Politecnico Di Milano

School of Industrial and Information Engineering

Master of Science in Computer Science and Engineering

POLITECNICO
MILANO 1863

MQTT+:
Enhanced Syntax and Broker Functionalities for Data Filtering, Processing and Aggregation

Supervisor: Alessandro Enrico Cesare Redondi
Co-supervisor: Matteo Cesana

Master Thesis of:
Riccardo Giambona
Student ID: 863204

Academic Year: 2017/2018

Acknowledgments

I want to thank my supervisor Alessandro Enrico Cesare Redondi and my co-supervisor Matteo Cesana for helping me writing this thesis.
I want also to thank dc-square GmbH, the company behind HiveMQ, for the license they gave us to allow us to compute the performance tests.

Riccardo Giambona

Index Of Contents

Index Of Figures

Index Of Tables

Abstract

In the last few years, the Message Queueing Telemetry Transport (MQTT) publish/subscribe protocol emerged as the de facto standard communication protocol for IoT, M2M and wireless sensor networks applications. Such popularity is mainly due to the extreme simplicity of the protocol at the client side, appropriate for low-cost and resource-constrained client devices. Other nice features include a very low protocol overhead, ideal for limited bandwidth scenarios, the support of different Quality of Services (QoS) and many others. However, when a client device is interested in performing processing operations over the data published by multiple sensors, the use of MQTT may result in high network bandwidth usage and high energy consumption for the end devices, which is unacceptable in resource constrained scenarios.
To overcome these issues, we propose in this thesis MQTT+, which provides an enhanced protocol syntax and enrich the pub/sub broker with data filtering, processing and aggregation functionalities.
MQTT+ is implemented starting from an extensible MQTT broker and evaluated in different application scenarios.

Sommario

Negli ultimi anni, il protocollo di tipo publish/subscribe MQTT (Message Queueing Telemetry Transport) è diventato lo standard di fatto per quanto riguarda i protocolli di comunicazione usati per IOT, M2M e reti di sensori wireless. Questa popolarità è dovuta all'estrema semplicità del protocollo lato client e questo lo rende appropriato per dispositivi low-cost e con poche risorse computazionali a disposizione.
Altre interessanti funzionalità del protocollo, includono un bassissimo overhead di gestione, ideale per casi in cui la banda di rete sia limitata, il supporto di diverse QoS (Quality Of Services) e molte altre.
Nonostante ciò, quando un client è interessato ad eseguire delle operazioni di elaborazione dei dati pubblicati da diversi sensori, l'utilizzo di MQTT può comportare un alto consumo di banda di rete e un elevato consumo di energia da parte dei client, il che è inaccettabile in scenari in cui i clients siano dispositivi con poche risorse computazionali.
Per superare questi limiti, in questa tesi viene proposto MQTT+, che fornisce un protocollo con una sintassi avanzata e introduce nel broker delle funzionalità di filtraggio, analisi e aggregazione di dati.
MQTT+ è implementato a partire da un broker estendibile e valutato in diversi scenari applicativi.

Chapter 1: Introduction

The Internet of Things (IoT) is day by day becoming a reality. Tiny and cheap devices equipped with sensors and wireless communication capabilities are being used more and more frequently in several application scenarios, such as wireless sensor networks, environmental monitoring, e-health, etc.
Regardless of the specific scenario, all IoT applications are characterised by common requirements: Sensor nodes operate with low-bandwidth wireless transceivers to transmit/receive data to/from a common data concentrator (sink node) or other IoT nodes. Such data may be then processed, according to the application's needs, either at the sink node or onboard other sensor nodes using low-power and energy-efficient microcontrollers. Such a resource-constrained environment stimulated in the last few years a vast body of research to design and optimise existing protocols at all layers of the communication stack. For what concerns the application layer, several efforts have been performed:
Protocols such as the Message Queue Telemetry Protocol (MQTT)[1], the Constrained Application Protocol (COAP)[2] and the Extensible Messaging and Presence Protocol (XMPP)[11] are the results of such efforts. Among the existing solutions, MQTT is certainly the one that has received the greatest attention in the last few years, practically becoming the standard de-facto in M2M and IoT applications. As a matter of fact, MQTT is becoming the most popular protocol to connect resource constrained devices to the major cloud platforms (e.g., Amazon AWS, Microsoft Azure, IBMWatson), which all expose their services through MQTT. The reasons of such popularity derive from MQTT's incredible simplicity client-side, which nicely fits in resource-constrained applications, yet supporting reliability and several degrees of quality of service (QoS). MQTT is based on the publish/subscribe pattern, and all communications between nodes are made available by a broker. The broker accepts messages published by devices and forwards them to clients subscribed to those messages, ultimately controlling all aspects of communication between devices.
There is however a set of common IoT and M2M applications scenarios where the use of MQTT causes an inefficient use of the available network and computing resources. Those are all cases where data consumers (subscribers) are interested in only a subset of the data produced (published) by sensor devices, while the broker still forwards the entire data available. Examples include clients interested in receiving data only if it respects some condition, clients interested in certain aggregation functions (e.g., cumulative sum, average) over a set of data published, or clients interested in the result of some processing task over such data, rather than the data itself. In all these cases, two main drawbacks can be identified: first, the data forwarded by the broker may potentially be discarded by subscribers, wasting network resources and (ii) subscribers need to perform additional processing operations, consequently decreasing their available computational and energy resources. To mitigate those issues, we propose in this paper an advanced MQTT broker (MQTT+) able to deal with such situations.
MQTT+ allows a client to subscribe to advanced functionalities on the data published, including rule-based data filtering, spatial and temporal data aggregation and data processing. All functionalities are provided reusing as much as possible the original MQTT protocol logical and syntactical rules and minimally modifying the client-side procedures. As a proof of concept, MQTT+ is implemented starting from a publicly available MQTT broker (HiveMQ) and evaluated in different application scenarios.

This thesis is structured as follows: Chapter 2 briefly discuss the MQTT protocol, highlighting its main features. Chapter 3 analyses the existing modified solutions of MQTT. Chapter 4 introduces MQTT+ logic and the proposed enhancements. Chapter 5 explains how this project has been implemented. Chapter 6 shows the results of the performance tests between MQTT and MQTT+ and finally, Chapter 7 talks about the future improvements that will be applied to this project.

Chapter 2: MQTT Overview

2.1 - Introduction

The Message Queuing Telemetry Transport is a lightweight publish/subscribe protocol whose design principles are to minimise both the end-devices requirements and the utilised network resources, still ensuring reliability and some degree of quality of service.
MQTT follows a traditional publish/subscribe pattern in which a client device publishes information relative to a particular topic, i.e., a multilevel string describing the data being published (e.g. kitchen/temp).
Other clients interested in such information subscribe to that topic. Information forwarding from the publishers to the subscribers is made possible by a broker, which is the core part of the system and is in charge of receiving data from the publishers and forwarding it to the subscribers.
Such design allows to decouple the publishing and subscribing processes: clients interested in a particular topic do not need to know who the publishers are, neither they have to be synchronised to the publishing operations.

2.2 - Usage

In this section, we are going to see the basics steps to use the MQTT protocol.

2.2.1 - Step one: Connect

Before being able to publish data or subscribe to any topic, each client needs to connect to a broker. Such connection is based on TCP/IP and implemented through a simple message exchange between the client and the broker. [15]

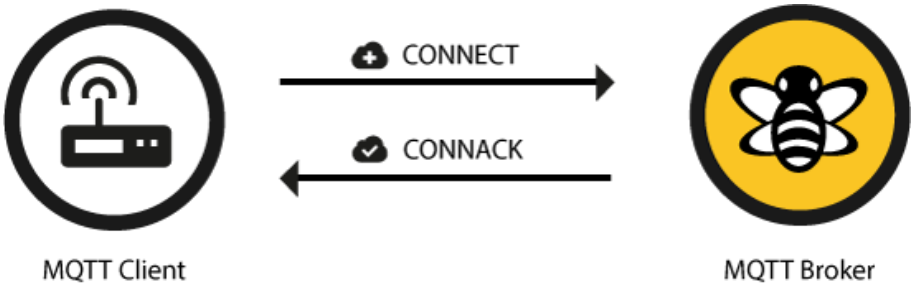


Figure 1: Connect-Flow (©dc-square GmbH HiveMQ MQTT Essentials)

During this process, a client communicates several information to the broker such as its client identifier, the connection keep alive time interval,the clean Session parameter and other optional parameters (authentication, last will topic and message, etc).
All this information is contained in the CONNECT message as shown below:

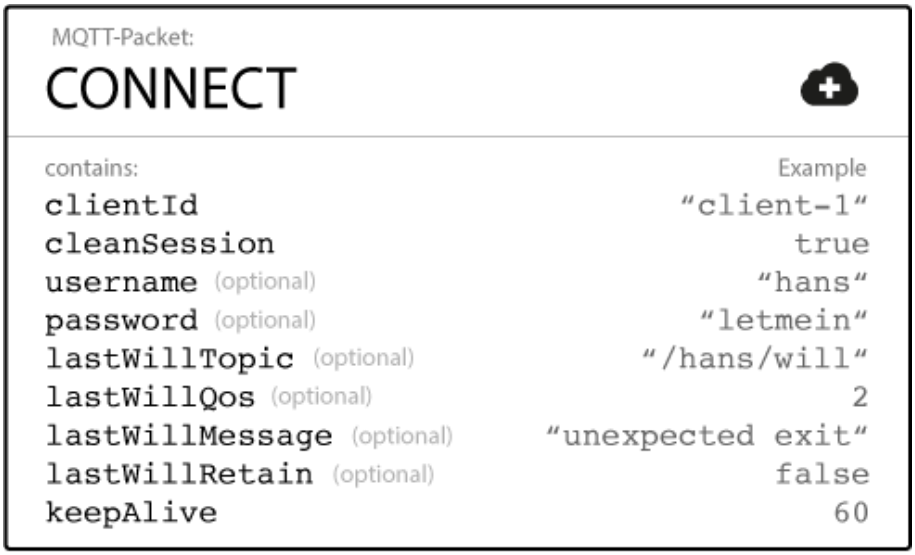


Figure 2: Connect-Message (©dc-square GmbH-HiveMQ-MQTT Essentials)

This CONNECT message is confirmed by a CONNACK message that indicates if the connection succeeded or not, specifying also the reason of the eventual failure.

Referring to the CONNECT message structure, it's important to understand what the 3 required attributes are:

- **ClientID**: Represents the name given to the client. It must be unique in the set of all the connected clients, since it needs to identify the single client that is connected to the broker. It can be set by the user, or generated randomly by the client library

- **CleanSession:** If it's set to true, then the session is clean, which means that after a client disconnects or loses the connection, all the subscriptions that it made during the session are lost and in case of reconnection it has to subscribe again to all topics. Instead if it's set to false, the session is persistent and in case of client disconnection, the client is still subscribed to all the subscriptions it made during the session.

- **KeepAlive:** This is the maximum interval of time (in seconds) in which a client can send no messages. If a client doesn't send any message after this period of time, the connection is automatically closed by the broker and therefore the client is disconnected.

2.2.2 - Step two: Publish/Subscribe

After the connection a client may directly start publishing data or subscribing to a certain topic using specific MQTT messages with minimal transport overhead (the fixed-length header is just 2 bytes).

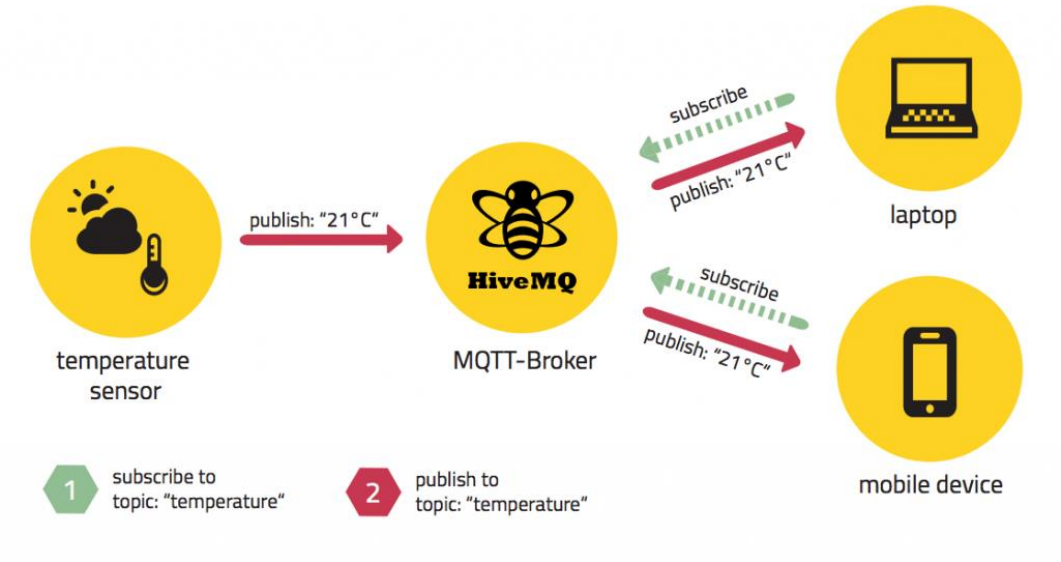


Figure 3: Publish-Subscribe-Flow (©dc-square GmbH-HiveMQ-MQTT Essentials)

In this example, we can see a single temperature sensor that publishes its detection, using the topic "temperature" and two different clients that subscribe to the same topic that receive the message forwarded by the broker.
It's important to know that the publish and subscription topics don't need to be initialized, which means that we don't have first to register the topic "temperature" to use it, but we can directly publish to that topic without any prior initialization. [16]

2.3 - QoS (Quality of Service)

For both publish and subscribe, clients have the possibility of choosing a Quality of Service (QoS) value, which impact on the way the broker handles the messages from/to the clients.

Three QoS levels are defined:

1. At most once (fire-and-forget), which relies on the underlying TCP connection

Figure 4:QoS (0) (©dc-square GmbH-HiveMQ-MQTT Essentials)

2. At least once, where the sender will retransmit a message until an ACK is received

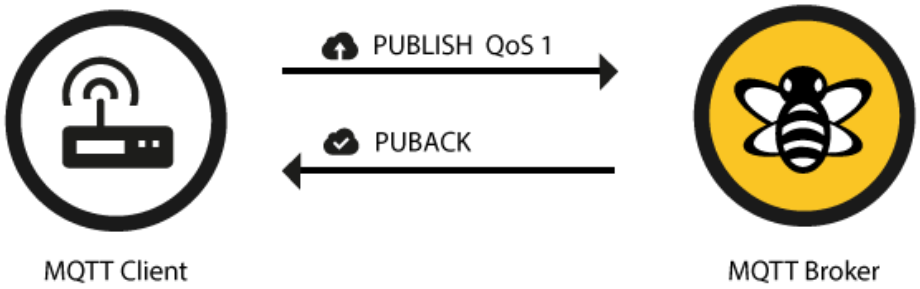


Figure 5: QoS (1) (©dc-square GmbH-HiveMQ-MQTT Essentials)

3. Exactly once, where it is guaranteed that a transmitted message is received only once by the counterpart.

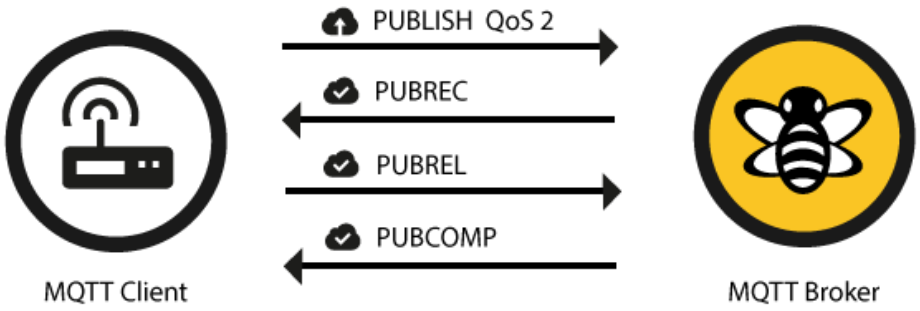


Figure 6: QoS (2) (©dc-square GmbH-HiveMQ-MQTT Essentials)

Of course, we can see from the flows that the higher the QoS, the higher will be the network traffic, since the process requires more acks for each single message sent. Also, the delay will be higher, since we have to wait all acks.

Note that the quality of service is not something defined globally, but it is defined for the messages sent from the broker to the subscribers and for the messages from the publishers to the broker.
Please note that the QoS for the messages forwarded by the broker to the subscribers it is equal to the QoS specified by the subscribers in their subscription message for each topic.
The message sent at the moment of subscription is structured as shown below:

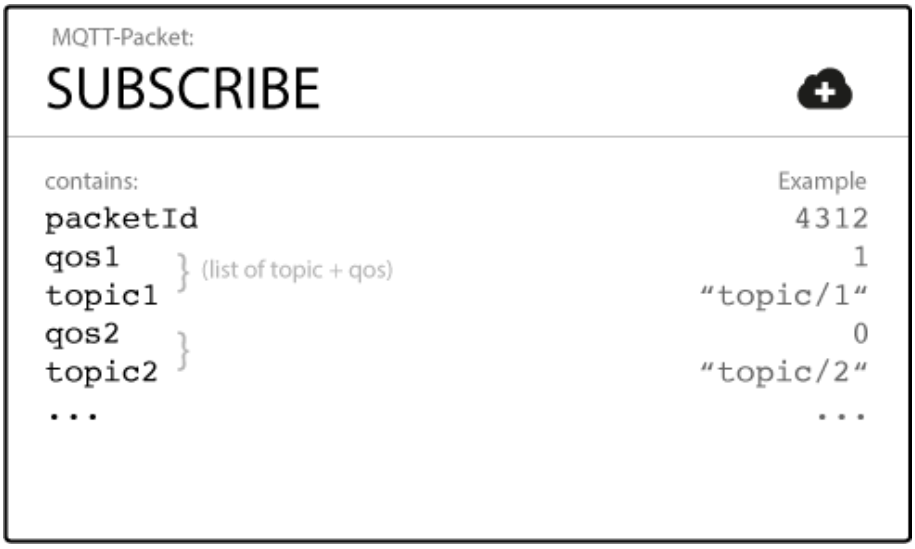


Figure 7: Subscribe message (©dc-square GmbH-HiveMQ-MQTT Essentials)

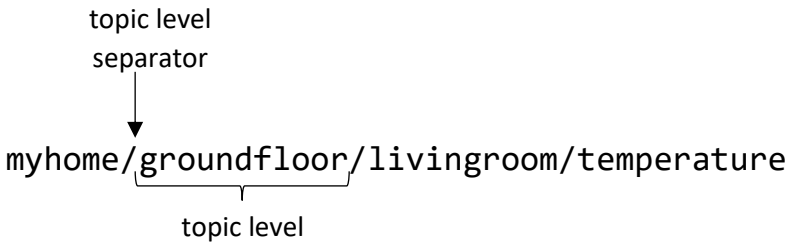
Instead, the QoS of the messages published by the sensors can be different from the QoS specified in the subscription process. For example, I can have QoS 0 for messages from publishers to broker and QoS 2 for messages from broker to subscribers (on certain topics). [17]

2.4 – Syntax

In this paragraph we are going to analyse the syntax that can be used to specify the topics for publishing or subscribing. [18]

2.4.1 – Topic

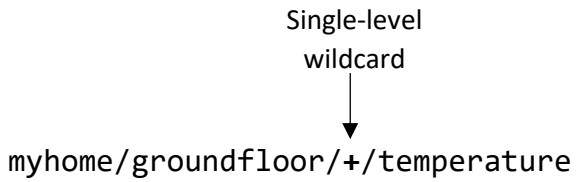
A topic is just a string separated by slash separators and each block is called level



One way to specify a topic is like it's shown in the example above, but what happens if we want to subscribe to several publishing topics with just one subscription? This is what wildcards are for.

2.4.2 – Single level wildcard (+)

This wildcard is also called single-level wildcard because it substitutes all the possible values of a single level with the + symbol. Let's clarify with an example:

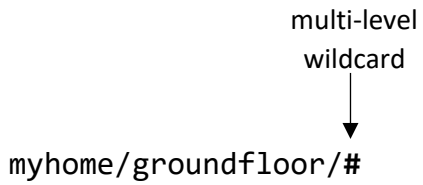



```
 [M] myhome/groundfloor/livingroom/temperature
 [M] myhome/groundfloor/kitchen/temperature
[NM] myhome/groundfloor/kitchen/brigthness
[NM] myhome/firstfloor/kitchen/temperature
[NM] myhome/groundfloor/kitchen/fridge/temperature
```

Using the + wildcard, if we subscribe to the topic: `myhome/groundfloor/+/temperature`
we will receive all the messages published with the topics marked with the symbol [M] (Matching), so basically in the level with the + we could put any possible value and the publish topic would match anyway with the subscription topic.
Of course the +, as said before, it's a single level wildcard, so the levels different of + must be specified exactly as the publish topic levels, this is why in the cases marked with the [NM] (Not matching) symbol, the subscription topic won't match the publish topic.

2.4.3 – Multi-level wildcard (#)

In this case the # is a multi-level wildcard, because it substitutes not only one single level of the topic, but it substitutes a variable number of levels starting from the one where the # is until the end of the topic. Please note that this wildcard can be put only as the last character of the topic and not in any position like the + wildcard. Let's clarify with an example:




```
 [M] myhome/groundfloor/livingroom/temperature
 [M] myhome/groundfloor/kitchen/temperature
 [M] myhome/groundfloor/kitchen/brightness
[NM] myhome/firstfloor/kitchen/temperature
```

In this case we can see that the subscription topic must be equal to the publish one, only for the first two levels (`myhome/groundfloor`), but the other levels can be any other thing and the subscription topic would match anyway the publish topic.

Chapter 3: State of art

3.1 – Related works research

In the last ten years, many research studies have proposed modifications and enhancements to the MQTT protocol. One of the most popular work is the one from Hunkeler et. al which propose MQTTSN[6], a version of MQTT focused particularly on constrained wireless sensor networks.
MQTT-S do not require clients to connect to the broker through a TCP/IP connection, therefore greatly simplifying their design. Other interesting features of MQTT-SN are the possibility of using an encoded format for publishing and subscribing topics (so as to save bandwidth) and the support for clients working according to a duty cycle.
Other solutions have been proposed that tackle different weaknesses of MQTT: the work in [8] tackles client mobility using memory buffers on publishers; in [12] a lightweight encryption technique based on Elliptic Curve Cryptography is proposed to increase the security of both MQTT and MQTT-SN protocols; in [5], authors analyse the end-to-end reliability of MQTT-SN considering several system parameters.
Two very recent works show contact points with what proposed in this paper: the work in [3], authors propose MQTT-CV (MQTT for communicating vehicles), in which vehicles publish sensor data and a control infrastructure is subscribed to such data. The main difference compared to MQTT is that the broker may accept some rule from the control infrastructure (e.g., forward only vehicle speed data greater or lower than a threshold). This is similar to the rule-based subscription available in the proposed MQTT+, although no details are given on how such rule-based subscriptions can be integrated in the MQTT syntax. Finally, the work in [8] proposes MQTT-NEG (Near-user Edge Gateway), a broker implementation that is able to interconnect different groups of sensors (i.e., content islands) and manage the published messages either locally (within each content island) or globally (distributing messages among different islands).
A more general body of research focuses on enhancing the capabilities of publish/subscribe systems. Li and Jacobsen propose PADRES, a pub/sub system which allows expressive and composite subscriptions tailored to the world of workflow management and business process execution. PADRES allow a subscriber to be notified when particular events (jobs in a workflow) happen in parallel, or in sequence, or repeat periodically.
On the same line, Demers et al. propose Cayuga [4] a pub/sub system allowing a user to express subscriptions spanning multiple events and supporting aggregation and parametrisation of subscriptions. The system is based on a non-deterministic finite automata and an event algebra which provides expressiveness and maps to the state of the automata.
Other recent works relative to aggregation of data in generic pub/sub systems are the ones from Pandey et al. In [9] and in [10] the authors propose a solution to aggregate data in a distributed way, among several brokers, together with an optimisation problem to minimise the communication cost of such distributed aggregation.

3.2 – Available standard MQTT brokers/platforms

After we made a research of the already existing solutions in the market, for improving the standard MQTT, the next research to do was to identify which broker, of the many available, to modify and extend with the MQTT+ logic and at the same time preserve the backward compatibility with the standard MQTT. From this research we listed all the found broker implementations and for each one of them we analysed its characteristics.

The criteria to evaluate each implementation and its characteristics are the following:

Language	C#, Java => Suitable	C => Not very suitable	Others => Not suitable
Documentation	Well documented	Poorly documented	No documentation
Extensibility	Easily extensible	Not easily extensible	Not extensible

Table 1: Implementations evaluation criteria

In the following tables we are going to list all the analysed implementations.

Name	Languages	Documentation	Extensibility
MQTTnet [Not very suitable]	Suitable	Poorly documented	Easily extensible
IBM Websphere MQ Telemetry [Not suitable]	Suitable	-	Not extensible
IBM IoT MessageSight [Not suitable]	Suitable	-	Not extensible
IBM Integration Bus Advanced [Not suitable]	Suitable	-	Not extensible
Really small message broker [Not very suitable]	See Mosquitto	See Mosquitto	See Mosquitto
Note: This project was made open-source thanks to the project Mosquitto, so we can refer to that project for the specifications.			
Moquette [Not very suitable]	Suitable	No documentation	Easily extensible
Emitter [Not suitable]	Not suitable	Well documented	Easily extensible
Eurotech Everywhere Device Cloud [Not suitable]	-	Well documented	Not extensible
Emqttd [Not suitable]	Not suitable	Well documented	Easily extensible
Xively [Not suitable]	-	Well documented	Not suitable

Table 2: Implementations list - Part 1

Name	Languages	Documentation	Extensibility
Yunba.io [Not suitable]	-	-	-
RabbitMQ [Not suitable]	Not suitable	Well documented	Easily extensible
Apache ActiveMQ [Suitable]	Suitable	Well documented	Easily extensible
NOTE: From this project the following two were born: **-ActiveMQ Apollo:** they say to be the improved version of ActiveMQ, but the documentation isn't very detailed, probably this is due to the fact that the project is very young -**Apache ActiveMQ Artemis:** Not blocking version, asynchronous version			
Mosquitto [Not very suitable]	Not very suitable	Well documented	Easily extensible
HiveMQ [Suitable]	Suitable	Well documented	Easily extensible
Mosca [Not suitable]	Not suitable	No documentation	Easily extensible
Litmus Automation Loo [Not suitable]	-	-	Not extensible
Solace Message Routers [Not suitable]	-	-	Not extensible
JoramMQ [Not suitable]	Suitable	-	Not extensible
VerneMQ [Not suitable]	Not suitable	-	
HBMQTT [Not suitable]	Not suitable	No documentation	
ThingScale IoT message broker [Not suitable]	-	-	
vertx-mqtt-broker [Not very suitable]	Suitable	Well documented	

Table 3: Implementations list - Part 2

3.3 – Final MQTT broker implementation

From the previous analysis we can see that the best two implementations for our purposes are the ActiveMQ versions and the HiveMQ one.
We decided to use the HiveMQ because, differently from ActiveMQ, it's not required to take the source code, study how that specific broker was implemented and understand where to put our modified logic to extend it to MQTT+, but everything works with events raised by the broker and that you can intercept registering the events and your modified code will run when those events are raised.
This simplifies a lot the work to do and allows you to focus only on the modifications you need to introduce on the broker, not worrying at all about internal broker logic.
So in the next sections and especially in the implementation sections we will refer to HiveMQ[14] broker.

Chapter 4 – MQTT+ Logic

4.1 - Introduction

In this chapter we are going to describe the work that has been done to improve the MQTT syntax and expressive power, compared to the actual state described before.
Also, as said before, what is true for the standard MQTT is also valid for the MQTT+.
The introduced changes are just additions to the MQTT functionalities, leaving untouched the standard logic. For this reason, in our project there are two types of topics:

- **Standard topics**: Topics that follow the standard MQTT syntax, which are managed by the already existing HiveMQ logic.

- **Special topics:** Topics that respect the MQTT+ syntax, which are managed by the added logic by this project.

This thing allows to have sessions where some subscribers use the MQTT syntax and other the MQTT+ syntax in the same session of the broker and this won't cause any kind of problem, allowing perfect backward compatibility.

4.2 – MQTT+ Syntax

First of all, it's important to note that the topic syntax used by the publishers is left untouched compared to the standard MQTT, because the new functionalities are used by the subscribers to ask some kind of elaboration on the data sent by the publishers to the broker.
For this reason, in this modified syntax, we will implicitly refer to the topics used by the subscribers only.

4.2.1 – Multiple value separator (;)

As explained in the previous sections, the MQTT allows to specify the + and the # wildcard, but what If we want to specify that we want to subscribe to only a subset of the possible values that a topic level can have?

In this case the # wildcard is not what we need because it includes multiple levels and the + wildcard neither, because it includes all possible values that a topic level can have.
That's why we introduced the possibility to specify a topic with the values separated by the ; symbol to list only the values that we want to consider. The following topic example will clarify the usage.

```
temperature/polimi/deib/room1/sensor1;sensor2;sensor3
```

```
 [M] temperature/polimi/deib/room1/sensor1
 [M] temperature/polimi/deib/room1/sensor2
 [M] temperature/polimi/deib/room1/sensor3
[NM] temperature/polimi/deib/room1/sensor4
```

As we can see in the example above, each one of the topic with the last level value in the specified set (sensor1, sensor2 and sensor3) is accepted, otherwise it's not.

4.2.2 – Special subscription topic syntax

The special subscription topics are in the following form:

```
$OperationsBlock/MatchingPart
```

First of all, we can notice that these topics must start with the special character "$".
This character is used by the broker to distinguish special topics from standard ones.

The `$OperationBlocks` must always be specified and can't be a wildcard (+ or #).
This is done because the broker needs to know the sequence of operations that needs to be computed on the published data.
The blocks in the `MatchingPart` can be anything that respects the MQTT standard (+ and # included).
Also each block can be a single value (e.g "sensor1") or a list of values ("sensor1;sensor2") as explained in the section before.
The MQTT matching rules apply only with the `MatchingPart` and not on all the subscription topic (`$OperationBlocks/MatchingPart`). The following example will clarify the concept:

Suppose we have the following publish topic:

```
temperature/polimi/deib/room1/sensor1
```

The following are some subscriptions with the indication if they match with the publish topic or not:

```
     OperationsBlock              MatchingPart
     ┌──────────┐ ┌──────────────────────────────────┐
 [M] $AVG$DAILYAVG/temperature/polimi/deib/room1/sensor1
 [M] $SUM$DAILYAVG/temperature/polimi/deib/#
[NM] $AVG$DAILYAVG/temperature/polimi/room2/sensor3
```

4.2.2.1 – OperationsBlock structure

The `OperationsBlock` is a sequence of single operation blocks and each block is separated from the others by a "$" char. For example, a valid operations block is: "AVGDAILYAVG"
The meaning of this sequence will be clear reading the next sections.
In this block there could be any compatible sequence of operations (the meaning of compatibility will be explained in the semantic analysis section) and each operation can be of 5 different types:

- **Last value:** This kind of operations work on the **last values** sent by the specified sensors or on an output given by a previous operation
- **Temporal:** This kind of operations work on the values sent by the sensors in a certain time window.
- **Periodic:** This kind of operations send periodically some kind of statistics (`AVG`,`SUM`,etc..) at a fixed interval of time and there can be **at most one** of this operations in the `OperationsBlock`
- **Rule based:** This kind of operations filter out the data given in input to them that don't respect the specified rule
- **Information extraction:** This kind of operations are used to extract information from a more complex data structure, for example to extract the number of people from an image.

4.2.3 – Available Operations

We are now going to describe all the available operations that can be specified in the operations block.

4.2.3.1 – Last Value Operations

Operation Name	Operation Description
$AVG	It computes the average on the last values assumed by the sensors or on the values generated by a previous operation
$SUM	It computes the sum on the last values assumed by the sensors or on the values generated by a previous operation
$MIN	It takes the minimum of the last values assumed by the sensors or on the values generated by a previous operation
$MAX	It takes the maximum of the last values assumed by the sensors or on the values generated by a previous operation
$COUNT	It counts the number of matching topics stored in the buffer or the number of elements passed by a previous operation (it will be clearer when the operation composition will be explained)

Table 4: Last values operations

Examples of valid subscriptions are:

Example topic	Description
`$SUM/tmp/#`	Subscribes to the sum of all temperature sensors that communicate with the system
`$MIN/tmp/+/+/room1/+`	Subscribes to the min temperature of the sensors that are in room1 in any organization or building.
`$AVG/tmp/polimi/deib/room1/sensor1;sensor2`	Subscribes to the average of the value of sensor1 and sensor2 in room1 of deib in polimi.

Table 5: Last value operations examples

4.2.3.2 – Temporal Operations

This kind of operations follow the following syntax:

`$TMP<LastValueOpId>;TimeWindow`

As we can see, a temporal operation is characterized by the fact that the operation ID starts with the TMP keyword (that stands for temporal).

The `TimeWindow` block follows this syntax:

`gg:hh:mm` -> It means two digits for days, two digits for hours and two digits for minutes.
For example 00:01:00 specifies a time window of 1 hour

The `LastValueOpId` can be replaced with any of the last value operations listed before.

The following table lists all the possible temporal operations:

Operation Name	Description
`$TMPAVG;gg:hh:mm`	This computes the average of the values assumed by the sensor, in the given Time Window.
`$TMPSUM;gg:hh:mm`	This computes the sum of the values assumed by the sensor, in the given Time Window.
`$TMPMIN;gg:hh:mm`	This takes the minimum of the values assumed by the sensor, in the given Time Window.
`$TMPMAX;gg:hh:mm`	This takes the maximum of the values assumed by the sensor, in the given Time Window.
`$TMPCOUNT;gg:hh:mm`	This counts how many values a sensor sent, in the given Time Window.

Table 6:Temporal operations

Examples of valid subscriptions are:

Example topic	Description
`$TMPSUM;00:01:00/tmp/#`	Subscribes to the sum of the values sent in the last hour by all the temperature sensors that communicate with the system
`$TMPMIN;00:01:00/tmp/+/+/room1/+`	Subscribes to the minimum temperature sent in the last hour by the sensors that are in room1,in any organization or building.
`$TMPAVG;00:01:00/tmp/polimi/deib/room1/sensor1;sensor2`	Subscribes to the average of the values sent by sensor1 and sensor2 in room1 of deib in polimi, in the last hour

Table 7: Temporal operations examples

4.2.3.3 – Periodic Operations

This kind of operations follows the following syntax:

`$TimeIntervalKeyword<LastValueOpId>`

As `TimeIntervalKeyword` the following values can be specified:

-`DAILY`: The event triggers at the end of the day (at 00:00 of each day)

-`HOURLY`: The event triggers at the end of each hour of the day (08:00,09:00,10:00,etc..)

-`QUARTERHOURLY`: The event triggers at the end of each quarter hour (08:15,08:30,08:45,etc..)

Instead, the `<LastValueOpId>` can be replaced with any of the last value operations listed before.

Examples of valid subscriptions are:

Example topic	Description
`$DAILYSUM/tmp/#`	At the end of the day, it sends the sum of the values sent in the last day by all the temperature sensors that communicate with the system
`$DAILYMIN/tmp/+/+/room1/+`	At the end of the day, it sends the minimum of the values sent in the last day by the sensors that are in room1, in any organization or building.
`$DAILYAVG/tmp/polimi/deib/room1/sensor1;sensor2`	At the end of the day, it sends the average of the values sent in the last day by sensor1 and sensor2 in room1 of deib in polimi

Table 8:Periodic operations examples

4.2.3.4 – Rule based operations

Operation Name	Operation Description
$GT;value	It lets pass the input value (published by a sensor or given by a previous operation) only if its value it's strictly greater than the fixed parameter "value" (the parameter value can be a generic double number)
$GTE;value	It lets pass the input value (published by a sensor or given by a previous operation) only if the value it's greater or equal than the fixed parameter "value" (the parameter value can be a generic double number)
$LT;value	It lets pass the input value (published by a sensor or given by a previous operation) only if the value it's strictly less than the fixed parameter "value" (the parameter value can be a generic double number)
$LTE;value	It lets pass the input value (published by a sensor or given by a previous operation) only if the value it's less or equal than the fixed parameter "value" (the parameter value can be a generic double number)
$EQ;value	It lets pass the input value (published by a sensor or given by a previous operation) only if the value it's equal to the fixed parameter "value" (the parameter value can be a generic double number)
$NEQ;value	It lets pass the input value (published by a sensor or given by a previous operation) only if the value it's different than the fixed parameter "value" (the parameter value can be a generic double number)
$CONTAINS;text	It lets pass the input string (published by a sensor or given by a previous operation) only if the string contains the fixed parameter "text"

Table 9: Rule based operations

Let's see some usage examples. In each example there will be the topic used by the sensor to publish a value and the value sent to the broker.

Publish Topic	Publish Value
tmp/sensor1	25
string/sensor2	"example"

Table 10: Rule based operations examples

Here there are some subscriptions with the indication if the published value passes the filter or not

```
[OK] $GT;20,25/tmp/+
[OK] $LT;30,76/tmp/+
[OK] $NEQ;26/tmp/+
[OK] $CONTAINS;exa/string/sensor2
[NO] GT;30/tmp/+
[NO] EQ;14/tmp/+
[NO] $CONTAINS;randomString/string/+
```

4.2.3.5 – Information extraction

Operation Name	Description
$COUNTPEOPLE	It takes the image passed as input and extracts the number of people present in that image
$COUNTMALE	It takes the image passed as input and extracts the number of males present in that image
$COUNTFEMALE	It takes the image passed as input and extracts the number of females present in that image

Table 11: Information extraction operations

Examples of valid subscriptions are:

Example topic	Description
$COUNTPEOPLE/image/sensor1	It extracts the number of people from the image sent by sensor1
$COUNTMALE/image/sensor1	It extracts the number of males from the image sent by sensor1
$COUNTFEMALE/image/sensor1	It extracts the number of females from the image sent by sensor1

Table 12: Information extraction operations examples

4.3 – Semantic

In the previous section, we have seen all the possible operations that it is possible to specify, but the real strength of the `OperationsBlock` comes from the combination of the available operations.
We have decided to let the users freely combine the operations, with just the limitation that the final `OperationsBlock` has a valid semantic.

The concept behind, that inspired this approach, is very similar to the logic of method calls.
So, with this similarity, just think each possible operation as a method with at most one input parameter and one output. For example, let's suppose we have these methods:

```
String CONTAINS(String par);

ArrayList<Double> DAILYAVG();

Double AVG(ArrayList<Double> elements);
```

Let's see the following method calls:

1. `CONTAINS(DAILYAVG())` -> Of course this call gives a compiler error since the `CONTAINS` takes a string as a parameter and instead the `DAILYAVG` returns an arraylist of double

2. `AVG(DAILYAVG())` -> This call, instead, it's accepted by the compiler, because the `DAILYAVG` returns an array of doubles and the `AVG` method accepts that array as parameter and returns a Double

Now, the passage from methods and our operation concatenation is very straightforward and the corresponding `OperationsBlock` are the following:

1. `$CONTAINS$DAILYAVG`
2. `$AVG$DAILYAVG`

Always remember that the `OperationsBlock` operations are executed from right to left, to respect the methods convention. In the next section we'll list the available data types that the various operations can take as inputs or return as outputs.

4.3.1 – Available data types

In this section we are going to explain all the data types that are used in our semantic and their hierarchy. Even in this case the similarities between this structure and the object-oriented programming is evident. Indeed, in the case of object-oriented programming, each object has always a parent object and the root of that tree is the Object class. Here, instead, as root (super-type) we have `Anything` and the children (sub-types) of the root are the other available data-types.

We split the data types in two different hierarchy trees: One for the single data types and the other for the set data types.

4.3.1.1 – Single data types

This kind of types are used from operations that take as input a single value or/and as output return a single value.

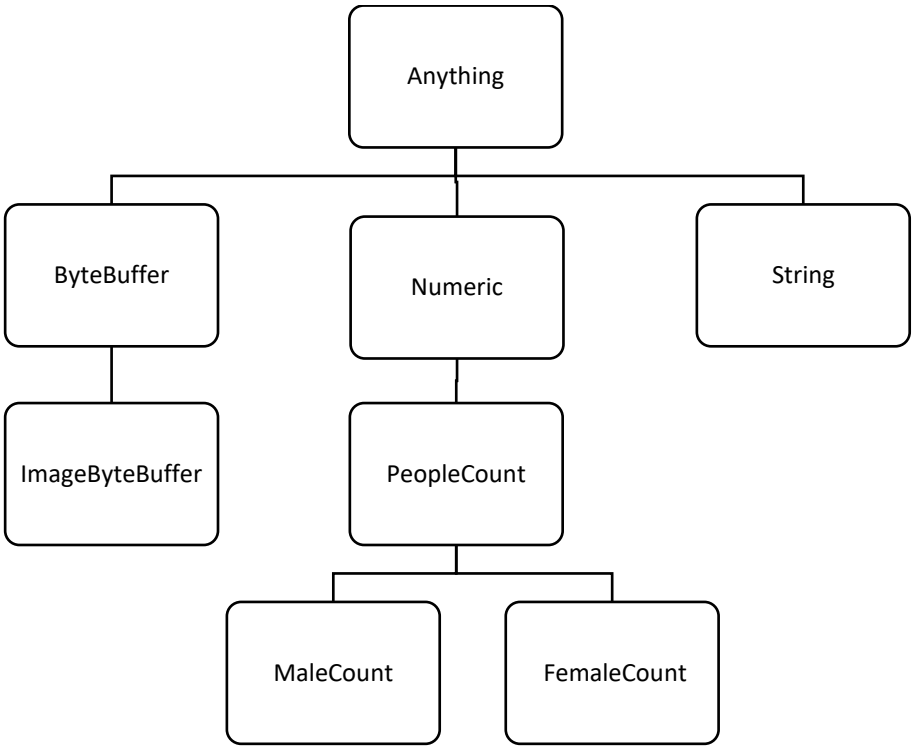


Figure 8: Single data types tree

Data Type	Description
`Anything`	Used as root and super type of all data types.
`ByteBuffer`	Indicates a generic buffer of bytes
`ImageByteBuffer`	Indicates a buffer of bytes that represents an image
`Numeric`	Indicates a generic number with no further meaning
`PeopleCount`	Indicates a number that represent the number of people
`MaleCount`	Indicates a number that represent the number of males
`FemaleCount`	Indicates a number that represent the number of females
`String`	Indicates a generic string

Table 13: Data types description table

4.3.1.2 – Set data types

This kind of types are used from operations that take as input a set of values or/and as output return a set of values.

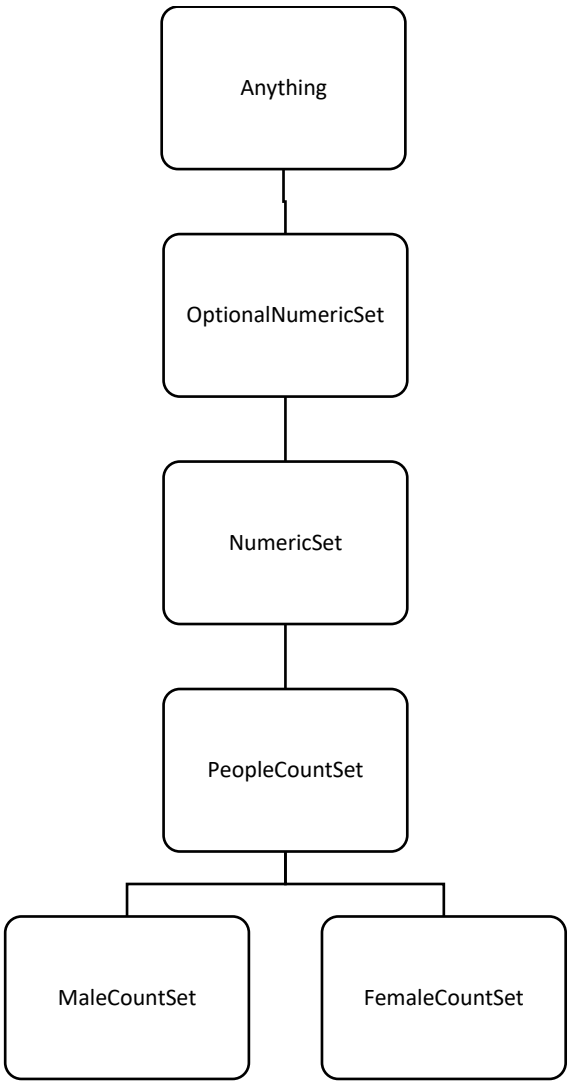


Figure 9: Multiple data types tree

Data type	Description
`Anything`	Used as root and super type of all types.
`OptionalNumericSet`	It indicates a set that can be specified or not. It is used to indicate that the input/output of the operation could be a set or not (it will be clear later on)
`NumericSet`	It indicates a set with its elements that are generic number with no further meaning
`PeopleCountSet`	It indicates a set with its elements that are numbers that represent the number of people
`MaleCountSet`	It indicates a set with its elements that are numbers that represent the number of males
`FemaleCountSet`	It indicates a set with its elements that are numbers that represent the number of females

Table 14: Multiple data types description

4.3.1.3 – Inner data type

The inner data type is a useful concept for the sets and indicates what kind of elements compose the set and for each set data type it's defined in this way:

Data type	Inner data type
`OptionalNumericSet`	`Numeric`
`NumericSet`	`Numeric`
`PeopleCountSet`	`PeopleCount`
`MaleCountSet`	`MaleCount`
`FemaleCountSet`	`FemaleCount`

Table 15: Inner data types

Actually, the inner data type is defined also for the other data types, but it's simply equal to the original type, so it's trivial and we won't list it here.

4.3.2 – Operations semantic

Now we are going to list all the previously explained operations with their definitions for the input and the output data types

Important Definition – Input Independency

If an operation has as input `Anything`, it means that the operation is computed working on the values in the buffer and never considers the input of previous operations.

4.3.2.1 – Last value operations

All last value operations are defined in this way:

Input data type	Output data type
`OptionalNumericSet`	`Numeric`

Table 16: Last value operations semantic definition

As we can see, they take the `OptionalNumericSet` as input data type.
This is due to the fact that the last value operations can work in two different ways:

1. If the previous operation doesn't return a `NumericSet`, the last value operation will take the values to work on from the last value buffer.

2. If the previous operation returns a `NumericSet`, the last value operation is performed on the `NumericSet` previously returned.

So, with this kind of behaviour, we need the optionality of the `NumericSet` as input, since in some cases there is a `NumericSet` given by the previous operation and in some other cases there isn't, but the operation works the same.

4.3.2.2 – Temporal operations

All the temporal operations are defined in this way:

Input data type	Output data type
`Anything`	`Numeric`

Table 17: Temporal operations semantic definition

These operations have `Anything` as input data type. This is because the elements to work on are always the ones that are in the temporal buffer, with a timestamp that is inside the time window specified with the temporal operation.

4.3.2.3 – Periodic Operations

For all periodic operations we have:

Input data type	Output data type
`Anything`	`NumericSet`

Table 18: Periodic operations semantic definition

As before, the Anything data type as input is due to the fact that also these operations take the data to work on from the buffer and not from previous operation.

The `NumericSet` set as ouput is necessary because, when the specified time passes, the operation returns the specified statistic (`AVG`,`SUM`,`MIN`,ecc..) for each topic stored in the buffer that matches with the subscription topic (or to be precise, that matches with the matching part of the subscription topic), so this is a set of values. This will be clear when we'll discuss about the statistics and their logic.

4.3.2.4 – Rule Based Operations

Operation name	Input data type	Output data type
`$GT;value`	`OptionalNumericSet`	`OptionalNumericSet`
`$GTE;value`	`OptionalNumericSet`	`OptionalNumericSet`
`$LT;value`	`OptionalNumericSet`	`OptionalNumericSet`
`$LTE;value`	`OptionalNumericSet`	`OptionalNumericSet`
`$EQ;value`	`OptionalNumericSet`	`OptionalNumericSet`
`$NEQ;value`	`OptionalNumericSet`	`OptionalNumericSet`
`$CONTAINS;text`	`String`	`String`

Table 19: Rule base operations semantic definition

The contains operation is trivial: It takes a string as input and returns the same string if the text is contained in the input string.

Instead, the understanding of the other rule-based operations is a bit more complex.
Consider these two different valid subscriptions:

1. `$GTE;25$DAILYAVG/tmp/+` -> I want to know the daily averages of the temperatures in the various rooms only If they are above or equal to 25 degrees

2. `$GTE;25/tmp/+` -> I want to receive all the temperature updates only if they are above or equal to 25 degrees

In this two cases the behaviour of the `GT` is different, because in the first case it has to filter out all the values coming from the `DAILYAVG` that are less than 25, so the `GTE` takes a `NumericSet` as input, instead in the second case, each time a temperature sensor publishes a value, it has to decide whether to let it pass or not, so in this case the `GTE` takes a `Numeric` data type as input (a single value).
This different behaviour explains why there is the optionality of the `NumericSet`, because in some cases the `NumericSet` is provided by the previous operation and in other cases it's not.

The `OptionalNumericSet` as output comes from a direct consequence of this behaviour, because if the `GTE` has as input a `NumericSet`, it returns a `NumericSet` with the filtered data, instead if it has a `Numeric` (single number) as input, it will return a `Numeric` (single number).

4.3.2.5 – Information extraction operations

Operation name	Input data type	Output data type
$COUNTPEOPLE	ImageByteBuffer	PeopleCount
$COUNTMALES	ImageByteBuffer	MaleCount
$COUNTFEMALES	ImageByteBuffer	FemaleCount

Table 20: Information extraction semantic definition

As we can see, the input is the same for every operation and it's a buffer of bytes that represents a valid image to be processed.

The output is different, because even though all those operations return a number as output, that output has a different meaning for each operation. In the case of `COUNTMALE` it is the number of males, instead in the case of `COUNTFEMALES` it's the number of females and, finally, for `COUNTPEOPLE` it is the number of generic people.
This difference is equivalent to the concept of unit measure: 1Kg or 1 m have both as value one, but they mean a different thing.
This concept of distinguish the numbers in `Numeric`,`PeopleCount`,`MaleCount` and `FemaleCount` will be crucial to buffer and retrieve correctly the various numbers, avoiding to mix numbers of different meanings in aggregations. This thing will be explained better when we'll show the buffer structure.

4.4 – Subscription processing

4.4.1 - Parsing

As we have seen in Chapter 2, the first step to do is to subscribe, in order to receive the published data by the various sensors. So, let's see how the subscription process works in MQTT+.

When a subscription arrives to the broker, the subscription topic must be parsed to verify if it's a valid subscription or not. The parsing includes two steps:

- **Syntax analysis**: This step checks that each operation block in the `OperationsBlock` follows the syntax rules that we specified in the previous sections.

- **Semantic analysis**: This step checks that the `OperationsBlock` is a semantically valid sequence of operations

The parsing process is important for several reasons:

1. **Avoid wrong input runtime errors:** If a subscription passes the parsing process successfully, we are guaranteed that, when that subscription needs to be elaborated, the computation will be successful and won't fail because of some incompatible operations that were put in sequence by an error of the user. Indeed, with this logic, if the initial input is correct (this must be always guaranteed by the user), the subscription will be elaborated correctly and this avoids to check, for every operation, if the input is correct.

2. **Avoid waste of computational resources:** The correct execution is important also for saving computational time. For example, if we have the current subscription:

   ```
   $CONTAINS;text$COUNTPEOPLE/image/+
   ```

   Without parsing, we wouldn't recognize that this subscription would fail in the step where the `$CONTAINS` expects a string as input and instead it receives a number. This would waist (for each image sent!) the computational time to analyse and extract the number of people from the image and (as we will see in the section dedicated to tests) this is a computationally intensive operation and must be done only when it's really needed.

3. **Define subscription returned data type:** The data type returned from the semantic analysis is crucial and necessary to access the right buffer to store and retrieve the correct data as we will see in the buffering section.

4.4.1.1 – Syntax analysis

This step is conceptually very simple, because basically it is sufficient to:

- Verify that the `OperationsBlock` starts with "$" and that it has at least one valid operation
- Split the `OperationsBlock` by "$", verify that each operation is a syntactically valid operation as we specified in section 3.2 and that there is not more than one periodic operation

If this first analysis is successful, then the semantic analysis is performed.
Instead, if the subscription syntax is not valid, the subscription is discarded and not considered as a valid MQTT+ subscription.

4.4.1.2 – Semantic analysis

This is the second and (more complex) final step of parsing.
If we arrive at this step, then the `OperationsBlock` contains for sure only syntactically valid operations and we can also be sure to work on a correctly formatted `OperationsBlock` string.

Before showing how the semantic analysis works, it is necessary to introduce the concept of type compatibility and type forwarding.

DEF. Type Compatibility

Suppose we have two data types:

- Type1
- Type2

Type 1 is **compatible** with Type2 only in two cases:

- Type1 is the same type of Type2
- Type1 is a sub-type of Type2

DEF. Type forwarding

Suppose we have the following operation and input:

1. Operation -> InputSpecificationDT: Type1, OutputSpecificationDT: Type2
2. Given input DT: Type3

The given input DT is forwarded as output of the Operation **if and only if** the following two conditions hold:

1. Type3 is compatible with Type1 (input-compatibility)
2. Type3 is compatible with Type2 (forwarding-compatibility)

So, from the previous definitions we can see that type compatibility, in the semantic analysis, is important for two reasons:

- To check if the type given as input, by the previous operation, is compatible with the input data type required by the current operation
- To forward the input data type as output if the input type is compatible with the output type

Let's make this process clearer with an example. Suppose to have this subscription, with valid syntax:

`$AVG$COUNTPEOPLE/image/+` -> Makes the average of the number of people in the images sent by all image sensors

This is how the semantic analysis works (remember that the operations are processed from right to left):

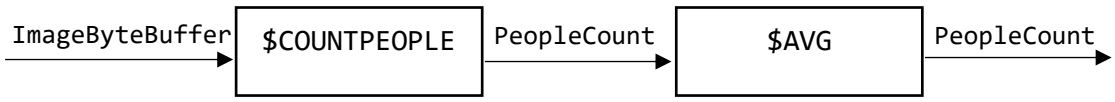


Figure 10:Semantic analysis example 1

NOTE: the input of the first operation (`$COUNTPEOPLE`) is always supposed correct and given as specified in the operations semantic specification table, because if the initial input is wrong, then it's not a problem of `OperationsBlock` semantic, but it's an error of the user that made a subscription to some sensors that publish data not compatible with the operation specified.

1. First step (analyse the `$COUNTPEOPLE`):
    a. Is the given input (`ImageByteBuffer`) compatible with the `$COUNTPEOPLE` input specification (`ImageByteBuffer`) ? Yes, because they are the same type
    b. Is the given input (`ImageByteBuffer`) compatible with the output specification of the operation (`PeopleCount`) ? No, because `ImageByteBuffer` is not a sub-type of `PeopleCount`, so don't forward and give as output the specified output DT (`PeopleCount`)

2. Second step (analyse the `$AVG`):
    a. Is the given input (`PeopleCount`) compatible with the `$AVG` input specification (`Anything`) ? Yes, because `PeopleCount` is a sub-type of `Anything`
    b. Is the given input (PeopleCount) compatible with the `$AVG` output specification (Numeric) ? Yes, because PeopleCount is a sub-type of Numeric, so the sub-type is forwarded and the output of the operation is PeopleCount.

3. Are there any more operations? No, then return the subscription data type (PeopleCount)

So, in this case the semantic analysis has successfully ended and the returned data type of the subscription is `PeopleCount`. The returned data type indicates what kind of data type the elaboration of all operations in the `OperationsBlock` will return and this is crucial, as said before, for the correct buffering of the data.

For this reason, it's very important the process of forwarding a compatible type from input to output, because let's assume that there wasn't this behaviour and that the operation always returns as output the specified output. Then, in the previous example, the `$AVG` would return `Numeric` as output and this data type is taken as the subscription return type, but it's wrong!
It's wrong, because we would lose the information that the `$AVG` is an average of the number of people (`PeopleCount`) and not a generic number.

This behaviour of type forwarding is also extended to be compatible between sets and single data types

DEF. Type forwarding between sets and single type

Set data type as input and single data type as output

Suppose we have the following operation, input and inner-input:

3. Operation -> InputSpecificationDT: SetType1, OutputSpecificationDT: SingleType2
4. Given input DT: SetType3

The **inner type** of the input DT is forwarded as output of the Operation **if and only if** the following two conditions hold:

- SetType3 is compatible with SetType1 (input-compatibility)
- **Inner-type** of SetType3 is compatible with SingleType2 (Set-single compatibility)

Single data type as input and set data type as output

Suppose we have the following operation and input:

5. Operation -> InputSpecificationDT: SingleType1, OutputSpecificationDT: SetType2
6. Given input DT: SingleType3

The **Set type with SingleType3 as inner-type** is forwarded as output of the Operation **if and only if** the following two conditions hold:

1. SingleType3 is compatible with SingleType1 (input-compatibility)
2. SingleType3 is compatible with **inner-type** of SetType2 (Single-Set compatibility)

Even though this concept is complex, the following two examples will clarify the logic behind. The first one will be the first case (Set data type as input and single data type as output) and the second one the second case (Single data type as input and set data type as output)

Set data type as input and single data type as output example

Suppose to have this subscription with valid syntax:

`$AVG$DAILYAVG/tmp/+` -> Makes the average of the average temperatures observed in a day in each room

This is how the semantic analysis works:

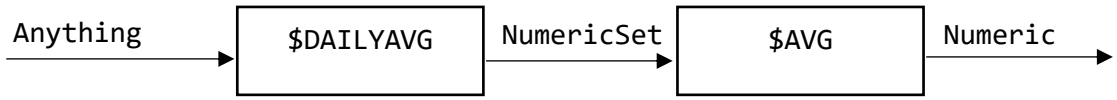


Figure 11: Semantic analysis example 2

1. First step (analyse the $DAILYAVG):
    a. Is the given input (Anything) compatible with the $DAILYAVG input specification (Anything)? Yes, because they are the same type
    b. Is the given input (Anything) compatible with the output specification of the operation (NumericSet)? No, because Anything is not a sub-type of NumericSet, then give as output the specified output DT (NumericSet)

2. Second step (analyse the $AVG):
    a. Is the given input (NumericSet) compatible with the $AVG input specification (OptionalNumericSet)? Yes, because NumericSet is a sub-type of OptionalNumericSet
    b. Is the **inner-type** of the given input (Numeric) compatible with the $AVG output specification (Numeric)? Yes, because they are the same type, so the **inner-type** of the given input (Numeric) is forwarded and the output of the operation is Numeric.

3. Are there any more operations? No, then return the subscription data type (Numeric)

Single data type as input and set data type as output example

Suppose to have this subscription with valid syntax:

$DAILYAVG$COUNTPEOPLE/image/+ -> Returns the average of the number of people in a day for each room

This is how the semantic analysis works:

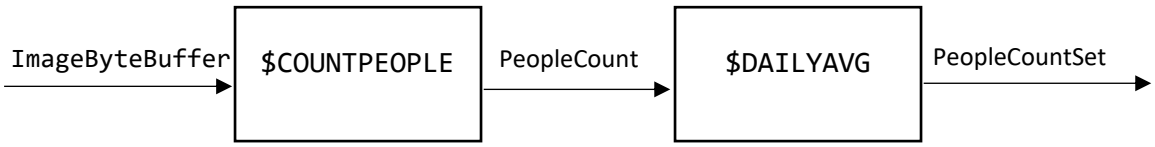


Figure 12: Semantic analysis example 3

1. First step (analyse the $COUNTPEOPLE):
    a. Is the given input (ImageByteBuffer) compatible with the $COUNTPEOPLE input specification (ImageByteBuffer)? Yes, because they are the same type
    b. Is the given input (ImageByteBuffer) compatible with the output specification of the operation (PeopleCount)? No, because ImageByteBuffer is not a sub-type of PeopleCount, then give as output the specified output DT (PeopleCount)

2. Second step (analyse the $DAILYAVG):
    a. Is the given input (PeopleCount) compatible with the $DAILYAVG input specification (Anything)? Yes, because PeopleCount is a sub-type of Anything
    b. Is the given input (PeopleCount) compatible with the $AVG output specification **inner-type** (Numeric)? Yes, because PeopleCount is a sub-type of Numeric.
    c. What is the Set data type that has as **inner-type** PeopleCount? PeopleCountSet, then return this type as output data type

2. Are there any more operations? No, then return the subscription data type (PeopleCountSet)

With this logic, we can see that even in the case of sets, the data type forwarding is guaranteed.
Note that without this extension for the mixed case, in which one data type (input/output) is a single data type and the other type a set data type, we would lose the information of a sub-inner-type given as input. For instance, in the last example, the returned data type would be NumericSet instead of PeopleCountSet, losing the information that in input we had PeopleCount as data type.

As last example, let's see the case in which the semantic analysis fails.

Suppose to have this subscription with valid syntax:

$COUNTPEOPLE$AVG/image/+

This is how the semantic analysis works:

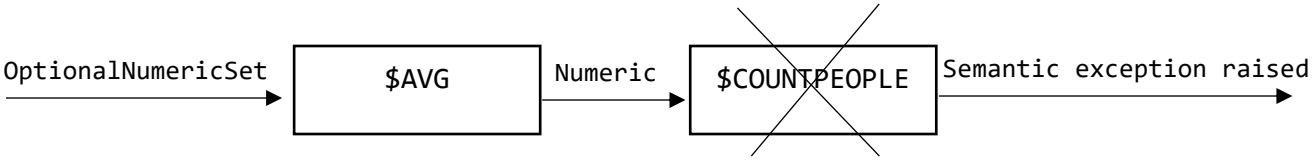


Figure 13: Semantic analysis example 4

In this case, when the $COUNTPEOPLE operation is analysed, the semantic analysis recognizes that the given input (Numeric) is not compatible with the required input (ImageByteBuffer), so it raises a semantic exception, the analysis stops and the subscription is discarded because it's not semantically valid.

The following flow chart summarizes all the previously explained steps of the semantic analysis

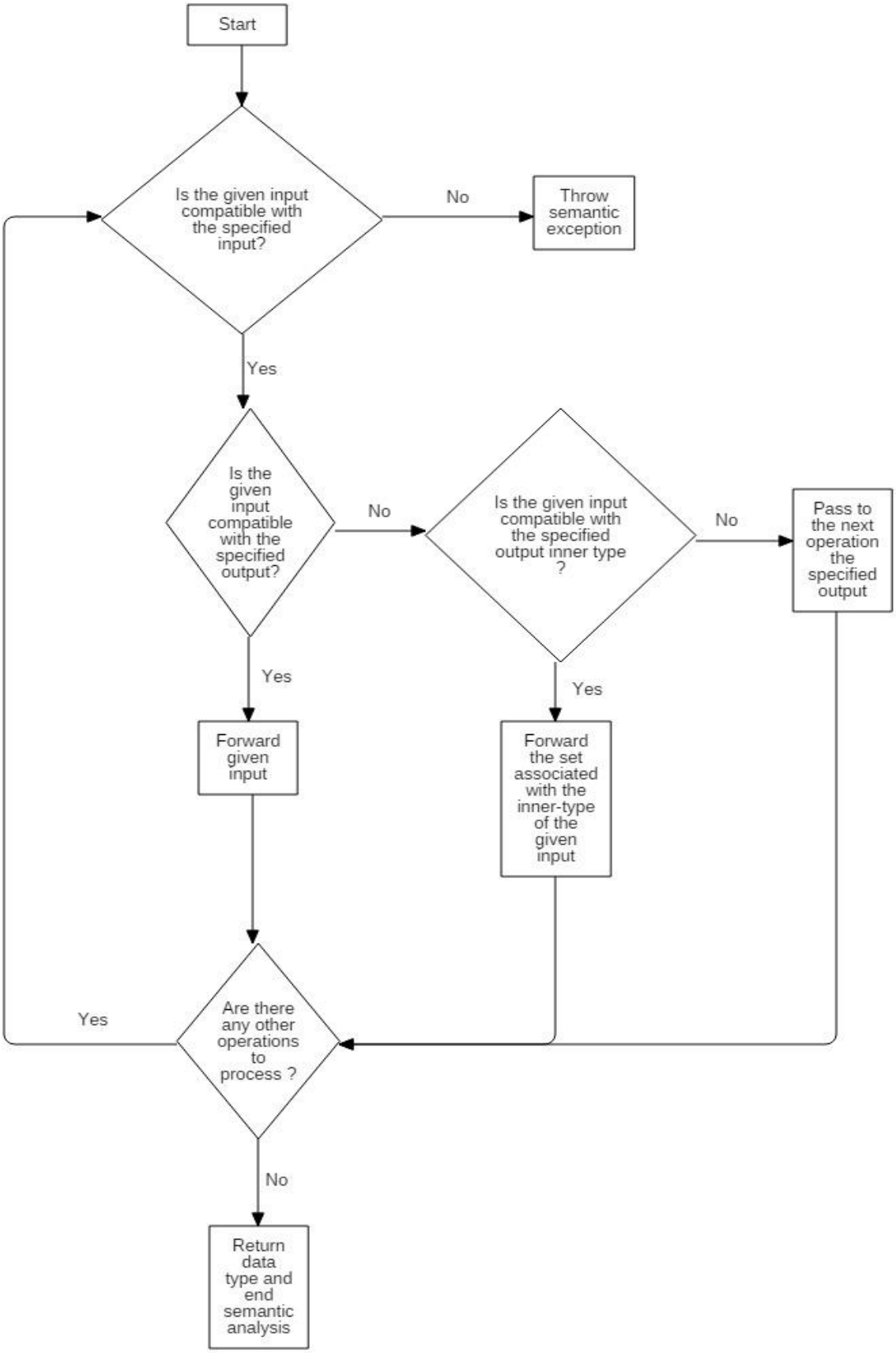


Figure 14: Semantic analysis control flow

4.4.2 – Subscription buffering

After the subscription has been parsed and recognized as a valid subscription, the HiveMQ broker saves it in a subscription buffer.

The entries of this buffer are as shown in the table below:

ClientID	Subscriptions
Client1	`$DAILYAVG/temperature/polimi/..`
	`$HOURLYSUM/temperature/..`
Client2	`$TMPMIN;00:01:14/temperature/..`

Table 21: Subscriptions table

So, for each client, we have all the subscription that it made.
Actually, the subscriptions made by a client are already managed and stored by the HiveMQ broker in an internal DB, but since each time a sensor publishes a message, these subscriptions must be retrieved (to see if the publish topic matches with some subscriptions), this will impact to the broker performances, as explained also in the HiveMQ guide, because it requires slow hard disk accesses (that we all know to be a PC's bottleneck) and therefore these queries must be done as few as possible.

For this reason, the subscriptions are retrieved from the DB only at broker start-up and after that they are read from the buffer.
Please note that this buffer is always in sync with the DB, because:

- Each time a client makes a subscription, that subscription is added to the buffer
- Each time a client unsubscribes, that subscription is removed from the buffer
- Each time a client, with a clean session, disconnects, all its subscriptions are removed from the buffer

This buffer is also used for other reasons than just on a publish from some sensors. For example, it's used in data buffering to enable or disable temporal buffering and it's also used to decide if to compute or not computationally expensive operations (people detection from images).
All these other usages of the subscription buffer will be explained later in the document.

4.4.2.1 – Resource consumption

Let's now see how much memory this buffer consumes.

Suppose to have:

1. N clients
2. M subscriptions for each client
3. Each subscription occupies an amount of memory equal to SRS (Subscription record size).
4. Each client id occupies an amount of memory equal to CIS (client id size).

Then, the amount of memory used is approximately:

$$Buffer\ size \approx CIS \cdot N + SRS \cdot N \cdot M = (CIS + SRS \cdot M) \cdot N$$

Now let's suppose that:

- N: We have 100.000 clients (which are a lot connected to the same broker!)
- M: Each client can make up to 100 subscriptions
- SRS: A subscription can have a length of 100 characters maximum (100 bytes)
- CIS: A client id can have a length of 15 characters maximum (15 bytes)

Then the buffer size is approximately:

$$Buffer\ size\ \approx\ (15 + 100 \cdot 100) \cdot 100.000\ bytes\ =\ 955\ MB$$

This value is not very small, but anyway it can be handled by a normal PC. Please note that this is the worst case possible, in which all 100.000 clients make 100 subscriptions each and with the maximum length of client id and subscription topic and they never disconnects (or they all make permanent connections), so this formula calculates the upper bound of the memory usage.
This is very unlikely that happens in a real context and in any case this memory upper bound can be fixed by the broker administrator, imposing the maximum number of subscriptions that a client can made, the maximum number of connected clients and subscription topic and client id maximum lengths, so this memory consumption can be easily taken under control.
Moreover, with the fact that whenever a client unsubscribes or disconnects, the buffer is updated and their subscription removed, it is very difficult that the upper bound limit is reached.

4.4.3 – Subscription elaboration

After the subscription has been parsed and stored in the buffer, the `OperationsBlock` in the subscription needs to be elaborated.
This elaboration can take place in two different moments:

- **Whenever a sensor publishes a message (with a topic that matches the subscription topic):** This is the case in which the `OperationsBlock` doesn't contain any periodic operation. This is called immediate subscription elaboration.

- **Whenever the timer triggers the time interval event:** This is the case in which the `OperationsBlock` contains a periodic operation. In this case, the elaboration is postponed until the time interval specified by the periodic operation has passed.

The way in which this is implemented will be discussed later in the document.

4.4.4 – Publish topic choice

After the subscription has been elaborated, we must decide which topic to use for publishing the message to the interested client.

Suppose that the current subscription has been elaborated:

```
$AVG/tmp/+
```

To publish the result, to the client that made the subscription, we can't use the same topic of the subscription, because this is illegal in the MQTT protocol. Indeed, the HiveMQ broker will raise an exception if you try to publish to a client with a topic containing wildcards.

So, since this approach is not possible, we have introduced to possible solutions:

- **Single keyword replacement (SKR):** We substitute each wildcard in the subscription topic with `$A`. So, in this case, the publish topic would be: `$AVG/tmp/$A`

- **Replacement with participating topics (RPT):** We substitute each wildcard in the subscription topic with every possible value that matches the subscription topic. For example, if in the buffer we have sensor1 and sensor2 for the last level of the topic, the publish topic would be: `$AVG/tmp/sensor1;sensor2`

In case we decide to use SKR, the bandwidth used is less, since we don't have to specify all the values on which the aggregation has been done, but the client doesn't know which topics the aggregation has been made on. This is why we introduced RPT that gives to the client this information (referring to the last example the client knows that the average has been made on the values of sensor1 and sensor2) at the price of a higher bandwidth used.

The choice of which approach to use is left to the broker administrator, depending on what it's more important between bandwidth used and the information given to the client, for the specific context that will be considered.

4.5 – Data Buffering

In this chapter, we are going to introduce another important concept: Data Buffering.

Either for temporal analysis, last value analysis and periodic analysis, a buffer is required. In the first case, to store all the values that a sensor publishes in a certain time window, in the second case to store just the last value sent by each sensor and in the third case to store some statistics about the values assumed by each in sensor in a certain interval of time.
Since the usage of the buffer from these three operations is different, we decided to use two different logic and structures to memorize the interested data: One for last value and periodic analysis and the other one for temporal analysis.

4.5.1 – Temporal analysis buffer

The entries of the buffer for temporal analysis are shown as in the table below:

Topic	Value	Detection Time
tmp/polimi/deib/room1/sensor1	18	21/02/2018 22:30:00
	19	21/02/2018 22:35:00
	20	21/02/2018 22:45:00
tmp/polimi/deib/room1/sensor2	21	21/02/2018 22:45:00
	20	21/02/2018 22:47:00
tmp/polimi/deib/room2/sensor1	25	21/02/2018 23:00:00
	23	21/02/2018 23:04:00

Table 22: Temporal analysis buffer

As we can see, for each topic (the identifier of a certain sensor/publisher), the detected values are stored with their detection time that indicates the date and time at which those values arrived to the broker.

4.5.1.1 – Temporal operations usage

This buffer is used only by temporal operations (`TMPAVG`,`TMPSUM`,etc..) and now we'll see how.
Suppose we have the following subscription to process:

```
$TMPAVG;00:02:00/tmp/polimi/deib/room1/+
```

[This subscription returns the average temperature detected by all sensors in room1 in the last 2 hours]

Suppose that the current instant is: 22/02/2018 00:37:00 and that the buffer table is the one above

The elaboration that takes the data from the buffer is split in 2 steps:

1. We have to see which topics, of the ones in the buffer, match the subscription.
   In this case, the first two topics in the buffer match it and the last one doesn't.

2. For each matching topic, we have to take the values that are in the interval of time [CurrentInstant-2h;CurrentInstant] (or in other words the values that are not older than two hours from now)

At the end of these two steps, the elements to compute the average on are:

Topic	Value	Detection Time
tmp/polimi/deib/room1/sensor1	20	21/02/2018 22:45:00
tmp/polimi/deib/room1/sensor2	21	21/02/2018 22:45:00
	20	21/02/2018 22:47:00

Table 23: Temporal analysis usage example

So, the computed average is: (20+21+20)/3 = 20,33333
And this value is sent to the client that made the subscription.

Note that the third topic of the buffer wasn't considered because it didn't match the subscription and also the first two detections of the first topic (sensor1) were not considered because those detections are outside the time window.

4.5.2 – Last value analysis buffer

The structure of the last value buffer is different from the previous one, because in this case, for each topic, alongside the value and its detection time, there are also some statistics on the received data and an expiration date, indicating when the value is too old to be considered valid as "last" received value. At the moment, this value is fixed at one hour after the detection time, which means that the TTL (time to live) is 1h.

Topic	Value	Detection Time	Expiration date	Statistics
tmp/polimi/deib/room1/sensor1	20	21/02/2018 22:45:00	21/02/2018 23:45:00	{See below}
tmp/polimi/deib/room1/sensor2	25	21/02/2018 22:47:00	21/02/2018 23:47:00	{See below}
tmp/polimi/deib/room2/sensor1	23	21/02/2018 22:55:00	21/02/2018 23:55:00	{See below}

Table 24: Last value analysis buffer

4.5.2.1 – Statistics

The statistics object contains useful information for the elaboration of periodic operations.
In particular it contains statistics values of some interesting quantities (`SUM`,`MIN`,ecc..) for each possible interval of time that a periodic operation can use (15 minutes,1 hour,1 day).

So, the statistics object is structured in this way:

Statistics for the quarter hour interval

Number of detections	QuarterHourlySum	QuarterHourlyMin	QuarterHourlyMax
23	152,45	12,5	23,76

Table 25: QuarterHourly stats

Statistics for the hour interval

Number of detections	HourlySum	HourlyMin	HourlyMax
50	234,12	10,23	34,35

Table 26: Hourly stats

Statistics for the day interval

Number of detections	DailySum	DailyMin	DailyMax
123	430,23	8,9	43,39

Table 27: Daily stats

All these three statistics are in the same record of each topic, which means that
Statistics = Statistics for the quarter hour interval + Statistics for the hour interval + Statistics for the day interval

Note that the `AVG` statistic is not memorized in the statistics because it would be useless, since when it's needed, it's sufficient to take the sum of the interested interval of time and divide it by the number of detections of that interval of time. For example, for the daily interval we have:

DailyAvg = DailySum/Number of detections

How are these statistics updated?
Each time a new value (NV) is stored in the buffer, the statistics (for each time interval) are updated in this way:

- IntervalSum = IntervalSum + NV
- IntervalMin = Min(IntervalMin, NV)
- IntervalMax = Max(IntervalMax, NV)
- Number of detections = Number of detections + 1

The last value analysis buffer, it's used by the last value analysis operations (`AVG`,`SUM`,`MIN`,etc..) and the periodic operations (`DAILYAVG`,`QUARTERHOURLYSUM`,etc..). Let's see how in the next two sections.

4.5.2.1 – Last value operations usage

Suppose we have the following subscription to process:

`$AVG/tmp/polimi/deib/#`

[This subscription returns the average temperature detected by all sensors in the deib department]

Suppose also that the Current instant is 21/02/2018 23:46:00 and the buffer is the one in the table 19.

The elaboration to take the data from the buffer is split in 3 steps:

1. Do we have another operation, before AVG, that returns a set that the AVG can use? No, so the AVG must work on the buffer and not on a previous input.

2. We have to see which topics, of the ones in the buffer, match the subscription.
   In this case all the topics in the buffer match the subscription.

3. For each matching topic, we have to take the values that are still valid (with an expiration date that is after the Current instant)

At the end of these three steps, the elements to compute the average on are:

Topic	Value	Detection Time	Expiration date	Statistics
tmp/polimi/deib/room1/sensor2	25	21/02/2018 22:47:00	21/02/2018 23:47:00	-
tmp/polimi/deib/room2/sensor1	23	21/02/2018 22:55:00	21/02/2018 23:55:00	-

Table 28: Last value operation usage example

So, the computed average is: (25+23)/2 = 24

And this value is sent to the client that made the subscription.

Note that the first line was not considered (room1/sensor1) because its value was not valid anymore (CurrentInstant > 21/02/2018 23:45:00)

4.5.2.2 – Periodic operations usage

Suppose we have the following subscription to process:

`$DAILYAVG/tmp/polimi/deib/#`

[This subscription returns the average temperature detected by all sensors in the deib department in the whole day]

This execution of this subscription is postponed until the end of the day, when the day event triggers.

The elaboration to take the data from buffer is split in 2 steps:

1. We have to see which topics, of the ones in the buffer, match the subscription.
   In this case all the topics in the buffer match the subscription.

2. For each matching topic, we have to take from the statistics the DailySum and the number of detections.

At the end of this step, suppose that the retrieved data from the statistics is:

Topic	DailySum	Daily number of detections
`tmp/polimi/deib/room1/sensor1`	420,24	25
`tmp/polimi/deib/room1/sensor2`	230,24	10
`tmp/polimi/deib/room2/sensor1`	625,23	35

Table 29: Periodic operation usage example

From this table we sum all the DailySums to obtain the total sum and the number of detections to obtain the total number or detections:

Total Sum = 420,24 + 230,24 + 625,23 = 1257,71
Total number of detections = 25 + 10 + 35 = 70

From these two values the DailyAvg is computed:

DailyAvg = TotalSum/Total number of detections = 1257,71 /70 = 17,97

The DailyAvg is then sent to the client that made the subscription.

4.5.3 – Buffer access

As we said before, in semantic analysis, we have to buffer data with different semantic meaning in different buffers.
To guarantee this fact, the two types of buffers (last detection buffer + temporal analysis buffer) are put in a single object (that we'll indicate as { buffer object }) and this object must be replicated for each different semantic type, so we need a mapping between the {buffer object} and its semantic type.
This mapping can be represented as the table shown below:

Stored data type	Buffer
`Numeric`	{buffer object}
`PeopleCount`	{buffer object}
`FemaleCount`	{buffer object}
`MaleCount`	{buffer object}

Table 30: Buffer mapping

So, how can we decide which buffer to use for the elaboration of a subscription?
Very simple: If the subscription is valid (it has passed the parsing process), we know its returned semantic data type. From this data type we get the inner data type and with this one we access the previous table and retrieve the correct {buffer object} to read/store data from/into.

4.5.4 - Resource consumption

Buffering can be resource consuming, especially for the RAM, but memory consumption is very different between last detection buffer (the one used for last value and periodic operations) and temporal buffer (the one used for temporal operations).

4.5.4.1 – Last detection buffer

Suppose that:

- Each sensor publishes its detections with a different topic
- Each detection occupies an amount of memory equal to LDRS (Last detection record size).
- We have N sensors

Now, since for each topic there is just one record the amount of memory used is approximately:

$$Buffer\ size\ \approx\ N \cdot LDRS$$

So, it's linear in the number of sensors sending data.

Just to have a rough idea of idea of the memory consumption, an LDR (Last detection record) is composed of:

- Topic size: 29 bytes (with the topic structure like the one in table 19)
- Value: 8 bytes (Double)
- Detection time: 18 bytes (with the date structure like the one in table 19)
- Expiration date: 18 bytes (with the date structure like the one in table 19)
- Statistic object x 3 (one for each time interval: quarterhourly,hourly,daily): 28 * 3 = 84 bytes
    - Number of detections: 4 bytes (Integer)
    - TimeIntervalSum: 8 bytes (Double)
    - TimeIntervalMin: 8 bytes (Double)
    - TimeIntervalMax: 8 bytes (Double)

So, making the sum of every component of the LDR, we obtain that:

$$LDRS\ \approx\ 154\ bytes$$

So, let's assume that we have 100.000 sensors, the memory usage of this buffer in memory would be approximately:

$$\text{Buffer size}_{100.000} \approx 100.000 * 154 \text{ bytes} = 14{,}69 \text{ MB}$$

That is a quantitative of memory that, nowadays, every computer has, so the last detection buffer isn't a problem for the RAM resource. Moreover, once we know the maximum number of sensors that can connect and send data to the broker (this limit can be set by the broker administrator) the upper bound of this buffer size is fixed and won't go more than that (because each sensor can occupy at most one record in the buffer).

For this reason, this buffer doesn't need cleaning techniques to reduce the memory used.
A total different approach, instead, it's used for the temporal buffer.

4.5.4.2 – Temporal buffer

Suppose that:

- Each sensor publishes its detections with a different topic
- Each topic occupies an amount of memory equal to TS (topic size)
- Each detection occupies an amount of memory equal to DRS (detection record size)
- We have N sensors
- Each sensor sends each detection with a frequency of λ (sec^{-1})
- The interval of time in which temporal buffering is on is T (sec)

Then the buffer size is approximately:

$$Buffer\ size \approx N \cdot TS + N \cdot \lambda \cdot T \cdot DRS = (TS + \lambda \cdot T \cdot DRS) \cdot N$$

In this case we can see that the buffer size depends by four different variables (the DRS is fixed):

- The number of sensors
- The frequency at which these sensors are sending data
- The interval of time in which the buffer must store all the data
- The topic size

Now, let's see how a DR (detection record) is composed:

- Value: 8 bytes (Double)
- Detection time: 18 bytes (with the date structure like the one in table 17)

So, making the sum of every component of the DR we obtain that:

$$DRS \approx 26\ bytes$$

Now, let's assume to have:

- N: 100.000 sensors
- TS: topics of 50 characters (50 bytes)
- λ: 1 message every 100 ms (10 messages per seconds)
- T: 1 day (86400 seconds)

Then the buffer size is approximately:

Buffer size ≈ (50+(1/10)*86400*26)*100.000 bytes = 20,93 GB

That is a huge number and it's very likely that a single broker doesn't have all that RAM, but let's even consider a more optimistic scenario with less sensors and with a lower sending frequency, so let's consider:

- N: 1.000 sensors
- TS: topics of 50 characters (50 bytes)
- λ: 1 message every 1 sec
- T: 1 day (86400 seconds)

Buffer size ≈ (50+1*86400*26)*1.000 bytes = 2,09 GB

Even in this case, the RAM required to buffer all the data sent by the sensors in one day is not irrelevant.

From these considerations, it's clear that some kinds of techniques are required to optimize the usage of the memory from this buffer.

4.5.4.3 – Techniques to save memory

As we have seen before, the temporal buffer can reach a significant size in terms of ram used, so in this section we are going to explain what techniques have been introduced to mitigate this problem for the temporal buffer.

The following ones are the techniques that have been introduced:

- **Start buffering only when temporal subscription arrives**
- **Clear the buffer from old values**
- **Add a new sensor value only if there is a subscription interested in that sensor**
- **Disable buffering if there is not anymore an interested subscription**

Start buffering only when a temporal subscription arrives

This optimization comes from the fact that it is a waste of resources memorizing values that will never be used by any subscription, so the buffering is started only when in the subscriptions buffer there is at least one subscription with a temporal operation in its `OperationsBlock`.

Clear the buffer from old values

Since we know all the subscriptions made by clients (stored in the subscriptions buffer), we also know what is the biggest temporal window size used.

For example, suppose to have these subscriptions in the buffer:

```
1. $TMPAVG;00:02:00/tmp/polimi/deib/#
2. $TMPAVG;00:30:00/tmp/polimi/deib/room2/+
3. $COUNTPEOPLE/img/polimi/deib/room2/imgSensor1
```

So, in this case, the biggest window size used by subscriptions that contain temporal analysis operations is 2 hours (00:02:00). Let's call this window MWS (Maximum window size), so in this case MWS = 2 hours.

This MWS is updated each time a new subscription is added or removed. For example, if the first subscription is removed, the MWS is updated to 30 minutes. Instead, if the following subscription is added:

```
4. $TMPAVG;00:04:00/tmp/polimi/deib/#
```

Then the MWS is updated to 4 hours.

In any case, whatever is the MWS computed in the way explained above, whenever a sensor publishes a detection that needs to be buffered, all the detections that are older than the MWS are removed from the temporal buffer.
This logic follows the principle that if a detection is older than the MWS, then it's not used by any subscription, so it's useless to take it in the buffer.

Note that this approach has the drawback explained with the following sequence of events:

1. The MWS is set to 2 hours, analysing the subscriptions from 1 to 3
2. 4 hours of time have passed
3. A sensor publishes a detection, with a topic matching subscription 1, that needs to be stored in the temporal buffer
4. The subscription 1 is computed and it finds all the values sent in the last 2 hours in the buffer
5. The buffer cleans all detections older than MWS (2 hours)
6. The subscription 4 is added and the MWS is set to 4 hours
7. A sensor publishes a detection, with a topic matching subscription 4, that needs to be stored in the temporal buffer
8. The subscription 4 needs to be computed, but it finds only the values of the last 2 hours and not 4, because they older ones have already been deleted when MWS was set to 2 hours!

Even though there is this drawback, we decided to give anyway more importance to memory consumptions (since we saw that the buffer can reach a significant size), rather than avoiding that a subscription finds an empty buffer or a buffer with less elements than needed. The solution to avoid this drawback (consuming more memory) would be to leave in the buffer all the detections, hoping that at a certain moment in the future, it will arrive a subscription that will use them.

Add a new sensor value only if there is a subscription interested in that sensor

Suppose to have these subscriptions in the subscriptions buffer:

- `$TMPAVG;00:02:00/tmp/polimi/deib/room1/sensor1;sensor2`
- `$TMPAVG;00:30:00/tmp/polimi/deib/room1/sensor1`

If a sensor publishes a detection with the following topic:

```
tmp/polimi/deib/room1/sensor5
```

This topic doesn't match any subscription of the ones currently in the buffer, so it has no sense to buffer a detection from a sensor that no one is interested in.
This behaviour is very useful to save memory if I have a lot more publish topics than subscriptions and these sensors send their detection at high frequency.

Of course, the drawback is the same as the case before: If, at a certain point in the future, it will be added a subscription interested in a temporal analysis of sensor5, the data sent until that moment, by sensor5, won't be in the buffer (and the detection set for sensor5 will be empty), but again even in this case we preferred to optimize memory usage, instead of taking in the buffer more data than needed, for future use.

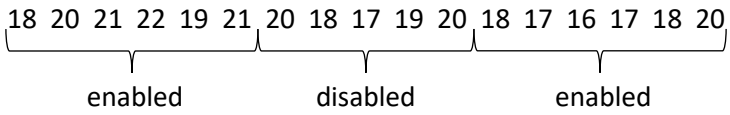
Disable buffering if there is not anymore an interested subscription

As explained in the subscriptions buffering, whenever a client unsubscribes to a certain topic, that topic is removed from the subscriptions buffer. This is true also when a client with a clean session disconnects: all the topics of subscriptions made by that client are removed from the buffer.
So, with this logic, it can happen that at a certain point all the "temporal subscriptions" (subscriptions with a temporal operation) are removed.
In this case, the temporal buffer is disabled and no more data is buffered until a new temporal subscription arrives.
When this happens, all the detections previously stored in the buffer are deleted. Note that to guarantee a correct functioning, the operation of deleting detections is not optional and we can't leave the detections there for future use.

Now, to understand why it's mandatory to delete the detections, let's suppose that the detections are not deleted and that the following sequence of events happen:

1. A client makes a temporal subscription on sensor1 and this is the only temporal subscription in the subscriptions buffer
2. The temporal buffer is enabled
3. 1 hour of time passes
4. The client unsubscribes from that subscription
5. The temporal buffer is disabled
6. Another client makes a temporal subscription (with a window size of 2 hours) on sensor1
7. The temporal buffer is re-enabled

Let's suppose also that the following detections have been sent by sensor1 in the meantime that the previous events happened:



Then, whenever the client that made the subscription at step 6 wants to elaborate its subscription, since it has a window size for the temporal operation equal to 2 hours (big enough to include all values sent), takes all the values in which the buffer was enabled and it computes the operation on them, but this is wrong, because the correct computation would need to consider also the values received when the buffer was disabled, values that unfortunately are not available since they were not buffered. So, since a user would expect that a temporal operation is computed on a continuous sequence of detections, without gaps due to buffer enabling and disabling, this behaviour must be avoided.
Then, the only correct solution is to delete all the detections when the buffer is disabled (this also frees the memory occupied by the buffer with a potentially high saving in term of RAM used) and starting from a clean state (the empty buffer) when the buffer needs to be re-enabled again.
Also, to be consistent with the previous technique, if there is not anymore at least one subscription interested in a certain sensor, all the detections stored of that sensor are removed from the buffer.

4.5.4.4 – Considerations on techniques to save memory

Even with the previous techniques employed, it can be possible that anyway the buffer saturates all the available RAM of the broker, because the previous techniques are useful to free memory as soon as possible, but they don't fix an upper bound of the RAM used by the buffer.
Unfortunately, to fix an upper bound, we must fix a maximum value for each one of the variables that contribute to increase the buffer size. So, let's report here the formula of the temporal buffer size:

$$Buffer\ size\ \approx\ (TS + \lambda \cdot T \cdot DRS) \cdot N$$

As we said before, the variables are 4: TS, λ, T and N.

To fix a limit on these variables it's necessary to know the context in which the broker will operate.
For example, in order to fix:

- **TS**: We must know how is the structure of the topics used to identify each sensor/publisher
- **λ:** We must know if we can control the behaviours of each sensor that send messages to the broker or we can't and in the case we can, we must establish what is an acceptable sending frequency for our context
- **T:** We must know what is the maximum interval of time that a client is interested in for temporal operations
- **N:** We must decide how many sensors can connect to the same broker

So, since our broker was made to be used in any context in which the MQTT protocol can be used, these choices must be left to the administrator of the broker that knows the application context and broker resources. Knowing this, he can set their values accordingly.

4.6 – Complex operations

For complex operations we mean computationally intensive operations.
In our case, the only complex elaboration is the image people detection process, but this of course, is a concept that can be extended for any other kind of complex operation.

4.6.1 – Complex operation processing

Before processing a complex operation, the broker logic evaluates if this computation is really necessary or, in other words, if the result of this computation will be used by any subscription or not.

For example, let's suppose that an image, sent by a sensor, arrives to the broker. Before extracting the number of people (or males, or females) from the image it is checked that at least one subscription in the subscriptions buffer has an operation `$COUNTPEOPLE` (or `$COUNTMALE`, or `$COUNTFEMALE`).
If this condition is satisfied, then the elaboration is started, otherwise the image is not elaborated, saving computational resources.

4.6.2 – Complex operation caching

Caching the result of a complex operation is a required thing to save even more computational power. Suppose that:

- An image sensor publishes a message with topic:
  `img/polimi/deib/room1/imgSensor1`

- There are 100 clients subscribed to the same topic:
  `$COUNTPEOPLE/img/polimi/deib/room1/imgSensor1`

Without caching, we would have to elaborate the published image 100 times, one time for each subscription made. Instead, with caching, we can extract the number of people just once, save the result in the complex operation cache and if the result it's needed by any other subscription, the value stored in the buffer is read.
With this approach, it's easy to see that if there is only one subscription requiring image elaboration or if there are 100, the computational time required is the same, because the elaboration is always done one time only. This saves a lot of computational resources!

The cache structure is very simple and for the case of the `$COUNTPEOPLE` operation it can be imagined as a table like this one:

Publish Message ID	People count
1255	5
1758	8

Table 31: Cache structure example

As we will see in the implementation part, the ID of the published message is known through all the elaboration process and so, whenever the `$COUNTPEOPLE` operation is required the cache is accessed using the publish id as key.
Please note that this table won't cause memory issues, because a record of the table is deleted as soon as the elaboration of all the interested subscriptions is terminated for that publish message.
Note also that the only case in which the table can have multiple records is the case in which two or multiple publish messages are elaborated simultaneously (using multi-threading).

Chapter 5 - MQTT+ Implementation

In this chapter we are going to see how the logic explained in the previous chapter is actually implemented.

First of all, it's important to know that in HiveMQ everything that happens in the broker can be handled by registering a callback that the broker can call when the specific event happens.

In particular, we handle the following events:

- **Publish Received:** This event is raised whenever a publisher publishes a message. It gives information about the published message (payload, topic, QoS, etc..) and about the publisher (ClientID)
- **Client Connect**: This event is raised whenever a client connects to the broker. It gives information about the connected client (ClientID) and on the session type (clean or persistent).
- **Client Disconnect**: This event is raised whenever a client disconnects from the broker. It gives information about the disconnected client (ClientID) and if the disconnection was made gracefully with a MQTT DISCONNECT message or if there was a tcp connection loss.
- **Subscribe Received:** This event is raised whenever a client subscribes to one or more topics. It gives information about the client that made the subscription (ClientID) and about all the topics that the client wants to subscribe to.
- **Unsubscribe Received:** This event is raised whenever a client unsubscribes to one or more topics. It gives information about the client (ClientID) and about the topics that the client wants to unsubscribe from.

5.1 – Subscription processing

After the interested clients have connected to the broker, they have to subscribe to the interested topics. Let's see how a subscription is accepted in the MQTT+ logic.

5.1.1 – Involved entities

- **Client:** A generic client that subscribes to a certain topic
- **SubscribeReceived:** The class that handles the subscription event
- **Subscription:** The class that represents a valid subscription. It also has methods to parse a subscription.
- **SubscriptionManager:** The class that has all the valid subscriptions made by clients (it implements subscription buffering)
- **MessageStore:** The class that manages all the existing buffers (one for each semantic type). It has all the operations that allow to read and write buffered data.

The following sequence diagram shows the interactions between these entities.

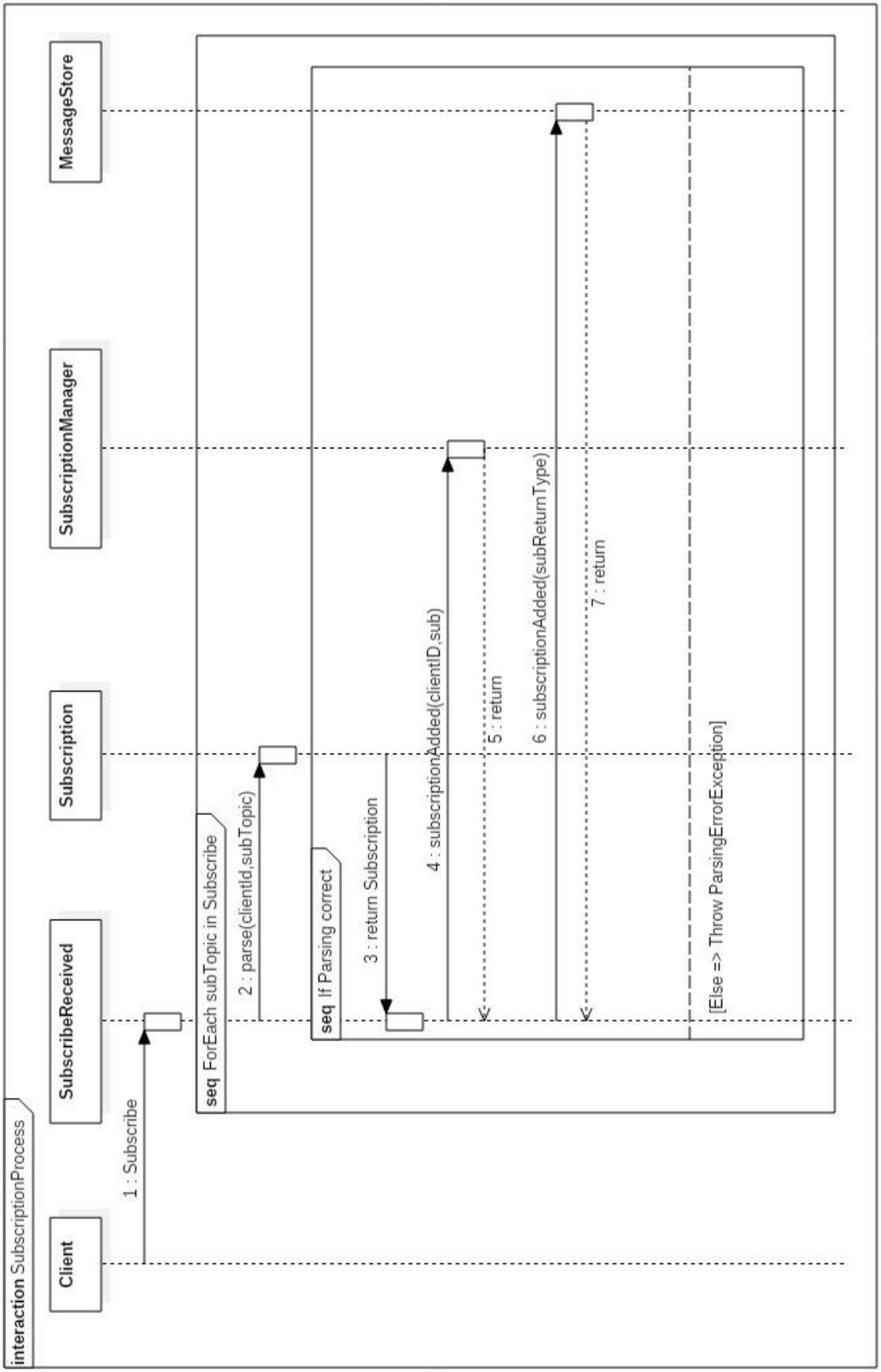


Figure 15: Subscription process

5.1.2 – Elaboration steps

1. The client sends the subscription message to the broker, specifying all the interested topics.
2. Parse the topic and verify if it's syntactically and semantically valid. If parsing isn't correct, throw ParsingErrorException and end computation
3. Get the initialized subscription object
4. Call the SubscriptionManager to add the new valid subscription in the subscription buffer.
5. Notify the MessageStore that a new subscription has been added to the subscription buffer. This notification is needed to verify if the new subscription is a temporal subscription and, in that case, enable the temporal buffer associated to the semantic data type of the subscription, if it was disabled.
6. If there are other topics, return to step 2

5.2 – Subscription parsing

Let's focus now on what happens on subscription parsing.

5.2.1 – Involved entities

- **SubscriptionSemanticAnalyzer:** The class that analyses the semantic of a subscription and it has a method that computes the data type returned by the OperationsBlock sequence of the subscription.
- **Subscription:** The class that represents a valid subscription.
- **SubscriptionSyntaxAnalyzer:** The class that analyses the syntax of a subscription.
- **ElaborationBlockInfo:** The class that contains all the information regarding a single operation (periodicity, ID, parameters, etc..)

5.2.2 – Elaboration steps

1. The class SubscriptionSyntaxAnalyzer is initialized with the subscription topic to be parsed.
2. The structure of the topic is checked to verify that it respects the MQTT+ syntax (it starts with $, it has at least one operation,etc..). If it doesn't, then the SyntaxAnalysisException is thrown and the parsing stops.
3. Take the first elaboration block in the OperationsBlock and initialize the ElaborationBlockInfo with it.
4. Check if the string representing the elaboration block is a valid operation, specified with the correct syntax or not. If it's not, then the SyntaxAnalysisException is thrown and the parsing stops.
5. Add the parsed elaboration block to the list that contains all the elaboration blocks in the OperationsBlock
6. Move to the next elaboration block (the next operation starting with $)
7. Repeat from step 4, until all the elaboration blocks are analysed.
8. Start semantic analysis to get the returned semantic type of the subscription
9. If the subscription is semantically valid, then stop the parsing procedure and initialize the Subscription object with the topic passed and the results from semantic and syntax analysis (the elaboration block info list and the returned data type). If it's not, then throw the SemanticAnalysisException and stop parsing.

The following sequence diagram shows all the steps explained here.

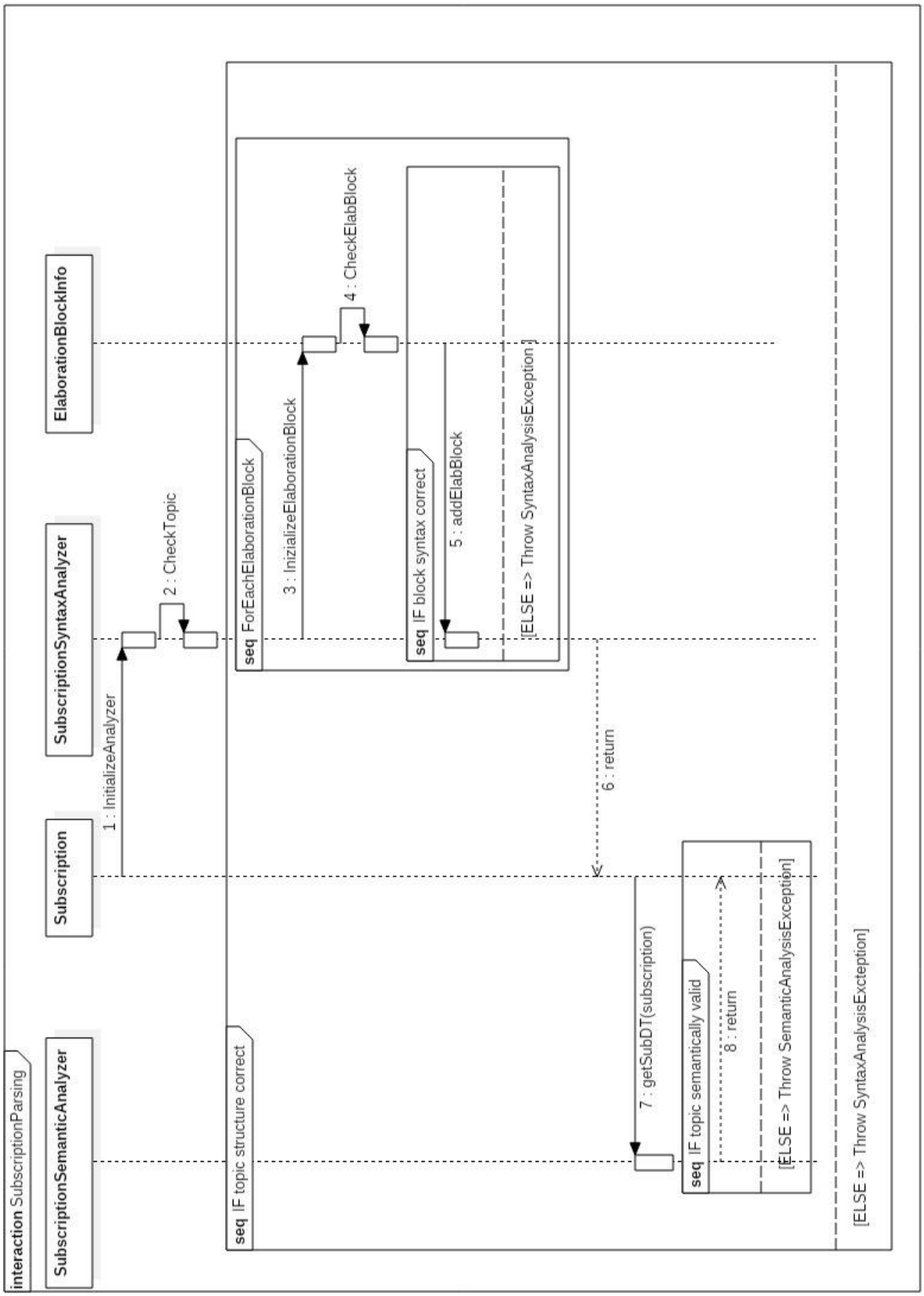


Figure 16: Subscription parsing

5.3 – Subscription elaboration

After a client subscribed to all the topics it is interested in, each one of those subscriptions, at a certain moment, needs to be elaborated. There can be two types of elaborations:

- **Immediate:** It applies to subscriptions that doesn't contain any periodic operation and It starts whenever a sensor publishes a message with a topic that matches the subscription topic.
- **Periodic:** This applies to periodic subscriptions and it starts whenever the interval of the subscription periodicity (QUARTHERHOURLY, HOURLY, DAILY) has passed.

5.3.1 – Immediate elaboration

5.3.1.1 – Involved entities

- **SubscribedClients:** All the clients that subscribed to one or more topics, matching the topic of the published message.
- **Sensor:** A generic sensor that publishes a message
- **PublishReceived:** The class that handles the publish event
- **InformationExtractor:** The class that extract some information from complex data structures (e.g. images) and buffers the extracted information or the published numeric value.
- **PublishRedirect:** The class that establishes which clients are interested in the published message and after each matching subscription has been elaborated, it sends the results of the various elaborations to the correct clients.
- **PublishVirtualizer:** The class that elaborates the OperationsBlock of each matching subscription

5.3.1.2 – Elaboration steps

1. A sensor sends a message to the broker
2. The InformationExtractor class analyses the payload of the message and establishes if it's an image, a number or a generic string. If it's an image, it extracts useful information from it and it buffers that data, if it's a number, it just buffers the publish data and if it's a string, nothing is buffered (we store only numbers: published or extracted from images).
3. The PublishRedirect class is called and it establishes the subscriptions that match with the publish topic
4. The OperationsBlock is elaborated and the result of this sequence of operations is put into the payload of a publish message (called VirtualPublishMessage, because it contains a payload that it's not the original payload sent, so it's not actually sent by any real sensor).
5. The virtual publish message is then added to a list that contains all the publish messages that the broker must send to the interested clients
6. Repeat from 4 until all subscriptions have been elaborated

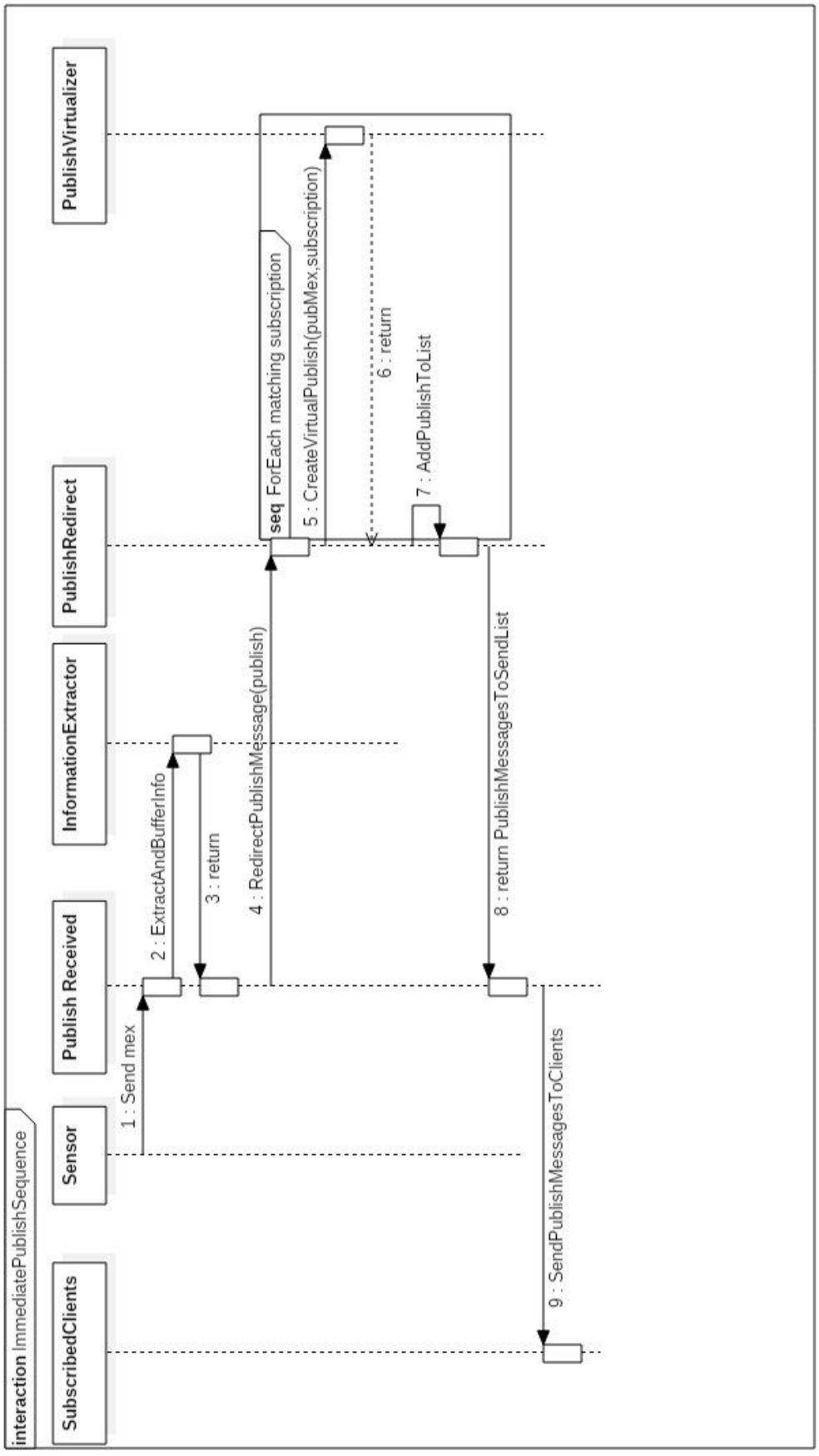


Figure 17: Immediate subscription elaboration

5.3.2 – Periodic elaboration

5.3.2.1 – Involved entities

- **Clients:** Clients that made some periodic subscriptions
- **SubscriptionManager:** The class that has all the valid subscriptions made by clients (it implements subscription buffering)
- **PeriodicTask:** The class that periodically sends the results of periodic operations to the interested clients.
- **PublishVirtualizer:** The class that elaborates the OperationsBlock of each subscription

5.3.2.2 – Elaboration steps

1. 15 minutes have passed
2. Takes all the periodic subscriptions with QUARTERHOUR as periodicity
3. The PublishVirtualizer class creates all the publish messages that need to be sent to the interested clients
4. The PeriodicTask sends all these messages to the interested clients
5. Repeat from 3 until all periodic subscriptions (with QUARTERHOUR as periodicity) have been elaborated

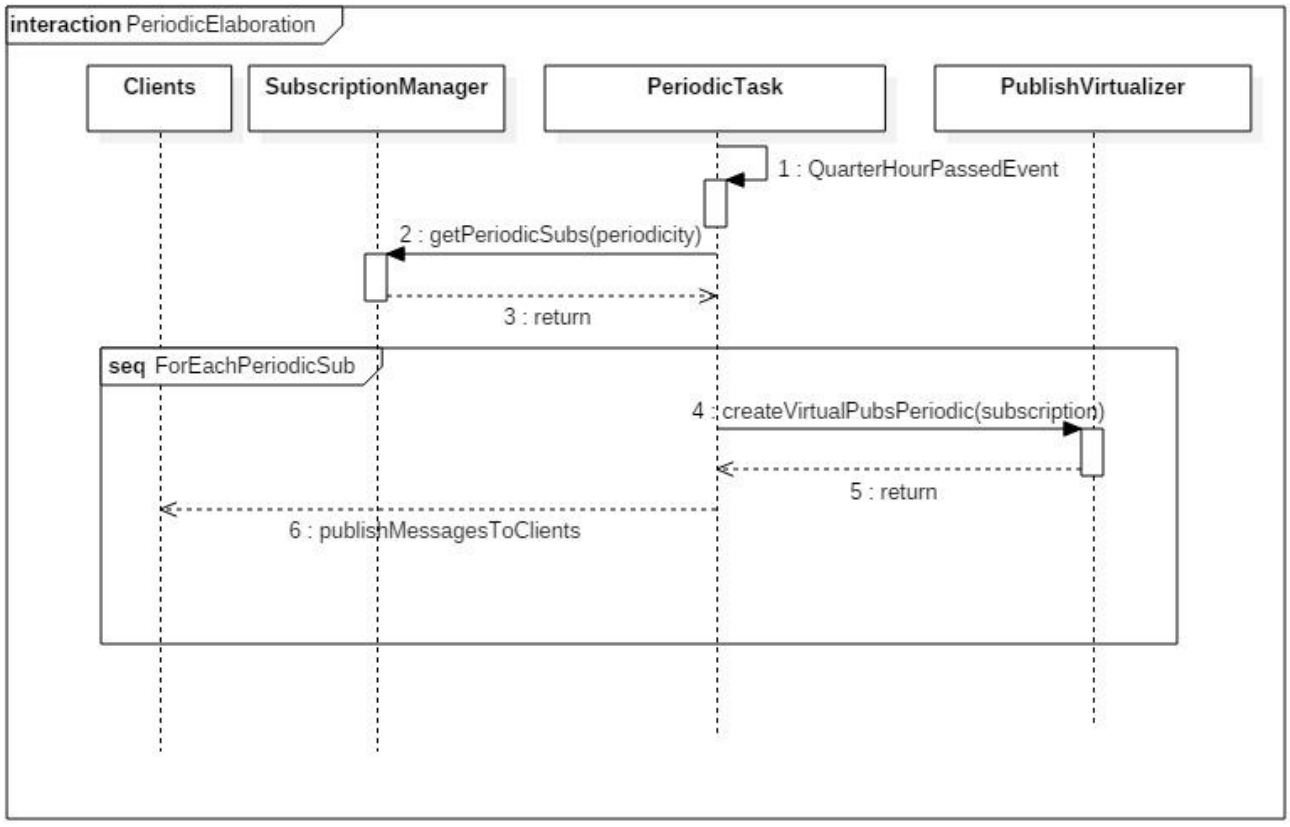


Figure 18: Periodic subscription elaboration

5.4 – Information extraction and buffering

Now we are going to focus on how the information is extracted and buffered. In the previous sequences we have seen that this task is given to the InformationExtractor. Let's see now how it works.

NOTE: We supposed to use OpenCV[20] as image library that has only the CountPeople function. If COUNTMALE or COUNTFEMALE are needed, Face++[19] must be used, but the working principle is practically the same to the one of OpenCV, so it won't be shown here.

5.4.1 – Involved entities

- **InformationExtractor:** The class that extract some information from complex data structures (e.g. images) and buffers the extracted information or the published numeric value.
- **ImageAnalyzer:** The class that actually extracts information from images
- **ComplexObjectsBuffer:** The class that implements complex operations caching
- **MessageStore:** The class that manages all the existing buffers (one for each semantic type). It has all the operations that allow to read/write data from/to the buffer.
- **NumericBuffer:** The class that implements a numeric buffer.

5.4.2 – Elaboration steps

1. The InformationExtractor establishes what kind of data the payload contains (string, image, number)
2. In case it's a string, return and end the elaboration, because the string doesn't need to be buffered or elaborated
3. In case it's a number, set the valueToBuffer to the published value (remember that the InformationExtractor is always called by the PublishedReceived class after a publish) and set the data type to Numeric
4. In case it's an image:

4.1 Extract the number of people from the image
4.2 Cache the result in the ComplexObjectsBuffer to be re-used by other subscriptions
4.3 Set the valueToBuffer to the extracted number of people
4.4 Set the bufferDataType to CountPeople

5. Call the MessageStore to store the valueToBuffer
6. The MessageStore class takes the right buffer to put the valueToBuffer in, checking if the data type associated to the buffer it's the bufferDataType
7. Store the valueToBuffer in the selected buffer

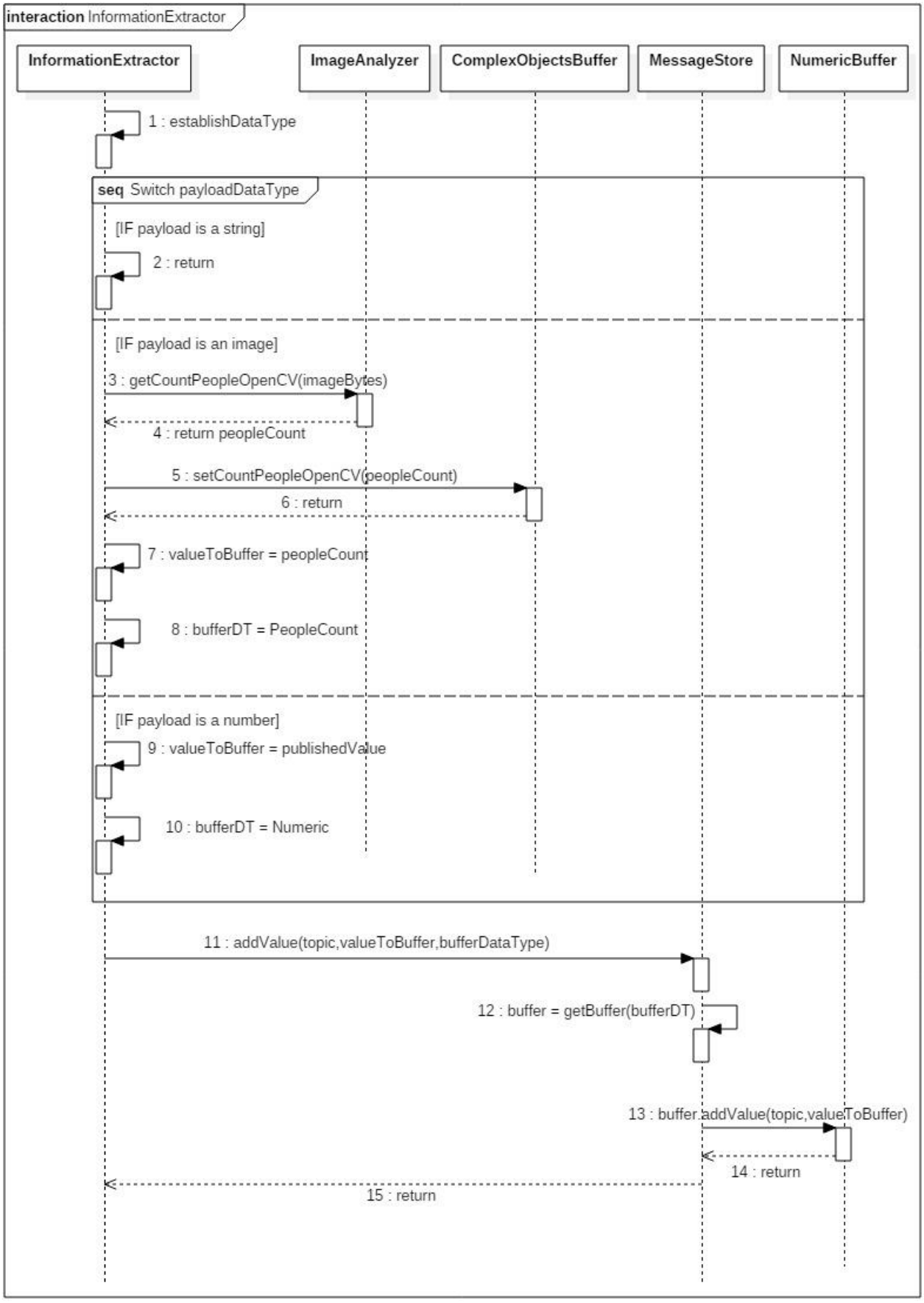


Figure 19: Information extraction and buffering

5.5 – Operation elaboration

Now let's see how an operation in the OperationsBlock is computed.
All the possible operations that we can specify in a subscription are split in three different categories.

Operation type	Operations list	Notes
`Arithmetic`	`AVG,SUM,MIN ,MAX,COUNT`	In this category, there are also the operations that contains these tokens in their name. For example `TMPAVG;00:01:00` is arithmetic, because it contains `AVG` and also `QUARTERHOURMIN` it's in this category because it contains `MIN`. This has a logic, because independently on which set of data these operations work on or the moment in which these operations are executed, they are always arithmetic operations.
`OnImage`	`COUNTPEOPLE, COUNTMALE, COUNTFEMALE`	In this category, there are all the operations that work on images
`RuleBased`	`GT,GTE,LT,LTE, EQ,NEQ,CONTAINS`	In this category, there are all the operations that filter the input data, using some kind of rule to decide which data let through or block.

Table 32: Operation types

Let's see how the operations of these three categories are computed.
Note that for arithmetic operations we have to split the explanation in two different categories:

- **Not periodic:** The operation takes the input data from the buffer or from the previous operation
- **Periodic:** The operation takes the input data from the stored statistics.

5.5.1 – Not periodic arithmetic operation

5.5.1.1 – Involved entities

- **PublishVirtualizer:** The class that elaborates the OperationsBlock of each subscription
- **ArithmeticElaborationService:** The class that prepares the data to be computed by the ArithmeticOpsElaboration class.
- **ArithmeticElaborationOps:** The class that has the logic to compute all the arithmetic operations, starting from a given data set.

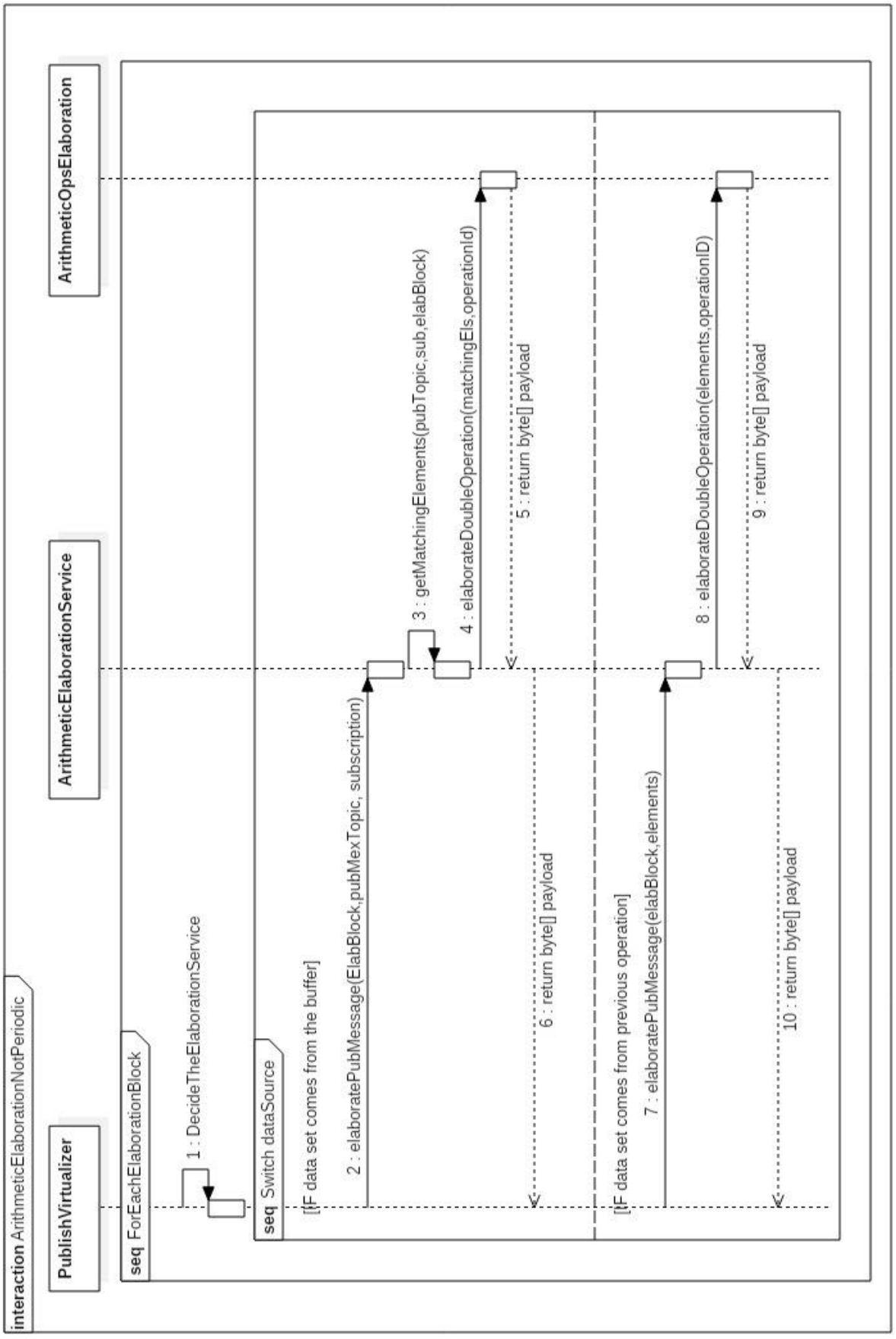


Figure 20: Arithmetic elaboration not periodic

5.5.1.2 – Elaboration steps

1. The operation is analysed to identify if it's an Arithmetic, OnImage or RuleBased operation. Of course, in this case, it is an Arithmetic operation.
2. If the operation takes the data set from the buffer:

2.1 The ArithmeticElaborationService is called with: The elaboration block, the publish message topic and the interested subscription

2.2 Take from the buffer all the elements with a topic that matches with the specified subscription topic

2.3 Call the ArithmeticOpsElaboration to compute the arithmetic operation on the retrieved data set

3. If the operation takes the data set from the previous operation:

3.1 The ArithmeticElaborationService is called with the elaboration block and the elements to process

3.2 Call the ArithmeticOpsElaboration to compute the arithmetic operation on the elements passed

5.5.2 – Periodic arithmetic elaboration

Note that in this sequence diagram, we suppose that the required operation is an AVG operation, but for other operations the steps would be the same, it would be only necessary to change the method called for the StatisticsManager.

5.5.2.1 – Involved entities

- **PublishVirtualizer:** The class that elaborates the OperationsBlock of each subscription
- **ArithmeticElaborationService**: The class that prepares the data to be computed by the ArithmeticOpsElaboration class.
- **ArithmeticElaborationOps:** The class that has the logic to compute all the arithmetic operations, starting from a given data set.
- **StatisticsManager:** The class that manages the statistics in the last detection buffer.

5.5.2.2 – Elaboration steps

1. The ArithmeticElaborationService is called with the elaboration block and the interested subscription
2. Call the ArithmeticOpsElaboration to get all the averages matching the subscription topic
3. Call the StatisticsManager, with the interested subscription, to retrieve the required statistics

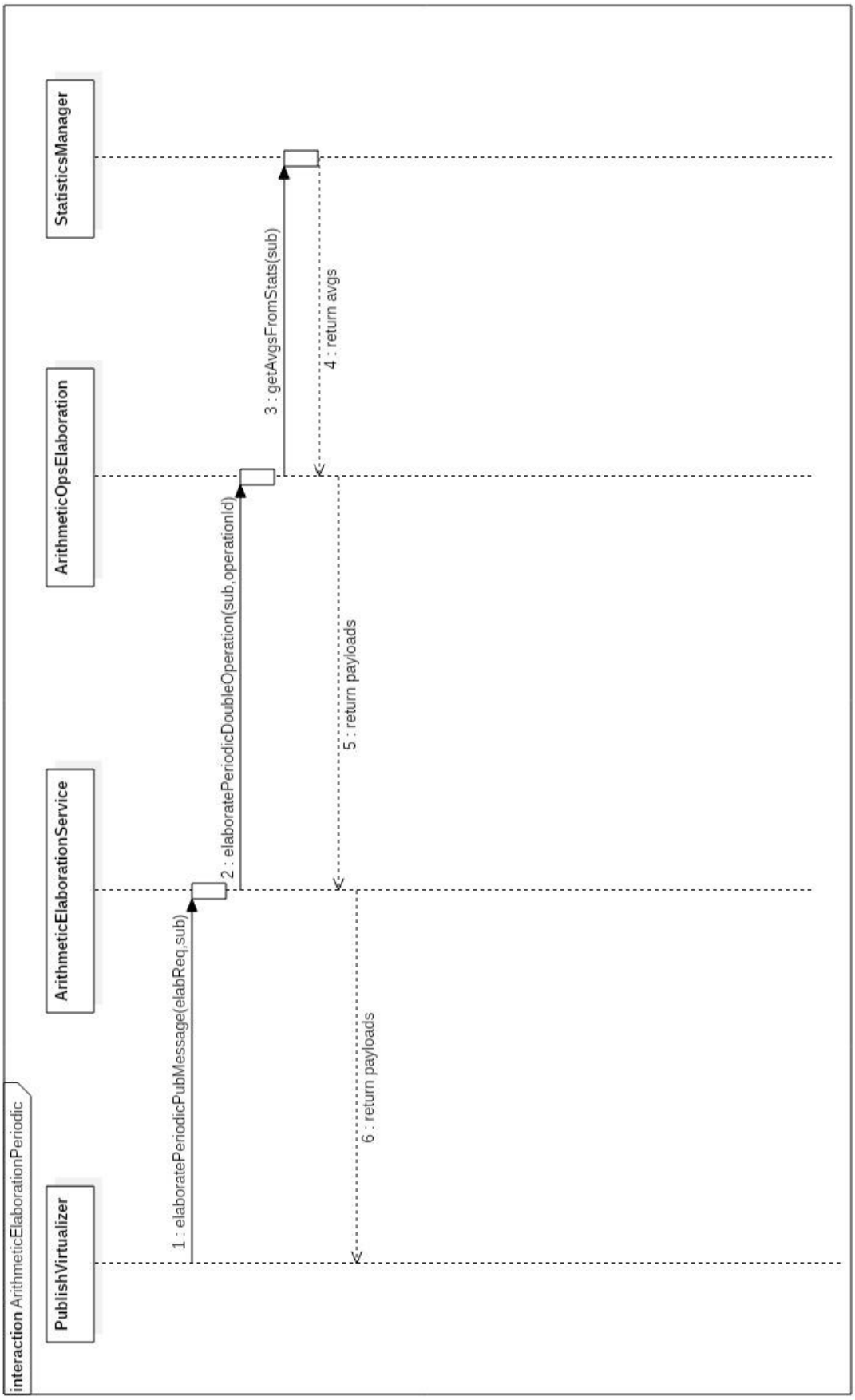


Figure 21: Arithmetic elaboration periodic

5.5.3 – OnImage elaboration

For this type of operation, we suppose to elaborate a COUNTPEOPLE operation and to use OpenCV as image library.

5.5.3.1 – Involved entities

- **PublishVirtualizer:** The class that elaborates the OperationsBlock of each subscription
- **OnImageElaborationService:** The class that prepares the data to be computed by the OnImageOpsElaboration class.
- **OnImageOpsElaboration:** The class that has the logic to compute all the OnImage operations.
- **ComplexObjectsBuffer:** The class that implements complex operations caching

5.5.3.2 – Elaboration steps

1. The operation is analysed to identify if it's an Arithmetic, OnImage or RuleBased operation. Of course, in this case, it is an OnImage operation.
2. The OnImageElaborationService is called with the elaboration block, publish message topic, the interested subscription and the publish id useful to get the cached image elaboration result
3. Call the OnImageOpsElaboration to compute the OnImage operation
4. Call the ComplexObjectsBuffer to get the cached computation for the image operation

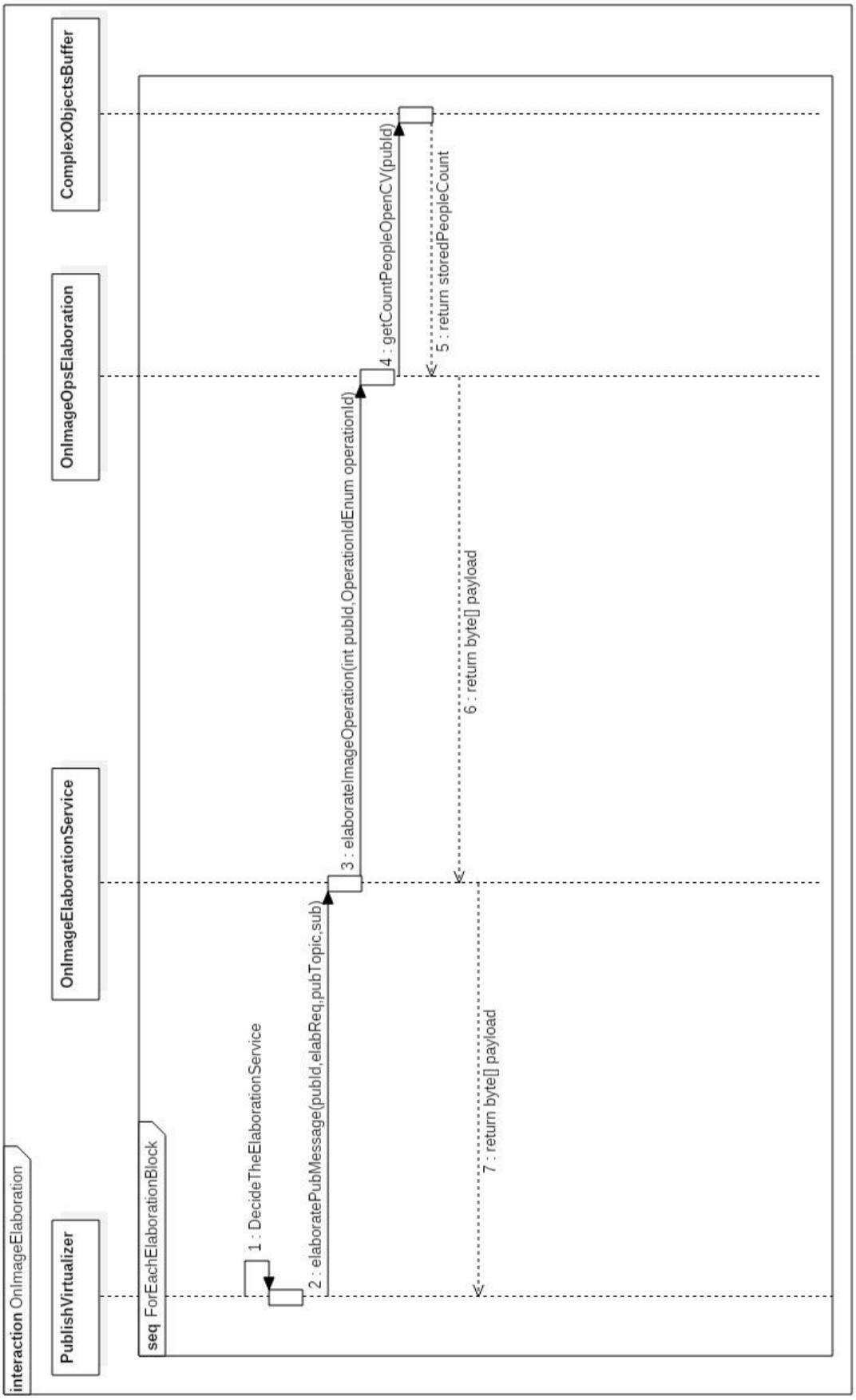


Figure 22: OnImage elaboration

5.5.4 – RuleBased elaboration

5.5.4.1 – Involved entities

- **PublishVirtualizer:** The class that elaborates the OperationsBlock of each subscription
- **RuleBasedElaborationService:** The class that prepares the data to be computed by the RuleBasedOpsElaboration class.
- **RuleBasedOpsElaboration:** The class that has the logic to compute all the RuleBased operations.

5.5.4.2 – Elaboration steps

1. The operation is analysed to identify if it's an Arithmetic, OnImage or RuleBased operation. Of course, in this case, it is a RuleBased operation.
2. The RuleBasedElaborationService is called with the elaboration block and the payload of the published message
3. Call the RuleBasedOpsElaboration to compute the RuleBased operation
4. If the specified rule is satisfied, return the published payload as it is, otherwise throw a BlockComputationException. This exception causes the elaboration process of the subscription to stop.

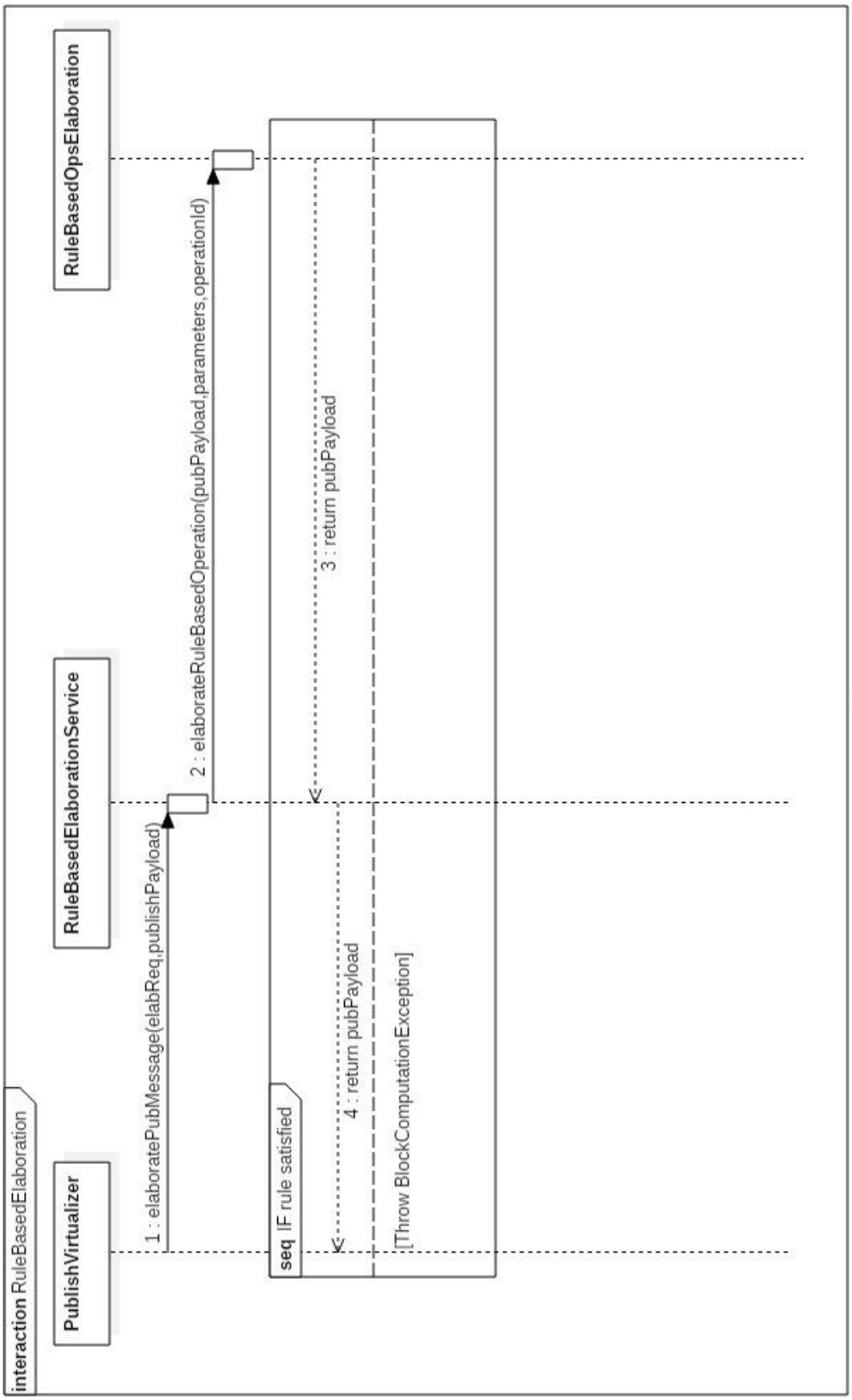


Figure 23: RuleBased elaboration

Chapter 6 – Performance tests

In this chapter we are going to discuss the performance tests that have been made to compare the resources used in the case of the non-modified broker with MQTT and the modified broker with MQTT+ .

First of all, let's see how these tests have been made.

6.1 – Technical specifications

The tests have been executed on a PC with the following specifications:

- **CPU:** Intel i7-6700@3,4 GHz
- **RAM:** 8 GB
- **Storage technology:** SSD
- **O.S.:** Windows 10 Pro (Version: 1803)

The CPU execution time and RAM usage have been monitored using the Get-Process[21] command, used within a PowerShell script, instead the Bandwidth used has been monitored using the tshark[22] command line tool.

The clients and sensors have been generated using two simple applications created ad hoc in java.
These applications generate the required clients and sensors for each test case. The MQTT clients and sensors are generated using the Eclipse Paho for Java library[23].

6.2 – Convention on CPU load

Since we noticed that the CPU execution time, given by the Get-Process command, varies a lot depending on the sending frequency of the sensors, we normalized the value obtained by a reference value in this way:

$$C_{n,m} = \frac{C_{n,m}}{C_{\mathrm{MQTT}1,1}}$$

Where n specifies the number of sensors and m the number of clients used.
Basically, this definition normalizes the CPU execution time (returned by the get-process command) by the CPU load with 1 sensor and 1 client in the case of MQTT. In this way, if the various absolute CPU executions time change, the normalized value stays the same, since the proportion between the absolute value and the reference value remains the same.

The comparison has been made on three types of resources: CPU, RAM and Bandwidth.
For each resource considered the comparison has been split in two, based on the type of data that sensors sent to the broker: Numbers or Images.
In the following sections we will show the comparisons for different number of sensors (the publishers) and clients (the subscribers).

6.3 – Numbers

For these tests, the following topics have been used:

- **Publish topic:** `numeric/polimi/deib/room1/sensorID`
- **Subscription topic:** `$AVG$QUARTERHOURLYAVG/numeric/polimi/deib/room1/+`

The sensorID used in the publish topic is different for each sensor used (Sensor1,Sensor2,Sensor3,etc..)

Since the $QUARTERHOURLYAVG operation would have required to wait 15 minutes for each test (with n sensors and m clients), the computation of all the required tests, with real timing, would have taken too long.
For this reason, the timings have been rescaled dividing the real timings by 15. So, since in a real context we supposed that sensors sent their detections every 20 seconds, we obtain the current rescaled timings:

- **$QUARTERHOURLYAVG:** One message every (15/15 minutes = 1) minute
- **Sensors sending interval:** One message by each sensor every 1,333 (20/15) seconds

NOTE: All messages are sent with QoS 0. This is done to avoid the influence of the acks, that higher QoS have, in the bandwidth
NOTE 2: In the bandwidth comparison we compare only the downlink usage, since the uplink usage is the same for MQTT and MQTT+, because in both cases the detections must be sent from the sensors to the broker.

6.3.1 – Bandwidth usage

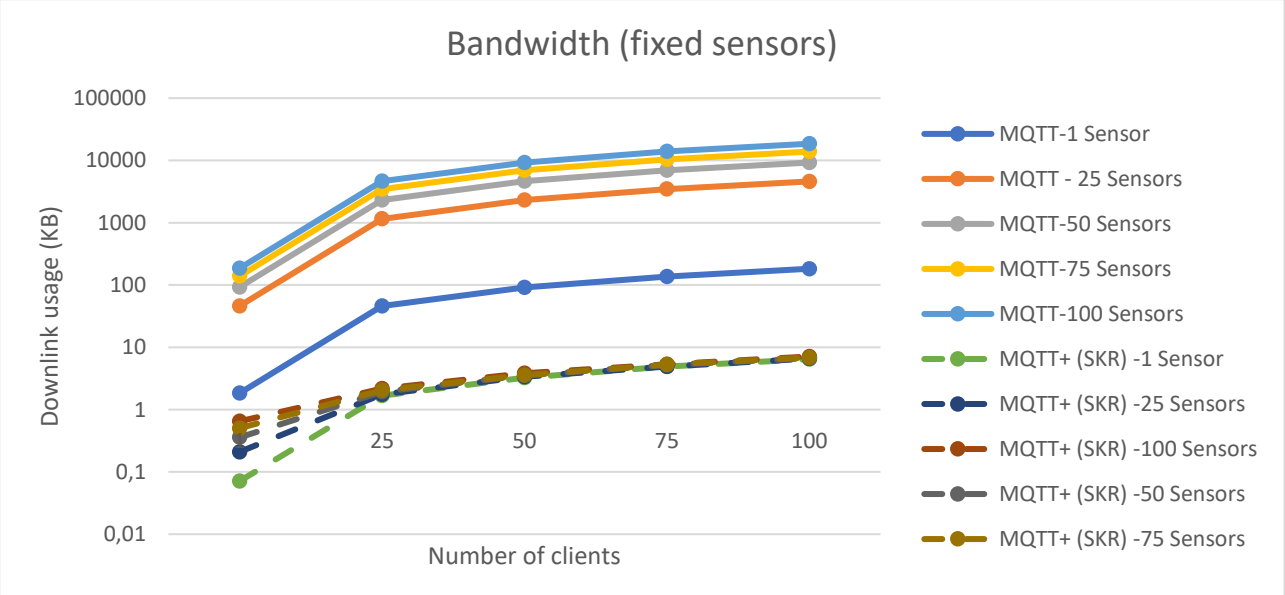


Figure 24: Numeric bandwidth usage - fixed sensors

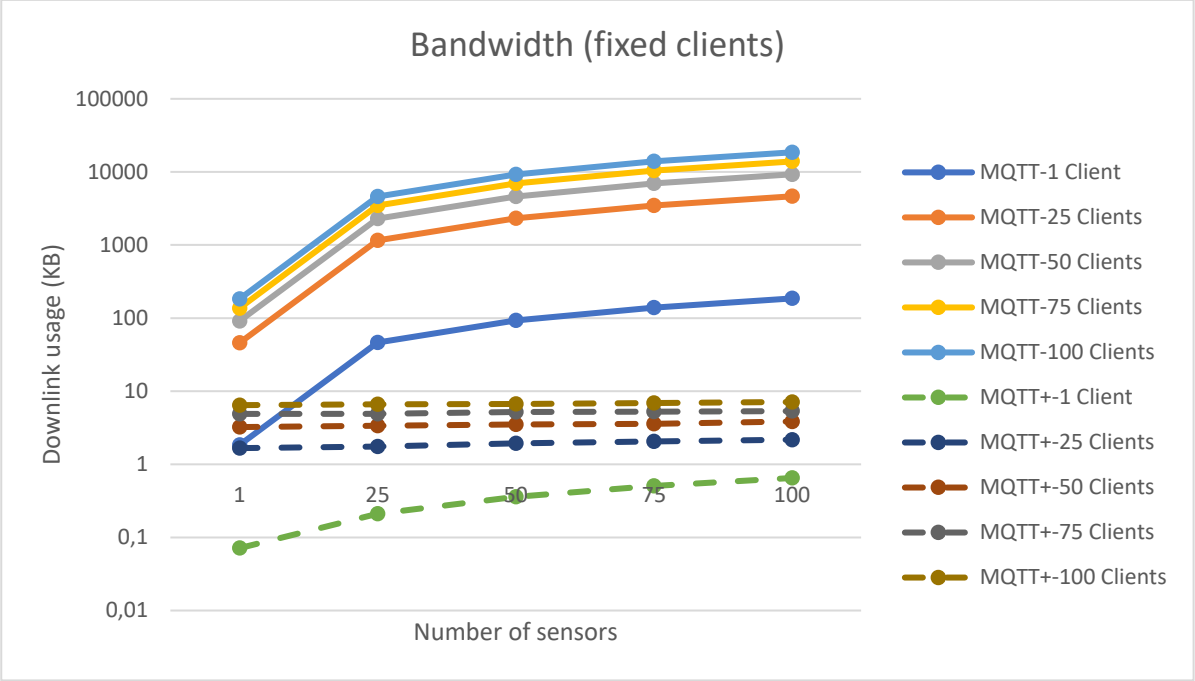


Figure 25: Numeric bandwidth usage - fixed clients

From these two charts we can see that the MQTT+ always uses less bandwidth than the standard MQTT. Also, as expected, the bandwidth used increases as much as we increase the number of clients and this is valid both for MQTT and for MQTT+.
The difference in MQTT+ is that the bandwidth used remains constant, increasing the number of sensors. We can see this behaviour looking to the second chart. In this chart, if we fix a certain number of clients (we fix the line we are looking to), changing the number of sensors doesn't affect the bandwidth and this can be easily noted by the fact that the MQTT+ lines are horizontals for any number of sensors.
This behaviour, apparently, seems to be not respected by the curve with just 1 client, but this is caused by the fact that for just one client, there is only one message to send at the end of the minute interval, so the bandwidth consumed by the message payload is minimal (around 50 bytes). The other bytes used, depend by the acks exchanged for the messages, because even if we are using QoS 0 (with no acks), the MQTT protocol relies on the TCP protocol that requires an ack message for each packet sent, so increasing the number of packets published (obtained by increasing the number of sensors that publish data), the acks that the broker must send back to the publishers increase and this contributes to minimally increase the bandwidth used.
In the other cases (from 25 to 100 clients) the bandwidth used by the payload of the messages sent is higher than the one used by the TCP acks, so the first component "hides" the second one and since the first one doesn't depend by the number of sensors (it is always sent just one packet in the test interval), we can see the expected behaviour for the bandwidth: constant bandwidth consumption for every number of sensors.

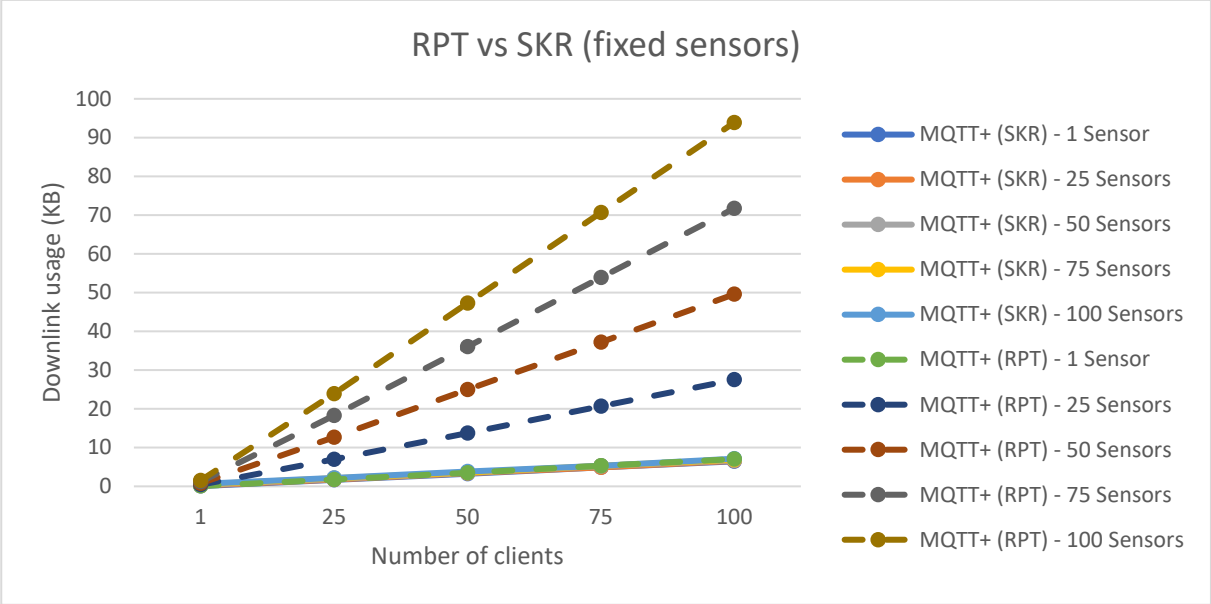


Figure 26: RPT vs SKR - fixed sensors

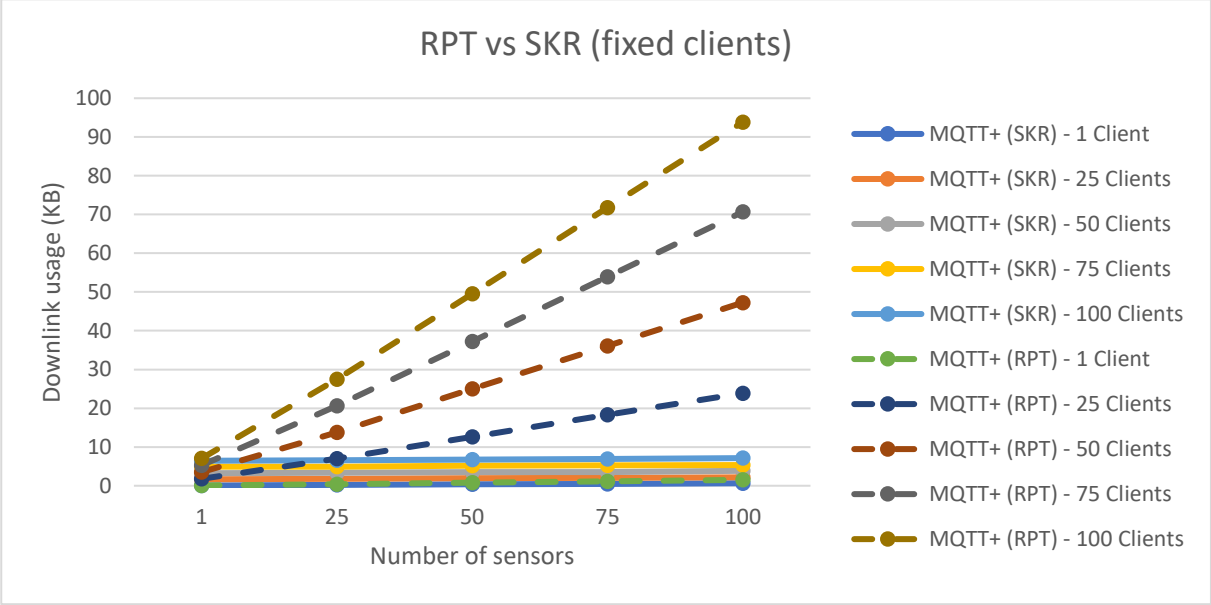


Figure 27: RPT vs SKR - fixed clients

These last two charts show a comparison between the SKR and RPT method, for the publishing topic.
From the first chart we can see that both methods bandwidths depend by the number of clients used, but in case of SKR, the slope of the lines is smaller and so the SKR consumes less bandwidth than RPT for any number of clients or sensors. This is expected by the logic with which these two methods build their publish topics.
From the second chart it's clear that the SKR bandwidth doesn't depend on the number of sensors (we have horizontal lines), because the publish topic is the same whether if we have 1 or 100 sensors.
Instead, this is not true for RPT that has to list, in the publish topic, all the sensor ids used. For this reason, it's also sensor dependent.

So, **for the specific subscription we made,** we obtain the following comparison for bandwidth:

MQTT+ (SKR)	MQTT
It's independent from the number of sensors	It's not client or sensor independent
It always consumes less bandwidth than MQTT	It always consumes more bandwidth than MQTT+

In terms of gain we obtain these results:

Minimum gain (MQTT+ (SKR) 1-1 vs MQTT 1-1)	Maximum gain (MQTT+ (SKR) 100-100 vs MQTT 100-100)
-1,765 KB; -96%	-18,11 MB; -99%

That is a great advantage in terms of bandwidth, given the fact that we are just considering numbers (with very small payloads) and that for sure in a real environment there will be more than 100 sensors and 100 clients, so this gain will be higher.

Instead for SKR and RPT we obtain the following comparison:

MQTT+ (SKR)	MQTT+ (RPT)
It's independent from the number of sensors	It's not client or sensor independent
It always consumes less bandwidth than MQTT+ (RPT)	It always consumes more bandwidth than MQTT+ (SKR)
It gives less information to the clients	It gives more information to the clients

In terms of gain we obtain these results:

Minimum gain SKR 1-1 vs RPT 1-1	Maximum gain SKR 100-100 vs RPT 100-100
0 KB; 0%	-86,72 KB; -92%

The absolute value of the maximum gain is not that high, but anyway it could be useful for contexts in which bandwidth is a very critical resource.

In general, for subscriptions with an aggregation (e.g `AVG`, `SUM`, etc..) after a periodic operation (e.g. `QUARTERLYHOURAVG`, etc..), like the one we used, we can introduce some theoretical formulas.

If we call:

- **m**: The number of clients
- **n**: The number of sensors
- **λ**: messages/second sent by the sensors
- **λ_A:** messages/second sent by the broker to the clients
- **P:** length of each publish message for MQTT (bytes)
- **T:** Observation interval (seconds)
- **Q:** length of each publish message for MQTT+ (bytes)
- **W:** length of sensorID (bytes)
- **O:** length of the MQTT+ OperationsBlocks

DEF

We define Q as:

$$Q \approx \begin{cases} P + O, & \text{if SKR is used} \\ P + O + W \cdot m, & \text{if RPT is used} \end{cases}$$

$$B_{MQTT} = \lambda \cdot \mathrm{T} \cdot (m \cdot n \cdot P)$$

$$B_{MQTT+} = \lambda_A \cdot \mathrm{T} \cdot (m \cdot Q)$$

$$Absolute\ gain = B_{MQTT} - B_{MQTT+}$$

$$Percentage\ gain = -\frac{Absolute\ gain}{B_{MQTT}} \cdot 100$$

6.3.2 – CPU consumption

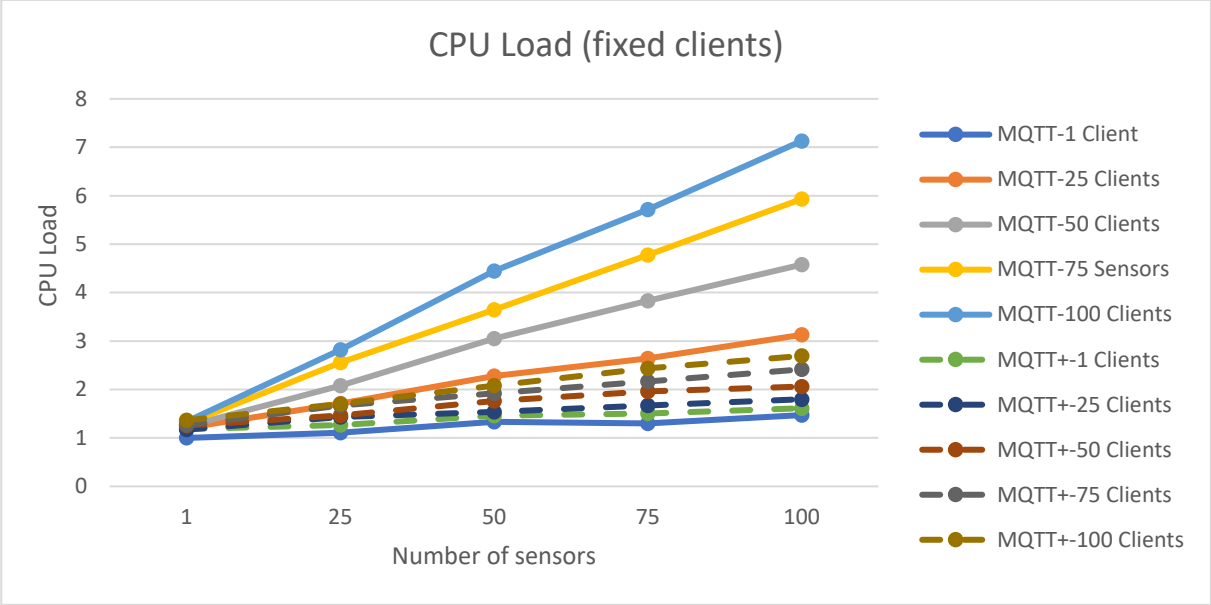


Figure 28: Numeric CPU load - fixed clients

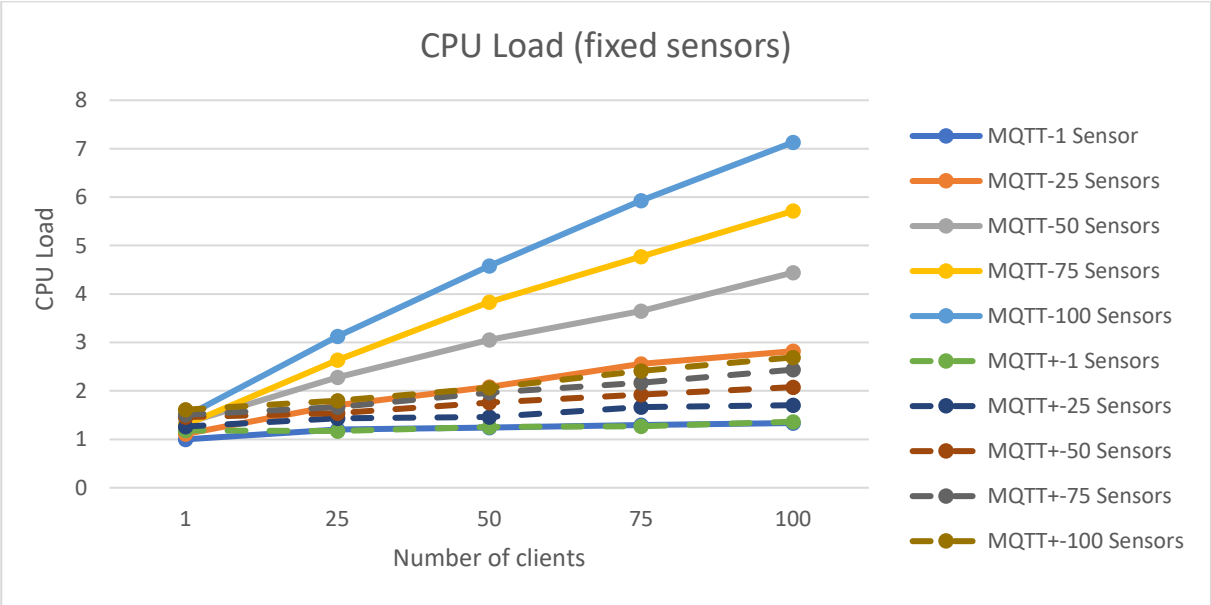


Figure 29: Numeric CPU load - fixed sensors

We see from these charts that the CPU load, for both MQTT and MQTT+, depends from the number of clients and sensors, but we can notice that in the MQTT+ case this load is always less or at most equal (in case of 1 fixed client or 1 fixed sensor) compared to the MQTT.
This is due to the fact that, in case there are numbers to be elaborated, the most complex operation to do is deciding which are the interested clients to a certain publish topic and since in MQTT+ we send less messages (just one per client) than MQTT, as a consequence, the CPU load of MQTT+ will be lower.
Also note that, even in the case of a fixed number of clients (first chart), the CPU load increases as the number of sensors increases. This is because, for each message sent, the MQTT+ logic needs to buffer the data and this contributes to, minimally, increase the computation load.

So, **for the specific subscription we made,** we obtain this gain for the CPU Load:

Minimum gain (MQTT+ 1-1 vs MQTT 1-1)	Maximum gain (MQTT+ 100-100 vs MQTT 100-100)
0; 0%	-5,77; -81%

The gain obtained for 100 sensors and 100 clients is significant.

6.3.3 – RAM Usage

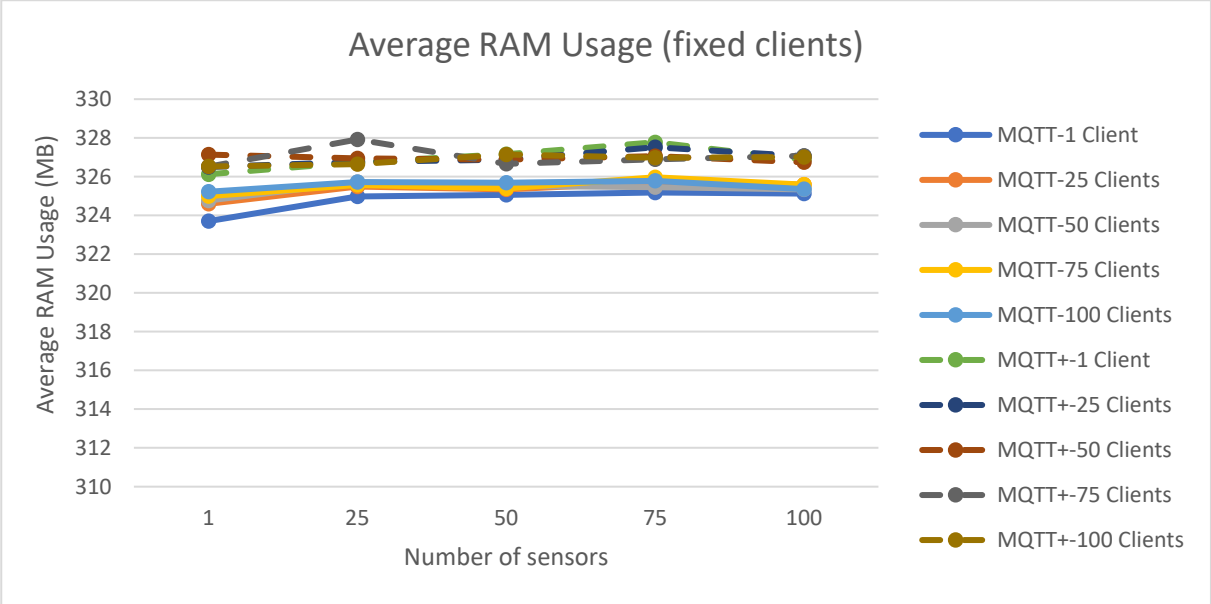


Figure 30: Numeric RAM usage - fixed clients

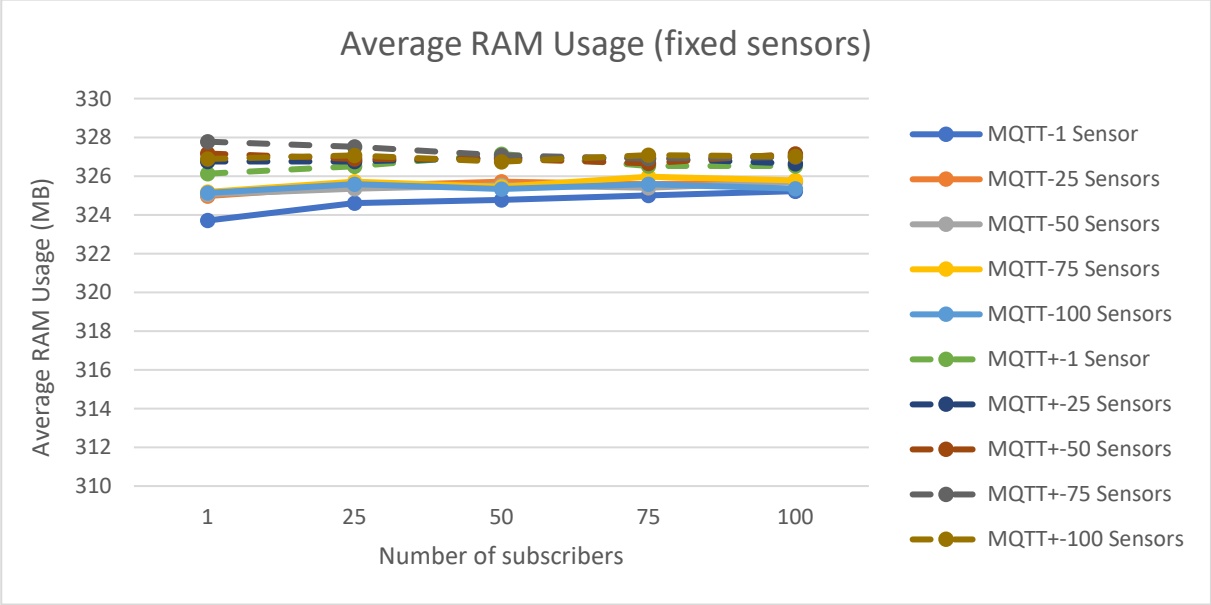


Figure 31: Numeric RAM usage - fixed sensors

From these charts we can see that the RAM consumption remains more or less the same for any number of sensors or clients, moreover the RAM used by the MQTT+ it's quite similar to the one used by MQTT.
This is positive, since it means that the MQTT+ logic doesn't impact much on RAM resources compared to the original broker.

6.4 – Images

For these tests the following topics have been used:

- **Publish topic:** `image/polimi/deib/room1/SensorID`
- **Subscription topic:** `$COUNTPEOPLE/image/polimi/deib/room1/+`

The sensorID used in the publish topic is different for each sensor used (Sensor1, Sensor2, Sensor3, etc..)

The sensors send their images one at a time (sequentially: first sensor1, then sensor 2 and so on..) with a delay between each message of 20 seconds.
Even in this case, all messages are sent with QoS 0.

6.4.1 – Bandwidth usage

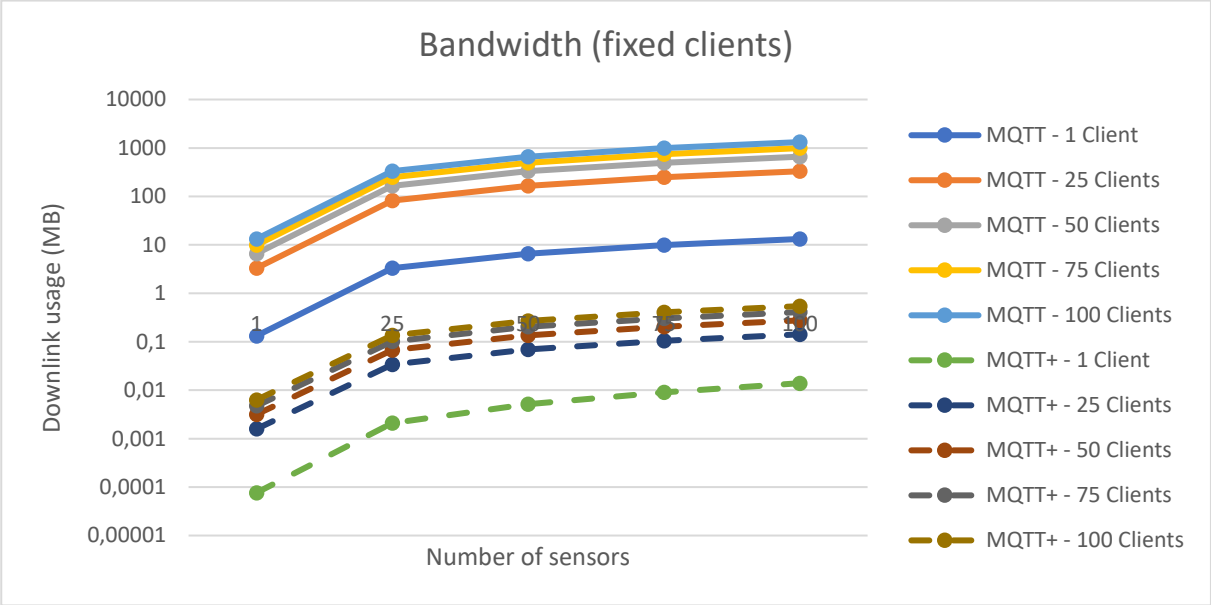


Figure 32: Images bandwidth - fixed clients

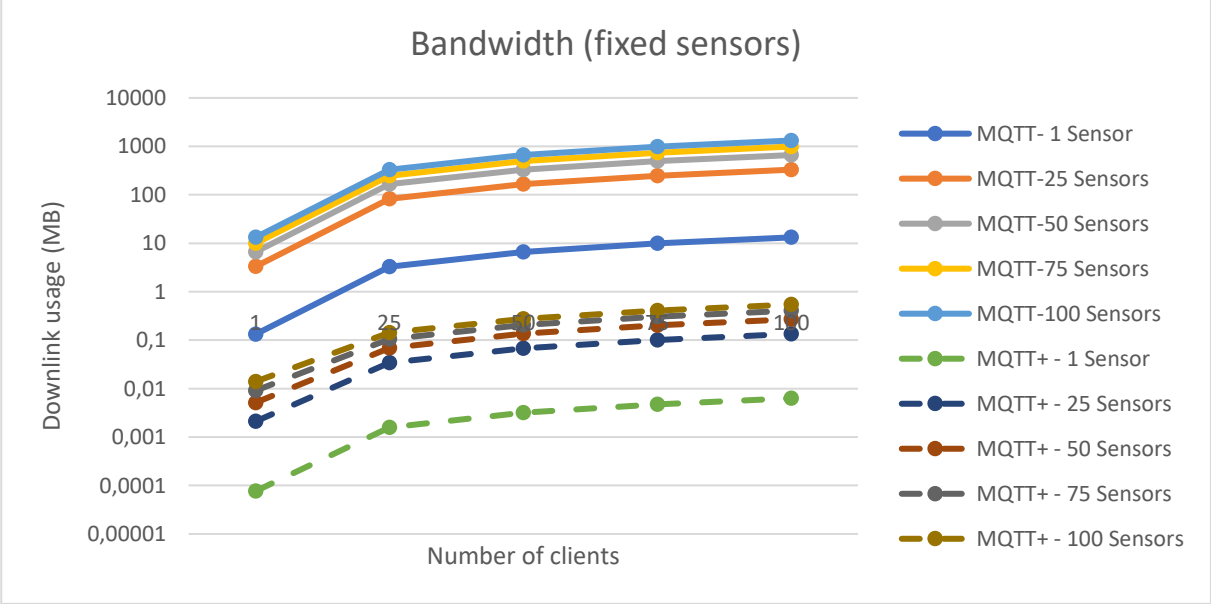


Figure 33: Images bandwidth - fixed sensors

Looking at the previous two graphs, we can see that, **for this particular subscription**, the number of sensors contributes in the same way of the number of clients in the bandwidth. In fact, the two charts are identical. This has perfect sense if we think about the logic on which this subscription is elaborated.
Let's suppose we have 1 sensor that sends one message and 3 clients that receive that message, so the traffic to the clients is 1*3*P (if P is the size of one packet). So, if I want to double the traffic, it's indifferent whether If I double the number of sensors or the one of the clients (2*3*P = 1*6*P).
This proves the results that we got in the tests.

Anyway, an important thing to notice is the gain in term of bandwidth with respect to the MQTT:

Minimum gain (MQTT+ 1-1 vs MQTT 1-1)	Maximum gain (MQTT+ 100-100 vs MQTT 100-100)
-0,13 MB; -99%	-1,29 GB; -99%

The gain in terms of bandwidth is huge in case of 100 sensors and 100 clients and this is an important result, because it justifies the introduction of MQTT+ in context where bandwidth is a critical resource.

6.4.2 – CPU usage

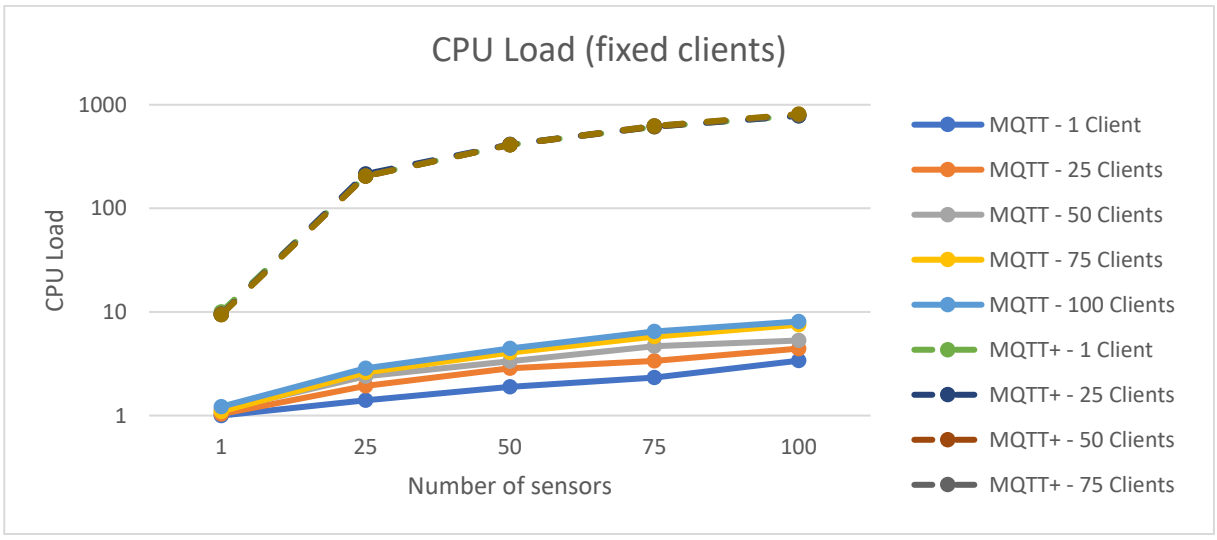


Figure 34: CPU load - fixed clients

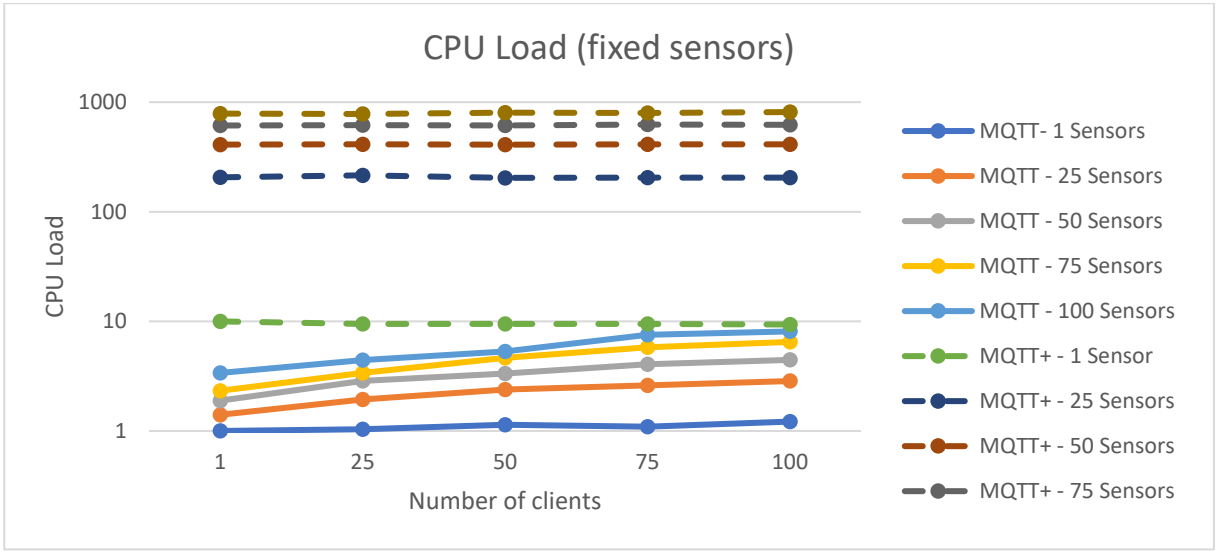


Figure 35: CPU load - fixed sensors

From the previous charts we can see that the CPU load for MQTT depends both by clients and sensors. Instead, for MQTT+, we can see that the CPU load doesn't depend by the number of clients. This is a consequence of the caching for complex operations. With caching, as said before, the image is processed only once per message (one elaboration per sensor), but after it's cached, that value is sent to any interested client, without having to make the computation again.
So, this explains the horizontal lines for MQTT+ in the second chart (or equivalently the overlapped lines in the first one).
In these charts we can also see the main drawback of the MQTT+: An increase of the computational resources required.
Indeed, the CPU load required by the MQTT+ is always higher than the one of the MQTT, in particular we obtain the following results:

Minimum increase (MQTT+ 1-1 vs MQTT 1-1)	**Maximum increase (MQTT+ 100-100 vs MQTT 100-100)**
+9; +900%	+803,91; +9937%

This is the price we have to pay if we want to avoid the image elaboration by each client, but consider that:

1. For the image elaboration we used OpenCV with a general purpose script for recognizing people[25] that has been trained with few examples. The performances of this elaboration can be improved training much more the recognition algorithm and maybe using more optimized algorithms.

2. In a real context the broker will be correctly sized (maybe a cloud could be used) for the computational load required by the context. In this way, this additional load can be managed.

3. We can consider of using an online service for image recognition (Face++, Amazon Rekognition[24], etc..). In this way, we reduce the load on the broker, but we increase the bandwidth that needs to be used to send the images to the external elaboration service.

4. In these tests we used a high-resolution image (1280x720 with 24 bit/pixel). In a video-surveillance system is highly probable that the images used have a lower resolution and so the image analysis software would take less computational resources to be computed.

6.4.3 – RAM Usage

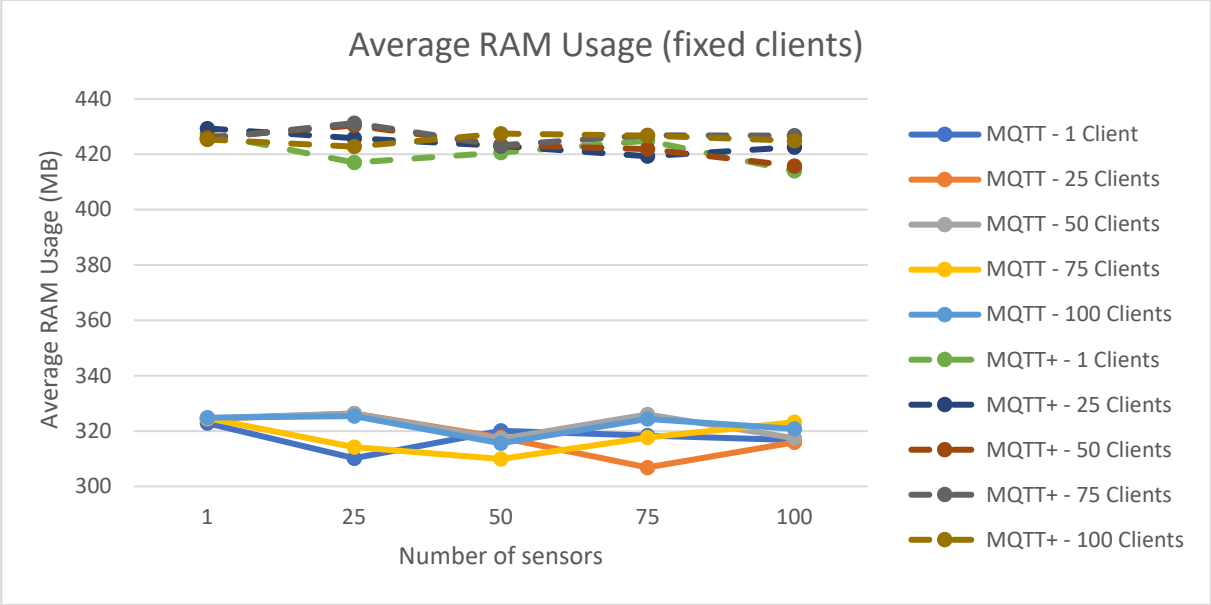


Figure 36: RAM usage fixed clients

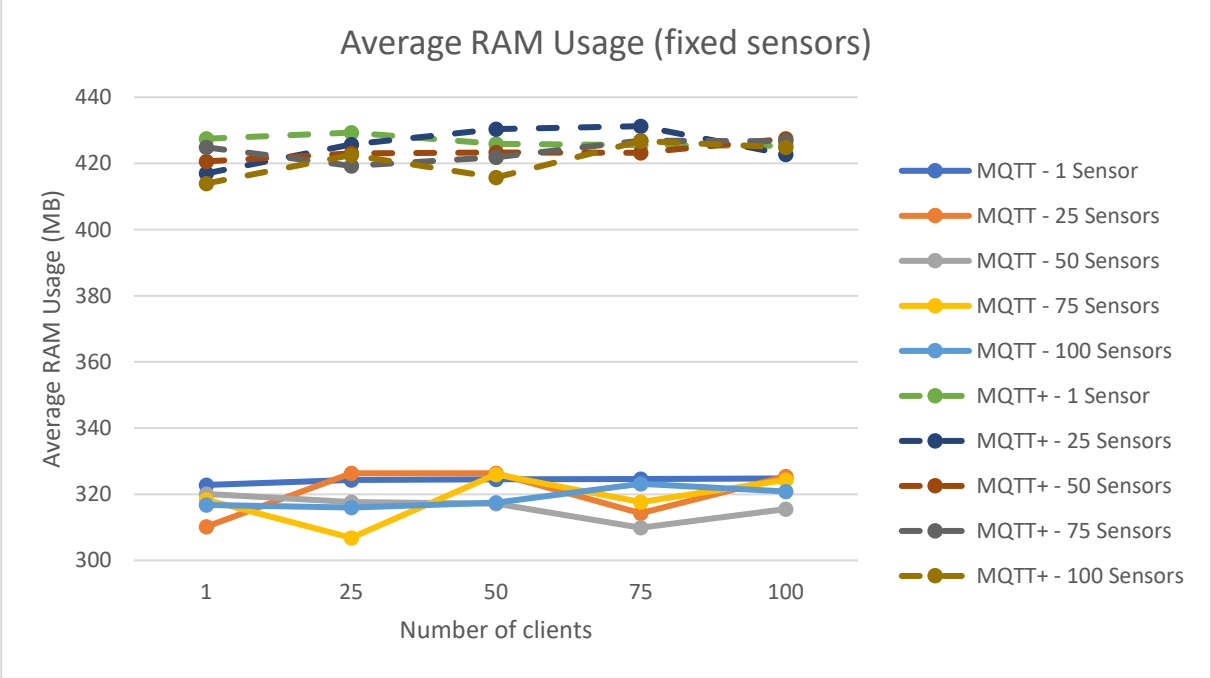


Figure 37: RAM usage - fixed sensors

From these charts we see that the RAM remains more or less constant for any number of sensors or clients. We can also notice that, due to the image elaboration process, the RAM used by MQTT+ is higher of about 100 MB, compared to the one used by MQTT.
This is not a serious problem, since we notice that the RAM for MQTT+ remains around 420 MB, that is an acceptable quantitative of memory for any computer.

Chapter 7 - Conclusions

In this thesis we have proposed MQTT+, an advanced version of MQTT which allows clients to use an enhanced syntax to exploit a broker's computation power to perform different operations.
MQTT+ supports rule-based subscriptions, spatial aggregation (last value operations) and temporal aggregation (temporal operations) of data and advanced data processing tasks. Such basic operations can also be combined together with composite subscriptions.

7.1 – Future works

In this section we are going to explain the possible future improvements that can be applied to the MQTT+ logic.

7.1.1 – Broker capabilities

With the current implementation, each client must know what operations the broker can compute and knowing this, make valid subscriptions to the broker.
An interesting functionality would be to give the clients the possibility to first ask the broker what operations can do and after that, decide what subscriptions to do.
A possibility to do that, would be to use the already existing reserved topic $SYS, to get information about the broker capabilities. For example, a client could ask the broker this information by subscribing to the topic $SYS/capabilities and the broker, at the moment of a subscription by a new client to this topic, could answer with a JSON response like the following one:

```
[{ "keyword": "$CNTPPL",
   "desc": "counts people in an image",
   "returns": "value"
  },
  {"keyword": "$CNTMALE",
   "desc": "counts males in an image",
   "returns": "value"
  },
  {
   "keyword": "$CNTFEMALE",
   "desc": "counts females in an image",
   "returns": "value"
  },
  {
   "keyword": "$RECOGNIZE",
   "desc": "recognizes objects in an image",
   "returns": "json"
}]
```

Figure 38: Capabilities answer example

In this way the client knows the keyword to use for each operation, the description of what that operation does and what kind of data that it returns.
Note that another implicit improvement, needed to implement this, is to introduce the support of answering with JSON messages.

7.1.2 – Advanced semantic analysis

Actually, we can give only one semantic definition for each available operation (only one data type as input and one data type as output), but sometimes it's necessary to have multiple definitions for one operation (like the overload of the methods).
For example, this is useful to define more precisely the behaviour of the rule-based operations that work on numbers (`GT,GTE`,etc..), because with the current logic of the semantic analysis we defined as input and output of these operations the `OptionalNumericSet` data type. This was done because these operations can take/give as input/output both `Numeric` or `NumericSet` and the only way to guarantee both behaviours was to specify the `OptionalNumericSet` as input and output.
A more elegant way to do this is to define the overloads of these operations like this:

Overloads	Input data type	Output data type
Overload 1	Numeric	Numeric
Overload 2	NumericSet	NumericSet

With this modification, the semantic analyser checks in which of the two cases we are and proceeds with the semantic analysis, choosing the right overload. For example, if we have:

```
$GT;25$DAILYAVG/temperature/+
```

The second overload is chosen for the `GT` operation. Instead, if we have:

```
$GT;25$COUNTPEOPLE/images/imgSensor1
```

The first overload is chosen.

This, of course, adds complexity to the semantic analyser that has to choose dynamically the suitable definition, between the available ones, looking at the given input, but it gives a more precise way of defining operations with different possible behaviours, like the rule-based operations.

7.1.3 – Specify explicitly the TTL

In the current implementation the TTL for the detections is fixed statically in the code to 1h, but this could be suitable for certain contexts and not in others.
To let the user choose explicitly the TTL, an idea would be to insert the TTL in the variable header of the publish message. The addition of a new field in the variable part of the header causes no issues since the length of the variable header is carried in the fixed part of the header, but this of course implies to modify a bit the libraries that send the publish messages, to include this additional header.

7.1.4 – Reply to invalid subscriptions

Currently, if a client makes an invalid subscription (the parsing process raises an exception), the subscription is simply discarded by the broker, but the client doesn't know anything about it.
A way to let the client knows about a parsing error, would be to include this information in the suback message, but this implies to modify the client library that handles the messages sent by the broker, since in the current implementation of MQTT, if the subscription is not accepted by the broker, the client continues to try subscribing until the broker accepts the subscription.

7.1.5 – Resource dependent subscription acceptance

Actually, the subscription is accepted whenever the parsing process is valid.
Another improvement could be to accept a valid subscription only if we have the resources to handle it.
For example, let's suppose that a client wants to make a temporal subscription like this one:

```
$TMPAVG;01:00:00/temperature/+
```

If we have the same parameters for the sensors of the example made in section 4.5.4.2, we can estimate that the memory required, to store all the data for this subscription, is more or less 20,93 GB. If the broker knows that the available memory isn't sufficient to guarantee all this memory, then it can reject the subscription to avoid memory issues.

7.1.6 – Advanced operation caching

Actually, the operation caching is useful only to cache the results of image elaborations, but this caching could be extended to cache also the results of subscriptions that are used by several clients.
For example, suppose that we have 1000 clients that subscribe to the same subscription:

```
$TMPAVG;01:00:00/temperature/+
```

As we have seen in the previous example, this subscription could take a lot of memory, therefore the computation of the average on such a big data set could be an intensive operation.
In the actual implementation, this computation must be done 1000 times (one for each subscription made), instead if we cached the result of the entire subscription, it would be necessary to make this computation just one time.
Also, we could apply this reasoning to re-use common operations between different subscriptions, for example if we have 1000 clients subscribed to:

```
$TMPAVG;01:00:00/temperature/+
```

and other 1000 clients subscribed to:

```
$GT;20$TMPAVG;01:00:00/temperature/+
```

Even though these two subscription topics are different, they share the last part of the topic (`$TMPAVG;01:00:00/temperature/+`), so they also share the complex operation `TMPAVG;01:00:00.`
This common operation can be saved in the cache and used in the elaboration of both subscriptions, saving computational resources.
Of course, these modifications to the caching logic introduce more complexity in terms of what to cache and if it's possible to reuse the results in cache for other subscriptions.

Bibliography

[1] Andrew Banks and Rahul Gupta. 2014. MQTT Version 3.1. 1. OASIS standard 29 (2014).

[2] Carsten Bormann, Angelo P Castellani, and Zach Shelby. 2012. Coap: An application protocol for billions of tiny internet nodes. IEEE Internet Computing 16, 2 (2012), 62–67.

[3] Samir Chouali, Azzedine Boukerche, and Ahmed Mostefaoui. 2017. Towards a Formal Analysis of MQtt Protocol in the Context of Communicating Vehicles. In Proceedings of the 15th ACM International Symposium on Mobility Management and Wireless Access. ACM, 129–136.

[4] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. 2006. Towards expressive publish/subscribe systems. In International Conference on Extending Database Technology. Springer, 627–644.

[5] Kannan Govindan and Amar Prakash Azad. 2015. End-to-end service assurance in IoT MQTT-SN. In Consumer Communications and Networking Conference (CCNC), 2015 12th Annual IEEE. IEEE, 290–296.

[6] Urs Hunkeler, Hong Linh Truong, and Andy Stanford-Clark. 2008. MQTT-S- A publish/subscribe protocol for Wireless Sensor Networks. In Communication systems software and middleware and workshops, 2008. comsware 2008. 3rd international conference on. IEEE, 791–798.

[7] Guoli Li and Hans-Arno Jacobsen. 2005. Composite subscriptions in contentbased publish/subscribe systems. In ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing. Springer, 249–269.

[8] Pietro Manzoni, Enrique Hernández-Orallo, Carlos T Calafate, and Juan-Carlos Cano. 2017. A Proposal for a Publish/Subscribe, Disruption Tolerant Content Island for Fog Computing. In Proceedings of the 3rd Workshop on Experiences with the Design and Implementation of Smart Objects. ACM, 47–52.

[9] Navneet Kumar Pandey, Kaiwen Zhang, Stéphane Weiss, Hans-Arno Jacobsen, and Roman Vitenberg. 2014. Distributed event aggregation for content-based publish/subscribe systems. In Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems. ACM, 95–106.

[10] Navneet Kumar Pandey, Kaiwen Zhang, Stéphane Weiss, Hans-Arno Jacobsen, and Roman Vitenberg. 2015. Minimizing the communication cost of aggregation in publish/subscribe systems. In Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on. IEEE, 462–473.

[11] Peter Saint-Andre, Kevin Smith, Remko Tronçon, and Remko Troncon. 2009. XMPP: the definitive guide. " O'Reilly Media, Inc.".

[12] Meena Singh, MA Rajan, VL Shivraj, and P Balamuralidhar. 2015. Secure mqtt for internet of things (iot). In Communication Systems and Network Technologies (CSNT), 2015 Fifth International Conference on. IEEE, 746–751.

[13] Dinesh Thangavel, Xiaoping Ma, Alvin Valera, Hwee-Xian Tan, and Colin Keng-Yan Tan. 2014. Performance evaluation of MQTT and CoAP via a common middleware. In Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on. IEEE, 1–6.

[14] dc-square GmbH. *HiveMQ.* n.d. https://www.hivemq.com/.

[15] dc-square GmbH. *MQTT Essentials Part 3: Client, Broker and Connection Establishment.* n.d. https://www.hivemq.com/blog/mqtt-essentials-part-3-client-broker-connection-establishment.

[16] dc-square GmbH. *MQTT Essentials Part 4: MQTT Publish, Subscribe & Unsubscribe.* s.d. https://www.hivemq.com/blog/mqtt-essentials-part-4-mqtt-publish-subscribe-unsubscribe.

[17] dc-square-GmbH. *MQTT Essentials Part 6: Quality of Service 0, 1 & 2.* s.d. https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels.

[18] dc-square GmbH. *MQTT Essentials Part 5: MQTT Topics & Best Practices.* s.d. https://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices.

[19] Face++,Megvii. s.d. https://www.faceplusplus.com/.

[20] OpenCV. s.d. https://opencv.org/.

[21] PowerShell-GetProcess. s.d. https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/get-process?view=powershell-6.

[22] tshark. s.d. https://www.wireshark.org/docs/man-pages/tshark.html.

[23] Eclipse Paho Java. s.d. https://www.eclipse.org/paho/clients/java/.

[24] Amazon Rekognition. s.d. https://aws.amazon.com/it/rekognition/.

[25] Python people detection script. s.d. https://www.pyimagesearch.com/2015/11/09/pedestrian-detection-opencv/.