.2020.9149046.

[29] A. Banks, E. Briggs, K. Borgendale, R. Gupta, Mqtt Version 5.0, OASIS Standard.