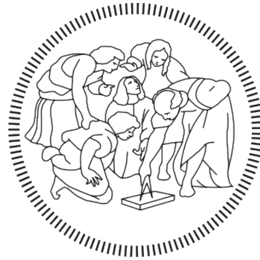
Development and comparison of MQTT distributed algorithms for HiveMQ



POLITECNICO
MILANO 1863

Filippo Antonielli
928440

Advisor: Prof. Alessandro E. C. Redondi

School of Industrial and Information Engineering
Politecnico Di Milano

This dissertation is submitted for the degree of
Laurea Magistrale in Computer Science and Engineering

Academic Year 2019-2020

To my family and friends.

Acknowledgements

Ringrazio di cuore tutti coloro che mi hanno accompagnato in questo percorso universitario. Ringrazio la mia famiglia, che non mi ha mai fatto mancare nulla e mi ha sempre sostenuto. Un ringraziamento va alle nuove amicizie nate e cresciute in questi anni universitari e un ringraziamento per quelle più vecchie risalenti agli anni del liceo o delle medie.

Per la realizzazione di questa tesi ringrazio in primo luogo il Professor Alessandro Redondi per essere stato il relatore di questo mio ultimo lavoro universitario. Desidero inoltre ringraziare Edoardo Longo per essere sempre stato disponibile a rispondere a ogni mia domanda in tempi sempre celeri e per avermi aiutato a risolvere alcuni problemi sorti verso la fine del mio lavoro.

Un ultimo ringraziamento sentito va in particolare ai miei genitori, Nicoletta e Mauro, a mio fratello Andrea, e a mia nonna Giorgia, che sono le persone che più mi sono state vicine nella mia vita.

Abstract

MQTT is one of the most popular protocols for IoT. Its operation consists of a central broker to which many clients are connected; the clients make requests and publish data through a publish/subscribe architecture. This system is clearly in antithesis with the Multi-Access Edge Computing (MEC), one of the main pillars of the fifth generation of cellular data (5G). A distributed architecture of brokers, where multiple brokers are connected and work together, would bring closer the MQTT protocol to MEC.

This work's goal is to use the MQTT protocol with the HiveMQ platform to create a network of HiveMQ's MQTT distributed brokers, loop-free, robust against failures, organized through a spanning tree. To achieve such a goal, we developed different HiveMQ's plug-ins that integrate the already present Bridge extension. While the Bridge extension is necessary to connect the brokers, all the developed extensions contain the logic to avoid loops and to rebuild the tree in case of failures. In addition, we present advanced distribution features, such as an ad-hoc forwarding table that keeps a record of the brokers interested in a Topic to forward the MQTT publishes only when necessary. The developed extensions have been tested in different scenarios and topologies, monitoring the network overhead and the physical resource consumption of every broker.

It turns out that the extension without any forwarding improvements in some scenarios can consume even 200% more bandwidth than the extension with the tables.

Sommario

MQTT è uno dei principali protocolli per IoT. Il suo funzionamento consiste in un broker centrale a cui sono connessi diversi clienti; i clienti effettuano richieste e pubblicano dati tramite una architettura publish/subscribe. Questo sistema è chiaramente in antitesi con il Multi-Access Edge Computing (MEC), uno dei pilastri fondamentali della quinta generazione di rete mobili (5G). Una architettura di broker distribuiti, dove molteplici broker sono connessi tra di loro e lavorano insieme, farebbe avvicinare il protocollo MQTT al MEC.
Questo lavoro si propone di utilizzare il protocollo MQTT, insieme alla piattaforma HiveMQ, per creare un network di broker distribuiti, senza loop, resistente alle disconnessioni, organizzati attraverso uno spanning tree. Per raggiungere tale obbiettivo, abbiamo sviluppato delle estensioni per HiveMQ da usare insieme alla Bridge extension. Mentre l'estensione Bridge è necessaria per connettere i broker, tutte le estensioni sviluppate consentono di evitare i loop e di ricostruire l'albero in caso di guasti. Inoltre, proponiamo delle funzione avanzate di distribuzione, come una tabella di forwarding che tiene traccia dei broker interessati a un Topic e inoltra le pubblicazioni MQTT solo quando necessario. Le estensioni sviluppate sono state testate in diversi scenari e topologie, monitorando il sovraccarico di rete e il consumo delle risorse fisiche di ciascun broker.
Si è scoperto che l'estensione senza nessun miglioramento di forwarding in certi scenari consuma fino al 200% più banda dell'estensione con le tabelle.

Contents

1 Introduction **13**

2 State of the art **15**

2.1 IoT Protocols 15

2.2 Distributed MQTT 15

3 Software development and implementation **17**

3.1 Protocols 17

3.1.1 MQTT 17

3.1.2 STP 17

3.2 HiveMQ 18

3.3 Development 20

3.3.1 HiveMQ-Benchmark 21

3.3.2 Flooding 22

3.3.3 Subscription Table 24

3.3.4 Recap 25

4 Experiments and results **27**

4.1 Scenario Metrics 27

4.1.1 Locality 29

4.2 Test 1: 50 Publishes 29

4.2.1 Tree 30

4.2.2 Star 32

4.2.3 Line . . . 36
4.2.4 50 Publishes: considerations . . . 38
4.3 Test 2: 500 Publishes . . . 38
4.3.1 Tree . . . 38
4.3.2 Star . . . 42
4.3.3 Line . . . 44
4.3.4 500 Publishes: considerations . . . 46
4.4 Test 3: 1500 Publishes . . . 47
4.4.1 Tree . . . 47
4.4.2 Star . . . 50
4.4.3 Line . . . 52
4.4.4 1500 Publishes: considerations . . . 55
4.5 Summary . . . 55
4.6 Cluster comparison . . . 57
4.7 Subscription Table in-depth . . . 57
4.8 End to end Delay . . . 62
4.8.1 50 Publishes . . . 63
4.8.2 500 Publishes . . . 65
4.8.3 Cluster . . . 65
5 Conclusions **68**
5.1 Future Works . . . 69

1 Introduction

Driven by the visions of Internet of Things and 5G communications, recent years have seen a paradigm shift from centralized to distributed computing. One of the main pillars of 5G is Multi-Access Edge Computing (MEC). MEC technology will bring computational power, storage resources, and service infrastructures to the edge, reducing the resources needed in the core network and decreasing latency.

In a nutshell, the main target of wireless systems, from 1G to 4G, was the pursuit of increasingly higher wireless speeds to support the transition from voice-centric to multimedia-centric traffic. As wireless speeds approach the wireline counterparts, the mission of 5G is different and much more complex: to support the explosive evolution of ICT (Information and Communication Technologies) and Internet. In terms of applications, a wide-range of new applications and services for 5G are emerging, such as real-time online gaming, virtual reality and ultra-high-definition (UHD) video streaming, which require an unprecedent high access speed and low latency. It is also predicted by Cisco that about 50 billions IoT devices will be added to the Internet by 2020, most of which have limited resources for computing, communication and storage, and have to rely on Clouds or edge devices to enhancing their capabilities. It is now widely agreed that relying only on Cloud Computing is inadequate to realize the ambitious millisecond-scale latency for computing and communication in 5G. This makes it essential to supplement Cloud Computing with Multi-Access Edge Computing that pushes traffic, computing and network functions towards the network edges.
In recent years, the Internet of Things (IoT) has drawn significant research attention. IoT is considered as a part of the Internet of the future and will comprise billions of intelligent communicating 'things'. However MQTT (Message Queuing Telemetry Transport) , that is the de-facto standard protocol for IoT solutions, is a centralised protocol where there is a broker that is the single endpoint to which all the clients are connected to. This protocol therefore is in contrast with the MEC that is one of the main innovation of 5G.

The goal of this work is to connect multiple brokers using HiveMQ that is a MQTT based messaging platform designed for the fast, efficient and reliable bi-directional movement of data between device and the cloud. The brokers will create a network among themselves that is completely transparent to the clients. A client can subscribe to any topic on any broker and will receive the publishes related to such topic even if they come from another broker in the network. The brokers in fact communicate with each other and exchange information.
In this way there will not be a single broker with all the clients connected to him, sim-

ilar to the classical cloud-based architectures, but there will be many brokers closer to the clients, more like Multi-Access Edge Computing. Such a distributed network of brokers located near the clients can be very helpful to satisfy delay-sensitive applications, like autonomous driving and health-monitoring, since the computing tasks are no longer processed at the distant single-broker.

The starting point to create the distributed network of HiveMQ's MQTT brokers is the HiveMQ's Enterprise Bridge extension that enables HiveMQ to bridge to one or more MQTT brokers. Then we started the development of a plug-in that would work together with the Bridge extension to create the loop-free distributed network. In fact the Bridge extension enables only the bridges between the brokers, but it does not contain any logic to avoid the creation of loops or an advanced routing strategy.

We developed three different extensions: *HiveMQ-Benchmark*, *Flooding* and *Subscription Table*. Every of this extension is to be used together with the Bridge extension and every one succeeds in the creation of the distributed network, but they are all different to one another and presents different distribution features, as explained the the following sections.

2 State of the art

2.1 IoT Protocols

Several authors have done a comparative analysis between protocols for the IoT, discussing the criteria for selecting protocols such as MQTT [1], CoAP [2], AMQP [3], and HTTP[4] [5]. The main focus of such works is generally to study the performance of the protocols in term of end-to-end delay, bandwidth consumption and number of supported devices [6]. [7] compares the fraffic generated and the average delay for MQTT and CoAP changing the network packet loss rate through a network emulator (WANEM). In [8], the authors analyze the performance of mosquitto and RabbitMQ regarding latency in the uplink data packets and present advantages and drawbacks of the candidates in a Smart City scenario. They use the brokers in cloud and as clients a Raspberry Pi and a common laptop.[9] use a peer-to-peer communication model to select which MOM (Message-Oriented-Middleware) use for industrial application. They find out that AMQP, KAFKA and ZeroMQ can achieve a throughput of more than 1000mps while MQTT cannot. Kafka has low latency but an higher overhead. [10] puts Mosquitto, BevyWiseMQTT, and HiveMQ in a small-scale, single broker cloud scenario comparing their performance by subscription throughput using mqtt-stresser and mqtt-benc . They find no big difference in performance when MQTT broker are applied to a domestic deployment use case.

2.2 Distributed MQTT

The research for distributed MQTT solutions has been very active in the last years. [11] proposes a distributed broker system for large-scale location-based IoT services. [12] presents an edge-enabled publish-subscribe middleware, named EMMA, that continuously monitors network QoS and orchestrates a network of MQTT protocol brokers. It transparently migrates MQTT clients to brokers in close proximity to optimize QoS. [13] compares different distributed MQTT brokers for performance, scalability, resilience, security, extensibility, and usability in an enterprise IoT scenario deployed to an edge gateway cluster; among them there is also HiveMQ, that is the only one that shows no message loss. The paper [14] proposes Interworking Layer of Distributed MQTT brokers (ILDM), which enables arbitrary kinds of MQTT brokers to cooperate with each other. In [15] is proposed a distributed MQTT architecture based on the RPL protocol.
These papers study in deep different and interesting aspects of connecting distributed MQTT brokers, but none of these focuses on creating a loop-free network without having a static topology.

Instead in [16] is presented MQTT-ST, a protocol able to create a distributed architecture of brokers organized through a spanning tree. The main difference between this work and the one written in the paper, is that the latter is build upon Mosquitto, where as this work uses HiveMQ. There are therefore some differences, starting from the language of development: C for Mosquitto and Java for HiveMQ. In the signalling phase we had to use only Publish messages, where with Mosquitto in possible to use PINGREQ messages and to append some information to it, like the IP address of the Root broker. Both the works succeed in the creation of a loop free network. We also implemented, in the third extension, a more advanced routing strategy, where every broker knows if a specific message is interested by another broker and only in this case it publishes the message. Such advanced routing strategy is based on PADRES [17]. The PADRES system is a distributed content-based publish/subscribe system which consists of a set of brokers connected by a peer-to-peer overlay network. The overlay network connecting the brokers is a set of connections that form the basis for message routing. The overlay routing data is stored in Overlay Routing Tables (ORT) on each broker. Specifically, each broker knows its neighbors from the ORT. Advertisements are effectively flooded to all brokers along the overlay network using the ORT. A subscriber may subscribe at any time. The subscriptions are routed according to the Subscription Routing Table (SRT), which is built based on the advertisements in that broker.

3 Software development and implementation

3.1 Protocols

3.1.1 MQTT

There are two big families of application layer protocols for the IoT: Client/Server and Publish/Subscribe. COAP (COnstrained Application Protocol) and MQTT (Message Queuing Telemetry Transport) are the two most used protocols, the first belongs to the family of Client/Server and the latter to the one of Publish/Subscribe. The main difference between the two is that COAP uses a system of Request/Response, a "PULL" type, so the data is obtained by issuing and explicit request. With MQTT [Figure 1] instead there is a PUSH paradigm: every client publishes and subscribes to a broker, and the data are sent by the broker to the subscribers as soon as are available. Every subscription sent by a client to a broker contains at least a Topic; every following publishes that arrive at the broker, will be sent to all the subscribers interested in that topic. MQTT runs over TCP/IP; each MQTT client opens one TCP connection to the MQTT broker. There are different kind of messages: CONNECT and CONNACK, PUBLISH and PUBACK, SUBSCRIBE and others. Every Publish message can have a value of Quality of Service (QoS): 0, 1 or 2.
MQTT is becoming more and more popular in IoT, such that the four major cloud computing services all adopt MQTT as protocol for connecting IoT devices to their endpoints: Amazon AWS, Google Cloud Platform, IBM cloud and Microsoft Azure. In addition to this in a network of distributed brokers, as the one in this work, MQTT's publish/subscribe system is to be preferred over COAP's request and response system: in such a scenario there could be hundreds of brokers with thousands of clients each. With so many clients every seconds there could be lots of new data and thanks to MQTT's publish and subscribe feature the clients can always be up to date.

3.1.2 STP

The Spanning Tree Protocol (STP) [Figure 2] is a network protocol that builds a loop-free logical topology. The basic function of STP is to prevent bridge loops and the broadcast radiation that results from them.

The spanning tree is obtained by electing a root switch and blocking some of the output ports of the other switches: blocked ports do not forward data frames, thus avoiding broadcast storms. To agree on the root node and on which ports should be

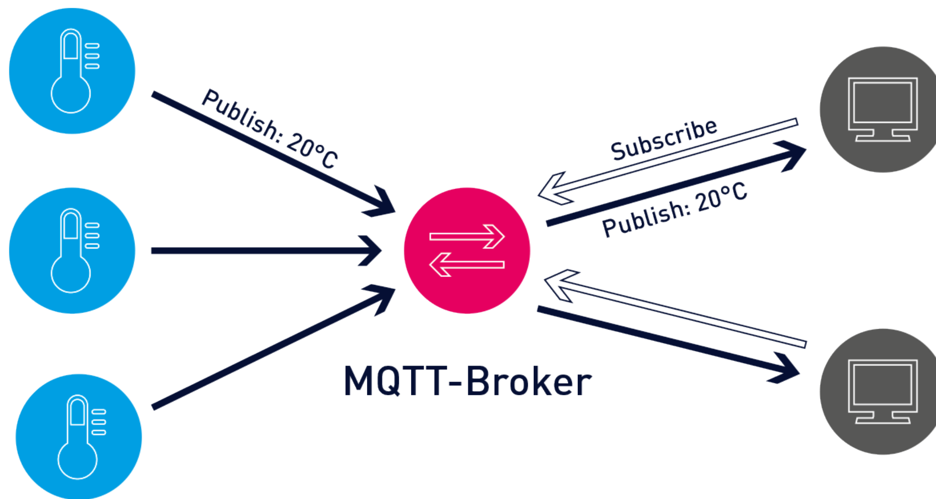


Figure 1: MQTT architecture

blocked, switches exchange control packets known as BPDU (Bridge Protocol Data Unit).
At startup, each node sets itself as root and start broadcasting BPDU. Each BPDU contains (among other parameters) the identifier of the node and the transmitting port, the identifier of the current root node selected by the transmitting node and the root path cost. The node identifier, composed of both the node MAC address and a configurable priority value is used for root selection: the node with the lowest identifier is elected as root.
Upon the reception of a BPDU, a node reconfigures its state by modifying the identifier of the (believed) root node and updating the root port (the port that leads to the least-cost path to the root). The rest of the active ports are labeled as either designated (used for forwarding traffic) or blocked. To avoid loops, nodes agree on which port should be designated or blocked, based again on the least-cost path to the root or the lowest identifier, in case of ties.
BPDUs are periodically transmitted by the root, and forwarded by all other nodes, to keep the topology updated.

3.2 HiveMQ

The aim of this work is to implement the Spanning Tree Protocol for Distributed MQTT Brokers using HiveMQ's MQTT brokers.
HiveMQ is an MQTT broker and a client based messaging platform designed for the fast, efficient and reliable movement of data to and from connected IoT devices. It uses the MQTT protocol for instant, bi-directional push of data between devices

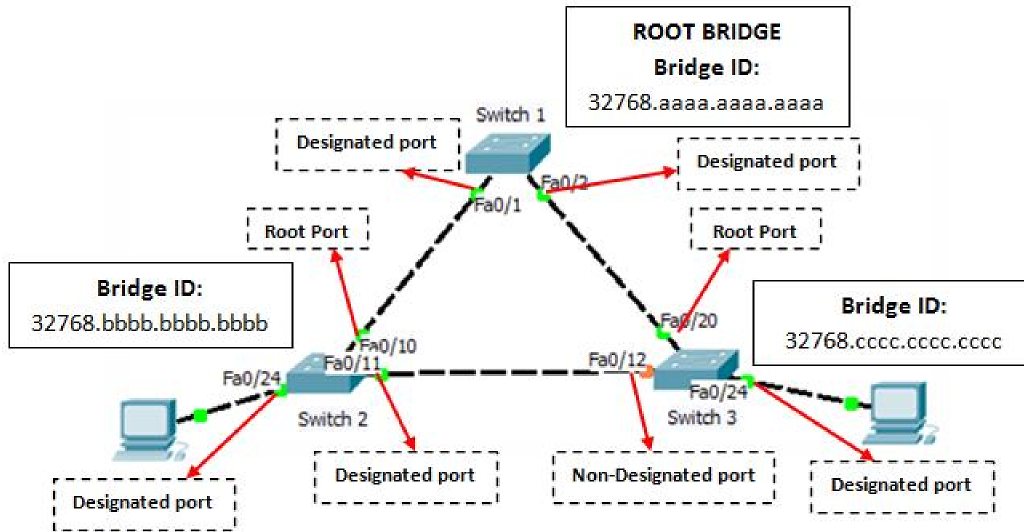


Figure 2: Spanning Tree Protocol

and the enterprise systems.
HiveMQ offers the possibility to create a cluster of brokers, which is very helpful to create a service with high availability and can be utilized to achieve horizontal scaling. However HiveMQ's cluster does not support tree topologies with more than one leaf, but only mesh network: in a cluster the brokers are all connected with one another. Therefore in the next section where we will show and present the experiments that we have done, we will only compare the cluster with the cases in which all the brokers were fully connected.
Respect to the clustering, bridging ensures the possibility to create any network topology, so we started to develop an extensions for HiveMQ that would work in synergy with the HiveMQ Enterprise Bridge Extension. The HiveMQ Enterprise Bridge Extension enables HiveMQ to bridge to one or more MQTT brokers for scalable, reliable, and bi-directional exchange of MQTT messages.

In order to test the code, we used Docker to launch different containers where every container is a HiveMQ's MQTT Broker. For instance the following command starts a container named "hive1" on port 1883, it uses the image "hivemq4:4.3.1" and copies everything there is in the folder case3_subTable/tree/broker_A in the folder "extensions" of hivemq-4.3.1 and it also copies the license. The license is necessary to use the Enterprise version of HiveMQ which is needed to run the Enterprise Bridge Extension. To run multiple container we simply run a similar command changing the port, so for example broker_B would have port 1884 instead of 1883.

```
docker run --rm -p 8080:8080 -p 1883:1883 --name hive1 -v
/Users/filippoantonielli/Desktop/Tesi/case3_subTable/tree
/broker_A/ :/opt/hivemq-4.3.1/extensions/ -v
/Users/filippoantonielli/Desktop/Tesi/case3_subTable/tree
/license :/opt/hivemq-4.3.1/license/  hivemq/hivemq4:4.3.1
```

3.3 Development

We developed all our extensions using IntelliJ IDEA, which is a IDE for Java. We used Docker to integrate our extensions with HiveMQ and to launch multiple containers at the same time. Every container is build on top of a HiveMQ image. We used Docker Stats to collect the data regarding the output network and CPU usage of every container. Finally we also used Docker to launch another container with *tcpdump* to save in an external file all the packets captured during our tests.
Every broker in our tests is a HiveMQ's enterprise MQTT broker. Thanks to the usage of a licence given us by HiveMQ, we are able to use the enterprise version, and consequently to use the enterprise Bridge extensions to connect the brokers. We are also able, thanks to the enterprise version, to connect multiple clients and to publish many more messages than with the default and free version.

We developed three extensions: *HiveMQ-Benchmark*, *Flooding* and *Subscription Table*. Every extension is to be used together with the Enterprise Bridge Extension of HiveMQ. In the configuration file of the Bridge Extension every broker needs to have an active bridge with another broker in order to be part of the network. If more bridges are active between brokers, every extension will create a loop-free tree for the exchange of the messages. The creation of the tree is RTT based: every broker will keep using the bridges in order to have the lowest RTT to reach the Root. For example, if there are three brokers A,B and C, all connected with each other and C is the Root, in most situations *broker_A* and *B* will keep active the bridge through *broker_C* and not use the bridge between *A* and *B* since the RTT to reach the Root is lower, in most cases, then to go through another broker.
The Root is the broker that has the highest Capacity, as explained in the following sections.
Once the tree is created, the bridges that are not to be used are not disabled: the brokers know which bridges are to be used because, after the Setup phase, every one will have a table with all the brokers it is connected to, and only the brokers that are its *next_hop* or one of its *prev_hop* use an active bridge and are meant to receive its messages.
The topology of the tree stays the same, as long the Root is the same. Every time a

broker *PINGs* another broker, a *PUB* message is exchanged between the two with the information about the Root and its capacity. If the Root broker disconnects, the brokers that are directly connected to the Root will advise all the other brokers and a new Root will be selected and the topology of the tree will change accordingly. During this phase all the *next_hop* and *prev_hop* will change, depending on the tree.

3.3.1 HiveMQ-Benchmark

The first extension is *HiveMQ-Benchmark*. It is the simplest way to connect multiple brokers to form a tree without the creation of loops of messages. Once a broker starts, it uses the following instructions to retrieve its number of CPU cores, how many RAM memory it has available and its IP address.

```
long maxMemory = Runtime.getRuntime().maxMemory();
int cpu_cores = Runtime.getRuntime().availableProcessors();
String own_address = InetAddress.getHostAddress();
```

With the values of maxMemory and cpu_cores, every broker calculates its Capacity value C:

$$C = \alpha \cdot maxMemory + \beta \cdot cpu_cores + \gamma$$

Every broker has a Listener with the method *onMqttConnectionStart*: when a broker tries to connect to another broker as described in the bridge configuration file, this method is triggered on both sides. The brokers add in a table the IP address of the other broker. The *PingReqInterceptor* is triggered every time a broker receives a ping. Since with HiveMQ can only be sent Publish messages and not other kinds of messages like *PINGREQ*, we made that every time a *PING* is received, the broker sends a Publish message to the other to ask for its C value and RTT if they are not already in the table.

Every broker has also a *PublishInterceptor*. It is triggered when a *PUB* is received. If the *PUB* comes from another broker that is present in the table, the broker checks the Topic. Thanks to switches and if cases, depending on the Topic string, some actions are triggered. For instance if the Topic starts with "$SETUP", then it checks the remaining of the Topic: if it is "c_value" it publishes a message with its Capacity value. If it is "rtt_value" then a couple of messages are exchanged between the brokers to calculate the RTT between the two.

When the Table in every broker is complete, every PING corresponds to a Publish with topic "$TOPIC/root_values". With these Publishes the brokers decide which

is going to be the Root of the tree and they choose it based on the one broker that has the best Capacity. So every broker sends as payload of this *PUB* the IP address, the Capacity and RTT of the broker with the highest Capacity in their table. Since it could happen that not every broker is connected to every other ones, if a broker receives the *PUB* and the Capacity value is bigger than the one that is the maximum in its Table, the broker updates the root and sends another *PUB* to every other broker it is connected to to inform them about this change.
In this phase the brokers also come to knowledge about the fastest way to reach the Root: if a broker is directly connected to the Root it will set as its *next_hop* the IP address of the root; otherwise if it is not connected to the root it will set as its *next_hop* the IP address of the broker that has advised him about the new Root. In this second case the broker that was set as *next_hop* in the first broker, will set this broker as his *prev_hop*. The root will set all the IP address in its table as *prev_hop*.

The *HiveMQ-Benchmark* extension works in this way: when a broker receives a *PUB* from a client, it forwards the *PUB* in the direction of the root, so to his *next_hop*. Another limitation that we've found with HiveMQ is that we can Publish a message to only a client only if the client has a Subscription with the specific topic. In this situation therefore, the broker has to forward the message to all the brokers it is connected to. In order to avoid loops, in the *PublishInterceptor* there is a check when a message is received: if the message is sent by a broker and this broker is not set as *prev_hop*, the message is not meant for the broker that has received the message and therefore the message is blocked from actually reach the broker. On the other hand, if the message is received from a broker where its *prev_hop* is equal to the IP address that has published the message, there are two cases: or the broker is the Root, otherwise is another broker that has to forward the message to the root. If we are in this second case, the broker adds to the Topic "$FORWARD/" and then Publishes the message. In this way the Root will receive the message, it will check if it has any client interested in the Topic, and then it will publish the message back to all the other brokers adding in the Topic "$ROOT/". When a broker receives the Pub that starts with "$ROOT/", it knows is meant for him and checks if there is any client subscribed to the Topic and then, if it has any broker set as his *prev_hop*, it will publish the message adding *forward* to the topic.

3.3.2 Flooding

The second extensions we developed is *Flooding*. The phase for the setup of the tree among the brokers is the same as the one in the first extension. The biggest problem of the first extension was that the brokers had to send a message to all the

bridges that are available and not to a single broker.
The possibility to send a message to only one broker is the main change with respect to the first extension. When a broker receives a *PUB* from a client it forwards the *PUB* in the direction of the Root using his *next_hop*. The difference in this case is that now the broker sends the message only to his *next_hop*, while in the *HiveMQ-Benchmark* extension the *PUB* would be received by all the other brokers connected to the one that is sending. Once the message arrives to the Root, the Root will send the Pub to all the brokers in order to reach every broker, similar to a real Flooding. Once a broker receives the message from the Root, checks if there is a *prev_hop* and in case sends the Publish to that broker. This is possible because in the bridge configuration file *bridge-configuration.xml* instead of having only one bridge that forward everything with the filter "#", now we have one bridge with different filters: the first is "$SETUP/#" and it serves for the formation of the tree in the connection phase of the brokers; the second filter is "$ROOT/#", useful for receiving the messages from the root; the third instead is "$BROKER/ip_address/#", where "ip_address" is the ip of the broker that is connected with the bridge. For example, *broker_A* has ip "172.17.0.2" and *broker_B* has ip "172.17.0.3". The two brokers are connected with a bridge; the "bridge-configuration.xml" file therefore will be:

```
<bridge>
   <name> AtoB </name>
   <remote-broker>
            <connection>
                <static>
                    <host>172.17.0.3</host>
                    <port>1883</port>
                </static>
            </connection>
            <mqtt>
                <keep-alive>11</keep-alive>
            </mqtt>
        </remote-broker>
        <topics>
            <topic>
                <filter>$BROKER/172.17.0.3/#</filter>
                <mode>PUB</mode>
            </topic>
            <topic>
                <filter>$SETUP/#</filter>
                <mode>PUB</mode>
            </topic>
```

```
            <topic>
                <filter>$ROOT/#</filter>
                <mode>PUB</mode>
            </topic>
        </topics>
    </bridge>
```

Thanks to the filter "$BROKER/ip_address/#" two brokers can now communicate without needing to send messages to all the other brokers. If *broker_A* sends a Publish with Topic "$BROKER/172.17.0.3/test", the message will only be received by *broker_B*, because even if *broker_A* is connected to *broker_C*, *broker_C* will have another ip address and therefore the filter in the "bridge-configuration.xml" file will be different.

3.3.3 Subscription Table

The third and final extension we developed we named it *Subscription Table*. The configuration file of the bridge is the same as the one in the second extension, so that a broker can send a message to only one other broker. The main difference in this extension, in comparison with the previous two, is that every broker has a table that keeps record of the subscriptions that are interested to his neighbors brokers. This function has been developed keeping in mind the PADRES system, [17], regarding the Network and the Broker Architecture.
The setup phase is the same as the other extensions. The key difference is that, when a broker receives a Subscription from a client:

- it inserts in its subscription table (*SubTable*) the Subscription with its IP address
- then sends a Publish to all the other brokers it is connected to to inform that it has a client interested in that Topic.
- The brokers that receive the message will insert in their table that the broker with its IP address is interested in the topic and will send another Publish to inform all the other brokers in the tree.

With this configuration [Figure 3], if *broker_D* receives a *SUB* on the topic "test", it will send a message to *broker_E* and broker_C. *Broker_E* and *broker_C* will insert

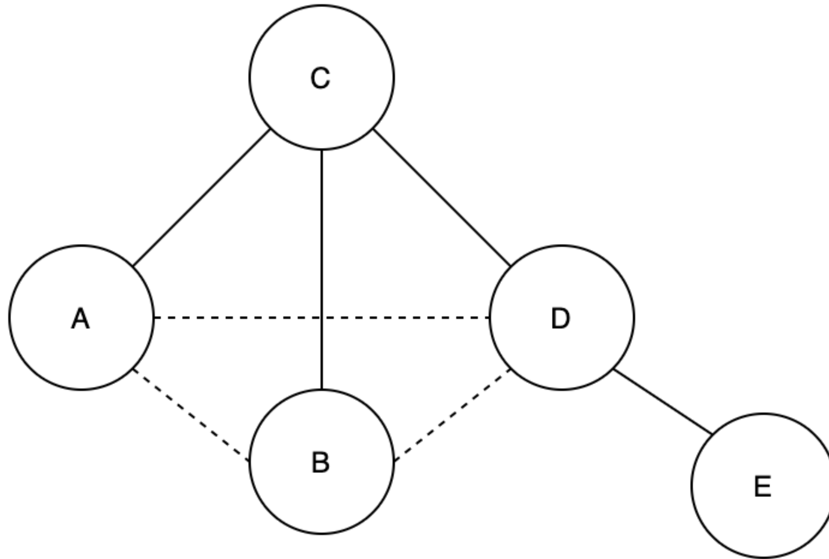


Figure 3: Tree

in their table that *broker_D* is interested in the topic "test". Then *broker_E* has no other broker to inform, while *broker_C* will inform *broker_A* and *broker_B*. *Broker_A* and *broker_B* will insert in their table that *broker_C* is interested in the Topic, since they have no way to reach *broker_D* directly.
At the end, when a broker receives a *PUB*, it checks if has any client interested in the message; then it checks its *SubTable*: if a broker is interested in that topic, it will forward the message to that broker. Once the broker has received the *PUB*, after checking for clients interested in the topic, checks in its *SubTable* for other brokers interested in the Topic and so on.

3.3.4 Recap

We developed three different extensions and every one succeeds in the goal of creating a loop-free tree of distributed HiveMQ's MQTT brokers when used with the HiveMQ's Enterprise Bridge Extension. However the three extensions don't have the same functionalities, but every one is an improvement with respect to the previous one.

- *HiveMQ-Benchmark* is the simplest one: it meets the goal of a loop-free tree

of distributed brokers and has the feature to rebuild the tree in case the Root disconnects. The main drawback is the impossibility to Publish a message to only one broker.

- *Flooding* presents the same functionalities of *HiveMQ-Benchmark* but it also has the possibility to Publish a message to only one broker.
- *Subscription Table* is the most advanced extension. After the Setup phase, that is the same between all the extension, and therefore the creation of the Tree, this extension keeps track of the brokers that are interested in a specific topic. Therefore the Publishes between the brokers are exchanged only if there is the need, so only if there is at least a client that is interested in the topic in some broker of the tree.

4 Experiments and results

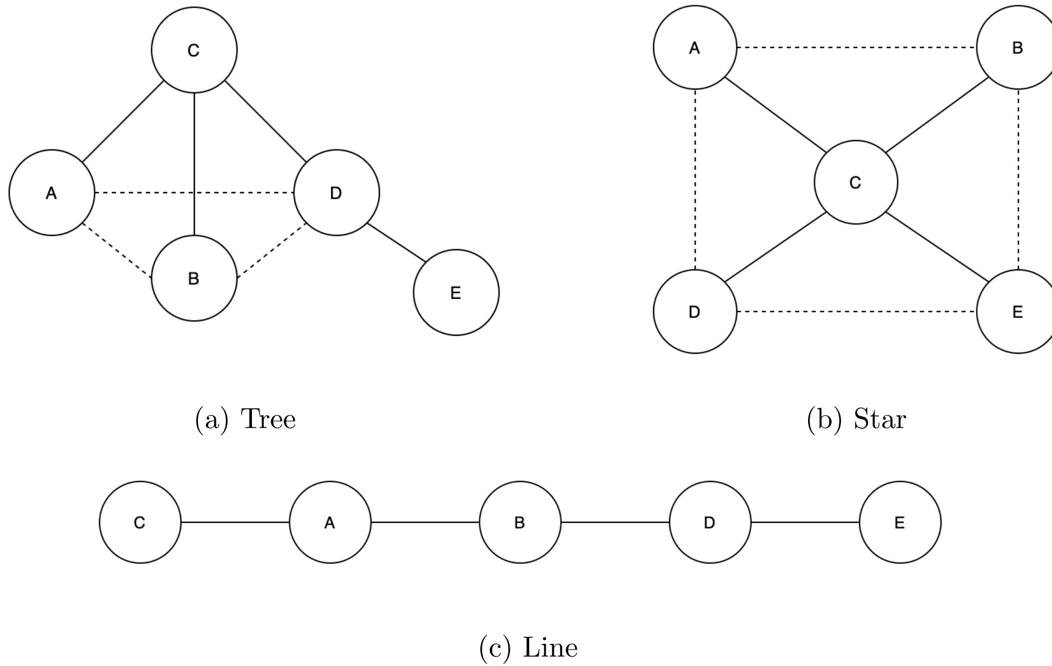
During the tests we focused our attention on the network overhead and on the resources utilization, in particular on the CPU. All our tests are done on my personal machine, a MacBookPro early 2015, processor 2,7 GHz Intel Core i5 dual-core and memory 8 GB 1867 MHz DDR3. Therefore, all the containers, and consequently all the brokers, have the same resources. In order to have the tests as similar as possible, the Root is always *broker_C* and there are 5 brokers in every test. This is only done for reducing the randomness: in a normal situation every broker could be the Root, it only depends on the Capacity of the broker.

4.1 Scenario Metrics

We made different tests focusing mainly on how many Bytes are necessary to exchange the messages within the brokers keeping an eye on the usage of the CPU. For checking the CPU we used Docker Stats: since every broker is a docker container, Docker Stats shows us the percentage of CPU in use in every container; since every container has at its disposal 4 CPU cores, the maximum value we can find in docker stats is 400%. In order to save these data we copied the values of Docker Stats in an external file every second. For checking how many Bytes were circulating in the broker network we also used Docker Stats for taking the value regarding the Network Input and Network Output at the end of any test. For every test we also used *tcpdump* to save in a .pcap file every packet that has been exchanged during the test; then we used these file to see how many bytes were MQTT packets, or how many bytes it took an extension to create a table for instance.

The tests we made consist in sending Publishes to a broker, and having clients interested in the topic of the *PUB* in other brokers. The topic in every Publish is the same.
We tested the extensions with 50 Publishes, 500 Publishes and 1500 Publishes, so we collected data about 81 different scenarios (every test consist in 27 different situations). We also used the Cluster offered by HiveMQ to make a comparison between our extensions and the default way to connect brokers. 9 are the tests regarding the cluster, since we can only change the locality but not the conformation of the tree since the brokers are always fully connected.
Taking for example our first test with 50 messages published, we have the following scenarios:

- three extensions (*HiveMQ-Benchmark, Flooding, Subscription Table*)



(a) Tree

(b) Star

(c) Line

Figure 4: Network topologies

- for every extension we test three different topology of the network (Tree, Star, Line)
- for every topology we make three different tests about the locality of the messages (100% locality, 0% locality, n% locality)

The first is locality 100% so 50 Published are on *broker_A* and the only client interested in the messages is on *broker_A*; 0% locality so the 50 publishes are on *broker_A* and the only client interest in the messages is on *broker_E*; the last is n% locality where the total number of *PUBs* are divided in every broker, so 10 in every one, and every broker has a client interested in the topic.

In every test we always used five brokers. We used three different shape to connect the brokers: Tree, Star and Line. [4]

The dotted lines indicate an active bridge between the two brokers and the lines indicate the tree that has been formed after the setup phase.

4.1.1 Locality

We used three different localities to test our extensions.

- 100% locality: all the publishes are on a Broker and all the subscribers are on the same Broker
- 0% locality: all the publishes are on a Broker and all the subscribers are on a different Broker
- n% locality: every Broker has the same number of publishes and every Brokers has the same number of clients interested in the publishes

In this way more data regarding different, but all plausible, scenarios are collected and confronted to find the best extension to be used in every situation.

4.2 Test 1: 50 Publishes

In [Figure 5] are shown the output data in all the three extensions in the case of 50 Publishes and the brokers linked to form a Tree. Every line represents the sum of all the traffic in exit of every five brokers; "SubTable-100%" is the sum of all the five brokers when testing 50 Publishes with the third extension *Subscription Table* with locality 100%, therefore all the Publisher were in *Broker_A* and the only client interested in the topic was in the same broker. In the total Byte of the traffic there are of course the messages the brokers publish between themselves and the messages the broker exchange every time there is a Ping Request to maintain the tree. In fact, if the Root broker disconnects, the other brokers would rebuild another tree choosing a new Root. This is possible only if the brokers are all connected between themselves; if a broker is only connected to a single broker, and this one crashes, the first broker cannot be reached by any other broker. Besides these messages there are also the *ACK* for every publishes, since in every test we used *QoS1* for the publishes: *QoS1* offers the best compromise between bandwidth usage and delivery guarantee. These are the MQTT packets. In addition to these MQTT packets there are other TCP packets which are indispensable to allow communication between the brokers and the clients.

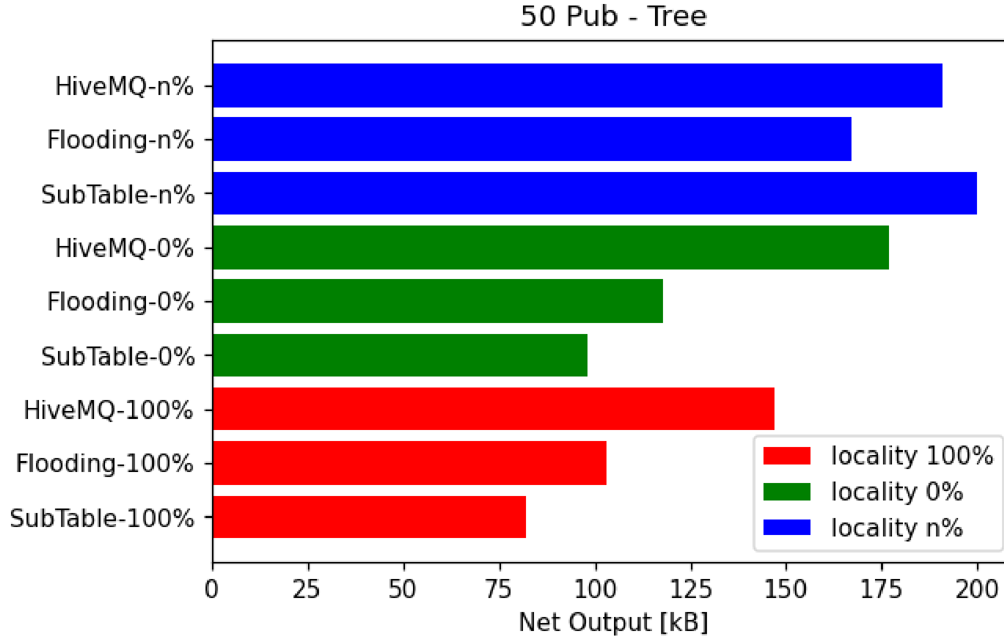
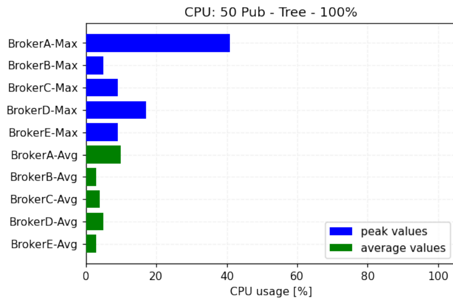


Figure 5: 50 Publishes, Tree topology

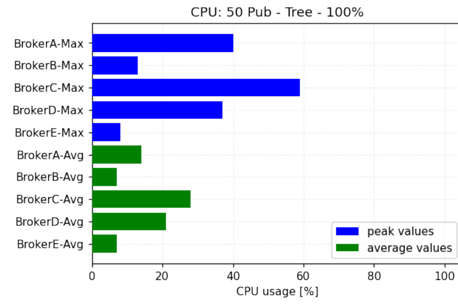
4.2.1 Tree

In the figure [5], the topology of the network is the Tree and even if the messages are only 50, in the cases with locality 100% and locality 0%, it is possible to notice the advantages of the *Flooding* extension and the *Subscription Table* extension. With these 2 extensions is possible to exchange a message with only 2 brokers without the need to send the messages to all the other brokers connected.

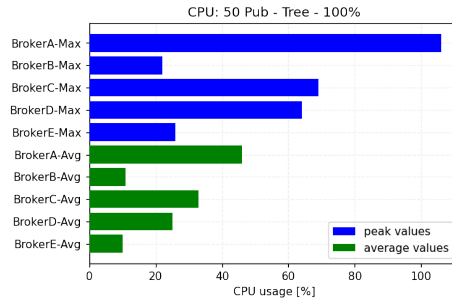
In [Figure 6] are shown the peak CPU values (in percentage with a maximum of 400%) of every broker and the average. The values are taken from the moment the first message is published. Even if the number of messages are not many, there is a key point that can be noticed: in the extension *Subscription Table* (a) the *broker_A*, that is the one that receives the Publishes and is the only broker that has a client interested, is the only broker with a moderate usage of CPU; with this extension in fact *broker_A* knows that there are no other brokers interested in the messages therefore the other brokers don't receive any of these messages. With the other extensions instead *broker_C*, that is the Root broker, has a pretty high usage of CPU because is the one that receives the publishes from *broker_A* and has to forward them to the other brokers. Even *broker_D* has a usage not negligible since he has to rely the messages to *broker_E*.



(a) Subscription Table

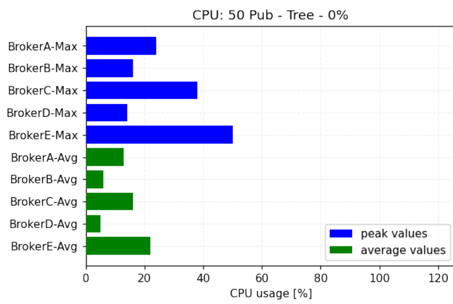


(b) Flooding

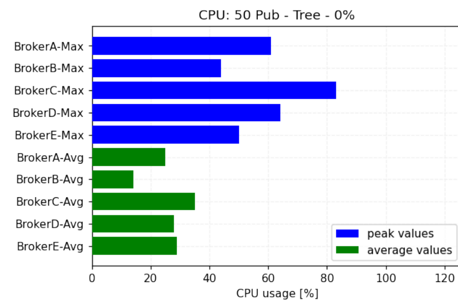


(c) HiveMQ-Benchmark

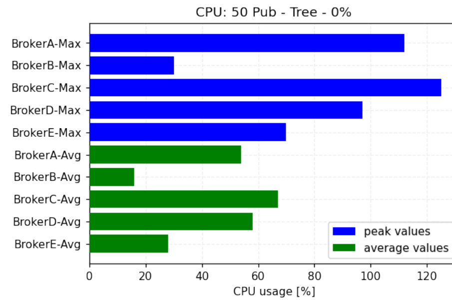
Figure 6: CPU usage with Tree topology and 100% locality



(a) Subscription Table

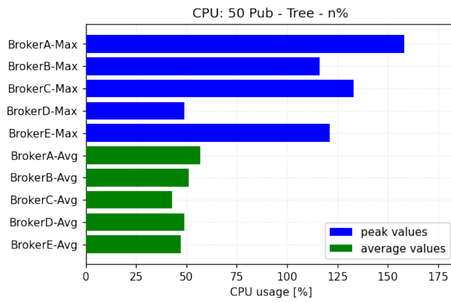


(b) Flooding

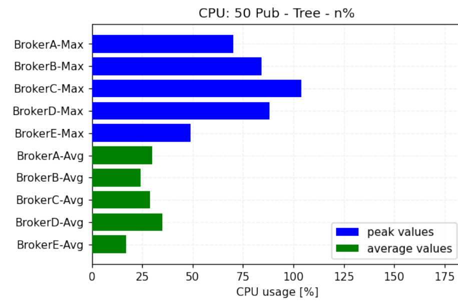


(c) HiveMQ-Benchmark

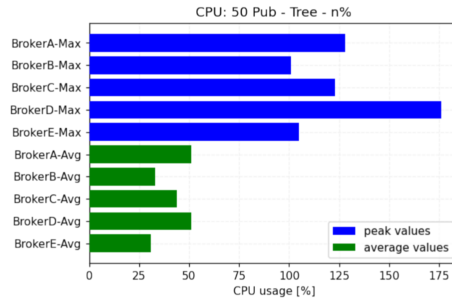
Figure 7: CPU usage with Tree topology and 0% locality



(a) Subscription Table



(b) Flooding



(c) HiveMQ-Benchmark

Figure 8: CPU usage with Tree topology and n% locality

[Figure 7] and [Figure 8] refers to the usage of CPU in the tests with 0% and n% locality. In the test with 0% locality the messages are published on *broker_A* and *broker_E* has a client interested in the topic. In all the extensions *broker_A*, *broker_C* and *broker_E* have a moderate usage of CPU: for *broker_A* and *broker_E* is pretty obvious and *broker_C* is the Root. In average, the extension *Subscription Table* has a lower consumption of CPU, followed by *Flooding* and then *HiveMQ-Benchmark*. With n% locality all the brokers have a pretty high usage of CPU since every one receives publishes and has clients interested in the messages. In this situation the extension *Subscription Table* is the one with the highest usage in average, followed by *HiveMQ-Benchmark* and the extension that has the lowest consumption of CPU is *Flooding*.

4.2.2 Star

In [Figure 9] are displayed the data in output when the brokers are linked to form a Star with *broker_C* as the Root and the center of the star. The results are pretty similar in the case of the Tree, even if here the extensions have less difference in terms of the sum of the output of every single broker. Even in this situation, with

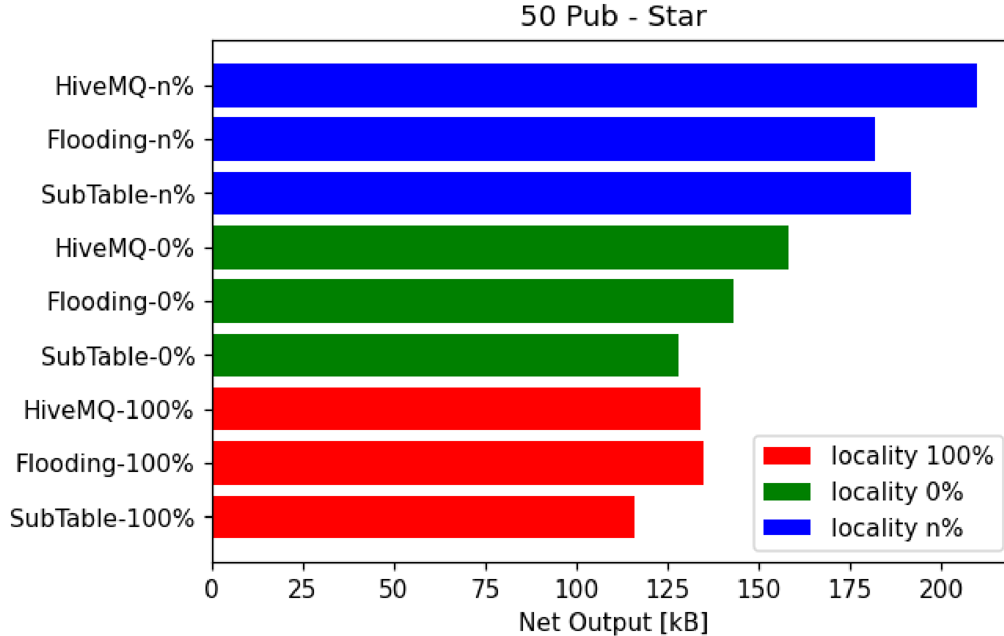


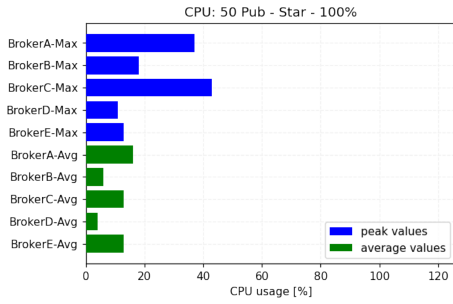
Figure 9: 50 Publishes, Star topology

few messages that are sent, in the test with n% locality, so where every broker has some publishes and every one has clients interested in the topic, the *Subscription Table* is outperformed by the *Flooding* extension, where in the other two cases of locality the extension *Subscription Table* performs a little better that the others.

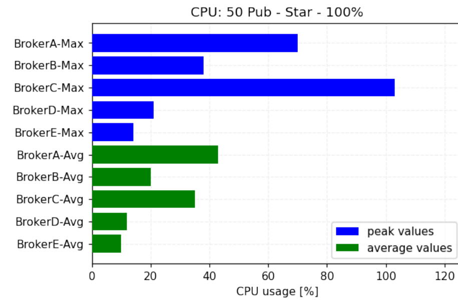
[Figure 10] presents the usage of CPU during the test with 50 *PUB*, with the brokers linked as a star. The extension *Subscription Table* performs much better than the other two; the peak in (a) is around 40% while in (b) and (c) is above 100%. Even the average of the brokers is lower with the *Subscription Table* extension, around the half of the other extensions.

In [Figure 11] are shown the usage of CPU of every broker in the three extensions. *Subscription Table* average values are around the half with respect to the other extensions and the peak values are also very low. In the *Flooding* extensions the Root *broker_C* has a very high peak value and even the average is high, similar situation regarding the average values can be found in the *HiveMQ-Benchmark* extension.

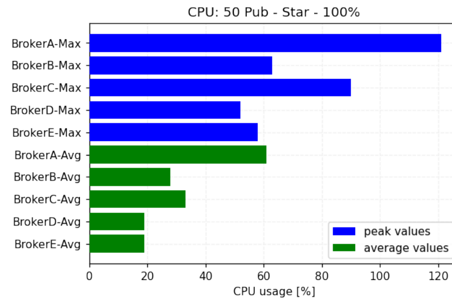
With n% locality [Figure 12] every extension has a very high peak with the Root *broker_C*. *Flooding* is the extension with the highest values, followed by *HiveMQ-Benchmark* which performs a little better. *Subscription Table* has the lowest percentage of CPU usage in every broker, both regarding peak values and average.



(a) Subscription Table

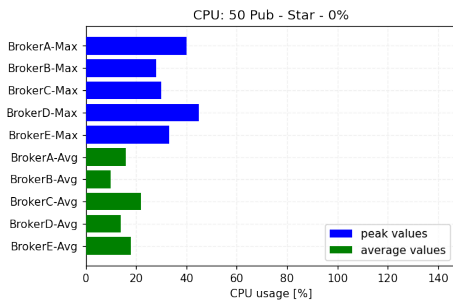


(b) Flooding

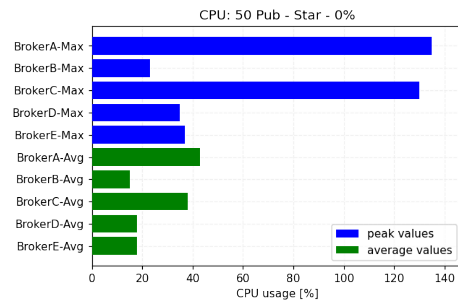


(c) HiveMQ-Benchmark

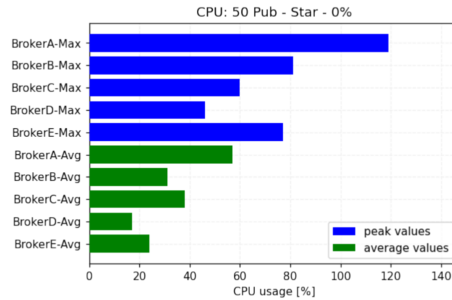
Figure 10: CPU usage with Star topology and 100% locality



(a) Subscription Table

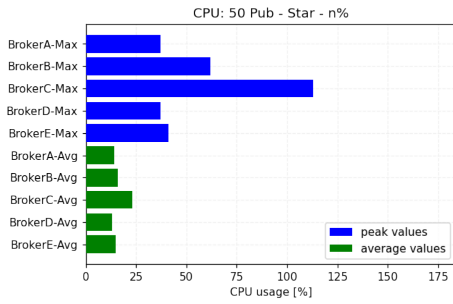


(b) Flooding

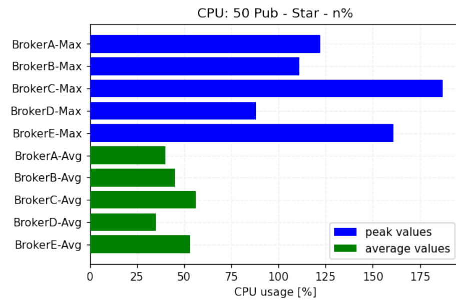


(c) HiveMQ-Benchmark

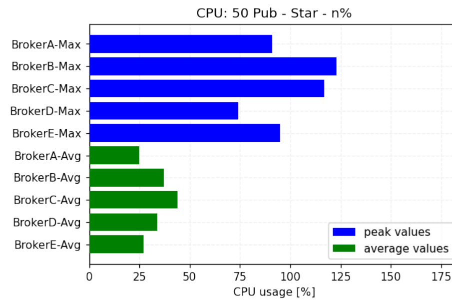
Figure 11: CPU usage with Star topology and 0% locality



(a) Subscription Table



(b) Flooding



(c) HiveMQ-Benchmark

Figure 12: CPU usage with Star topology and n% locality

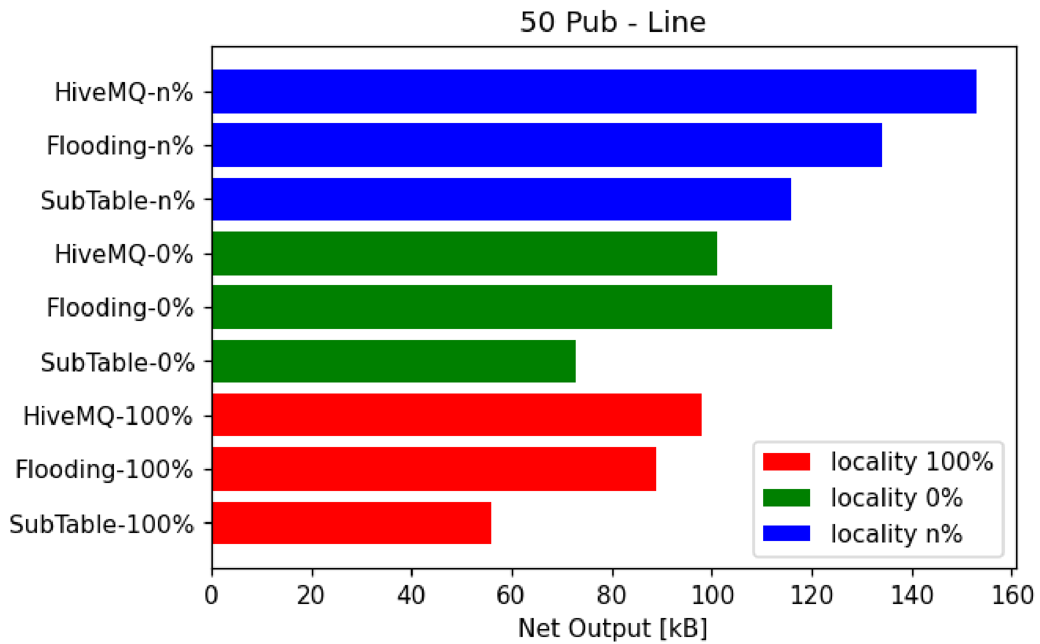
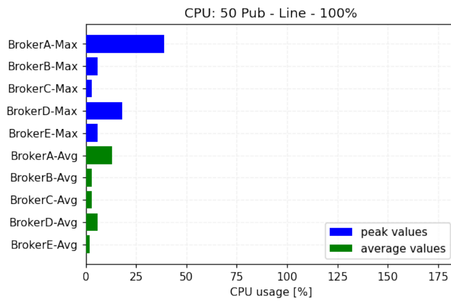
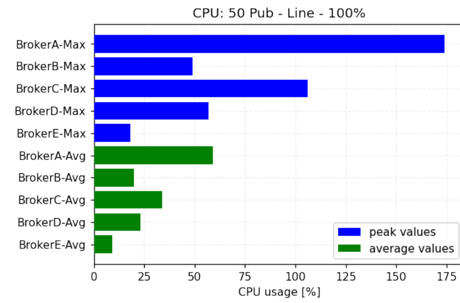


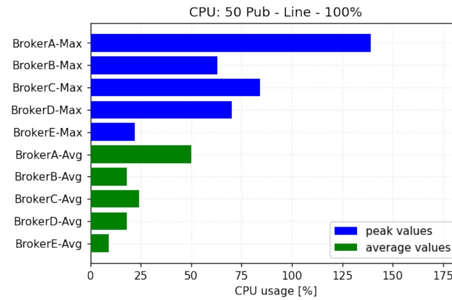
Figure 13: 50 Publishes, Line topology



(a) Subscription Table



(b) Flooding



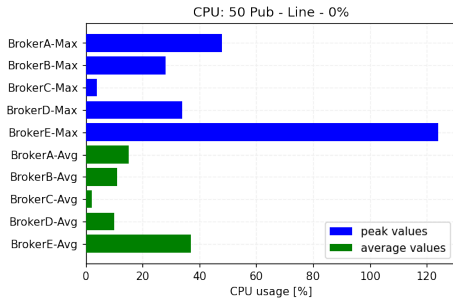
(c) HiveMQ-Benchmark

Figure 14: CPU usage with Line topology and 100% locality

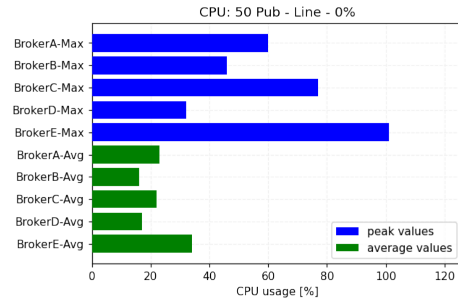
4.2.3 Line

The last set of test with 50 Publishes is with the broker linked in Line. Even in this situation the *Subscription Table* extension performs better in every situation. The biggest advantage is obtained with locality 100%, since *Subscription Table* knows when there is no need to forward the messages. Anyway even in this test the sum of the output of every broker is very similar in the three extensions, probably caused by the few messages that are published.

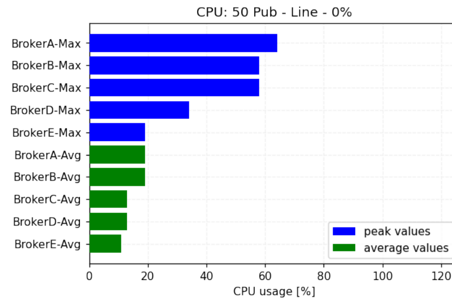
[Figure 14], [Figure 15] and [Figure 16] show the CPU usage of the brokers in the three extensions. In all the extensions, in the case of 100% locality *broker_A* has the highest consumption of CPU, but in *Subscription Table* is remarkably lower than in the other extensions; the other brokers have a minimal impact on the CPU with *Subscription Table*, unlike the other extensions where the CPU's usage of the other brokers is not negligible. With 0% locality, *broker_E* has the highest impact on the CPU in *Subscription Table* and *Flooding*; in *HiveMQ-Benchmark broker_E* is not the one with the highest values, probably because the other brokers have to forward the messages to reach the end of the line. The average values anyway are pretty similar in the three extensions. Finally, in the test with n% locality *Subscription Table* has a slightly better usage of CPU in average compared with the other two



(a) Subscription Table

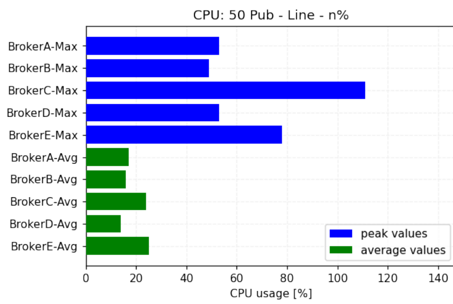


(b) Flooding

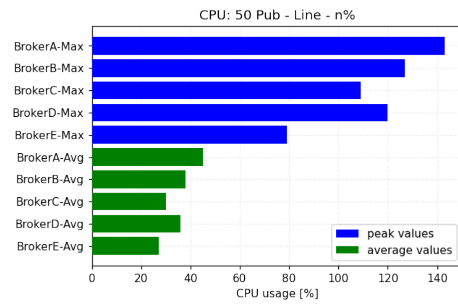


(c) HiveMQ-Benchmark

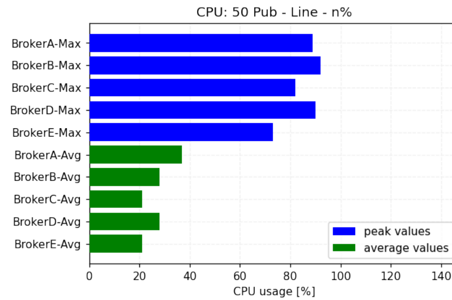
Figure 15: CPU usage with Line topology and 0% locality



(a) Subscription Table



(b) Flooding



(c) HiveMQ-Benchmark

Figure 16: CPU usage with Line topology and n% locality

extensions. Anyway all the extensions have similar results.

4.2.4 50 Publishes: considerations

In this first set of tests there were a few number of publishes, only 50. Regarding the traffic analysis, the extension *Subscription Table* performs better than the other extensions in almost every situations, being outperformed by the other extensions in only a couple of situations; *Flooding* comes in second place, almost every time with a slightly higher values than *Subscription Table* but better than *HiveMQ-Benchmark*. This last extension is the one that consumes more Byte, as expected. In fact it does not have any kind of optimization that the other two extensions have. On the CPU side, even here usually *Subscription Table* is the one that performs best, followed by *Flooding* and *HiveMQ-Benchmark*, even if in some circumstances *Flooding* takes the upper end. In both the situation, traffic analysis and CPU usage, anyway the differences are not that big: the average consumption of CPU is pretty similar among all the extensions and taking a look at the Byte circulating in the network of the brokers, the *Subscription Table* extension in most cases have a better value but the difference with the other two extensions is not that remarkable.

4.3 Test 2: 500 Publishes

This second set of tests is very similar with the previous one, only that now there are 500 Publishes instead of 50; this means that in locality 100% *broker_A* receives 500 Publishes and sends 500 Publishes to a client, in locality 0% *broker_A* receives 500 Publishes that need to reach *broker_E* that has a client interested and in locality n% every broker receives 100 publishes and every broker has a client interested in these messages.

4.3.1 Tree

[Figure 17] shows the network traffic between the broker, that is the sum of the Bytes in output of every single broker in every extension, when the broker are linked as a Tree. Is evident that the extension *Subscription Table* is the one with the least usage of data between the extensions, and the difference is pretty evident in the test with locality 100%. The *Flooding* extension is always the second best and, besides the case with locality 100% where is more similar to *HiveMQ-Benchmark*, usually has a very good result, similar to the one of *Subscription Table*. *HiveMQ-Benchmark* instead is always the one with the highest values.

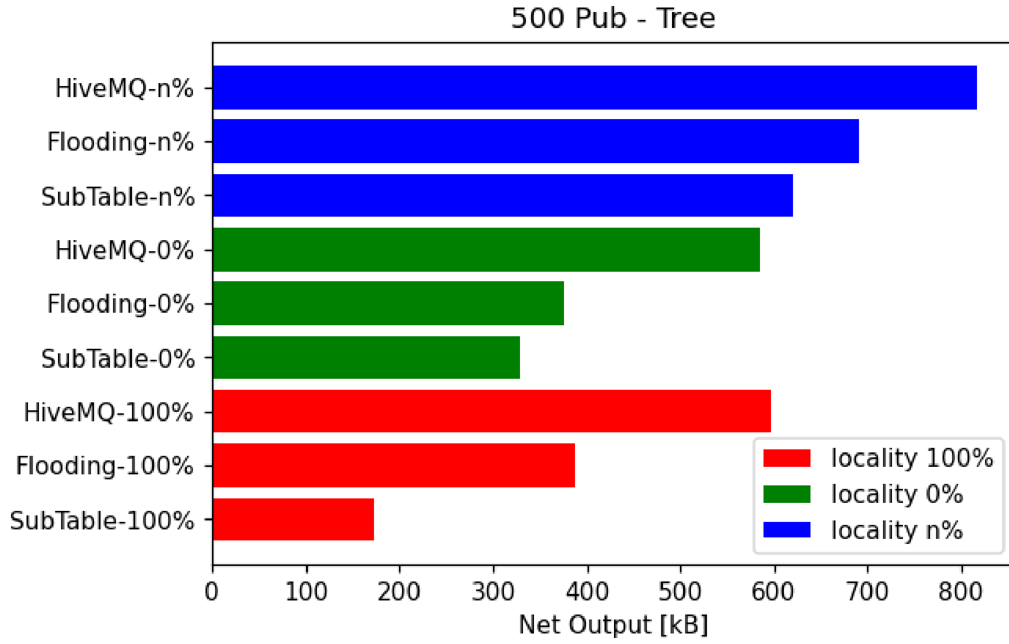
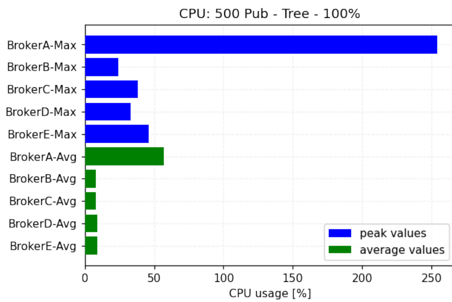
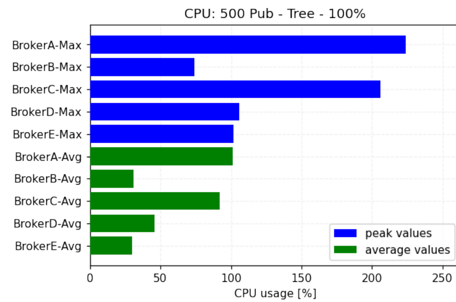


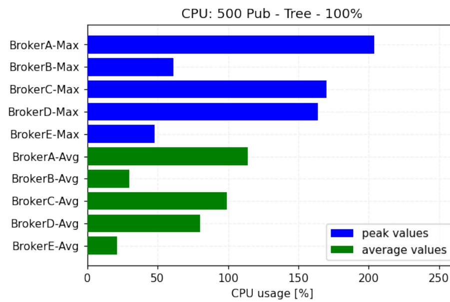
Figure 17: 500 Publishes, Tree topology



(a) Subscription Table

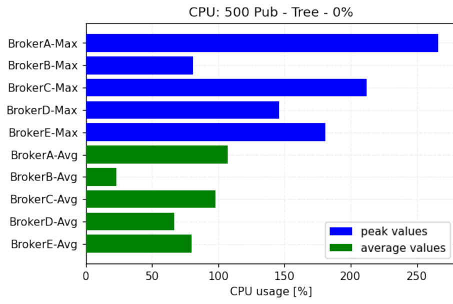


(b) Flooding

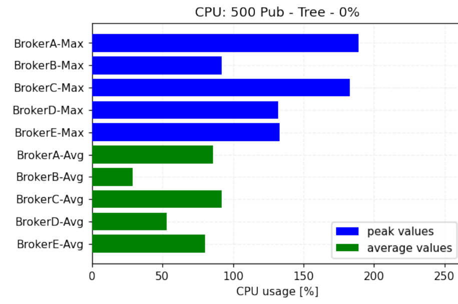


(c) HiveMQ-Benchmark

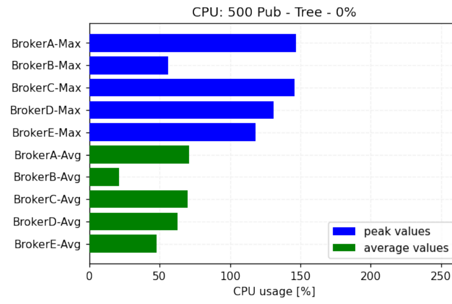
Figure 18: CPU usage with Tree topology and 100% locality



(a) Subscription Table

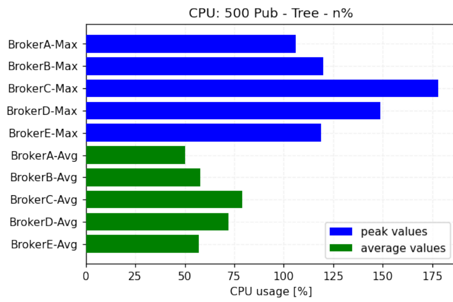


(b) Flooding

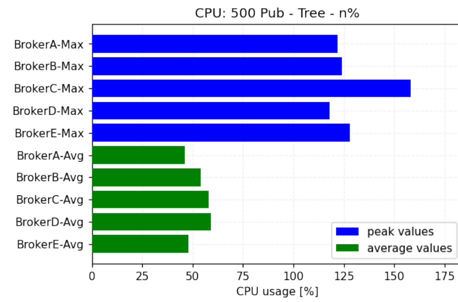


(c) HiveMQ-Benchmark

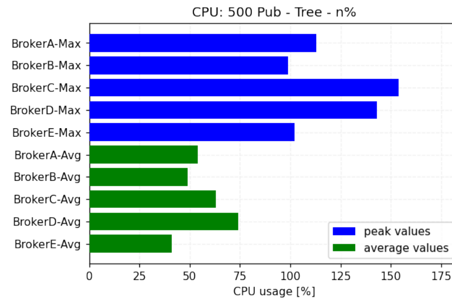
Figure 19: CPU usage with Tree topology and 0% locality



(a) Subscription Table



(b) Flooding



(c) HiveMQ-Benchmark

Figure 20: CPU usage with Tree topology and n% locality

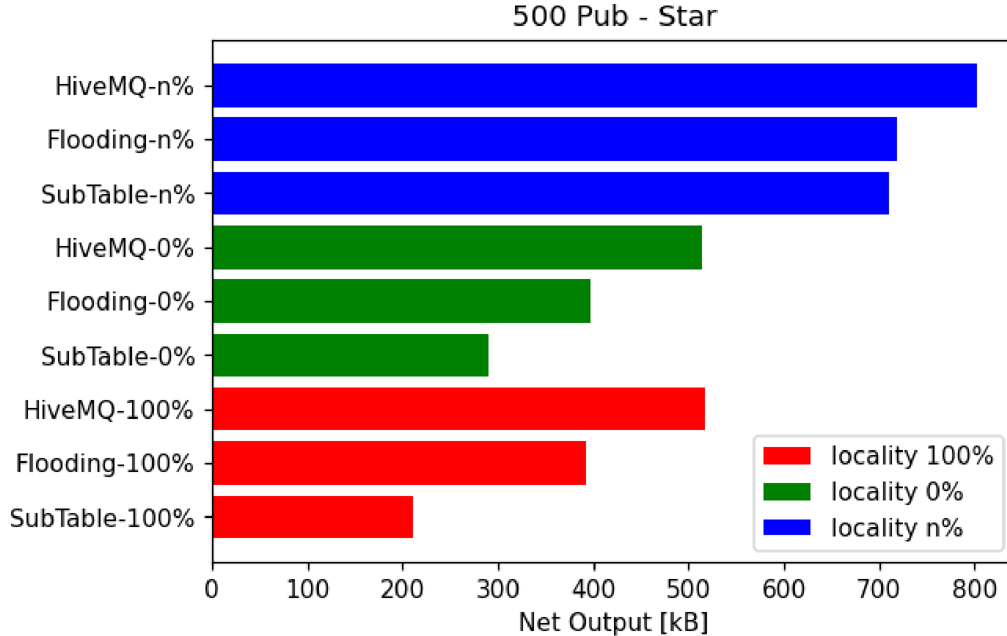


Figure 21: 500 Publishes, Star topology

In [Figure 18] there are the percentage values regarding the CPU usage of every broker taken with Docker Stats. *Subscription Table* has the highest peak value on *broker_A* than every extension, but the averages values are the lowest, and, besides *broker_A* that is in charge of all the publishes of the messages with this locality, the other brokers have almost zero consumption. In (b) and (c) all the brokers have a non negligible impact on the CPU, specially with *broker_A* and *broker_C* that is the Root. With locality 100% is evident the better performance of *Subscription Table*, but it is not always the case with the other localities. In [Figure 19] *Subscription Table* has the highest peak value on *broker_A* and overall has a slightly higher usage of CPU. In this test with locality 0% is HiveMQ-Benchmark that holds the best results, both regarding the peak values and the average values. With locality n% [Figure 20], *Flooding* and *HiveMQ-Benchmark* are head to head while *Subscription Table* is the one with sligty higher peak values and average values. *Flooding* anyway is overall the winner: the peak values are very similar to the ones of *HiveMQ-Benchmark* but the averages are slightly lower. Until now, with 500 Publishes, we have found that *Subscription Table* always performs better regarding the network traffic, but the same is no longer valid for the CPU.

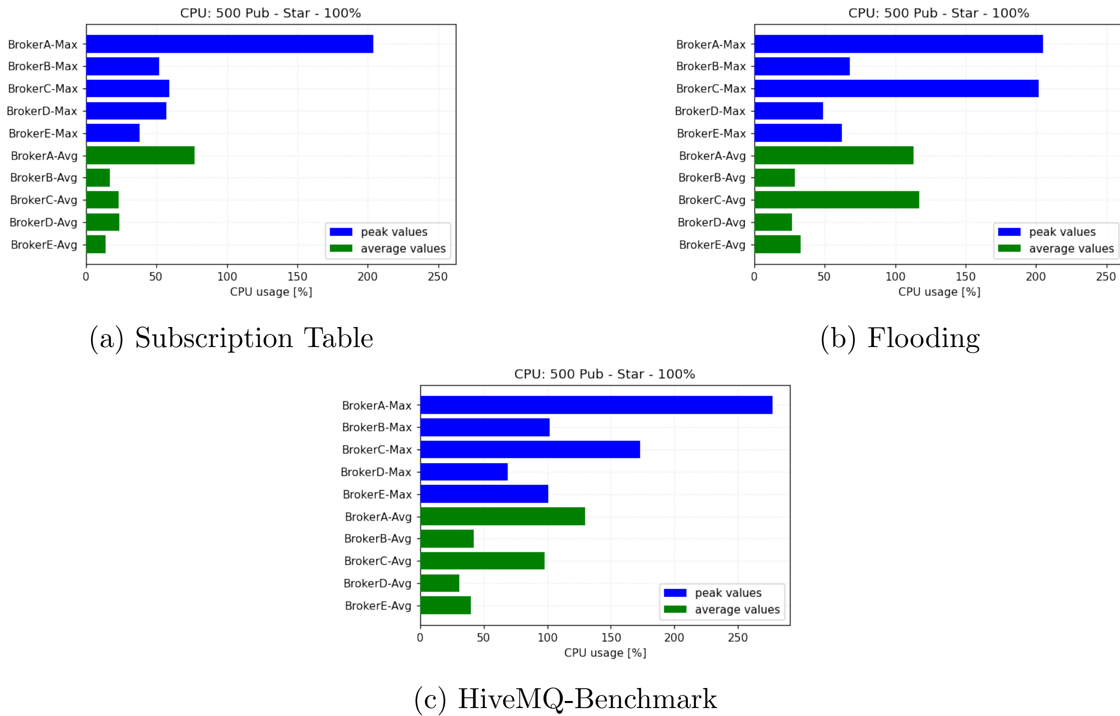
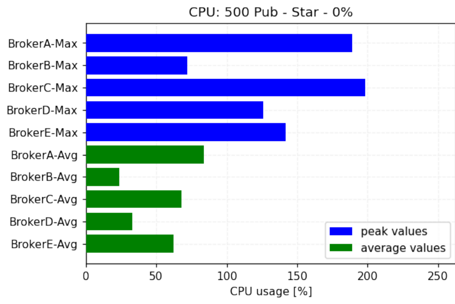


Figure 22: CPU usage with Star topology and 100% locality

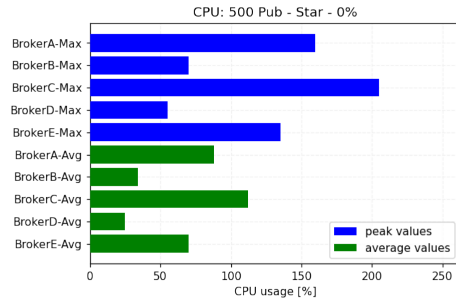
4.3.2 Star

[Figure 21] shows the traffic output when the brokers are fully connected with *broker_C* as the center of the star. The values are very similar to the previous case: *Subscription Table* always performs better, specially in the case of locality 100%. Even here, with locality n%, that is where every broker is interested in the messages, is where the values are pretty similar, specially between *Subscription Table* and *Flooding*.

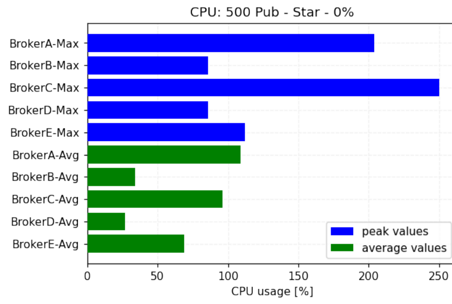
Regarding the CPU usage, [Figure 22] shows the value of every single broker of every extension when there is a Star conformation, with *broker_C* as Root, and 100% locality. As usual with 100% locality *Subscription Table* shows the best performance, with only *broker_A* that has a significant impact on the CPU while the other brokers are very low, specially with the average values. *Flooding* and *HiveMQ-Benchamark* have higher values, specially on the Root *broker_C*, but *Flooding* performs a little better with lower average and peak values. Analogous situation with 0% locality, [Figure 23]. *Subscription Table* has the lowest peak and average values, followed by the *Flooding* extension and finally *HiveMQ-Benchmark*. In every extension in clear that *Broker_A*, *broker_C* and *broker_E* are the ones with an average consumption that is much higher than the other brokers, that are not interest in the messages.



(a) Subscription Table

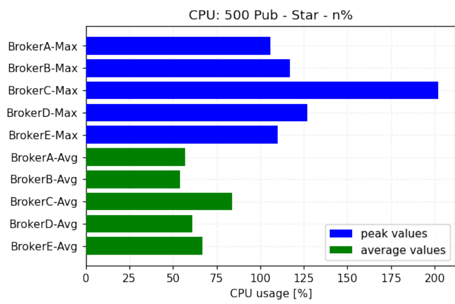


(b) Flooding

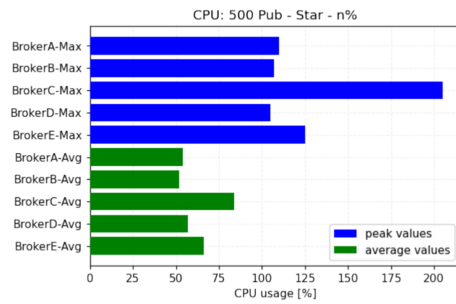


(c) HiveMQ-Benchmark

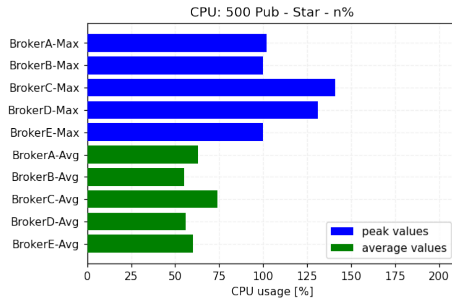
Figure 23: CPU usage with Star topology and 0% locality



(a) Subscription Table



(b) Flooding



(c) HiveMQ-Benchmark

Figure 24: CPU usage with Star topology and n% locality

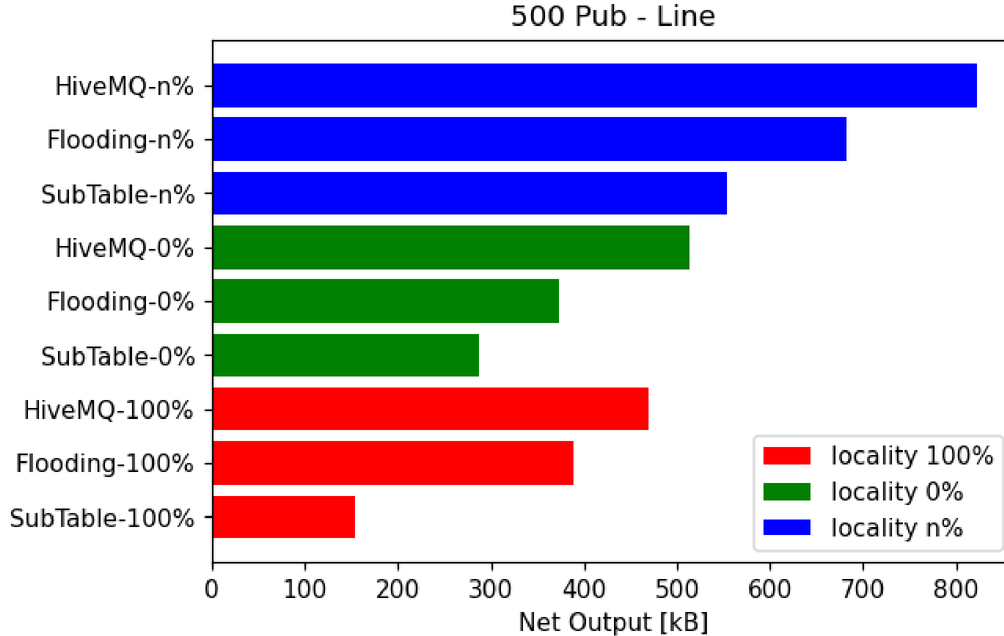


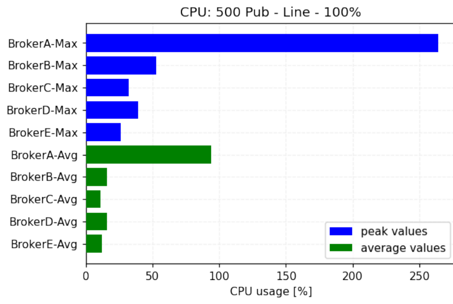
Figure 25: 500 Publishes, Line topology

With locality n% *HiveMQ-Benchmark* is the one that show the best values in [Figure 24]. Every extension has an elevate values with the Root *broker_C*, since is responsible to forward all the publishes to all the brokers. (c) anyway has the lowest peak values and the lowest average values, followed by *Subscription Table* and then *Flooding*.

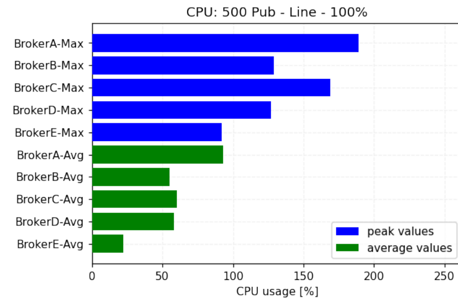
4.3.3 Line

Finally, [Figure 25] shows the Net output when the broker are all linked in Line. Even in this case *Subscription Table* is always the one with the best performance, and as usual the situation where is more evident is in locality 100%. In this situation even when all the brokers are interested in the publishes there is a pretty difference regarding the total Byte in output between *Subscription Table* and *HiveMQ-Benchmark*. *Flooding* is always the second best while *HiveMQ-Benchmark* is always the worse.

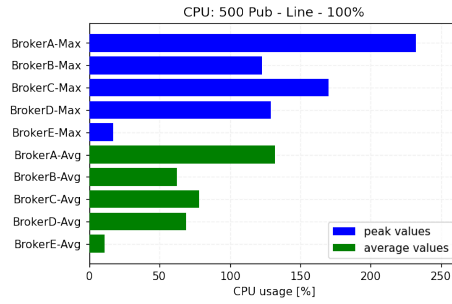
[Figure 26] shows how the extension *Subscription Table* only utilizes *broker_A*, in contrast with (b) and (c). *Subscription Table* has the lowest average values in every broker beside *broker_A*, where he has a very high peak value and a moderate average. *Flooding* and *HiveMQ-Benchmark* have high average values with all the brokers, not only *broker_A*. With locality 0% [Figure 27], all the extensions have very



(a) Subscription Table

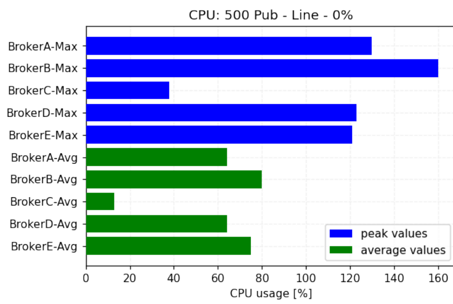


(b) Flooding

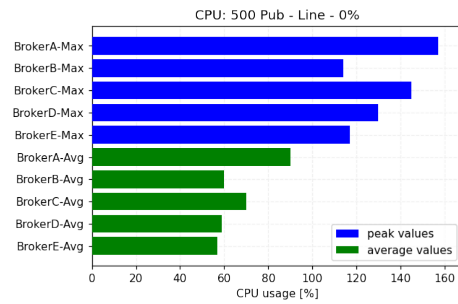


(c) HiveMQ-Benchmark

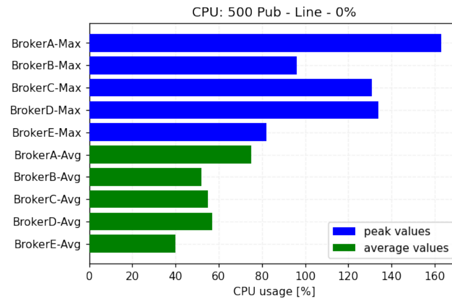
Figure 26: CPU usage with Line topology and 100% locality



(a) Subscription Table

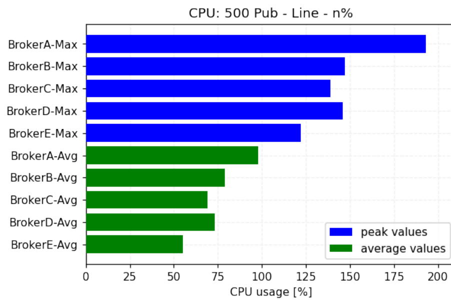


(b) Flooding

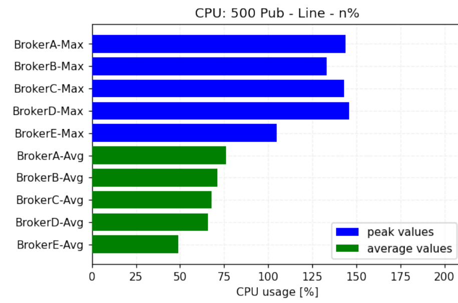


(c) HiveMQ-Benchmark

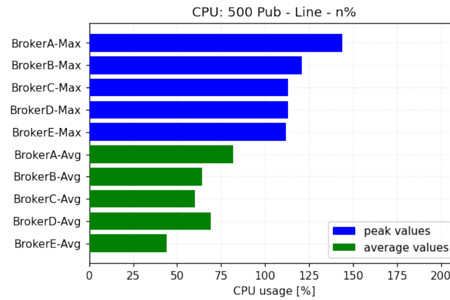
Figure 27: CPU usage with Line topology and 0% locality



(a) Subscription Table



(b) Flooding



(c) HiveMQ-Benchmark

Figure 28: CPU usage with Line topology and n% locality

similar values; to be noticed is the behaviour of *Subscription Table* on *broker_C*: since *broker_A* already knows that its *prev_hop broker_E* is interested in the messages, there is no need to reach *broker_C* that therefore has a very low usage of CPU. Finally [Figure 28] with n% locality, *Subscription Table* is the one with the highest values, where *Flooding* and *HiveMQ-Benchmark* have very similar results.

4.3.4 500 Publishes: considerations

With a higher number of Publishes is more evident the advantages of the extension *Subscription Table* with regards to the other two in terms of network output. *Subscription Table* always uses fewer Bytes, and the difference is very evident with the tests with locality 100%. *Flooding* always performes better than *HiveMQ-Benchamrk* but worse than *Subscription Table*. *HiveMQ-Benchamrk* is always the one that the highest network output, and even with locality n%, that was the case where with 50 Publishes was very similar if not better than the other extensions, now the difference is pretty evident. On the CPU side, with locality 100% *Subscription Table* is the one with the lowest averages values overall, but with a very high peak value and average one on *broker_A*. With 0% locality the values are almost the same in every extension. With n% locality instead, *Subscription Table* has the

highest impact on the CPU while *HiveMQ-Benchmark* has the lowest, even if the differences are not so marked.

4.4 Test 3: 1500 Publishes

In order to take these data, we repeated the past test of 500 Publishes three time in a row: as soon as the 500 messages are published and received by all the brokers interested in them, another 500 publishes are sent and after being received another 500. The results that we are going to expect are the same of the 500 Publishes, but with an higher difference among the extensions, in particular in favor of *Subscription Table* that, even with only 500 messages, was the one that performed better regarding the network output.

4.4.1 Tree

The first set of test uses the tree topology [Figure 29]: as expected *Subscription Table* is the one with the lowest values, and the difference with *HiveMQ-benchmark* is always bigger. In locality 100% is where *Subscription Table* shows the best advantages, and in this situation is more clear than ever: with around 500 kB in output adding all its brokers, is the half of *Flooding* (around 1000 kB) and less than a third than *HiveMQ-Benchmark* (around 1600 kB); there is a difference of more than 1100 kB, which is more than the traffic of *Subscription Table* and *Flooding*. When we are in locality 0% the differences are shortened but is evident that *Subscription Table*, with around 800 kB, is the best choice; not much distant there is *Flooding* with 1000 kB. *HiveMQ-Benchmark* instead is over 1500 kB, almost the double of *Subscription Table* with a difference of around 700 KB. With locality n% is where the differences are the least, as the past tests have shown. However the more the number of messages, the more the differences of Bytes between *Subscription Table* and *HiveMQ-Benchmark*. In this instance, *HiveMQ-Benchmark*, with 2400 kB, consumes 700kB more than *Subscription Table*, at 1700 kB. Very similar, as in the past tests, are the results of *Subscription Table* and *Flooding* with this particular locality: the feature of *Subscription Table* regarding the possibility to send a message only when a broker is interested into it is not so useful when all the brokers receive publishes and every one is interested.

On the CPU side, [Figure 30] shows the usage of CPU with a Tree conformation and 100% locality. As usual, *Subscription Table* only has an high impact on the CPU with *broker_A*, while the other brokers have an average usage very low; the peak value however is the highest among all the extension. *Flooding* and *HiveMQ-*

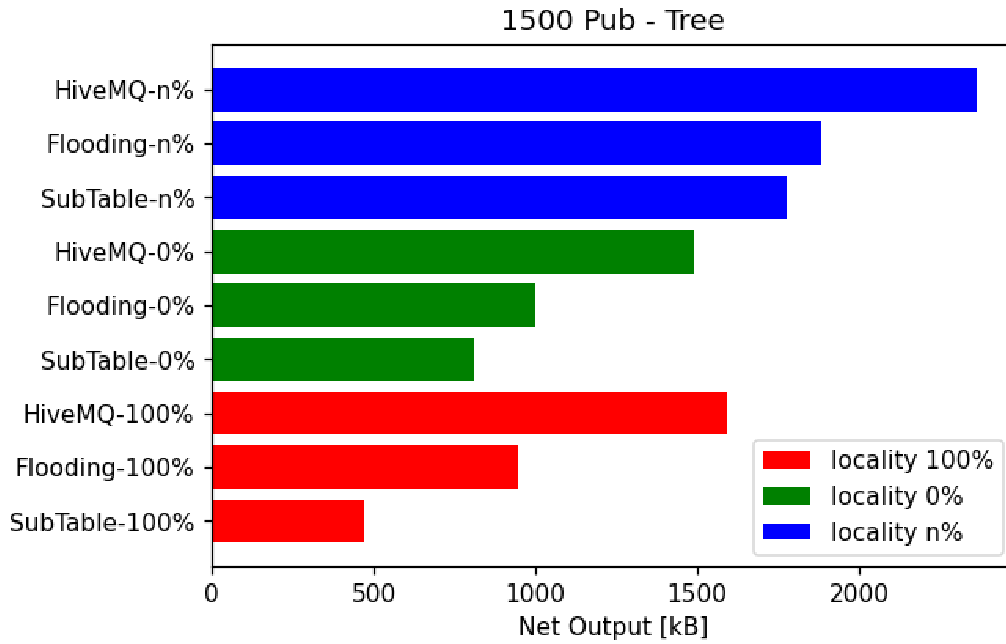
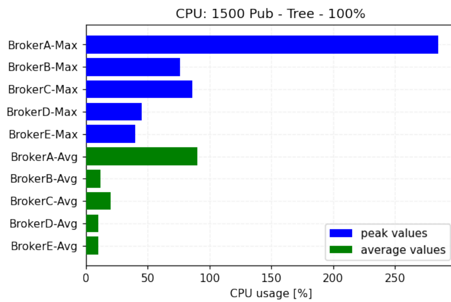
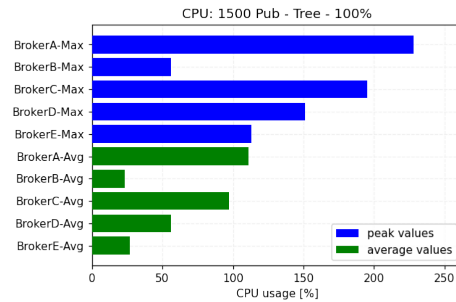


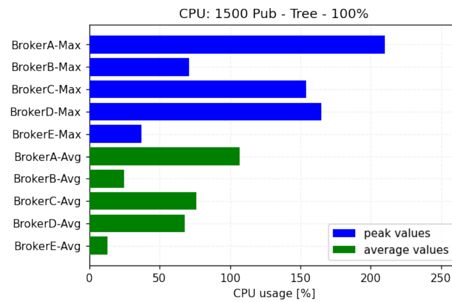
Figure 29: 1500 Publishes, Tree topology



(a) Subscription Table

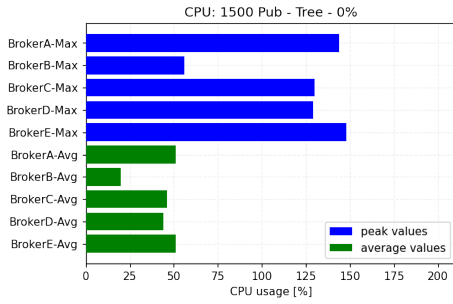


(b) Flooding

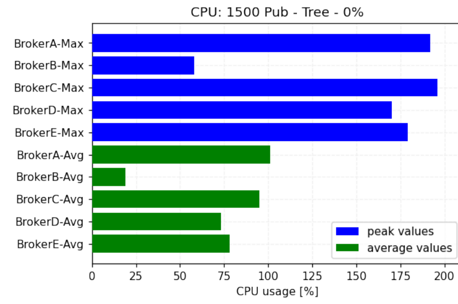


(c) HiveMQ-Benchmark

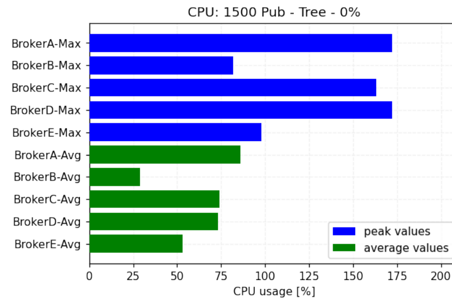
Figure 30: CPU usage with Tree topology and 100% locality



(a) Subscription Table

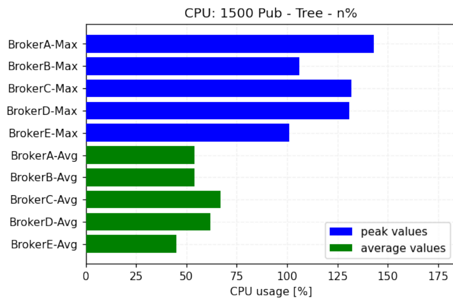


(b) Flooding

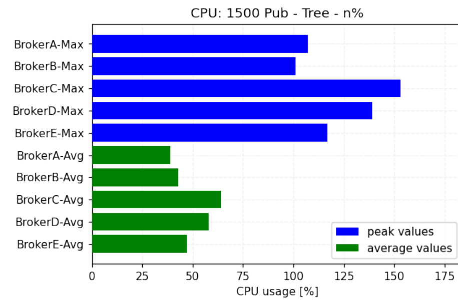


(c) HiveMQ-Benchmark

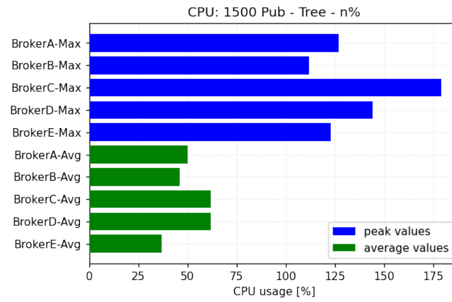
Figure 31: CPU usage with Tree topology and 0% locality



(a) Subscription Table



(b) Flooding



(c) HiveMQ-Benchmark

Figure 32: CPU usage with Tree topology and n% locality

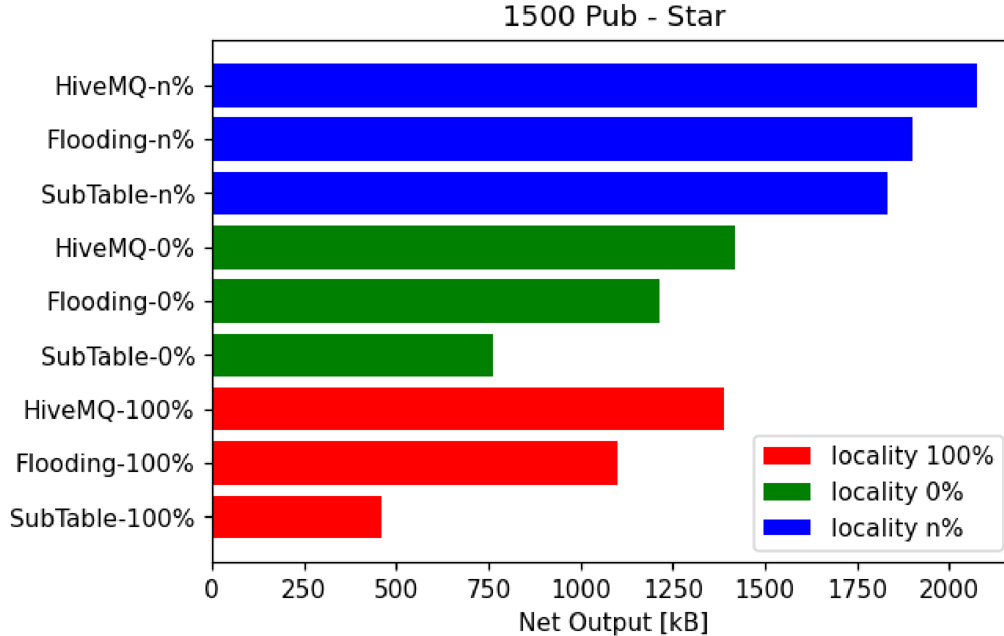


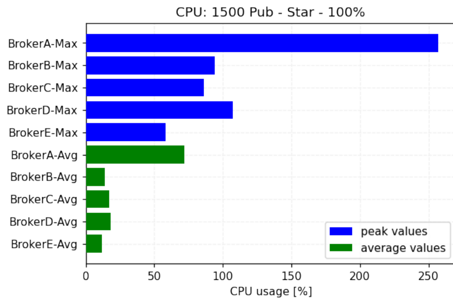
Figure 33: 1500 Publishes, Star topology

Benchmark have very similar values, in which *Broker_A*, *broker_C* and *broker_D* (which is in charge to forward the messages to *broker_E*) have not negligible values. With 0% locality [Figure 31] in all the extensions *broker_B* is the one that consumes the least as expected. However *Subscription Table* has an overall lower average and lower peak values, where *Flooding* is the one with the highest values. In the last locality [Figure 32], the three extensions show similar pattern, but *Flooding* is the one with a slightly lower average than the other two extensions.

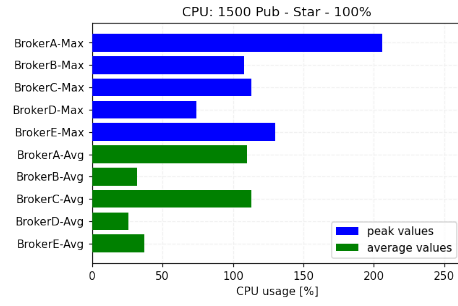
4.4.2 Star

[Figure 33] shows the data regarding the network output when the brokers are connected to form a Star with *broker_C* as center of the Star. The results are very similar at the case of the Tree. The extension *Subscription Table* is the one that always consume less data, in particular in the case of 100% locality. In n% locality, *Flooding* is really close to *Subscription Table*, as already seen in other circumstances. *HiveMQ-Benchmark* instead is always the worst.

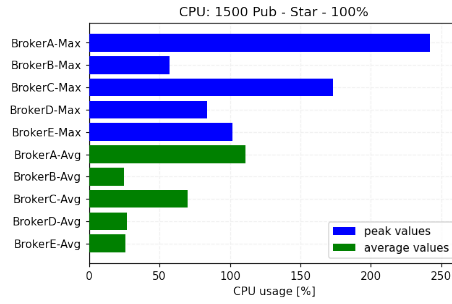
[Figure 34], [Figure 35] and [Figure 36] display the usage of CPU. With 100% locality, extension *Flooding* and the extension *HiveMQ-Benchmark* have much higher peak values than *Subscription Table*. The average values are pretty similar, but a



(a) Subscription Table

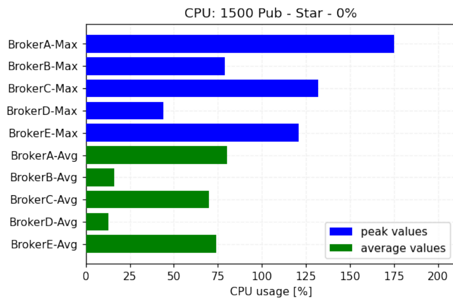


(b) Flooding

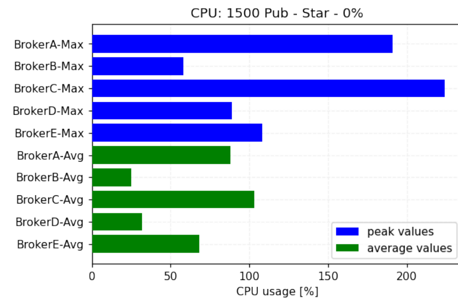


(c) HiveMQ-Benchmark

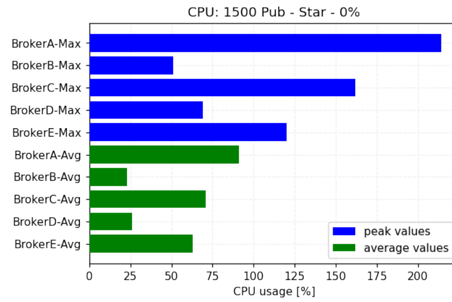
Figure 34: CPU usage with Star topology and 100% locality



(a) Subscription Table

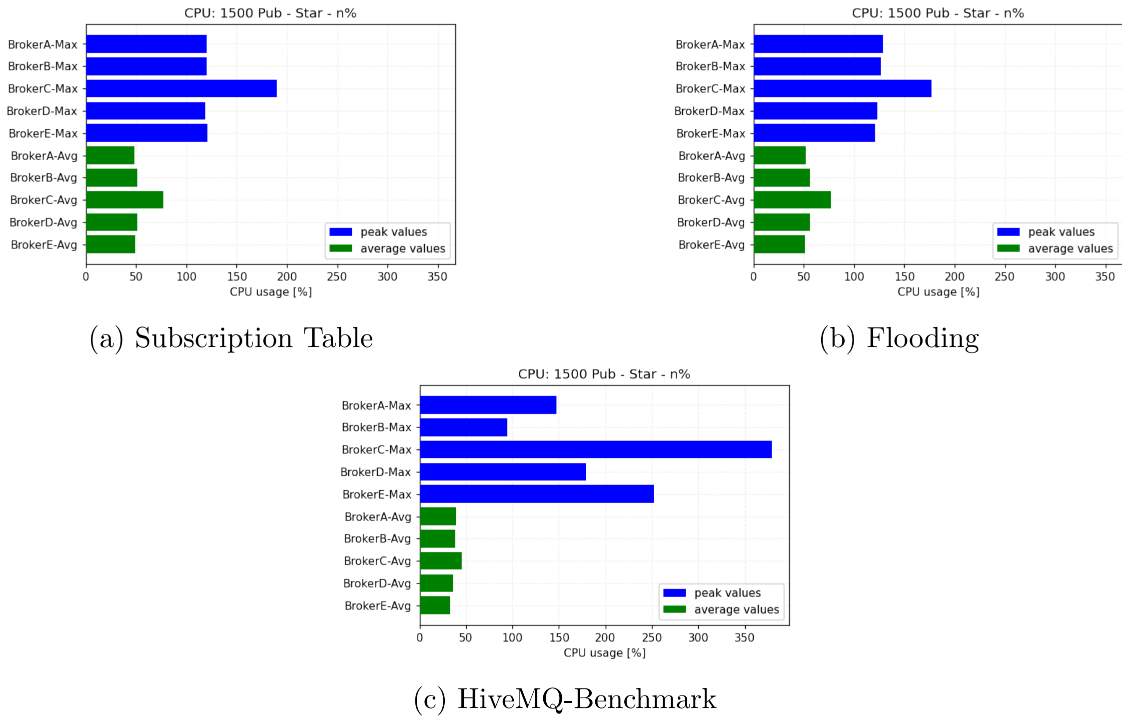


(b) Flooding



(c) HiveMQ-Benchmark

Figure 35: CPU usage with Star topology and 0% locality



(a) Subscription Table

(b) Flooding

(c) HiveMQ-Benchmark

Figure 36: CPU usage with Star topology and n% locality

bit lower on *Subscription Table* that confirms itself as the best extension in 100% locality. With 0% locality *Subscription Table* has a lower peak value and an overall lower average values, but very close to the results of *HiveMQ-Benchmark*; *Flooding* instead has a bit higher values. Finally with n% locality *HiveMQ-Benchmar* has some extremely high peak values with the Root *broker_C*, but has the lowest average values. *Subscription Table* and *Flooding* instead have almost identical results.

4.4.3 Line

The last test is with the broker connected to form a Line [Figure 37].

Even in this case, *Subscription Table* is the extension that uses less data, followed by *Flooding* and then *HiveMQ-Benchmark*. As we have already seen with locality 100% *Subscription Table* is unbeatable with around 400 kB of data in output, when *Flooding* has around 1050 kB and *HiveMQ-Benchmark* around 1500 kB. When the locality is 0%, *Subscription Table* is still the best choice, but *Flooding* performs really close. With n% is when there are fewer differences between the extensions, but nevertheless *Subscription Table* consumes around 800 kB less than *HiveMQ-Benchmark* to do the job.

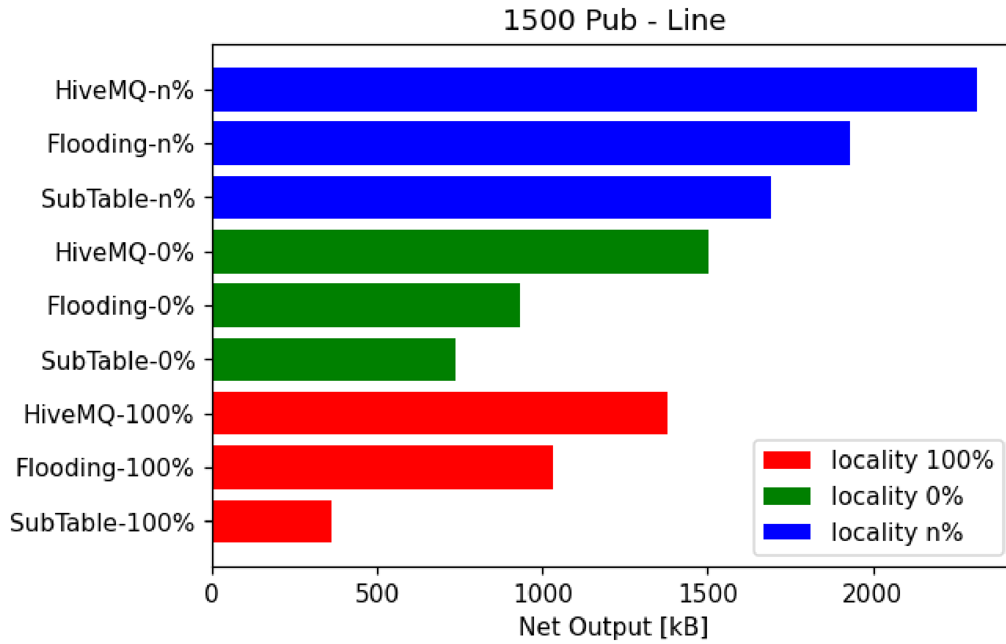
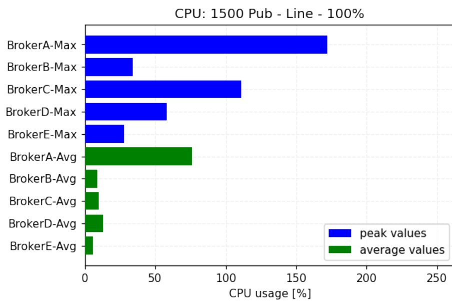
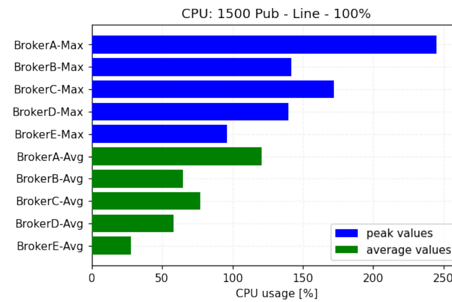


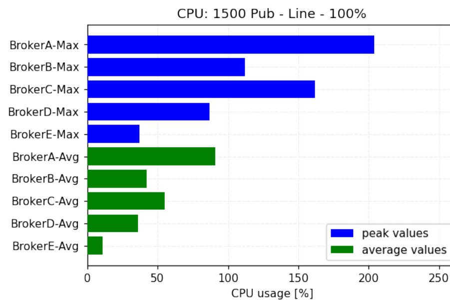
Figure 37: 1500 Publishes, Line topology



(a) Subscription Table

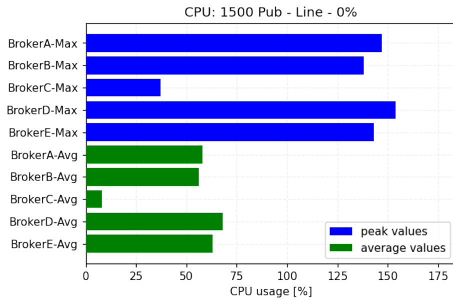


(b) Flooding

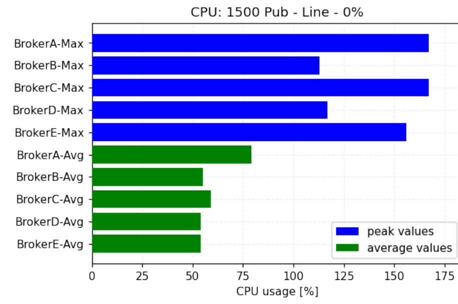


(c) HiveMQ-Benchmark

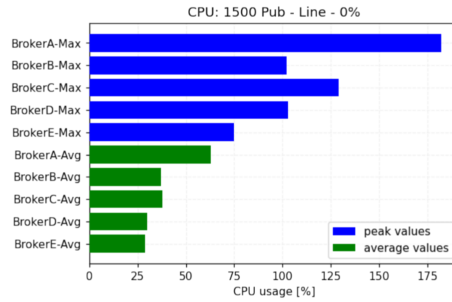
Figure 38: CPU usage with Line topology and 100% locality



(a) Subscription Table

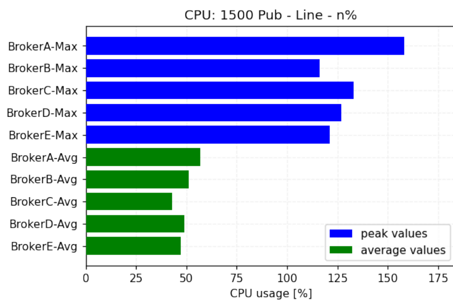


(b) Flooding

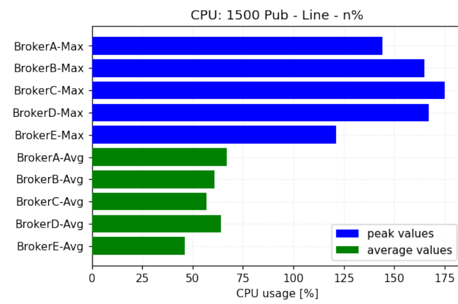


(c) HiveMQ-Benchmark

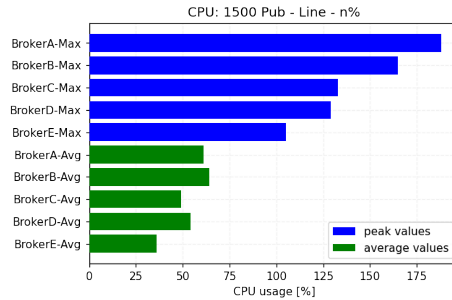
Figure 39: CPU usage with Line topology and 0% locality



(a) Subscription Table



(b) Flooding



(c) HiveMQ-Benchmark

Figure 40: CPU usage with Line topology and n% locality

Taking a look to the CPU graphs [Figure 38], it is clear that *Subscription Table* has a minor impact on the CPU with respect to the other extensions. With 0% locality [Figure 39] peculiar is the CPU consumption of the Root *broker_C* in *Subscription Table*: since *broker_C* is at the top of the line, *broker_A* does not need to reach the Root since it already knows if *broker_B*, that is its *prev_hop*, wants the messages; therefore the messages are published from *broker_A,* received and forwarded by *broker_B*, received and forwarded by *broker_D* and finally received by *broker_E*. Despite all of this, *HiveMQ-Benchmark* is the extension that consumes less CPU resources, with *Subscription Table* that performs slightly better than *Flooding*. [Figure 40] shows how, with n% locality, the results are very similar in all the extensions. *Subscription Table* seems the one with slightly lower values, but very close to the one of *HiveMQ-Benchmark*

4.4.4 1500 Publishes: considerations

The more the messages the more evident are the advantages of the *Subscription Table* extension regarding traffic data with respect to the other extensions. Taking in consideration only the goal to minimize the network traffic, *Subscription Table* is the best choice. To keep in mind that all the tests we have done have many messages published but only on one Topic, topic "test". The real limitation of this *Subscription Table* extension is when there are many clients interested in different topics; in a following section there will be a deepening on this subject.

If the goal is to minimize the network traffic, while keeping an eye on the usage of the resources, *Subscription Table* is still the best choice. In some specific situations could be better the Flooding extension: in n% locality this extension performs really well, very similar to *Subscription Table* in the data output and the CPU usage sometimes is slightly better that *Subscription Table*. But overall the *Subscription Table* extension is the best choice for these kind of tests.

4.5 Summary

This section presents three images that show side by side the network overhead in every extension. Figure 41 represents the data when the topology of the network is a Tree, figure 42 when the topology is a Star and figure 43 when the topology is a Line. On the y axis are highlighted the number of publishes and to which locality the data refers to.

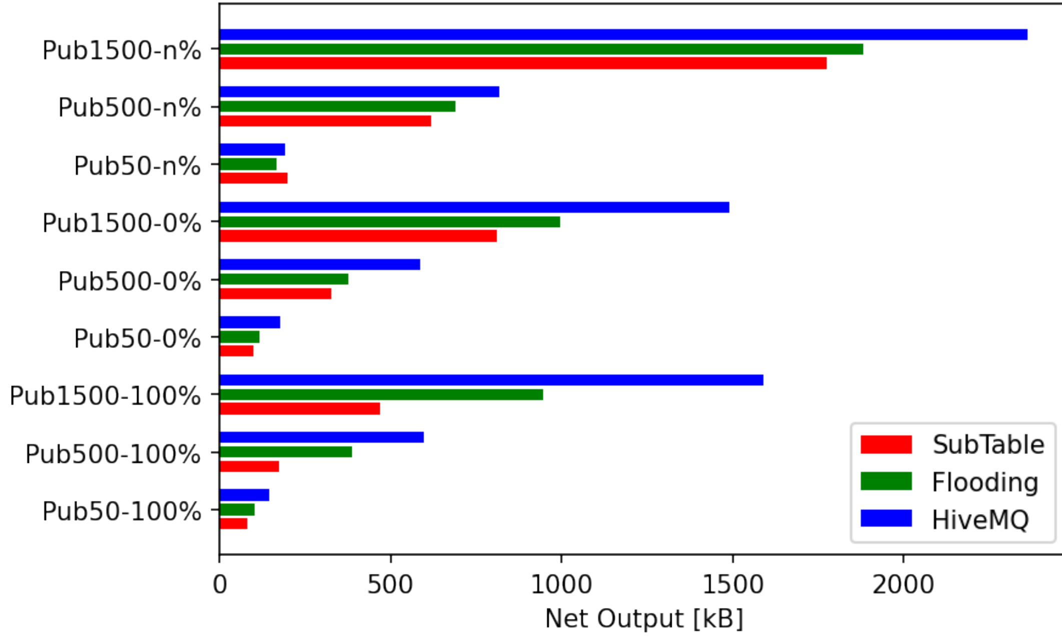


Figure 41: Tree topology

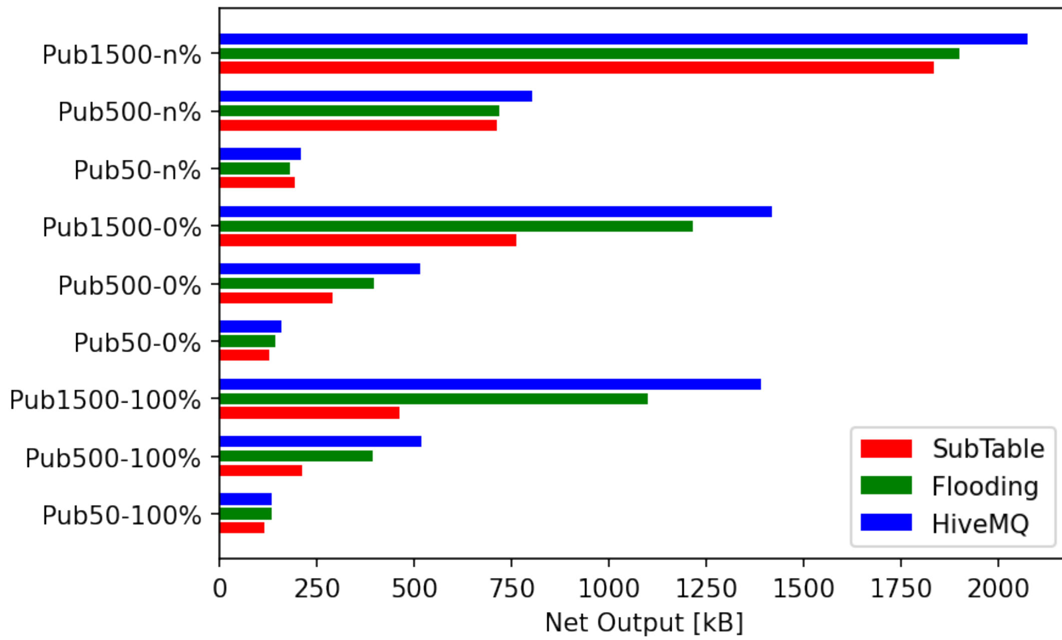


Figure 42: Star topology

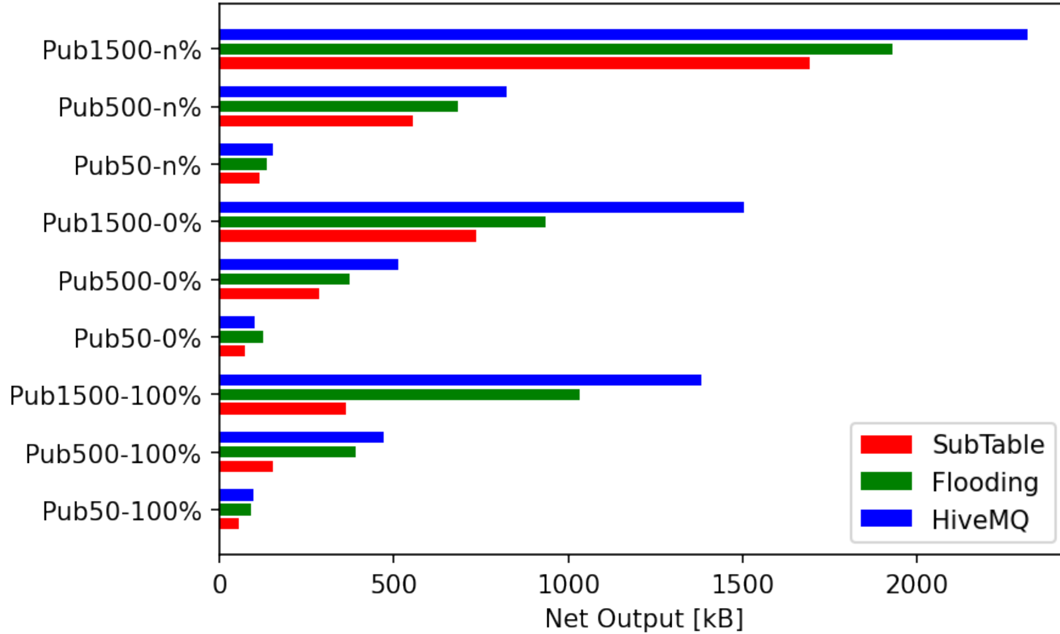


Figure 43: Line topology

4.6 Cluster comparison

HiveMQ offers a default way to connect the brokers: the cluster. This is ideal to create a service with high availability, but not ideal in terms of network consumption. In the following graphs, the data of the extension are taken always with the broker connected as a Star, with the three localities and with 50, 500 and 1500 publishes. The confront is only with the Star because in a HiveMQ's cluster the brokers are fully connected, and therefore the most similar situation is when the brokers are in the Star situation.

[Figure 44], [Figure 45] and [Figure 46] show the network output of all the extensions side by side with the Cluster. It is evident that the Cluster consumes a lot of data more than the extensions, in particular in the n% locality. The Cluster consumes also a lot of data not only to create the cluster, but also to maintain it, probably because of the high reliability and availability that the Cluster offers.

4.7 Subscription Table in-depth

In almost every test the extension *Subscription Table* has a lower network output with respect to the other two extensions. This is specially true the more the Pub-

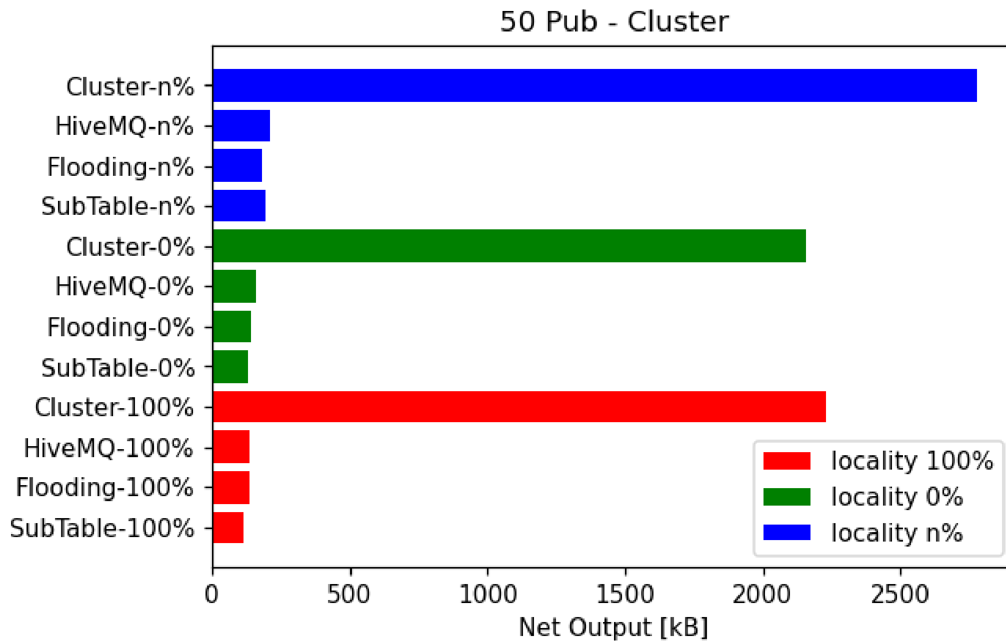


Figure 44: 50 Publishes, Cluster

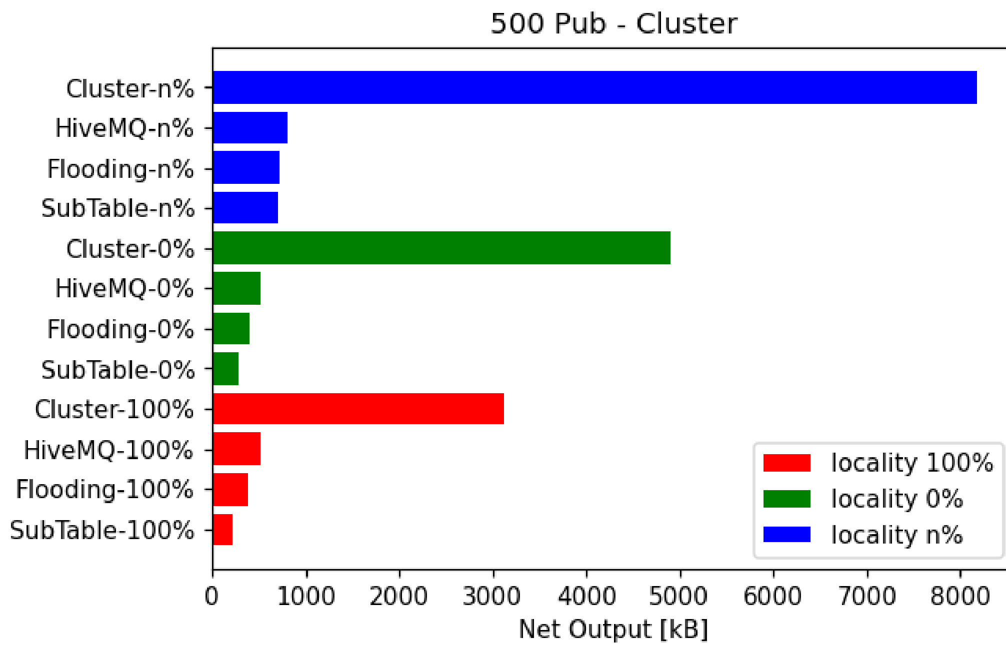


Figure 45: 500 Publishes, Cluster

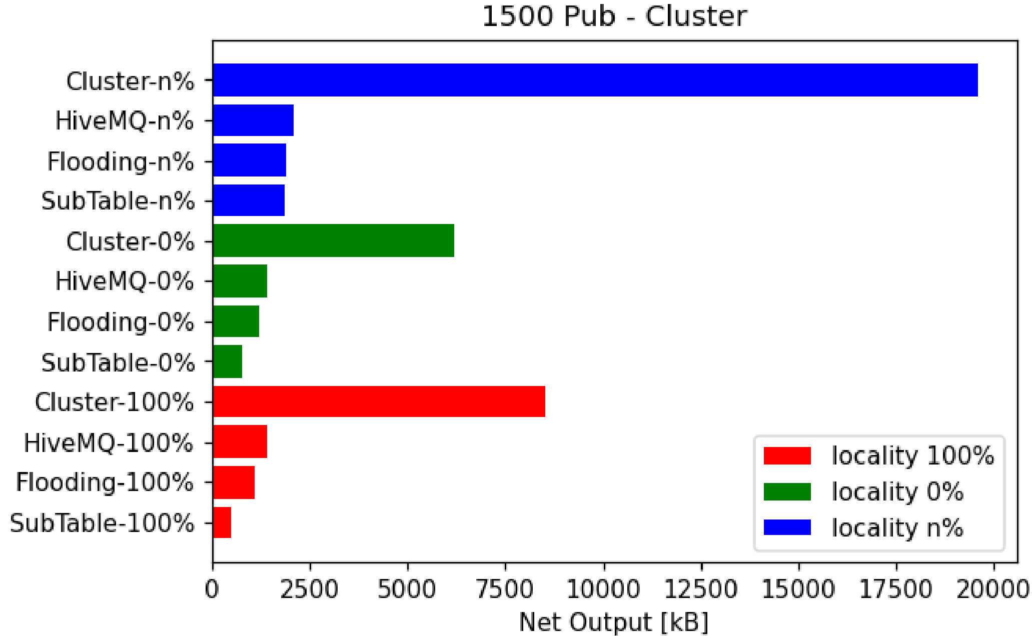


Figure 46: 1500 Publishes, Cluster

lishes are exchanged between the brokers. This is because the brokers know when one of its neighbour is interested in a specific topic, because it has been advised. Analysing the traffic data, taken with *tcpdump*, with Wireshark, is possible to see how this phase of "advising" works. With the following filter only the Publishes the brokers exchange to inform about the interest of a topic are shown.

```
mqtt.topic == "$BROKER/172.17.0.4/$TOPIC" or
mqtt.topic == "$BROKER/172.17.0.2/$TOPIC" or
mqtt.topic == "$BROKER/172.17.0.3/$TOPIC" or
mqtt.topic == "$BROKER/172.17.0.5/$TOPIC" or
mqtt.topic == "$BROKER/172.17.0.6/$TOPIC"
```

Every of this Publish has the Topic in the form of "$BROKER/Ip_address/ $TOPIC", so by putting all the IP addresses of the brokers all the messages are displayed. Keeping in mind that we've used 5 brokers in all our tests, we noticed that in the tests with locality 100% and 0% there are only 4 Publishes. In this tests there is only one client interested in the topic, on *broker_A* with 100% locality and on *roker_E* with 0% locality. The number of Publishes, 4, is the same with the tests of 50 published and with 500, both with the broker linked to build a Tree, a Star and a Line. This is because, taking for instance the locality 100%, with a Tree

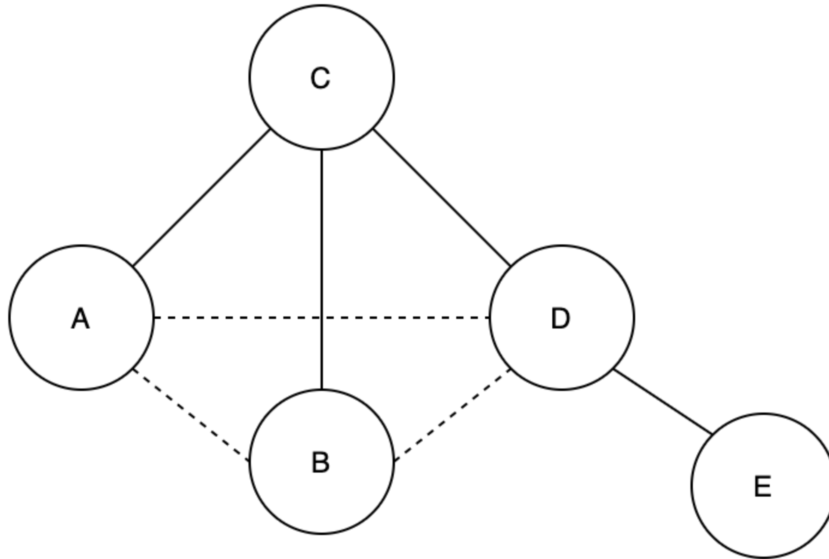


Figure 47: Tree conformation

[Figure 47] we have that *broker_A* receives the Subscription request by a client with topic "X", *broker_A* advises *broker_C*, *C* advises *B* and *D* and finally *D* advises *E*. So given N the number of brokers in the network, there is the need of N-1 Publishes to advise all the brokers. Taking this formula, with n% locality, that is where every broker has a client interested in the topic, there will be 20 Publishes, $4 \cdot 5$. And it is the same that the Wireshark filter shows in the capture with the same filter. The number of these Publishes could be less if some optimizations are implemented. For example, let's have a look at the Star conformation. [Figure 48]

In this situation the 20 messages are the following:

```
1: A->C, C->B, C->D, C->E
2: B->C, C->A, C->D, C->E
3: C->A, C->B, C->D, C->E
4: D->C, C->A, C->B, C->E
5: E->C, C->A, C->B, C->E
```

This means that *Broker_A* receives a *SUB* from a client "Y" with topic "X", so *broker_A* add to one of its table that the client "Y" is interested in "X" and publishes a message to *broker_C* with Topic "$BROKER/172.17.0.4/$TOPIC" and as payload "X". *Broker_C* receives the message, and insert in its table that *broker_A*

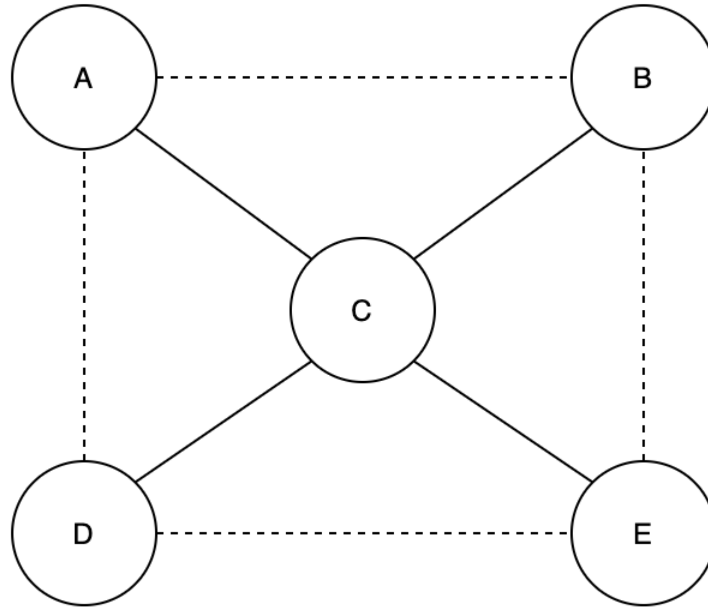


Figure 48: Star conformation

is interested in topic "X", then publishes a message to *broker_B*, *broker_D* and *broker_E*, changing the topic accordingly and with the same payload. *Broker_B*, *broker_D* and *broker_C* will insert in their table that *broker_C* is interested in the topic "X". The same is repeated for all the other Subscriptions on every broker. If another client will then subscribe with the same topic "X" on *broker_A*, the procedure will not be triggered since *broker_A* already has a client interested in that topic so the other brokers have been already advised. With a proper optimization this result can be achieved:

```
1: A->C, C->B, C->D, C->E
2: B->C, C->A
3:
4: D->C
5: E->C
```

In this situation the are no doubles publishes: this can be achieved by not forwarding immediately the publishes; however *broker_C* should keep track of the broker he has already advised in another table and this could cause more disadvantages than advantages in terms of CPU and memory consumption.

To recap, with the extension *Subscription Table* and five brokers we have discovered that every new Subscription with a different topic on a broker triggers 4 Publishes, and 4 *ACKs* since we are using *QoS1*, in order to inform the rest of the network. To abstract, given N is the number of brokers in a network, a new Topic triggers at minimum $N - 1$ publishes and at maximum $N \cdot (N - 1)$ if in every broker there is at least a client interested in the topic. Indicating with "n" the number of brokers where there is at least a client interested in the new topic, we have $n \cdot (N - 1)$ Publishes. All of this can have some negative effects: since every new topic on a broker causes 4 new publishes, if there are very few publishes in the network with such topic, this extension can have a bigger network output than without the extension. Another consideration to keep in mind is that, if there is a very large number of different topic on a broker, this can cause an excessive usage of resources, in particular of memory, since every new topic is added to a table and the same table is checked every time there is a new subscription.
After all of these considerations, *Subscription Table* remains the best extension among the three, to be avoided only in very specific cases, where the *Flooding* extension can be preferred.

4.8 End to end Delay

We have used MQTT-Box for every test. Thanks to this software, we have also collected the timestamps of when the messages are sent by the publishers and when they are received by the subscribers interested. In this way is possible to calculate the end to end delay to receive all the messages, and to calculate how many messages are sent and received per second.
To be noticed that with MQTT-Box in every test there must be set the number of messages to publish and the Run time, which is the number of seconds in which the messages must be published. In all the test we have performed, the Run time is always 5 seconds; this means that in the tests with 50 Publishes there are 10 PUB per second while with 500 Publishes there are 100 PUB per second in average. The tests with 1500 Publishes are done by repeating the 500 case three times in a row, therefore the results we are gonna to show refers only to the tests with 50 or 500 Publishes.
In addition to this, the data displayed in the following sections refers only to the tests with locality 100% and 0%. With locality n%, the results where very similar among all the subscribers, one in each broker, because by the time we manually started the fifth and last set of Publishes, the previous publishes were already received in large part by the subscribers, and therefore the data obtained at the end would not reflect the real end to end delay of the test.

4.8.1 50 Publishes

The following tables show the data regarding the tests with 50 Publishes and the three topologies: Tree, Star and Line.
The column *Extension* indicates which extension the data refers to among *Subscription Table, Flooding* and *HiveMQ-Benchmark*. The column *Locality* indicates which kind of locality between 0% and 100%; *Publishes* is the number of Publishes in the test, 50 or 500; *Pub Time* is in how many seconds the PUB are sent; *Pub/s* indicates the ratio between the number of Publishes and the Time spent sending them, so, since the Pub Time is always 5 in all the tests, is 10 in the tests with 50 PUB and 100 with 500 PUB; *Total Time* are the seconds elapsed since the sending and the reception of all the messages at the interested subscriber; *Mgs/s* indicates the ratio between the number of Publishes and the Total Time to receive them at the subscriber.

Table [1] shows the data of the tests with 50 Publishes and Tree topology. The Total Times of *Flooding* and *HiveMQ-Benchmark* are very similar, but the ones of *Subscription Tables* are lower, specially in the test with locality 100%. This results in more messages received every second with the *Subscription Table* extension.
With the Star topology, Table [2], the Total Time of *Flooding* is very close to the one of *Subscription Table* in the 100% locality, with *HiveMQ-Benchmark* that instead uses a second more that the other extensions. With 0% locality instead, all the extensions performs really similar and therefore have almost an identical Msg/s result.
Finally, with a Line topology as shown in Table [3], we have that *Subscription Table* performs much better both with locality 100% and 0%. *Flooding* has a very similar Total Time with *HiveMQ-Benchmark* with locality 100% but much higher with 0% locality.

To sum up, with 50 Publishes *Subscription Table* presents a lower Total Time, and consequently a higher Msg/s, than the other extensions with 100% locality.
With 0% locality the results vary depending on the topology of the tree: with the Star topology all the extensions presents a very similar Total Time, while with the other topologies *Subscription Table* has a lower Total Time, but the differences are smaller than with 100% locality.

Table 1: 50 Publishes, Tree topology

Extension	Locality	Publishes	Pub Time[s]	Pub/s	Total Time[s]	Msg/s
SubTable	100%	50	5	10	5,632	8,89
Flooding	100%	50	5	10	6,758	7,40
HiveMQ	100%	50	5	10	6,811	7,34
SubTable	0%	50	5	10	6,78	7,37
Flooding	0%	50	5	10	7,46	6,70
HiveMQ	0%	50	5	10	7,473	6,69

Table 2: 50 Publishes, Star topology

Extension	Locality	Publishes	Pub Time[s]	Pub/s	Total Time[s]	Msg/s
SubTable	100%	50	5	10	5,131	9,745
Flooding	100%	50	5	10	5,698	8,775
HiveMQ	100%	50	5	10	6,532	7,655
SubTable	0%	50	5	10	6,607	7,568
Flooding	0%	50	5	10	6,736	7,423
HiveMQ	0%	50	5	10	6,577	7,602

Table 3: 50 Publishes, Line topology

Extension	Locality	Publishes	Pub Time[s]	Pub/s	Total Time[s]	Msg/s
SubTable	100%	50	5	10	5,267	9,493
Flooding	100%	50	5	10	6,514	7,676
HiveMQ	100%	50	5	10	6,726	7,434
SubTable	0%	50	5	10	6,801	7,352
Flooding	0%	50	5	10	8,769	5,702
HiveMQ	0%	50	5	10	7,487	6,782

Table 4: 500 Publishes, Tree topology

Extension	Locality	Publishes	Pub Time[s]	Pub/s	Total Time[s]	Msg/s
SubTable	100%	500	5	100	13,274	33,668
Flooding	100%	500	5	100	17,35	28,818
HiveMQ	100%	500	5	100	17,631	28,359
SubTable	0%	500	5	100	18,311	27,306
Flooding	0%	500	5	100	18,324	27,287
HiveMQ	0%	500	5	100	19,338	25,856

4.8.2 500 Publishes

The following three tables show the data of the tests with 500 Publishes.
Table[4] is with the Tree topology. With 100% locality *Subscription Table* has a much lower Total Time than both *Flooding* and *HiveMQ-Benchmark*; with 0% locality instead the results are all close to each other, specially between *Flooding* and *Subscription Table*.
Analogous situation is with the Star topology, Table [5]. With 100% locality *Subscription Table* has a much lower Total Time and therefore a much higher number of messages received every second at the subscriber. With 0% locality *HiveMQ-Benchmark* is the worst between the extensions with an higher Total Time, while *Flooding* and *Subscription Table* show similar results.
Finally, with a Line topology (Table [6]), *Subscription Table* performs the best both with locality 100% and 0%; *HiveMQ-Benchmark* is the extensions with the highest values of Total Time among the three extensions.

At the end, *Subscription Table* presents the lowest end to end delay among the three extensions in almost every situation. With 100% locality is always better *Subscription Table*, while with 0% locality, besides the case with the Line topology, the Total Times are similar among all the extensions.

4.8.3 Cluster

Table [7] shows the Total Times and the Messages received per second using the HiveMQ's Cluster. Taking in comparison the Total Times of *Subscription Table* with the Star topology, that is the extensions with the best results in the topology that is the most similar to the Cluster one, the Cluster has in every situation a lower

Table 5: 500 Publishes, Star topology

Extension	Locality	Publishes	Pub Time[s]	Pub/s	Total Time[s]	Msg/s
SubTable	100%	500	5	100	13,649	36,633
Flooding	100%	500	5	100	18,012	27,759
HiveMQ	100%	500	5	100	21,59	23,159
SubTable	0%	500	5	100	16,77	29,815
Flooding	0%	500	5	100	17,125	29,197
HiveMQ	0%	500	5	100	19,9	25,126

Table 6: 500 Publishes, Line topology

Extension	Locality	Publishes	Pub Time[s]	Pub/s	Total Time[s]	Msg/s
SubTable	100%	500	5	100	14,66	34,106
Flooding	100%	500	5	100	18,905	26,448
HiveMQ	100%	500	5	100	24,97	20,024
SubTable	0%	500	5	100	17,524	28,532
Flooding	0%	500	5	100	21,463	23,296
HiveMQ	0%	500	5	100	29,696	16,837

Table 7: Cluster

Extension	Locality	Publishes	Pub Time[s]	Pub/s	Total Time[s]	Msg/s
Cluster	100%	50	5	10	5,021	9,958
Cluster	0%	50	5	10	5,01	9,98
Cluster	100%	500	5	100	13,966	35,80
Cluster	0%	500	5	100	14,228	35,142

Total Time. While with locality 100% the results almost identical, with locality 0% the differences are notable.
This can be caused by the fact that the *Subscription Table* extension uses MQTT publishes with QoS1 to exchange information among the brokers while the Cluster uses its own proprietary protocol; with Quality of Service 1 every Publish packet needs an ACK, and therefore the time necessary to reach the broker that has the client interested in the messages can be higher. In fact, with locality 100%, *Subscription Table* doesn't have to send any Publish message to any broker since the subscriber is on the same broker that receives the first publishes, and therefore with such locality the Total Times are very similar to the ones of the Cluster.

5 Conclusions

With more and more "things" connected, MQTT, the main protocol for IoT, needs to move from a classical broker centric model to a distributed one with more brokers connected between themselves and closer to the clients. In this work we have developed three extensions for HiveMQ, in order to create a network of distributed brokers that is completely transparent to the clients.
All the extensions allow the replication of messages between the brokers, are loop-free and do not need a static topology. As long as a broker is connected through a bridge to at least another broker, the topology of the network will change in order to select as root the broker with the best Capacity. All the extensions are also resistant to failures: if the Root, for any reason, is no longer operative, the brokers will create another Tree electing a new Root; this is possible only if there are enough bridge connections since it is not possible to modify the bridge extension file to create new bridges at runtime.

The first extension, *HiveMQ-Benchmark*, is the simplest way to connect the brokers in a loop-free network. As the name suggests it has been used in the tests as a benchmark of HiveMQ.
The second extension, *Flooding*, presents some improvements. The main one is, thanks to the presence of more filters in the bridge configuration file, the possibility to send a Publish to only one broker.
In the third and last extension, *Subscription Table*, every broker has a table of subscriptions in which are marked the brokers that are interest in a Topic.
We have conducted several tests keeping under observations the network overhead and the CPU usage of the brokers. The tests consist in publishing a different number of messages (50, 500 and 1500) on different brokers (all on *broker_A*, all on *broker_E* and on every broker in equal number) in different network topologies (tree, star and line).

It comes out that the *Subscription Table* extension is the best in almost every situation. The more the messages are published the more advantages it has, with the same number of topics.
The downside of this extension is that every broker needs to save in its table which topic is of interest to whom. This, if the number of topics is very high, could lead to an excessive usage of CPU and memory resources, and to an increase of network overhead if the publishes on such topics are very few.
In these rare circumstances could be preferred the *Flooding* extension. This extension performs worse than *Subscription Table* almost every time regarding the

network overhead, but much better than *HiveMQ-Benchmark*.
HiveMQ-Benchmark is by far the worst one in network overhead, and the more the messages the greater is the difference with the other two extensions.
Regarding the CPU consumption, no extension presents an excessive CPU overload. *Subscription Table* performs better in the tests with locality 100%, where sometimes is a little more expensive than the other extensions with 0% or n%.
Overall, *Subscription Table* is in almost every situation the best extension to use with the HiveMQ's enterprise Bridge extension to create a network of distributed brokers.

In conclusion HiveMQ's MQTT brokers could not be the best candidate for creating a network of distributed brokers. It is possible to create it using the Bridge extension along with one of the extension we developed. HiveMQ offers in fact the possibility to develop and implement a plug in without the need to rewrite the code from scratch. The extensions are written in Java, therefore can be personalized in various ways, depending on the developer. I myself have developed three different extensions, adding functionalities from one to another. At this link [18] is possible to find all the APIs available for the development. They are many, useful to create interceptors or publish messages for instance. One problem we have found however, is the possibility to create on the fly only a limited type of messages. So we could create Publish messages but not *PINGREQ* ones, like it was done in [16], and this can cause an increment in the network overhead because of the size of a Publish packet. Another very important problem is the fact that, with the bridge extension, is not possible to send a message to only one brokers, but the message is sent to all the brokers that satisfy the filter in the bridge configuration file: therefore if, like in the first extension *HiveMQ-Benchmark*, all the filters in the file are "#", all the messages are sent and received by everyone and this can cause other problems, specially in the possibility of creating loops of messages. To avoid this situation, we had to implement a publish interceptor in every broker in order to prevent the reception of the messages that would cause the creation of loops.

5.1 Future Works

Starting with the extensions of this work, more brokers can be tested on different machines instead of only on a single one. This would be more like a real deployment; moreover, since on a single machine all the brokers have the same resources, this could lead to different network topologies since every broker could be the Root.
The same tests that we performed can be replicated with others MQTT brokers like verneMQ or EMQX, and the results can be compared in order to find the best to

create a network of distributed brokers.
Other metrics can be tested besides the network overhead and the CPU consumption. For instance the delay from when the Publish in sent by a broker and when the same is received by the subscribers (in this work we presented some data regarding the end to end delay, but the main focus of this work was, beside the creation of the network of distributed brokers, to study the network overhead); there can be differences in the results depending on the network topology: if the subscriber is on one leaf and the publish is in a broker on the opposite side of the Tree the delay can be higher than in the case where publish and subscribe are on the same broker. By studying these results, someone can propose a different way to build the Tree: if the traffic between some brokers is more delay-sensitive respect to the traffic between others, the Tree can be build by trying to minimize the delay between specific brokers, by using a direct bridge for instance, and not focusing on having the lowest RTT with the Root of the Tree.
One big limitation of this work is that the Bridge configuration file cannot be modified at runtime, therefore a new broker that has not already an active bridge with another broker, cannot be part of the network. Hence, future works can study in deep this aspect and find an alternative solution to bridge the brokers.

List of Abbreviations

MQTT: Message Queuing Telemetry Transport

IoT: Internet of Things

MEC: Multi-Access Edge Computing

ICT: Information and Communication Technologies

QoS: Quality of Service

IDE: Integrated Development Environment

COAP: Constrained Application Protocol

TCP: Transmission Control Protocol

IP: Internet Protocol

ACK: Acknowledge

PUB: Publish

STP: Spanning Tree Protocol

CPU: Central Processing Unit

List of Figures

Figure 1: MQTT architecture with a broker in the middle and 5 clients connected to him. Every client can Subscribe and Publish

Figure 2: Spanning Tree Protocol

Figure 3: 5 brokers connected to create a tree topology; the dash lines indicate an open bridge that could be used, the continuous lines indicate the active bridges.

Figure 4: the three topologies used to test my extensions. (a) is a tree, (b) is a star and (c) a line. The dash lines indicate an open bridge that could be used, the continuous lines indicate the active bridges.

Figure 5: output data obtained summing every broker's output traffic in the tests with 50 publishes with Tree topology and every locality.

Figure 6: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 50 Publishes, tree topology and 100% locality.

Figure 7: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 50 Publishes, tree topology and 0% locality.

Figure 8: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 50 Publishes, tree topology and n% locality.

Figure 9: output data obtained summing every broker's output traffic in the tests with 50 publishes with Star topology and every locality.

Figure 10: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 50 Publishes, Star topology and 100% locality.

Figure 11: CPU usage of every broker; the blue lines indicate the maximum

value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 50 Publishes, Star topology and 100% locality.

Figure 12: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 50 Publishes, Star topology and n% locality.

Figure 13: output data obtained summing every broker's output traffic in the tests with 50 publishes with Line topology and every locality.

Figure 14: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 50 Publishes, Line topology and 100% locality.

Figure 15: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 50 Publishes, Line topology and 100% locality.

Figure 16: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 50 Publishes, Line topology and n% locality.

Figure 17: output data obtained summing every broker's output traffic in the tests with 500 publishes with Tree topology and every locality.

Figure 18: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 500 Publishes, Tree topology and 100% locality.

Figure 19: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 500 Publishes, Tree topology and 0% locality.

Figure 20: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs

referes to the test with 500 Publishes, Tree topology and n% locality.

Figure 21: output data obtained summing every broker's output traffic in the tests with 500 publishes with Star topology and every locality.

Figure 22: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 500 Publishes, Star topology and 100% locality.

Figure 23: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 500 Publishes, Star topology and 0% locality.

Figure 24: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 500 Publishes, Star topology and n% locality.

Figure 25: output data obtained summing every broker's output traffic in the tests with 500 publishes with Line topology and every locality.

Figure 26: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 500 Publishes, Line topology and 100% locality.

Figure 27: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 500 Publishes, Line topology and 0% locality.

Figure 28: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 500 Publishes, Line topology and n% locality.

Figure 29: output data obtained summing every broker's output traffic in the tests with 1500 publishes with Tree topology and every locality.

Figure 30: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b)

to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 1500 Publishes, Tree topology and 100% locality.

Figure 31: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 1500 Publishes, Tree topology and 0% locality.

Figure 32: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 1500 Publishes, Tree topology and n% locality.

Figure 33: output data obtained summing every broker's output traffic in the tests with 1500 publishes with Star topology and every locality.

Figure 34: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 1500 Publishes, Star topology and 100% locality.

Figure 35: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 1500 Publishes, Star topology and 0% locality.

Figure 36: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 1500 Publishes, Star topology and n% locality.

Figure 37: output data obtained summing every broker's output traffic in the tests with 1500 publishes with Line topology and every locality.

Figure 38: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 1500 Publishes, Line topology and 100% locality.

Figure 39: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 1500 Publishes, Line topology and 0% locality.

Figure 40: CPU usage of every broker; the blue lines indicate the maximum value, the green lines the average value. (a) refers to the extension "SubTable", (b) to extension "Flooding" and (c) to extension HiveMQ-Benchmark. These graphs referes to the test with 1500 Publishes, Line topology and n% locality.

Figure 41: Network overhead in every test with the Tree topology.

Figure 42: Network overhead in every test with the Star topology.

Figure 43: Network overhead in every test with the Line topology.

Figure 44: output data obtained summing every broker's output traffic in the tests with 50 publishes with Star topology and every locality and compared with the traffic generated by the Cluster.

Figure 45: output data obtained summing every broker's output traffic in the tests with 500 publishes with Star topology and every locality and compared with the traffic generated by the Cluster.

Figure 46: output data obtained summing every broker's output traffic in the tests with 1500 publishes with Star topology and every locality and compared with the traffic generated by the Cluster.

Figure 47: 5 brokers connected to create a Tree topology; the dash lines indicate an open bridge that could be used, the continuous lines indicate the active bridges.

Figure 48: 5 brokers connected to create a Star topology; the dash lines indicate an open bridge that could be used, the continuous lines indicate the active bridges.

References

(1) Hunkeler, U.; Truong, H. L.; Stanford-Clark, A. MQTT-S—A publish/subscribe protocol for Wireless Sensor Networks. **2008**, 791–798.

(2) Shelby, Z.; Hartke, K.; Bormann, C. The constrained application protocol (CoAP). **2014**.

(3) Vinoski, S. Advanced message queuing protocol. *IEEE Internet Computing* **2006**, *10*, 87–89.

(4) Yokotani, T.; Sasaki, Y. Comparison with HTTP and MQTT on required network resources for IoT. **2016**, 1–6.

(5) Naik, N. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. **2017**, 1–7.

(6) Kokkonis, G.; Chatzimparmpas, A.; Kontogiannis, S. Middleware IoT protocols performance evaluation for carrying out clustered data. **2018**, 1–5.

(7) Thangavel, D.; Ma, X.; Valera, A.; Tan, H.-X.; Tan, C. K.-Y. Performance evaluation of MQTT and CoAP via a common middleware. **2014**, 1–6.

(8) De Oliveira, D. L.; Veloso, A. F. d. S.; Sobral, J. V.; Rabêlo, R. A.; Rodrigues, J. J.; Solic, P. Performance evaluation of MQTT brokers in the Internet of Things for smart cities. **2019**, 1–6.

(9) Sommer, P.; Schellroth, F.; Fischer, M.; Schlechtendahl, J. Message-oriented middleware for industrial production systems. **2018**, 1217–1223.

(10) Mishra, B. Evaluating the Performance of MQTT Brokers. **2018**, 37.

(11) Kawaguchi, R.; Bandai, M. A distributed MQTT broker system for location-based IoT applications. **2019**, 1–4.

(12) Rausch, T.; Nastic, S.; Dustdar, S. Emma: Distributed qos-aware mqtt middleware for edge computing applications. **2018**, 191–197.

(13) Koziolek, H.; Grüner, S.; Rückert, J. A comparison of MQTT brokers for distributed IoT edge computing. **2020**, 352–368.

(14) Banno, R.; Sun, J.; Takeuchi, S.; Shudo, K. Interworking Layer of Distributed MQTT Brokers. *IEICE TRANSACTIONS on Information and Systems* **2019**, *102*, 2281–2294.

(15) An, H.; Sa, W.; Kim, S. Design and Implementation of RPL-based Distributed MQTT Broker Architecture. *Journal of Korea Multimedia Society* **2018**, *21*, 1090–1098.

(16) Longo, E.; Redondi, A. E.; Cesana, M.; Arcia-Moret, A.; Manzoni, P. MQTT-ST: A spanning tree protocol for distributed MQTT brokers. **2020**, 1–6.

(17) Fidler, E.; Jacobsen, H.-A.; Li, G.; Mankovski, S. The PADRES Distributed Publish/Subscribe System. **2005**, 12–30.

(18) HiveMQ HiveMQ's API `https://www.hivemq.com/docs/hivemq/4.5/extensions-javadoc/index.html`.