UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS
UNIDADE ACADÊMICA DE GRADUAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO

FUAD FREDERIK SAUD

EXTENSIBLE ELASTIC INTEROPERABILITY MIDDLEWARE FOR THE INTERNET OF THINGS

São Leopoldo
2017

Fuad Frederik Saud

EXTENSIBLE ELASTIC INTEROPERABILITY MIDDLEWARE FOR THE INTERNET OF THINGS

Artigo apresentado como requisito parcial para obtenção do título de Bacharel em Ciência da Computação pelo Curso de Ciência da Computação da Universidade do Vale do Rio dos Sinos – UNISINOS

Orientador: Prof. Dr. Rodrigo da Rosa Righi

São Leopoldo

2017

EXTENSIBLE ELASTIC INTEROPERABILITY MIDDLEWARE FOR THE INTERNET OF THINGS

Fuad Frederik Saud*

Rodrigo da Rosa Righi**

Abstract: The Internet of Things has been growing at a fast pace, with constant evolution in the wide range of technologies used to enable it. More and more companies are entering the market, introducing new devices, use cases and solving common problems in novelty ways. This combination of the accelerated rhythm and the broadness of impact ends up creating a certain chaos when it comes to standardization. And while open IoT communities have been able to establish a few standards that allow for a more homogeneous ecosystem, interoperability between device networks is still a huge issue for IoT application developers. This project investigates the creation of a middleware that enables IoT applications and devices that are built with support for different communication protocols (such as HTTP, MQTT and CoAP), to interact with each other in such a way that the burden of interoperability is alleviated for the application developers. The middleware is designed to be agnostic to protocol details and communication model - allowing for both synchronous and asynchronous semantics - and to also allow elasticity when deployed in a cloud computing environment, where the decoupled nature of its components favours a granular level of control over the reaction to dynamic network and load conditions.

Keywords: Internet of Things, Middleware, Interoperability, Elasticity

1 INTRODUCTION

The Internet of Things, as defined by XIA et al., is the "networked interconnection of everyday objects", creating a wave of sensors, actuators, services and other applications available through the internet. Use cases range from smart cities and autonomous cars to intelligent agricultural systems and ubiquitous environments, like smart homes and hospitals.

For example, IoT enabled sensors can make environments like a large city, a crowded hospital or a manufacturing site more efficient, by enabling real time data collection and analysis, allowing systems and humans to make better decisions and act accordingly. They can make peoples life's safer by improving prediction of natural disasters and

* Aluno do curso de Ciência da Computação. Email: fuadfsaud@gmail.com

** Orientador, professor da Unisinos, pós-doutor pelo Korean Advanced Institute of Science and Technology (2013), doutor em Ciência da Computação pela Universidade Federal do Rio Grande do Sul e pela Tecnische Universitaet Berlin (2009), Mestre em Ciência da Computação pela Universidade Federal do Rio Grande do Sul (2005). Email: rrrighi@unisinos.br

therefore allowing for better evacuating conditions. Smart farms can reduce the loss by using real data for more accurate crop forecasting, which computers can perform better than humans. (NGU et al., 2017) (CHANDRA et al., 2017) (JAYARAMAN et al., 2016)

But in practice, the implementation of the Internet of Things that we observe today still has many open questions regarding security, scalability, service discovery, performance. One prominent pain point is the multitude of protocols that IoT enabled devices use to communicated with others. IoT networks are largely heterogeneous: different manufacturers build devices for different purposes and the lack of an open, well established standard, leads to the development of a wide range of different protocols, making it hard to integrate multiple devices and applications within a network. Solutions to this problem do exist, but the approaches vary a lot, with respect to purpose, level of interoperability, supported protocols, etc. (BOMAN; TAYLOR; NGU, 2014)

Alan Kay, the inventor of Object-Oriented Programming, during a conversation about the fundamentals of Smalltalk and OO concepts in a mailing list, mentioned the Internet as an analogy for the messaging capabilities of that programming paradigm:

> Think of the internet – to live, it (a) has to allow many different kinds of ideas and realizations that are beyond any single standard and (b) to allow varying degrees of safe interoperability between these ideas.

With that in mind, the goal of this project is to explore the development of an integrating middleware: one that allows for devices and applications that don't originally talk the same protocols to be integrated in a transparent fashion. The middleware must also be build in such a way that allows for elastic scalability, allowing it to adapt to rapidly changing networking loads by making easy to adapt adjust provisioning parameters.

To develop a scalable, extensible middleware for the Internet of Things capable of integrating multiple communication protocols used in IoT applications and design its own output protocol.

- Identify the most important communication protocols used in the IoT.
- Investigate a set of semantics that can be applied to map communication from

one protocol to another.

- Develop a middleware prototype that applies these semantics and can translate between the existing protocols.
- Design the middleware semantics in such a way that it can be extended to support numerous protocols.
- Allow the middleware to be horizontally scalable by efficiently making use of existing cloud elasticity solutions.

This work has been divided in 7 sections. This section introduces the theme and the problems to be solved. Following up, section 2 reviews the existing literature on the state-of-art concepts and technologies used in this work. Section 3 presents a series of works that describe and propose solutions to similar problems, shedding light over the existing gaps that this project covers. Section 4 describes in detail the proposed middleware solution, including design decisions and an architectural overview. Section 5 goes about the methodology used for development and testing while section 6 discusses the obtained results. Finally, section 7 closes the text with a review of the work done and considerations about future work.

2 LITERATURE REVIEW

.

This chapter describes relevant fundamental concepts in the area of Internet of Things, ubicomp and Cloud Computing, that are essential for the development of this project.

2.1 Internet of Things

The Internet of Things represents the interconnection of a wide range of everyday life devices, at an extremely large scale. From home appliances, wearable devices, and health care machinery to smart cars and smart cities, a multitude of wireless devices can be plugged to internet and weaving a complex network of sensors, actuators,

Figure 1 – An overview of IoT markets
Source: AL-FUQAHA et al.

interactive applications, and ubiquitous interfaces. (AL-FUQAHA et al., 2015) (DESAI; SHETH; ANANTHARAM, 2015)

IoT represents a great market opportunity for many industries, ranging from technology players like hardware manufacturers and applications developers, to actual businesses that can explore the concept to refine techniques, gain efficiency and improve quality of life. Figure 2 shows a projection of the economic growth for markets that are expected to make heavy use of IoT solutions in the next decade. Numbers for the health care and manufacturing markets stand ou considerably. (AL-FUQAHA et al., 2015)

Such scenarios bring many new challenges to hardware manufacturers and application developers. In order to enable the expansion of IoT and fostering the success of its wide adoption, conditions, devices and applications must address concerns as scalability, interoperability, transparency; networks and protocols must be loosely coupled in order to unlock the full power of interconnected environments. KATOLE et al.

Interoperability in the Internet of Things might happen in different ways, at different levels of the stack. Figure 3 shows common IoT protocols side by side with their counterparts in the Web stack, categorized by layer.

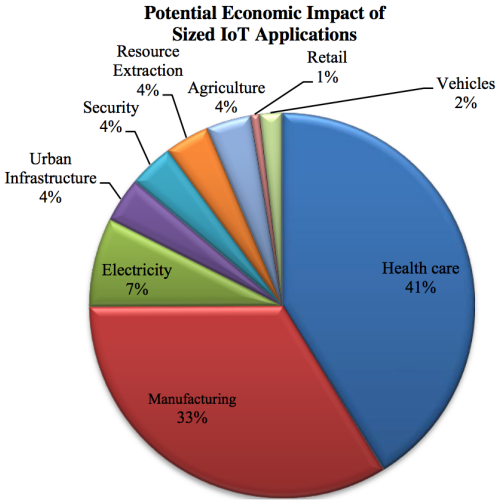

Figure 2 – Projected market share of dominant IoT applications by 2025
Source: AL-FUQAHA et al.

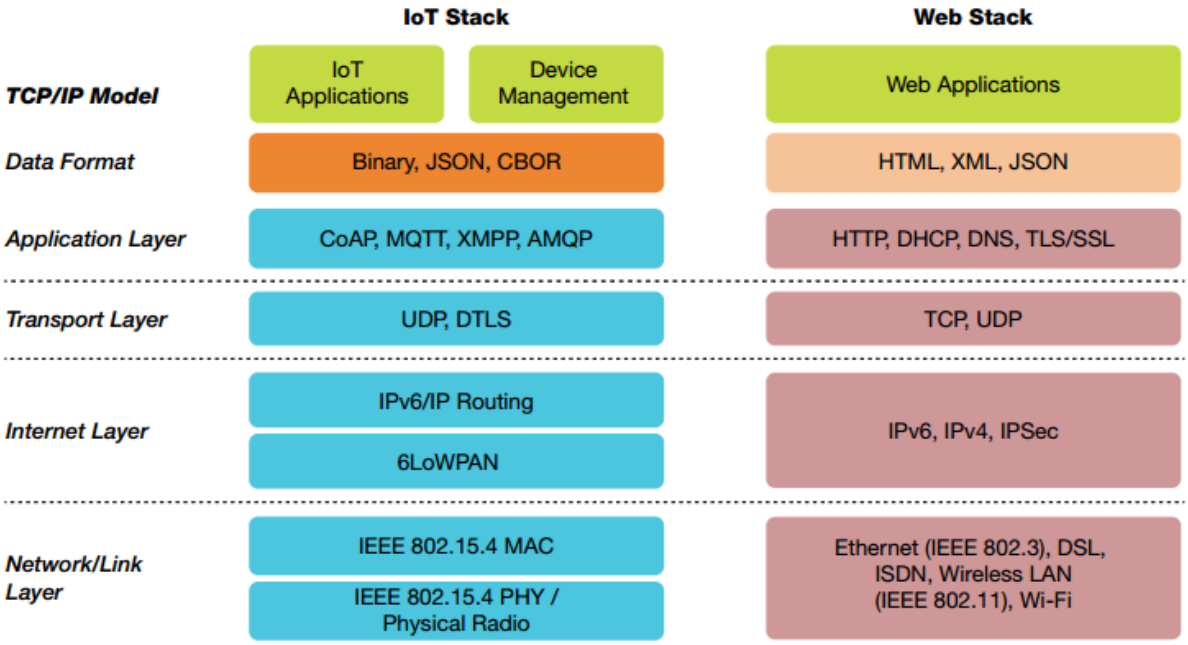

Figure 3 – Standardized IoT protocols, in contrast with the Web stack
Source: CHAKRABARTI

Application Layer	**HTTP**	**CoAP**	**MQTT**
Transport Layer	TCP	UDP	TCP
Network Layer	IPv4/IPv6	6loWPAN	
QoS		2 levels: Confirmable and Unconfirmable messages	3 levels: no guarantees, delivered at least once and delivered exactly once
Content Negotiation	Accept/Content-Type headers	Accept/Content-Type headers	
Messaging Model	Request/Response	Request/Response + Subscribe	Pub/Sub

Table 1 – Comparison between selected IoT protocols

2.2 Protocols

Since the solution proposed in this project - more specifically in chapter **??** - focuses on dealing with interoperability at the application layer, it is wise to research the most common protocols that operate at this level. These can be summarized as: CoAP, MQTT, HTTP, AMQP and XMPP. (DERHAMY; ELIASSON; DELSING, 2017) For the scope of this project, a subset is going to be considered initially - the three first -, while others can - and should - come up in future works.

2.2.1 HTTP

The Hypertext Transfer Protocol is the standard protocol for the World Wide Web. It is based on a client/server, request/response, mostly synchronous interaction model. Asynchronicity can somewhat be achieved by making use of webhooks (TRIFA et al., 2010). It is usually conjoined with REST semantics which advocate for interaction by means of resource state transfer.

Interactions between clients and servers start with the client issuing a request of a certain type - defined by the request method, which is represented by a verb, the most common being `GET`, `POST`, `PATCH`, `PUT` and `DELETE` to a given server. Each request carries a resource locator (URL) which contains the server's host address, along with a resource identifier - to locate the server host - and a path that identifies the resource o which to operate within that server.

Nor HTTP nor REST themselves specify resource discovery mechanisms, but abstractions such as hypermedia API's are built to address that. (ALARCON et al., 2015) Since HTTP runs on top of TCP, security is handled at the transport layer by TLS (in what's called Secure HTTP, or HTTPS). (RESCORLA, 2000)

2.2.2 CoAP

The Constrained Application Protocol is an Application Layer Protocol specified by the Internet Engineering Task Force (IETF) Constrained RESTFul environments (CoRE).

It is built on a client/server, request/response interaction model. It is based on the REpresentational State Transfer model, and the semantics are largely shared same defined by HTTP. Clients issue requests by specifying an action - the same ones present in HTTP, e.g. `GET`, `POST`, `PATCH`, `PUT` and `DELETE`, etc - to be performed upon a resource located in the server, identified by an URL. The server accepts the request and issues back a response containing a status code - that indicates the result of the request actions - and, optionally, a representation of the requested resource. (SHELBY; HARTKE; BORMANN, 2014)

CoAP differs from HTTP in that it relies on UDP for the transport layer, rather than on TCP. This means that reliability is not guaranteed by the transport layer; instead, it is handled at the application layer level, by specifying four types of messages: confirmable, non-confirmable, acknowledgement and reset. The messaging layer of CoAP, which takes care of coordinating the transmission of data over UDP, is responsible for detecting duplicate messages, avoiding network congestion and treating errors. Figure 4 depicts the four CoAP message types in action. (AL-FUQAHA et al., 2015) (SHELBY; HARTKE; BORMANN, 2014) (ECLIPSE NEWSLETTER: MQTT AND COAP, IOT PROTOCOLS, 2014)

The main goal of CoAP is to enable RESTful interactions to be performed by IoT devices - which are usually constrained when it comes to computation and networking capabilities. (AL-FUQAHA et al., 2015) It also facilitates peer-to-peer interaction among IoT devices, by having them act simultaneously as both RESTful clients and servers. (JIMENEZ et al., 2015)

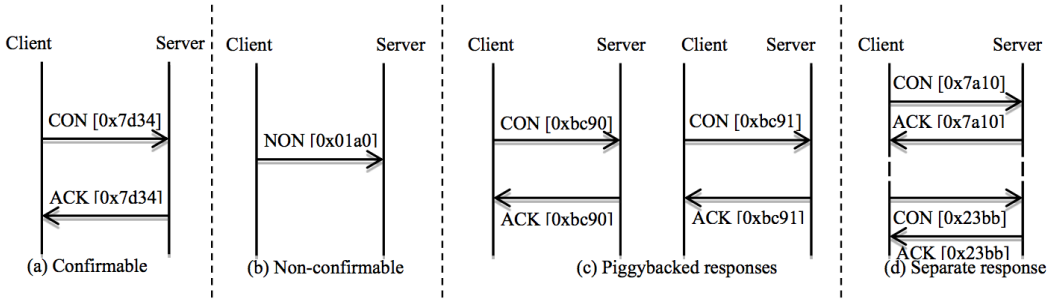

Figure 4 – Different CoAP message types
Source: AL-FUQAHA et al.

CoAP has baked-in basic resource discovery and caching capabilitiees. Security is handled by making use of DTLS, typically with RSA/AES encryption. (ECLIPSE NEWSLETTER: MQTT AND COAP, IOT PROTOCOLS, 2014) (CHAKRABARTI, 2015)

2.2.3 MQTT

The Message Queue Telemetry Transport is a messaging protocol based on a publish/subscribe model. It was developed in 1999 and became an standardized at OASIS in 2014. (BANKS; GUPTA, 2014)

In MQTT, clients connect to a broker which defines topics for communication. Clients can publish messages to specific topics and also subscribe to topics, which means they'll listen to and be informed about new messages published by other clients on these topics. Topics are identified by their names and are created automatically when a client publishes a message to it.

Figure 5 illustrates an example interaction between IoT sensors and actuators as well an application. Sensors like IoT enabled thermometers or beacon enabled apps running on a smartphone could publish messages containing data points - like the current temperature in a room or the identifier of the closest beacon - to topics on the broker. Other clients interested on these data streams can subscribe to the specific topics in order to be notified of any updates. Applications running on a server can subscribe to many topics and consume data points coming from various sensors. That allows them to define complex business rules based on these inputs and decide how to react to different scenarios. The output of the application can be a message published to the very same broker, to be consumed by other IoT-enabled actuator de-

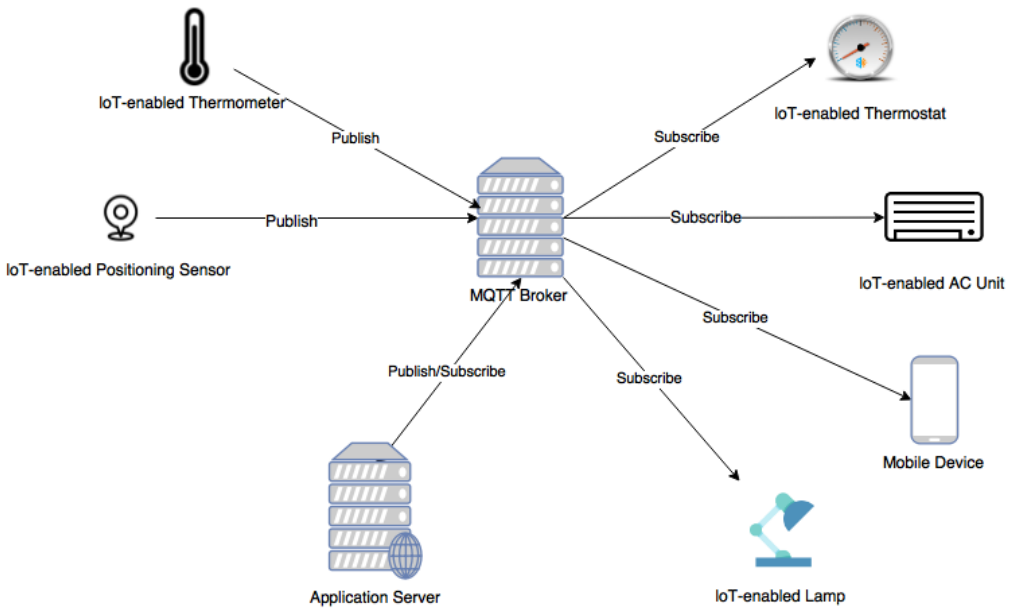

Figure 5 – Example MQTT network interaction

vices. For example, a home automation application can subscribe to topics where a set of thermometers - installed throughout the house - publish regular updates on the room temperature. It can also subscribe to indoor position data coming from the user's smartphone, obtained through a beacon positioning system. It can then use this information to coordinate with other devices, like a thermostat, an individual ac unit or a lamp and issue new messages on topics these devices are listening to, making them react to changes in the environment. This allows the application to turn the lights on and adjust the room temperature when it identifies that someone walks in.

2.3 Cloud Computing

The concept of Cloud Computing comes from nature of decoupling physical machines from computing resources. Deploying applications on the Cloud means to have an infrastructure stack that's fully managed by a Cloud services provider. The advantages brought by this model are huge: cost is reduce because the Cloud provider can optimize the use of hardware across virtual computing resources allocated to different customers and customers don't need to worry about the hassle of maintaining physical data centers. One great effect of this is that it enables for application elasticity, which

is defined by HERBST; KOUNEV; REUSSNER as "the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available re- sources match the current demand as closely as possible". This means cloud applications can scale automatically and react to changing demands over time. (AAZAM et al., 2014)

The Cloud is one big enabling factor behind IoT. It enables the processing of big data coming from the multitude of sensors distributed across Wireless Sensor Networks. In this project, the Cloud is explored as the deploy environment for the proposed middleware. Combined with a service oriented architecture, cloud service providers like Amazon Web Services [1], and Google Cloud Platform [2], service instances can be provisioned and deprovisioned automatically, which goes well with the dynamic nature of IoT applications. SoA Beside general Cloud services for application deployment, management and monitoring, there's also a multitude of IoT specific products, that make it easier for integrating complex networks. Amazon, for example, offers the AWS IoT platform, which can be explored for automatically scaling MQTT message brokers.

3 RELATED WORK

NGU et al. classify three major types of existing IoT middleware solutions: service-based - where IoT devices are deployed as services (eg. LinkSmart (LINKSMART® MIDDLEWARE PLATFORM PORTAL, 2017) and GSN (GLOBAL SENSOR NETWORKS - A MIDDLEWARE FOR PROCESSING SENSOR DATA IN THE INTERNET, 2011))-, cloud-based - in which the types of devices that can be deployed are reduced, but the process of collecting and analyzing data is facilitated by the fact that use cases are pre-defined, and usually controlled by the cloud provider (eg. Google Nest (NEST, 2017), Google Fit (GOOGLE FIT, 2017) and Xively (XIVELY, 2017)- and actor-based - where the middleware can be deployed at any layer of the network (the sensory layer, a mobile access layer and the cloud) and it explores a plug-and-play IoT architecture (eg. Calvin (PERSSON; ANGELSMARK, 2015), Ptolemy (PTOLEMY IOT - GOOGLE SEARCH, 2017) and (NODE-RED, 2017)). Most of the works described in this section, as well as the solution proposed by this work would fall in to the service-based

[1] https://aws.amazon.com
[2] https://cloud.google.com

category.

According to (AL-FUQAHA et al., 2015), the biggest challenges in IoT are "availability, reliability, mobility, performance, scalability, interoperability, security, management, and trust". Researching and addressing these problems is a central point in maturing the IoT environment and making it become a widely adopted, universal concept. (DESAI; SHETH; ANANTHARAM, 2015) approaches the subject arguing about the existence of vertical silos of communication that arise from each specific domain, precluding interconnectivity between domain stacks at the Network, Messaging and Data Model layers. Incompatibility at the network level is mostly characterized by domain specific protocols operating at the hardware level, but the paper doesn't dive into this problem, instead, focusing on the application level - which includes the messaging protocols semantics and data model.

In (THANGAVEL et al., 2014), a simple gateway middleware is developed. It is an unidirectional gateway for transporting aggregated sensor data from sensor networks to backend servers and brokers, and the main purpose of the project is to allow backend applications to consume data from sensor networks independently from the protocol used by the WSN itself and the data aggregation gateway, similar to the one proposed by DESAI; SHETH; ANANTHARAM. Since the data flow is unidirectional, the semantics are oversimplified and therefore the solution doesn't favour the flow of communication from IoT applications to sensor networks, or even between two sensor networks setup with different protocols. The gateway supports two output protocols, CoAP and MQTT, but is extensible in such a way that new protocols could be included; input is based on the designed API which supports two calls: one for publishing messages to the gateway and another for checking the if the message was successfully forwarded. Moreover, the paper describes some experiments performed to assess and compare the packet traffic in the network for both protocols and the impact of packet loss in maintaining a reasonable QoS (MQTT level 1 and CoAP confirmable messages).

Regarding interoperability, AL-FUQAHA et al. makes a critique of the existing attempts to solve the problem. It regards (PONTE, 2013) as one of the biggest contenders, as it provides "uniform" open API's to enable automatic conversion between various IoT protocols". Ponte defined by its authors as a "multi-protocol Internet of

Things/Machine to Machine broker; it supports MQTT and REST API's over HTTP and REST. Ponte focuses on exposing the MQTT API through REST in a transparent way, such that developers can access pub/sub semantics through a request/response interaction model. The article, however, states that solutions like Ponte are not ideal for a couple of reasons: it doesn't take into consideration the common resource constraint of limited IoT devices - since it requires device clients to implement communication through TCP/IP -, and since it's an application level any-to-any protocol translator, it implies in increased packet sizes, making communication more verbose and less performant - although it doesn't show explicit examples of how this could be the case, nor actual numbers.

Ponte is also criticized in (LEE et al., 2016). One of the flaws raised by the author is the fact that the broker requires all clients to be connected to the middleware that's the deployed in the cloud, preventing devices from communicating directly with each other. The proposed solution for that problem is to use a Software Defined Network switch to intercept packet and make decisions based on an URL within the packet: if the packet is coming from an MQTT network to a CoAP network - or vice-versa - the switch sends the packet to middleware, listens for the response and forwards it to the original target. Another point of concern is that the middleware semantics disregard the common scenario for the use of CoAP where the constrained devices play the role of the server. Since CoAP is designed to allow direct device-to-device communication, it is expected that devices and applications play both the role of client and servers. Therefore, a CoAP application interested in reading data from a CoAP sensor issues a `GET` request to the device's endpoint and interprets the response. Pub/sub based protocols don't incorporate the notion of pulling data like that; instead, information gets pushed to interested clients by the producers of the data. Keeping generic, transparent semantics has the upside of keeping a natural API for clients, but has the downside of keeping only the natural semantic intersection between the protocols. The suggested solution for this is to have the SDN switch intercept messages coming from the MQTT broker and issue an "out-of-band" `GET` request to the CoAP endpoint, grab the response and send it back to the MQTT broker through the middleware. The author, however, do not get into the details of this "data-request" semantics for the MQTT clients.

Still covering the interoperability topic, DERHAMY; ELIASSON; DELSING raise

similar problems. They criticize solutions that approach interoperability by implementing protocol gateways, which usually operate on the network layer, as these do not address application layer protocols. A few middleware solutions are mentioned, like Starlink (BROMBERG; GRACE; RéVEILLèRE, 2011), uMiddle (NAKAZAWA et al., 2006), INDISS (BROMBERG; ISSARNY, 2005) and UIC (ROMAN; KON; CAMPBELL, 1999) but they are regarded by authors as a less than ideal solution, since they lock the applications to a particular technology and create interoperability problems with other middlewares, or having security and scalability problems. The paper goes on to propose an architecture based on the Arrowhead Framework [3]. It describes concerns such as service registry, authorization and orchestrations, but the most relevant section is the one that touches protocol translations. It suggests with a direct protocol-to-protocol translation scheme, in which, independently of the multitude of protocols that the tool might support, translation always happen in an isolated context, where only the input and output protocols are considered. This is done so to reduce the semantic information loss (as the semantic intersection tends to shrink across a large number of protocols). The translator is designed to run within the local IoT cloud.

Table 2 summarizes the analyzed interoperability solutions, comparing them with regard a few characteristics:

- **Actuation Level**: namely network or application level.
- **Direction of Communication**: sensor network to server or bi-directional (which might include device-to-device).
- **Supported protocols**: at least in the original implementation, since most of them support some level of extensibility.
- **Data format**: with regard to payload syntax, most protocols tend to be agnostic, simply forwarding the received data.
- **Security**: even though most solutions don't address security explicitly, some papers do make some considerations on the topic.
- **Extensibility**: how solutions address the addition and integration of new protocols for greater interoperability.

[3] http://www.arrowhead.eu/

Based on this research, this project aims to build on the architecture described in (COLLINA; CORAZZA; VANELLI-CORALLI, 2012) and evolved in (PONTE, 2013). The positive points of this architecture are the way it handles interoperability in a transparent way, allowing for maximum reuse of existing tools reducing the need for adaptations. And the way it decouples the protocol translation from the protocols semantics, by using an intermediary format. Inspiration is drawn from the other researched solutions, so to improve the middleware's scalability - by allowing a finer-grained control of how components scale independently from each other - and to augment the middleware's semantics.

4 MODEL

This section presents the proposed middleware solution to bridge the communication between devices and applications that implement different protocols. Subsections go into detail about the design decisions and trade offs that were considered, architectural overview, the intricacies of syntactic and semantic compatibility between the protocols that were evaluated and the proposed API for the middleware.

4.1 Design Decisions

Since the most prominent problem that this research tries to solve is the multitude of different protocols used by IoT enabled applications and devices, one desirable capability of a middleware that bridges them is extensibility: allowing new protocols to be added by just writing a new piece of adapter code that's capable of translating the syntax and semantics of a protocol to and from a common, core protocol. From this, it's noticeable the need for decoupling the parts of the middleware that deal with protocol specific syntax/semantics and the middleware's core semantics, which will define the common ground for protocols to be plugged in.

One attribute that is commonly used to categorize protocols is whether their communication semantics are synchronous or asynchronous. Synchronous protocols, those which mandate both sides of the communication wire to be online in order for information to be exchanged, are stricter that asynchronous ones, which allow information

Table 2 – Comparison between analyzed IoT interoperability solutions

	PONTE	**LEE et al.**	**AL-FUQAHA et al.**	**DERHAMY; ELIASSON; DELSING**	**THANGAVEL et al.**	**DESAI; SHETH; ANANTHARAM**
Actuation Layer	Application (protocol)	Application (protocol) and Network (packet routing)	Application (protocol)	Application (protocol)	Application (protocol)	Application (protocol)
Direction of communication	Bi-directional (with restrictions to request/response protocols)	Bi-directional	Bi-drectional	Bi-directional	From WSN to backend	From WSN to backend
Protocols covered	MQTT, CoAP, HTTP	MQTT, CoAP, HTTP	MQTT, DDS	MQTT, CoAP	MQTT, CoAP	MQTT, CoAP, XMPP
Data format	Agnostic	Agnostic	Agnostic	Agnostic	Agnostic	JSON and XML (for CoAP); XML-only for MQTT and XMPP
Security	Doesn't address; delegates higher level security to use of VPN's	Doesn't directly address	Doesn't directly address	Partially encrypted payloads; delegates authentication/authorization to external systems	Doesn't directly address	Discrimination of private and public sensor features accompanied by OAuth
Extensibility	The middleware is open-source and support for other protocols can be developed by implementing servers that interact with the clients and with the other protocol servers/brokers	Requires implementation of protocol boundaries in the cloud middleware and definition of routing semantics at the network level		Supports extension via definition of new high-level translation rules between protocols	Can be extended via implementation of protocol boundaries in the gateway, bridging incoming data and the gateway's semantic model to the that of the desired protocol	

consumption to be decoupled from information production, removing the necessity for both sides to be online simultaneously. Hence, to support both communication semantics at the same time, the core middleware solution should be designed with the looser model in mind, allowing protocol adapters to decide how to deal with the friction between the outside world and the middleware internals.

Being a Message-oriented middleware, the whole system revolves around the concept of messages. Since they are the most important building block and are shared across different components that will treat them using diverse semantic rules, it's a good practice to uniquely identify them. This allows for boundaries to have more control over One problem that can arise when two message brokers interact with each other is an infinite message loop, where brokers keep feeding each other the same messages over and over. This can be circumvented by having clients add UUID's to each message that gets published to the middleware. For the sake of simplicity, the UUID can be added as header, occupying the first 128bit of every message. (RFC 4122 - A UNIVERSALLY UNIQUE IDENTIFIER , UUID)

4.2 Architecture

This section describes the overall architecture proposed for a middleware implementation. In order to support the development of this research, a prototype middleware was implemented, serving as an aid to highlight design challenges and therefore facilitating the refinement of the model.

The two main components of the middleware are the middleware core and I/O boundaries. The middleware core can be described as a message broker with general pub/sub capabilities. It divides messages into topics (categories of messages) and allows for multiple interested clients to both produce and consume messages from topics. The prototype implementation, described in more detail on 5, uses the Apache Kafka message broker and streaming platform [4] to fulfill these needs. Besides behaving as message queue, Kafka also acts as a commit log, persisting every message that has ever been published to the broker. While this is a nice feature to have, it's not necessary to power the middleware's essential functionality; therefore, other message

[4] `https://kafka.apache.org`

queues implementations could be used to power the its publish/subscribe engine. The middleware core defines an API that can be used by other components interested in producing and/or consuming - more on that on section 4.4.

I/O boundaries are the brain of the middleware. They either translate incoming client communication semantics into messages that are produced in the middleware core, or consume messages from the core and communicate them to clients. Boundaries are completely separate components from the middleware core: they are run in separate processes and can be implemented using any programming language. Boundary components can be as simple as an adapter layer - which solely translates some incoming data from a client into a message for the core - or pretty complex, involving caching rules or a persistence layer. Application developers are therefore able to freely extend the middleware the way that's most useful to them.

Six boundaries were considered for implementation as a proof-of-concept: HTTP Server, HTTP Client, CoAP Server, CoAP Client, MQTT Subscriber and MQTT Publisher. Each boundary is a completely separate service, running in it's own process and has no direct relationship with other boundaries whatsoever. The HTTP Server boundary, for example, takes `POST` requests containing the message id, topic and payload. It publishes these messages using the middleware's core API and answers the client with a `202 Accepted` response. The HTTP Client boundary, on the other hand, takes WebHook-style subscriptions through `POST` requests, where clients specify a topic whose messages they are interested in, along with an URL through which they would like to be notified about them. The boundary consumes messages from the middleware core and issues `POST` request to subscribed clients. Figure 6 depicts and example of middleware deploy with the 6 aforementioned boundaries along with possible client interactions.

4.3 Addressing Syntax and Semantics translation

Protocol semantics are dealt with by the boundary components. They feed and are fed via the middleware's core message streaming API. Pushing protocol specific logic responsibility into small components that only rely on a simple and well defined API allows middleware development to scale better. One of the strengths of this strategy is

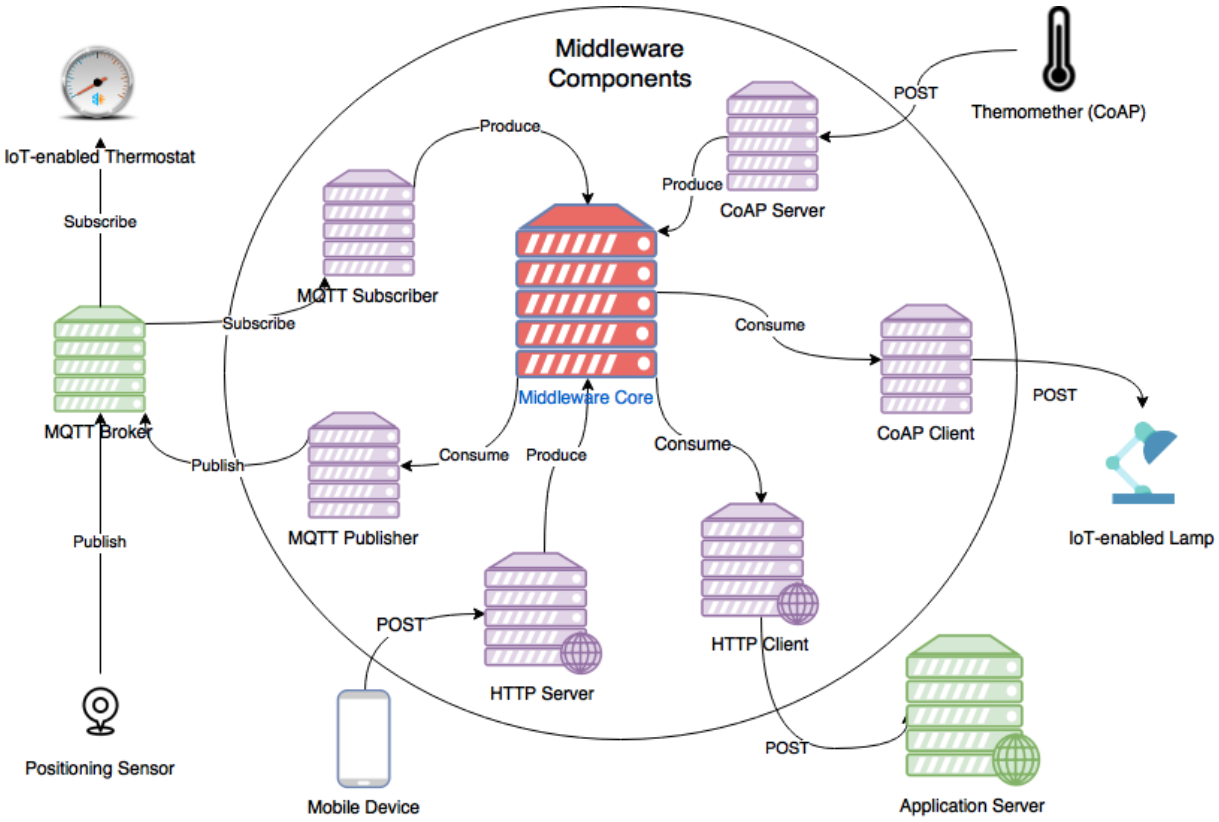

Figure 6 – Middleware boundaries mediate communication between clients and the middleware core.

the fact that it simplifies the translation of protocol syntax and semantics.

Following the publish/subscriber pattern, boundaries can offer simple operation semantics for clients, such as accept incoming messages from clients and publish them to the middleware core and consuming messages from the core and distributing to clients as they see fit.

The middleware is designed to be completely agnostic in terms of message payload as well. As long as the middleware is concerned payloads are simply byte arrays. Content negotiation explicitly addressed, since there's no universal way of representing this in every protocol. Clients can work around this by agreeing previously on which format to accept. Boundaries are also free to attach metadata to messages published in the middleware core and use them as appropriate within each protocol metadata capabilities.

4.4 Application Programming Interface

4.4.1 Middleware Core

Considering that the core of the middleware is a streaming communication platform between boundary peers, the API consist mostly of message production and consumption operations.

Boundaries that receive inbound connections from clients that want to enter some message into the system must supply two pieces of data when producing a message: the topic name, under which the message will be entered the system, the message payload (the actual message data) and the message id (a UUID that identifies the message to avoid it from being fed back into the system, as discussed in 4.1. Producing a message ensures the topic is set up to receive messages, so no prior setup should be necessary. The message id is attached as metadata. The message payload is an arbitrary byte array - the middleware does not make any assumptions about it's format, or schema, so it's up to clients to negotiate this.

Consuming a message is the inverse operation to producing. Those boundaries interested in consuming messages from the middleware can subscribe to topics either by specifying an exact topic name or a regex for catching various topics. As discussed in 4.1, overall, there's no guarantee that messages will arrive to consumers in the same order they were produced. Consumers receive the message topic, payload and id, the same way they were published.

Quality of service should be configured for the middleware core as whole. It should be able to guarantee basic QoS levels, ensuring that messages are delivered either at most once, at least once. Boundaries should be responsible for handling communication errors with clients and defining which strategy is the best to treat each case, since QoS may vary drastically from one protocol to another.

4.4.2 Boundaries

Boundaries are where the really interesting middleware logic lives. They are completely decoupled from each other and can offer whatever API fits best for their clients

to interact with the middleware. This means that there could be a multitude of different boundaries that interact with the same protocol, offering different kinds of input/output formats and communication models.

Figure 7 depicts a sequence diagram exemplifying a complex communication scenario. Two MQTT clients are connected to an MQTT broker which is connected to an MQTT Subscriber boundary. On the other side, a CoAP device subscribes to topic updates via a WebHook subscription with the CoAP Client boundary and a CoAP enabled application issues an observing request to the CoAP Observe Boundary, registering its interested on updates from the same topic. The MQTT IoT device publishes a message to the broker which delivers it directly to the MQTT enabled app and the MQTT Subscriber boundary, which produces a message to the Middleware Core, as a result. That message is consumed by the CoAP boundaries and clients are notified about it accordingly: either via CoAP resource representation update inside the context of an observing request, or through a `POST` request to clients subscribed via WebHook.

5 METHODOLOGY

For the sake of proving the feasibility of such architecture, a prototype implementation was developed. The middleware core has been implemented using the aforementioned Apache Kafka broker. 6 boundaries were implemented as services using the Clojure programming language [5], running on top of the Java Virtual Machine:

- **MQTT Subscriber:** a service that connects to an MQTT broker and listens for published messages; for every message that enters the MQTT broker, a corresponding message is produced into the middleware core.
- **MQTT Publisher:** a service that consumes all messages from the middleware core and publishes corresponding messages to the MQTT broker it's connected to.
- **CoAP Server:** a CoAP service that accepts `POST` requests from CoAP clients, containing the message topic (as an URL query parameter) and the message

[5] `https://clojure.org`

payload in the request body. Every incoming requests turns into a message produced into the middleware core.

- **HTTP Server:** an HTTP service, with a behavior similar to the CoAP server;
- **CoAP Client:** besides the misleading name, it is a boundary that implements a CoAP service that accepts subscriptions to topics. Clients can issue `POST` requests indicating which topics they are interested in along with an URL where they want to be notified about new messages. The boundary consumes all messages from the middleware core and issues `POST` corresponding requests to subscribed clients.
- **HTTP Client:** behaves similarly to the CoAP client, except it interacts with clients through HTTP.

Aiming to fulfill this work's objectives, two types of tests were performed. A qualitative test where, with all 6 boundaries running messages were produced by clients through the 3 boundaries and arrived and successfully arrived at clients listening through the 3 output boundaries. The middleware core broker and boundaries were run on a laptop; sending and receiving clients were run on a second laptop. Both machines were connected to and performed all communication through a Wireless Local Area Network.

The middleware was profiled by measuring the latency of each component (boundaries and core) according to table 3 and the measurements were analyzed according to table 4. Four translation scenarios were considered: $CoAP \rightarrow HTTP$, $HTTP \rightarrow CoAP\ MQTT \rightarrow CoAP\ MQTT \rightarrow HTTP$.

t_0	=	message leaves the sending client
t_1	=	message is being processed by the input boundary
t_2	=	message is being processed by the output boundary
t_3	=	message arrives at the receiving client

Table 3 – Test setup timing instrumentation

The same four translation scenarios were benchmarked against the Mosquitto MQTT broker throughput. Measurements were taking by publishing a sequential burst of messages with 60 seconds of duration and assessing the number of messages that the middleware translated and relayed to the single subscribing client.

T_0	=	$t_1 - t_0$
T_1	=	$t_2 - t_1$
T_2	=	$t_3 - t_2$

Table 4 – Timing calculations

6 RESULTS

The test fired 10000 messages in a serial way, which were consumed concurrently by clients. The worst case latency observed for a message to be transported from the sending client to the receiving end was around 630 milliseconds.

Table 5 shows the results of the profiling tests. Four different scenarios were run and 3 statistics calculated for each of them: the 99th percentile, the mean and the max value. 10000 messages were produced sequentially by the originating clients and consumed sequentially by destination clients.

	$CoAP \rightarrow HTTP$			$HTTP \rightarrow CoAP$			$MQTT \rightarrow CoAP$			$MQTT \rightarrow HTTP$		
	P99	Mean	Max	P99	Mean	Max	P99	Mean	Max	P99	Mean	Max
T_0	175	139	607	32	23	117	34	20	163	40	22	180
T_1	5	3	201	6	3	298	6	3	155	76	4	166
T_2	8	2	120	5	2	17	5	2	22	30	5	160

Table 5 – Profiling test results

Table 6 shows the results for the throughput measurements. Compared to the results obtained from benchmarking the MQTT broker alone, we can observe an approximate 6.3% throughput degradation in the translation from MQTT messages to CoAP requests, as well as a 6.8% degradation when translating MQTT messages to HTTP requests. When translating HTTP requests to CoAP requests, the degradation in throughput is more noticeable, getting up to around 65.5%. It's even worse when we consider CoAP to HTTP translation, where throughput can be more than 8 times smaller. These results are biased by implementation details of the middleware boundaries, since the HTTP and CoAP components are not taking full advantage of async processing - which can help with highly I/O blocking operations.

$MQTT \rightarrow MQTT$	$CoAP \rightarrow HTTP$	$HTTP \rightarrow CoAP$	$MQTT \rightarrow CoAP$	$MQTT \rightarrow HTTP$
53.51 msg/s	6.4 msg/s	35.08 msg/g	50.14 msg/s	49.89 msg/s

Table 6 – Throughput benchmark results

7 CONCLUSION

This work has presented an extensible middleware solution for interoperability in the Internet of Things. Compared to related works, it differs in that it is designed as a message-oriented middleware, having a completely asynchronous nature, it allows for unlimited extensibility, by decoupling protocol syntactic and semantic translation from the common representation, streaming and storage of messages. Developer of IoT applications can benefit from such design in many ways: freeing themselves to use IoT devices that implement different communication protocols without worrying about protocol specific logic inside their applications, creating data drive applications that consume messages directly from the middleware core, processing and mapping streams of messages to transform the middleware's input into output, among others.

To evaluate the viability of such solution, a prototype implementation was developed and tested. The prototype was able to bridge input and output communication between 3 protocols commonly used by Internet of Things applications and devices (CoAP, MQTT and HTTP). Benchmark tests were made to profile middleware components considering different communication scenarios.

The results obtained from the prototype implementation show that such model is viable and, in some scenarios, does not add significant overhead compared to direct communication between clients using the same protocols - given similar network latency conditions. There are, however, plenty of limitations that can be explored in future works. These include: addressing security concerns at the middleware level, evaluating middleware deploy strategies in diverse, real world scenarios as well as evaluating the middleware elasticity capabilities when deployed in a Cloud Computing platform, performing more reliable stress tests.

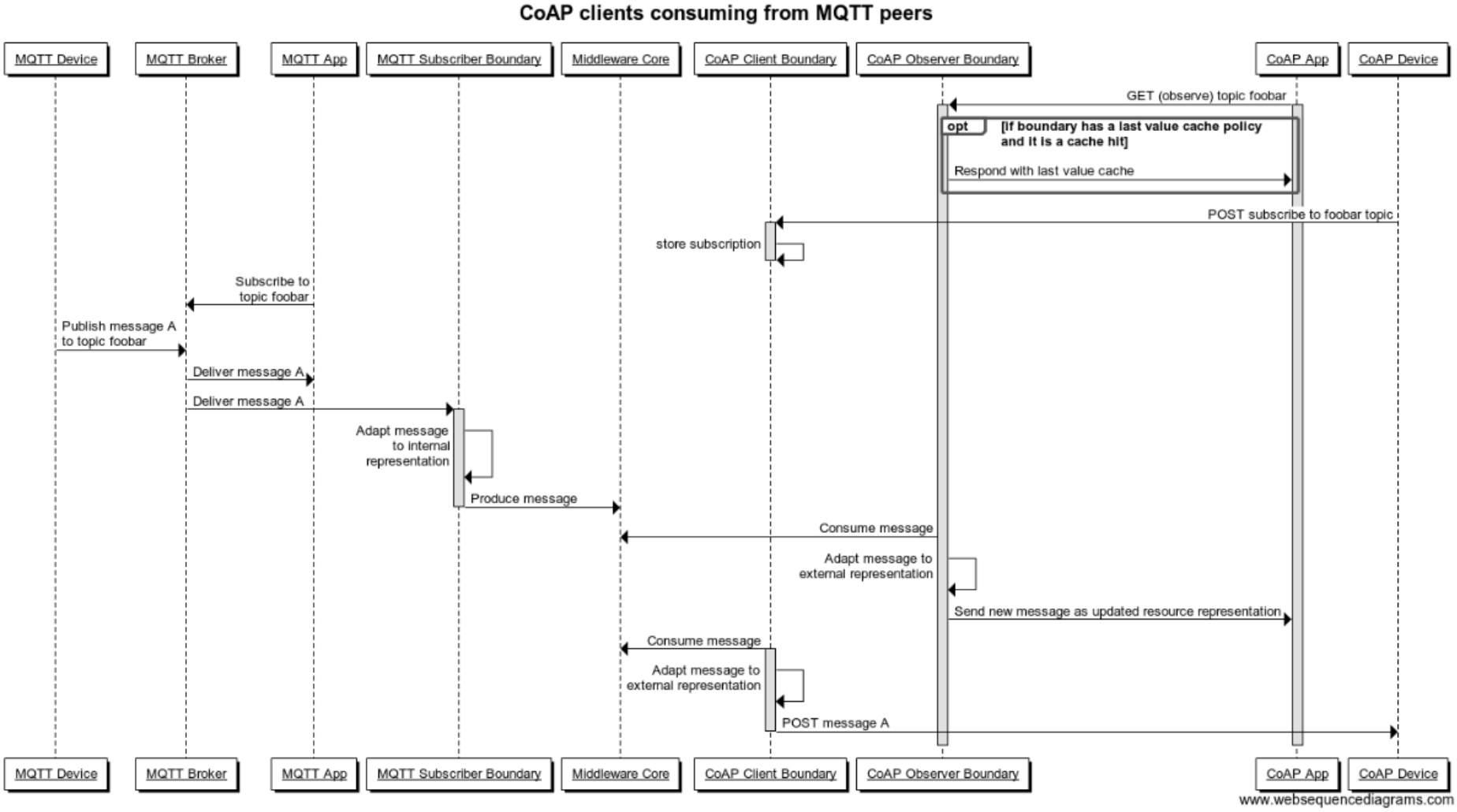

Figure 7 – Sequence diagram of message published by an MQTT client being delivered to other MQTT clients as well as to CoAP clients

REFERENCES

AAZAM, M. et al. Cloud of things: integrating internet of things and cloud computing and the issues involved. In: APPLIED SCIENCES AND TECHNOLOGY (IBCAST), 2014 11TH INTERNATIONAL BHURBAN CONFERENCE ON, 2014. **Anais...** 2014. p. 414–419.

AL-FUQAHA, A. et al. Internet of things: a survey on enabling technologies, protocols, and applications. **IEEE Communications Surveys Tutorials**, v. 17, n. 4, p. 2347–2376, Fourthquarter 2015.

ALARCON, R. et al. Rest web service description for graph-based service discovery. In: INTERNATIONAL CONFERENCE ON WEB ENGINEERING, 2015. **Anais...** 2015. p. 461–478.

BANKS, A.; GUPTA, R. Mqtt version 3.1. 1. **OASIS standard**, 2014.

BOMAN, J.; TAYLOR, J.; NGU, A. H. Flexible iot middleware for integration of things and applications. In: IEEE INTERNATIONAL CONFERENCE ON COLLABORATIVE COMPUTING: NETWORKING, APPLICATIONS AND WORKSHARING, 10., 2014. **Anais...** 2014. p. 481–488.

BROMBERG, Y. D.; GRACE, P.; RéVEILLèRE, L. Starlink: runtime interoperability between heterogeneous middleware protocols. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, 2011., 2011. **Anais...** 2011. p. 446–455.

BROMBERG, Y.-D.; ISSARNY, V. Indiss: interoperable discovery system for networked services. In: ACM/IFIP/USENIX 2005 INTERNATIONAL CONFERENCE ON MIDDLEWARE, 2005, New York, NY, USA. **Proceedings...** Springer-Verlag New York: Inc., 2005. p. 164–183. (Middleware '05).

CHAKRABARTI, A. **Emerging open and standard protocol stack for iot**. (Accessed on 06/23/2017), https://www.linkedin.com/pulse/emerging-open-standard-protocol-stack-iot-aniruddha-

CHANDRA, A. et al. A study of climate-smart farming practices and climate-resiliency field schools in mindanao, the philippines. **World Development**, v. 98, n. Supplement C, p. 214 – 230, 2017.

COLLINA, M.; CORAZZA, G. E.; VANELLI-CORALLI, A. Introducing the qest broker: scaling the iot by bridging mqtt and rest. In: IEEE 23RD INTERNATIONAL SYMPOSIUM ON PERSONAL, INDOOR AND MOBILE RADIO COMMUNICATIONS - (PIMRC), 2012., 2012. **Anais...** 2012. p. 36–41.

DERHAMY, H.; ELIASSON, J.; DELSING, J. Iot interoperability - on-demand and low latency transparent multi-protocol translator. **IEEE Internet of Things Journal**, v. PP, n. 99, p. 1–1, 2017.

DESAI, P.; SHETH, A.; ANANTHARAM, P. Semantic gateway as a service architecture for iot interoperability. In: IEEE INTERNATIONAL CONFERENCE ON MOBILE SERVICES, 2015., 2015. **Anais...** 2015. p. 313–319.

ECLIPSE newsletter: mqtt and coap, iot protocols. (Accessed on 06/23/2017), https://eclipse.org/community/eclipse$_n ewsletter/2014/february/article2.php$.

GLOBAL sensor networks - a middleware for processing sensor data in the internet. (Accessed on 06/21/2017), http://lsir.epfl.ch/research/current/gsn/.

GOOGLE fit. (Accessed on 06/21/2017), https://www.google.com/fit/.

HERBST, N. R.; KOUNEV, S.; REUSSNER, R. H. Elasticity in cloud computing: what it is, and what it is not. In: ICAC, 2013. **Anais...** 2013. p. 23–27.

JAYARAMAN, P. P. et al. Internet of things platform for smart farming: experiences and lessons learnt. **Sensors**, v. 16, n. 11, 2016.

JIMENEZ, J. et al. **A constrained application protocol (coap) usage for resource location and discovery (reload)**. RFC Editor, 2015. RFC. (7650).

KATOLE, B. et al. The integrated middleware framework for heterogeneous internet of things (iot). **intelligence**, v. 4, n. 7, 2015.

LEE, C. H. et al. Interoperability enhancement for internet of things protocols based on software-defined network. In: IEEE 5TH GLOBAL CONFERENCE ON CONSUMER ELECTRONICS, 2016., 2016. **Anais...** 2016. p. 1–2.

LINKSMART® middleware platform portal. (Accessed on 06/21/2017), https://linksmart.eu/redmine.

NAKAZAWA jin et al. a bridging framework for universal interoperability in pervasive systems. In: 26TH IEEE INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS (ICDCS'06), 2006. **Anais...** 2006. p. 3–3.

NEST. (Accessed on 06/21/2017), https://nest.com/.

NGU, A. H. et al. Iot middleware: a survey on issues and enabling technologies. **IEEE Internet of Things Journal**, v. 4, n. 1, p. 1–20, Feb 2017.

NODE-RED. (Accessed on 06/21/2017), http://nodered.org/.

PERSSON, P.; ANGELSMARK, O. Calvin – merging cloud and iot. **Procedia Computer Science**, v. 52, p. 210 – 217, 2015. The 6th International Conference on Ambient Systems, Networks and Technologies (ANT-2015), the 5th International Conference on Sustainable Energy Information Technology (SEIT-2015).

PONTE. (Accessed on 06/20/2017), https://www.eclipse.org/ponte/.

PTOLEMY iot - google search. (Accessed on 05/14/2017), https://ptolemy.eecs.berkeley.edu/.

RESCORLA, E. **Http over tls**. RFC Editor, 2000. RFC, http://www.rfc-editor.org/rfc/rfc2818.txt. (2818).

RFC 4122 - a universally unique identifier (uuid) urn namespace. (Accessed on 11/24/2017), https://tools.ietf.org/html/rfc4122.

ROMAN, M.; KON, F.; CAMPBELL, R. H. Design and implementation of runtime reflection in communication middleware: the dynamictao case. In: IEEE INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS. WORKSHOPS ON ELECTRONIC COMMERCE AND WEB-BASED APPLICATIONS. MIDDLEWARE, 19., 1999. **Proceedings...** 1999. p. 122–127.

SHELBY, Z.; HARTKE, K.; BORMANN, C. **The constrained application protocol (coap)**. RFC Editor, 2014. RFC, `http://www.rfc-editor.org/rfc/rfc7252.txt`. (7252).

THANGAVEL, D. et al. Performance evaluation of mqtt and coap via a common middleware. In: IEEE NINTH INTERNATIONAL CONFERENCE ON INTELLIGENT SENSORS, SENSOR NETWORKS AND INFORMATION PROCESSING (ISSNIP), 2014., 2014. **Anais...** 2014. p. 1–6.

TRIFA, V. et al. Web messaging for open and scalable distributed sensing applications. In: INTERNATIONAL CONFERENCE ON WEB ENGINEERING, 2010. **Anais...** 2010. p. 129–143.

XIA, F. et al. Internet of things. **International Journal of Communication Systems**, v. 25, n. 9, p. 1101, 2012.

XIVELY. (Accessed on 06/21/2017), `https://www.xively.com/`.