BORDER: A Benchmarking Framework for Distributed MQTT Brokers

Edoardo Longo, Alessandro Enrico Cesare Redondi, *Senior Member, IEEE*, Matteo Cesana, *Senior Member, IEEE*, and Pietro Manzoni, *Senior Member, IEEE*

***Abstract*—Message queuing telemetry transport (MQTT), one of the most popular application layer protocols for the Internet of Things, works according to a publish/subscribe paradigm where clients connect to a centralized broker. Sometimes (e.g., in high scalability and low-latency applications), it is required to depart from such a centralized approach and move to a distributed one, where multiple MQTT brokers cooperate together. Many MQTT brokers (both open source or commercially available) allow to create such a distributed environment: however, it is challenging to select the right solution due to the many available choices. This article proposes, therefore benchmarking framework for distributed MQTT brokers (BORDER), a framework for creating and evaluating distributed architectures of MQTT brokers with realistic and customizable network topologies. Based on isolated Docker containers and emulated network components, the framework provides quantitative metrics about the overall system performance, such as End-to-End latency as well as network and physical resources consumed. We use BORDER to compare five of the most popular MQTT brokers that allow the creation of distributed architectures and we release it as an open-source project to allow for reproducible researches.**

***Index Terms*—Benchmarking, distributed message queuing telemetry transport (MQTT) brokers, MQTT, MQTT bridging, MQTT clustering.**

I. Introduction

BOOSTED by applications needing a lightweight, reliable, and straightforward communication protocol, the message queuing telemetry transport (MQTT) has become the standard *de-facto* for Internet of Things (IoT) applications and Machine-to-Machine (M2M) communications [1]. As a matter of fact, all major cloud platforms (e.g., Amazon AWS,[1] Microsoft Azure,[2] and IBM[3]) expose their IoT services through MQTT, and many enterprises provide MQTT-based data collection and communication solutions.

Developed by IBM in 1999, MQTT aims to transport messages between devices requiring a small code footprint and with limited network bandwidth. According to the specifications [2], an MQTT client is any device, from a microcontroller up to a server, that connects to an MQTT broker for exchanging messages. The communication follows a topic-based publish/subscribe pattern with a broker acting as messages dispatcher. The broker is a central entity in charge of handling clients' connections, clients' subscriptions, and data publishing on specific topics. In this way, the broker allows to decouple data generation and data consumption both in space and time, removing direct communication between the publisher of the data and the receiver (subscriber). This aspect, combined with the protocol simplicity at the client side and the support for reliability and Quality of Service (QoS), makes MQTT an ideal candidate for resource-constrained application.

However, the centralized fashion of MQTT also brings drawbacks. First, the broker is typically the single point of failure of the system: no MQTT communication is possible if the broker is unavailable. Second, it does not scale well considering the massive numbers of IoT devices forecasted in the next future. Third, a network architecture with a single central broker is partially at odds with the recent interest for edge architectures such as the one envisioned by 5G cellular technologies [3], in which cloud services (including any broker instance) are moved to the edge, closer to the user devices to support low-latency applications.

A possible solution foresees the cooperation of multiple distributed MQTT brokers, acting as a single entity. In particular, distributed brokers are deployed on different machines or virtual environments (i.e., Docker [4]) connected together over a network. The result is a single logical broker that ensures high scalability, replication, elasticity, and resiliency to failures [5]. Specifically thought for being located in cloud-based enterprises datacenters, many commercial brokers (i.e., EMQX [6], HiveMQ [7], and others) already provide message distribution and clustering capabilities. Clusters of brokers ensure publishes and subscriptions forwarding between the nodes as well as advanced features, such as broker discovery or failure recovery. As a consequence, the lightweight principle of MQTT goes lost, often making communication overhead between the brokers nonnegligible or increasing latency. This is unfavorable in an IoT scenario where deployments are often in

Manuscript received 29 September 2021; revised 18 January 2022; accepted 26 February 2022. Date of publication 2 March 2022; date of current version 7 September 2022. This work was supported in part by the Project BASE5G under Project 1155850 funded by Regione Lombardia within the framework POR FESR 2014–2020. *(Corresponding author: Edoardo Longo.)*

Edoardo Longo, Alessandro Enrico Cesare Redondi, and Matteo Cesana are with the Department of Electronics, Information and Bioengineering, Politecnico di Milano, 20133 Milan, Italy (e-mail: edoardo.longo@polimi.it; alessandroenrico.redondi@polimi.it; matteo.cesana@polimi.it).

Pietro Manzoni is with the Department of Computer Engineering, Universitat Politècnica de València, 46022 Valencia, Spain (e-mail: pmanzoni@disca.upv.es).

Digital Object Identifier 10.1109/JIOT.2022.3155872

[1]https://docs.aws.amazon.com/iot/latest/developerguide/mqtt.html

[2]https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-mqtt-support

[3]https://www.ibm.com/docs/en/mapms/1_cloud?topic=devices-communicating-mqtt

constrained or frugal environments with brokers located at the edge of the network.

Other brokers, for instance, the popular open-source Mosquitto [8] implementation, use a mechanism called *bridging* to exchange publications among distributed brokers. Bridging allows to interconnect brokers and distribute publications on specific topics among them, directly exploiting MQTT primitives. On the one hand, bridging makes the system simpler and entirely MQTT-based; however, it generally relies on a static configuration that may not scale in complex environments or cause message loops if not configured correctly.

Such fragmented approaches may lead to noticeably different system performance based on the environment characteristics, i.e., underlying topology, network latency, clients distribution, etc. Even if the scalability of MQTT brokers connected in a cluster has already been demonstrated [9], [10], it is not clear how the network and hardware characteristics could affect the overall performance.

Therefore, in this work, we present a benchmarking framework for distributed MQTT brokers (BORDER), a framework capable of creating and evaluating clusters of MQTT brokers on an emulated realistic network that can be tuned according to the user's needs. Our contribution is twofold: first, we explain in detail the benchmarking framework, which is based on the *Containernet* network emulator [11], providing isolated brokers in different Docker containers; second, we use BORDER to compare four popular MQTT brokers that support native clustering and two bridge-based brokers ensembles, including MQTT-ST, a distribution protocol based on the Mosquitto broker [12]. We evaluate the results using quantitative metrics about the system latency as well as network and physical resources consumed in the cluster. We release BORDER as an open-source framework at *https://github.com/ANTLab-polimi/BORDER*.

The remainder of this article is organized as follows. Section II describes the related works; Section III provides an overview of MQTT distributed scenarios as well as a detailed description of the broker implementations tested in this work. Section V describes the BORDER framework architecture as well as the evaluation metrics; in Section VI, the performed experiments are described, and performance among the different brokers are compared. Finally, Section VII contains concluding remarks and future research directions.

II. Related Work

Comparisons studies of the performance of MQTT brokers have been actively performed by researchers and companies. Most works focus on the benchmarking methodologies and KPIs to determine the operational performance of MQTT brokers in an IoT scenario [13], [14].

Several authors have performed comparative analyzes between protocols for the IoT, discussing the criteria for selecting protocols, such as MQTT, CoAP, AMQP, and HTTP [15]–[19]. The main focus of such works is generally to study the performance of the protocols in terms of End-to-End (E2E) delay, bandwidth consumption, and supported devices. For instance, Sommer *et al.* [20] leveraged a peer-to-peer communication model to select which message-oriented middleware (MOM) to use for industrial applications. They find out that AMQP, KAFKA, and ZeroMQ can achieve a throughput of more than 1000 messages per second (mps) while MQTT (with the Eclipse Mosquitto implementation) cannot. In particular, compared to the other candidates, Mosquitto has a lower overhead but higher latency.

Similarly, MQTT brokers have been also analyzed in diverse applications scenarios. De Oliveira *et al.* [21] compared the performance of Mosquitto and RabbitMQ in a Smart City environment. Utilizing cloud-located brokers and Raspberry Pis as clients, the authors analyzed the two brokers with different configurations (payload size, number of packets, and bandwidth). Their results show that the server hardware features and the bandwidth consumed significantly impact the message latency, suggesting RabbitMQ for robust and low-latency applications, while Mosquitto for low data-flow ones.

The work of Mishra [22] puts Mosquitto, BevyWiseMQTT, and HiveMQ in a small-scale, single broker cloud scenario comparing their performance by subscription throughput using `mqtt-stresser`[4] and `mqtt-bench`,[5] two publicly available tools for MQTT benchmarking. They find no significant difference in performance when MQTT brokers are applied to small deployment scenarios.

In addition to pure performance, works focus also on qualitative metrics, such as portability, usability, and maintainability, classifying nine different open-source brokers by the ISO/IEC 25010 normative using a single machine testbed [23]. In a white paper, Scaleagent.com [24] benchmarks single node MQTT brokers, including Mosquitto, RabbitMQ, ActiveMQ, and JoramMQ to evaluate the client scalability in a *multi-publisher* scenario, including up to 1000 publishers and one subscriber. Similar to our work, they evaluated the CPU usage of the server, the E2E latency (called message transmission latency) and the delivered throughput. With 100 000 publishers producing 100 000 mps, they discover that JoramMQ is the only one capable of handling such a massive amount of traffic.

Similarly, HiveMQ developers demonstrate that a cluster of HiveMQ brokers can handle ten million concurrent connections and up to 1.74 million/s messages in a cluster of MQTT brokers [9]. The scenario reflects a Fan-Out test case with ten million clients subscribing and 40 publishing clients connected to 40 HiveMQ nodes deployed on as many Amazon Web Services (AWS) c4.2xlarge instances. In addition, the authors provide insights on average consumed RAM, CPU, and bandwidth for HiveMQ.

In order to solve throughput and latency problems, researchers have made multiple brokers cooperating between themselves, creating clusters of brokers [5], [12], [25]–[31]. Table I summarizes the main research contributions and products that provide such kind of distributed architectures. Distributed MQTT brokers can find several applications, from high availability and scalability clusters for massive data

[4]See https://github.com/inovex/mqtt-stresser. Last accessed: January 10, 2022.

[5]See https://github.com/takanorig/mqtt-bench. Last accessed: January 10, 2022.

TABLE I
DISTRIBUTED FEATURES OF THE MOST POPULAR MQTT BROKERS

Broker	Cluster	Bridge	Purpose	Language	Ref.
ActiveMQ	✓	✓	Product	Java	[36]
D-MQTT		✓	Research	C	[27]
DM-MQTT	Rendezvous system		Research	-	[28]
Emitter	✓		Product	Go	[37]
Emma		Dynamic	Research	Java	[5]
EMQX	✓	✓	Product	Erlang	[6]
FogMQ	Bridge-based		Research	-	[29]
HiveMQ	✓	✓	Product	Java	[7]
HbMQTT			Project	Python	[38]
ILDM	External Tool		Research	-	[35]
Aedes	✓	✓	Project	Javascript	[39]
Mosquitto		✓	Product	C	[8]
Moquette-io			Project	Java	[40]
MQTT-NEG	External Tool		Research	Python	[31]
MQTT-ST		✓	Research	C	[12]
RabbitMQ	✓	✓	Product	Erlang	[41]
Schmit et Al.		Dynamic	Research	C	[30]
VerneMQ	✓	✓	Product	Erlang	[42]

handling to low-latency applications developed at the edge of the network. Even if efforts to create heterogeneous clusters have been made [32], brokers are only supported to function for homogeneous clusters, i.e., only MQTT brokers from a single vendor cooperating with each other. Typically, each broker runs on a dedicated Virtual Machine located either on an on-premise data center or cloud infrastructures, such as AWS, Microsoft Azure, or Google Cloud. Then, they are connected to each other through the network for cluster communication.

Benchmarking architectures with distributed brokers can be found in several works [10], [32]–[34]. Koziolek *et al.* [10] created a cluster of two MQTT brokers in redundant edge gateway servers running the open-source edge virtualization platform StarlingX and orchestrated by Kubernetes (K8s). Tests with MZBench[6] showed that, among HiveMQ, VerneMQ, and EMQX, the latter shows the highest throughput when the cluster is deployed in an eight CPU cores limited scenario. Koziolek included additional metrics for the brokers' evaluation, such as availability, resilience, security, and extensibility. Finally, Banno *et al.* [35] proposed a mechanism called ILDM that enables heterogeneous MQTT brokers to cooperate with each other. In a testbed with five Mosquitto brokers, the system can improve the throughput up to four times than a single Mosquitto instance.

Rather than focusing on MQTT absolute performance, as done in the state of the art, this work offers BORDER, a benchmarking framework dedicated explicitly to applications for constrained/edge architectures for the IoT. To this end, in our work, we provide customization for MQTT brokers with interchangeable and isolated Docker containers and a fully personalized configuration of the network topology.

III. DISTRIBUTED MQTT BROKERS

The MQTT brokers' ecosystem provides a wide range of choices, available in many different implementations and programming models. According to MQTT.org,[7] there are about

[6]See https://github.com/satori-com/mzbench. Last accessed: January 10, 2022.

[7]See https://github.com/mqtt/mqtt.org/wiki/brokers. Last accessed: September 10, 2021.

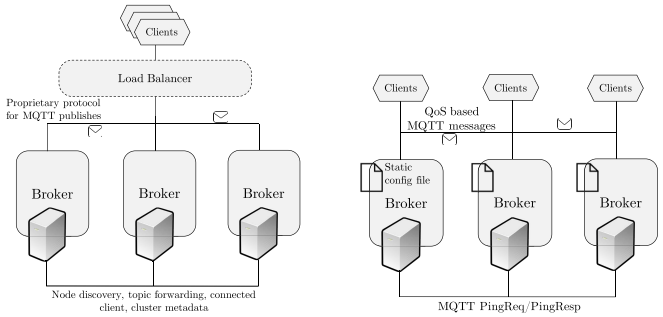


Fig. 1. Distributed MQTT brokers approaches.

30 broker implementations available to date, indicating the incredible success MQTT is experiencing in the last few years. Most of these brokers support natively both MQTT v3.1.1 and MQTT v5 (introduced by OASIS in 2019). In addition, other popular message platforms not originally built for MQTT (e.g., RabbitMQ and Apache Kafka), support the protocol through the addition of plug-ins to integrate the protocol functionalities.

Despite evolution and improvements, MQTT still remains a centralized protocol, with the broker acting as a single message dispatcher. This architecture may not fit well future IoT applications which require scalability, location awareness, reliability, and low latency. To circumvent such a single point of failure in the systems, a distributed ensemble of MQTT brokers may be created. Currently, the MQTT standard does not define any specific procedure to create such a distributed architecture; however, in the research literature as well as in products available commercially, two main design techniques are present (Fig. 1): 1) clustering and 2) bridging.

A. Clustering

A cluster of MQTT brokers is a set of brokers working together to create a single logical MQTT broker. Each broker in the cluster is generally run on a different virtual machine, usually on the cloud, and connected through the network. Load balancers are often used together with MQTT clusters to have a single entry point for all the clients. This is useful when brokers are located in data servers handling many clients with elevated message rates. As regards message distribution inside the cluster, broker vendors often use proprietary protocols and encrypted communication, making the message exchange invisible to the user. As shown in Fig. 1, MQTT brokers clustering may also expose extra features, such as broker discovery, failure detection, message replication, or other cluster status metadata. For these reasons, the clustering approach is likely to consume high volume of bandwidth and internal resources of the machine hosting the broker.

B. Bridging

In addition to clustering, MQTT provides a native (but nonstandardized) mechanism to exchange messages among brokers. This mechanism is called *bridging*, or bridge extension, and allows a broker to forward publish messages directly

to other brokers following the MQTT communication pattern. Using configuration files, it is possible to specify the IP addresses and ports of the brokers that are going to be connected to the instance that is started with that specific configuration. Once two or more brokers are connected, they can communicate with each other, forwarding publishes on specific topics. For example, a broker A, willing to receive publishes from a broker B, can subscribe to a topic (wildcard included) as a standard client. It is important to note that bridging only covers message routing between brokers, unlike clustering. Indeed, clients cannot resume their persistent sessions on other broker instances and cannot receive queued messages or unfinished QoS 1 and 2 message flows. Moreover, beyond its static nature, this approach lacks in terms of efficiency since bridging forwards the messages on each specified connection, without the possibility of knowing a priori the topics circulating among the nodes of the system. This can also cause message loops among the brokers connected through the bridging.

IV. Evaluation Tools

For our evaluation, we select five, among the most popular, MQTT brokers to create a multibroker environment using clustering or bridging. The candidates are implemented in different programming languages, fully supporting MQTT v3.1.1 and MQTT v5. All candidates brokers provide a free version, in some cases open source.

A. EMQX

EMQX [6] (Erlang/Enterprise/Elastic MQTT Broker) was launched in 2013 as an open-source project. It provides different products: an open-source broker, an enterprise-ready broker, and an IoT Cloud Hub. Moreover, it offers customized extendible plug-ins and compatibility with other IoT protocols, such as MQTT-SN, CoAP, and LwM2M. EMQX is based on the Erlang/Open Telecom Platform (OTP) platform that guarantees high availability, reliability, and scalability of the implementation. EMQX supports clustering exploiting the Erlang Distribution Protocol (ErlDP) for the intrabroker message exchange. Nodes communicate over TCP/IP on port 4370 (or adjacents), periodically checking the cluster health with specific pings. Inside the cluster, brokers forward each others clients connection/disconnection, subscription/unsubscriptions; as regards message publication, the broker forwards publishes only to the interested nodes in the cluster. EMQX implements both MQTT V3.1/V3.1.1 and V5.0 protocol specifications, and the open-source implementations are licensed under the Apache Version 2.0. EMQX products also include a lightweight variant called *EMQ X Edge* for resource-constrained edge devices, however, the edge variant does not support clustering features.

B. VerneMQ

Developed at Octavo Labs AG in Zurich (Switzerland) in 2015, VerneMQ [42] is the successor company of Erlio GmbH. It is an open-source project (Apache License 2.0) with no commercial versions, but the company offers commercial support for enterprises; for example, among the customers there are Microsoft and Volkswagen. VerneMQ supports clustering between MQTT brokers utilizing a master-less clustering approach. In particular, there are no special nodes like masters or slaves to consider when the topology changes or nodes are removed or added. VerneMQ offers two ready Docker images based on Debian and Alpine, supporting MQTT versions up to V5.0. Similar to EMQX, it is developed in Erlang (Erlang/OTP 21.2) and uses the ErlDP for the intrabroker message distribution over TCP/IP on the default port 4369. Also, VerneMQ nodes share with the others the clients' connection/disconnection, subscription/unsubscription, and other session metadata.

C. RabbitMQ

RabbitMQ [41] is a multiprotocol messaging broker developed from Rabbit Technologies Ltd. since 2007. It initially implemented only the AMQP protocol but, now, it provides plug-ins to support STOMP and MQTT v3.1.1. It is an open-source project and the code is released under the Mozilla Public License. In addition to the broker module, RabbitMQ includes several libraries for creating clients in different programming languages (i.e., Python, Java, PHP, C#, Go, etc.). It also provides Enterprise and Cloud Ready versions for private and public clouds. For example, it is used by the messaging platform of T-Mobile, Adidas Runtastic, and other several commercial customers. Similar to VerneMQ and EMQX, also the RabbitMQ server program is developed with the Erlang programming language and is built on the OTP framework for clustering and failover. RabbitMQ brokers are equal peers with no special nodes, where data and clients states are replicated across all the nodes in the cluster.

D. HiveMQ

HiveMQ [7] is a Java-based MQTT broker that supports MQTT 3.x and MQTT 5. The company, formerly named *dc-square*, started in Germany in 2012 with a commercial MQTT broker. It evolves to HiveMQ in 2018 releasing the Community Edition, an open-source variant licensed under Apache 2. Presently, HiveMQ is available in three different editions: 1) HiveMQ Enterprise; 2) HiveMQ Professional; and 3) HiveMQ Community. With regards to commercial editions, they are available for personal evaluation with a limited number of client connections. In addition, HiveMQ provides an IoT cloud platform variant with hourly subscription fees and a Java MQTT client. The company declares to have over 130 customers including BMW and Deutsche Telekom. HiveMQ also provides a rich blog about MQTT use cases, best practices, and tutorials. Although the company claims that HiveMQ can also run on embedded devices, it has some minimum hardware requirements to work adequately: at least 4 GB of RAM, 4 or more CPUs, 100 GB, or more free disk space. A key feature of the HiveMQ broker is the possibility to use and create custom extensions using a specific SDK. With the Cluster Discovery extension is possible to create an MQTT broker cluster that discovers automatically the components in the network. Also, the HiveMQ DNS discovery plug-in uses DNS discovery to add or remove brokers instances to the cluster at runtime. As

regards cluster messages transport, HiveMQ communicates by default using TCP on port 8000 but also offers the selection between UDP and secure TCP. In addition to the clustering, HiveMQ supports bridging with a specific enterprise extension.

E. Mosquitto and MQTT-ST

Mosquitto [8] is probably the most popular open-source MQTT broker for the IoT. Eclipse Foundation has recently released Mosquitto version 2.0, which implements the MQTT protocol versions 5.0, 3.1.1, and 3.1. Mosquitto is extremely lightweight and it is suitable both for low-power single board computers and full servers. The whole broker is written in C language and the Mosquitto project also provides a C library for implementing MQTT clients. Moreover, it offers command line MQTT clients, *mosquittopub* and *mosquittosub*. Unlike the brokers above, Mosquitto does not support broker clustering; nevertheless, bridging can be used to interconnect multiple Mosquitto broker instances. In order to handle dynamically different topology, we will use MQTT-ST [12] to create a tree of brokers in a distributed fashion. In particular, MQTT-ST is a distribution protocol able to create such a distributed architecture of brokers organized through a spanning tree. The protocol uses in-band signaling (i.e., reuses MQTT primitives for the control messages) and allows for full message replication among brokers, as well as robustness against failures. The tree of brokers is created automatically by the brokers, taking into account parameters, such as interbroker latency and computational/memory resources for the optimal path computation. MQTT-ST reuses and integrates the original Mosquitto source code, directly adding the additional features for the message distribution in the broker's logic. In particular, to achieve such a distribution, MQTT-ST replaces standard CONNECT and PINGREQ/PINGRESP messages with custom ones. In a nutshell, a broker willing to create a bridge with another one transmits a crafted MQTT CONNECT message containing all the node information (e.g., IP address and port); after the connection, to create and maintain the broker tree MQTT-ST reuses PING messages as STP BPDUs. Crafted PING messages are composed of IP address of the root broker, the broker capability value, and a root path cost. Once the tree is created, MQTT-ST message distribution relies only on MQTT publish messages. In this way, the ensemble creation, management, and message dissemination are entirely transparent from the clients' and users' points of view. Moreover, the tree is robust to broker failures, and it can reconfigure automatically upon malfunctions mimicking the cluster behavior.

V. BORDER Framework Scenario

The proposed framework, called BORDER, provides an easy way for creating distributed environments of MQTT brokers for in deep performance evaluation and experimenting with topologies. In order to allow extensibility as well as isolation between the framework components, BORDER orchestrates Docker Containers in a virtual network with real end-code as well as real Linux kernel and network stack. In a nutshell, a Docker container image is a lightweight, standalone, and executable package of software that includes code, system tools, system libraries, and settings. BORDER nodes are MQTT brokers and clients that become containers running in the Docker Engine, bundling the image core process with configuration files and environmental variables. In such a way, Containers isolate the software from its environment and ensure that BORDER works uniformly despite hardware differences between development and staging. Furthermore, BORDER Docker images are hosted on a dedicated DockerHub cloud repository for easy management and sharing of the components.

A. Containernet

The backbone of BORDER is Containernet [11], a fork of the Mininet[8] network emulator. Containernet inherits all the Mininet characteristics, i.e., creating a network of virtual hosts, switches, controllers, and links for custom topologies testing. Furthermore, it uses Docker containers as hosts in a Mininet emulated network. In such a way, Containernet can replicate real-world environments allowing to move testbeds to real systems with minimal code changes. Using Containernet, BORDER creates flexible MQTT clusters topologies, with the possibility of adding, removing, or replacing containers. Moreover, Containernet allows replacing the docker image of a specific node or pass to the container different custom configuration files. This allows BORDER to support interconnections among MQTT brokers with different topologies configurations. As regards hardware emulation, exploiting the Docker API, the proposed framework provides resource limitation to the machine, CPU control, as well as fine tuning of the RAM and physical memory of each node.

B. BORDER Architecture

The BORDER framework has been developed in Python 3.8 language exploiting the Containernet/Mininet API and the Python Docker SDK for the creations and the management of the framework components. It also uses the Containernet CLI for runtime management and control of the environment.

The framework aims at reflecting as closely as possible the network topology and the physical resources of a distributed MQTT brokers deployment. For this reason, each MQTT broker resides in a dedicated network, isolated from the others. As shown in Fig. 2, the architecture is composed of three main items.

1) *Router:* The router is in charge of forwarding data packets between different broker networks. BORDER separates the broker instances in different networks; thus, packets forwarding occurs thanks to routers. Each broker has a dedicated router that is in charge of the communication with the external networks. Technically, a BORDER router is a Linux node instance with IP forwarding toward the other networks enabled. Routers use IPv4 notation; at boot time the framework assigns the IP addresses following the pattern 10.*R*.0.0, with *R* the sequential ID of the router.
2) *Switch:* Real networks use switches for data packets forwarding directly between devices. As one can see from

[8]See http://mininet.org. Last accessed: October 21, 2021.

TABLE II
DOCKER IMAGES OF THE MQTT BROKERS CANDIDATES

	EMQX	**VerneMQ**	**RabbitMQ**	**HiveMQ**	**Mosquito**
Parent Image	erlang:21.3.6-alpine	alpine:3.9	alpine:3.12	openjdk:11-jre-slim	alpine:3.12
Image compressed size	56.67 MB	99.04MB	61.03 MB	222.67 MB	12.21MB
OS/arch	linux/amd64	linux/amd64	linux/amd64	linux/amd64	linux/amd64

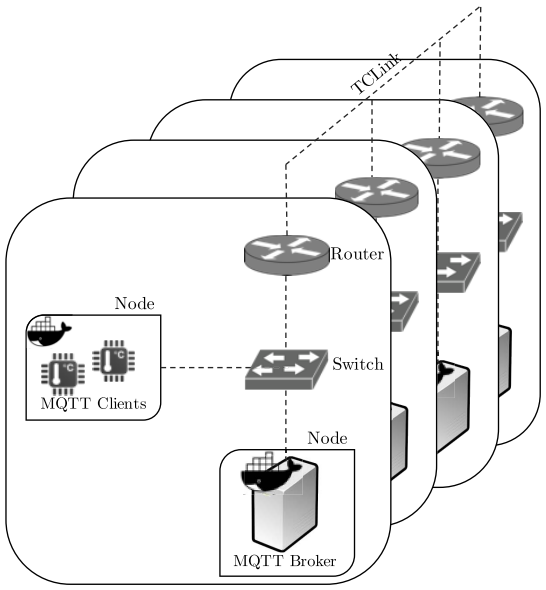


Fig. 2. Details of the proposed BORDER framework architecture.

Fig. 2, in BORDER, the switch acts as a network dispatcher between upper layers (i.e., routers) and lower layers (i.e., MQTT broker and clients). Moreover, the switch allows to handle multiple brokers in the same local network for advanced architectures (e.g., horizontal clustering in the same network).

3) *Node:* Nodes represent the server where the broker is deployed. Here, Docker images run to become containers with all the MQTT broker functionalities. BORDER passes to the node the broker configuration file (according to the instance deployed), the dedicated hostname, the IP address, port bindings, and environmental variables. Also MQTT clients are deployed on dedicated nodes, separated from the MQTT broker. In this case, the instance runs a specific docker image containing the clients' functionalities.

Finally, the framework items are connected through a timing compensated high-speed optical link (TCLink), with adjustable link delay and link bandwidth parameters. In this configuration, the brokers are isolated between each other and connected only to their associated router. On top, routers are connected between themselves with a TCLink for routing between the lower layers.

C. BORDER Broker Images

We built BORDER docker images on top of the pre-existing candidates' ones, i.e., EMQX, VerneMQ, RabbitMQ, HiveMQ, and Mosquitto. Except for HiveMQ, which use a custom base image, all Dockers parent images are based on Alpine Linux version 3, a lightweight and resource-efficient Linux distribution. In order to keep the docker images as close as the original ones, we only installed the networking packages needed for fulfilling the Containernet requirements[9] (i.e., `iproute2`, `iperf`, `tcpdump`, etc.). In Table II, we summarize the main characteristic of the Docker images used. As one can see, apart from HiveMQ, the other candidates use the same base OS, thus presenting minimal differences.

TABLE III
MOST RELEVANT BORDER PARAMETERS

Parameter	Value
Number of brokers	5
RTT delay router-router link	5, 50 ms
RTT delay switch-router link	0 ms
RTT delay broker-switch link	0 ms
Broker RAM upper limit	2 GB
Number of CPU per broker	2
Number of published messages	50
Number of subscribers	100
Number of publishers	1
MQTT broker port	1883
Message publishing rate	1 msg/s
Topics name	*dst/mqtt/*
Message size	18 bytes
Number of simulations	10
Quality of Service (QoS)	0,1,2

D. Framework Configuration

We designed the BORDER framework to allow for different MQTT brokers hardware configurations as well as network topologies. The Python program which orchestrates BORDER generates multiple MQTT broker instances according to the parameter passed via the CLI (e.g., number of brokers and broker type). As regards network topologies, the program automatically generates a fully meshed network of brokers, where links have adjustable round-trip time (RTT) and bandwidth. Table III resumes the exact parameters accepted by the BORDER framework. For the simulation, the user can also define, among the others, number of publishers and subscribers connected to the brokers, publishing message rate, and message number.

Taking as example the parameters used in the experiment scenario (Section VI), the framework will:

1) start the Linux Network Controller on the default port (6654);
2) define the IP address for the routers in the range $[10.100.0.0, 10.(100\langle+brokers\rangle).0.0]$, and create the corresponding routers;
3) create switches, defining the network interface name and IP addresses;

[9]See https://github.com/containernet/containernet/wiki/Container-Requirements-and-Compatibility. Last accessed: January 10, 2022.

4) start the MQTT brokers containers in the IP range [10.0.0.0, 10.⟨*router*⟩.0.⟨*brokers*⟩]. RAM and CPU are defined by the CLI parameters;
5) create the router–router, router–switch, and switch–broker links with users' input parameters;
6) run the Docker entrypoint of the MQTT brokers image and enable the node network interfaces;
7) after a fixed time interval, check brokers networking with pings ensuring the cluster health;
8) for each switch network, start the MQTT clients (100 subscribers and 1 publisher) and connect them to the respective brokers;
9) finally, start the messaging process logging metrics and producing results as explained in Section VI.

E. Metrics

The metrics used for the evaluation of the distributed MQTT brokers in an IoT scenario are as follows.

1) End-to-End Delay: E2E delay refers to the time in seconds that an MQTT message takes to be transmitted from a publisher p to a subscriber s. In a distributed scenario, this includes: 1) the time for a message to be received by a broker; 2) the broker queueing and processing time; 3) the intrabroker communication (if any); and 4) the broker–subscriber time. The delay is calculated taking the difference between the publishing timestamp and receiving timestamp at the subscriber side. Low E2E delay is fundamental in many IoT applications, especially when the latency is a concern, like edge applications where real-time actions are required (e.g., connected cars).

2) Physical Resources: Processing huge dataflows and handling the cluster of MQTT brokers are expected to be resource intensive. This can affect the normal operation of the system to the point of creating a performance bottleneck if server resources are not dimensioned correctly. For this purpose, we evaluate the average RAM and CPU consumption of the cluster of MQTT brokers.

3) Overhead Traffic: In addition to the MQTT traffic, the cluster of brokers generates a high volume of additional traffic for internal communication between the nodes of the group. The overhead traffic is composed of multiple factors: e.g., forwarding MQTT messages in the cluster, notifying new clients connection, forwards client subscriptions, or simply checking the cluster health. An excessive volume of traffic among the brokers may cause network saturation, increasing packets queue and system overloading.

VI. Experimental Results and Discussion

A. Test Infrastructure

To evaluate the broker candidates, we set up the BORDER framework on a single machine equipped with an Intel Xeon CPU E5-1660 with 16 CPU @ 3.00 GHz and 16 GB of RAM running Ubuntu 18.04. For what concerns the testbed environment, each framework component is isolated in a dedicated Docker container. For each broker, we allow to use two cores and we allocated 2 GB of RAM and 1 GB of swap memory. As for clients (publisher and subscribers), they can use all the remaining host machine resources. As explained in Section V-C, brokers are built from their base Docker image and then embedded in the framework. For what concerns clients, the publishers are built on top of the open-source *mqtt-benchmark* tool[10] developed in *golang* language, while the subscribers exploit the *paho-mqtt 1.5.1* library for Python 3.8.

For each scenario, tests have been run for all the MQTT QoS levels: (0, 1, 2) and repeated ten times to average the values and exclude outliers. The execution works as follows. First, the BORDER is started, then:

1) A specific number of homogeneous MQTT brokers is started. As explained in Section V-B, each one is connected to its correspondent switch/router, and eventually, the routers are connected to each other. The architecture *broker–router–switch* is connected through emulated link creating a full mesh topology, i.e., each broker has connections to all the others through routers. We vary the RTT between the routers in the range {5, 50} ms and we set the number of brokers in the cluster to 5. Subsequently, we wait a defined amount of rest time to allow the brokers to connect correctly.
2) Upon the timer expiration, a fixed number of MQTT clients (i.e., publishers and subscribers) are connected to their correspondent broker. Clients are organized according to a fan-out configuration, with a single publisher and 100 subscribers. The topic used by clients is `dst/mqtt/` corresponding to 9 bytes of overhead in the request header.
3) BORDER ensures the scenario correct behavior. Each component checks the connectivity with the others through a PING request/response exchange. If all the pings are correctly received the test-bed can start.
4) A specific number of messages are published on the topic mentioned above. The publisher sends a burst of 50 messages with a constant rate of 1 msg/s with a payload of 18 bytes containing the message creation timestamp. Finally, when the subscribers receive the publications, results are logged for further evaluation and clients are disconnected.

B. Clients' Localities

To evaluate different traffic scenarios, we consider three different client configurations according to publishers and subscribers location, as shown in Fig. 3.

1) *Locality 100%:* This is the maximum degree of locality, meaning that all publishers and all subscribers are connected to the same broker.
2) *Locality 0%:* Conversely, this case represents the scenario with the minimum degree of locality, where all publishers are connected to one broker and all subscribers are connected to a different broker.
3) *Locality 50%:* We also test an intermediate case, where publishers and subscribers are randomly distributed among the five brokers. In this case, we repeat the test ten times changing each time the distribution of

[10]See https://github.com/krylovsk/mqtt-benchmark. Last accessed: January 10, 2022.

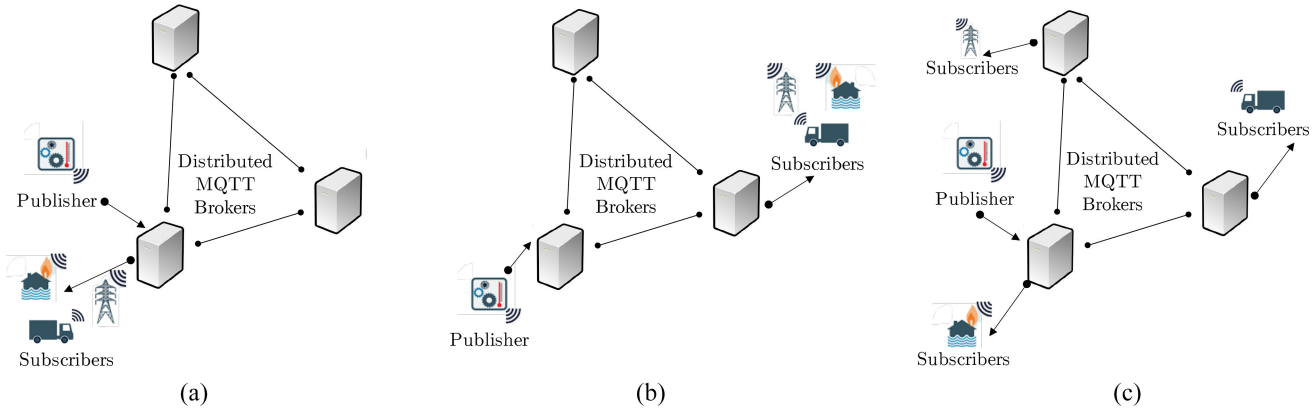


Fig. 3. Distributed MQTT brokers configurations. Clients positions represent different locality scenarios. (a) 100% locality. (b) 0% locality. (c) 50% locality.

publishers and subscribers and we show average results with their standard deviation.

C. Results

1) End-to-End Message Delay: First, we analyze the E2E message delay that, as explained in Section V-E, is the average time from the generation of the publish to the arrival of the message at the subscribers. The E2E delay is calculated at the subscriber side; when the publisher generates the message, it incapsulates into the payload the timestamp. Upon reception, the subscriber subtracts the current timestamp with the received one, obtaining the total message delay. (It is important to remember that since all Docker containers share the same host machine clock, brokers and clients are synchronized in time.) Fig. 4 summarizes the E2E delay results for each broker. The average values are shown with the 95%-confidence intervals as an indication for variations across experiments. As one can see, the two configurations, 5 and 50 ms RTT delay have a similar pattern emphasizing the differences when the delay between the brokers becomes higher. As regards locality 100%, all the different clusters show a very limited delay, especially when the QoS is 0. RabbitMQ and VerneMQ clusters get almost double E2E message delay in the cases when locality is 100% and QoS increases to 1 and 2; however, they generally outperform the rivals in all the other scenarios. MQTT-ST, which is the only broker using the bridging approach, has different results. On the one side, when locality is high (100%), it can deliver messages extremely efficiently, serving first the client subscribers and then the other brokers connected. On the other side, when subscribers are far away, the multistep fashion of the bridging fails, obtaining the worst average E2E delay among the candidates. In particular, in cases when QoS > 0, the total delay is even higher, due to the multiples ACKs between the brokers. Indeed, it is important to remind that the bridging approach reuses the MQTT message standard, thus, a broker has to wait for the internal MQTT ACKs between the brokers (e.g., PUBACK for QoS 1, and PUBREL and PUBCOMP for QoS 2) before delivering the messages to the clients.

In our simulation scenario, HiveMQ brokers (cluster-based and bridging-based) do not satisfy the suggested system requirements explained in Section IV. This results in message queue overload and eventually in significantly high E2E message delay. Without penalizing HiveMQ, we decided to exclude it from the E2E delay comparison.

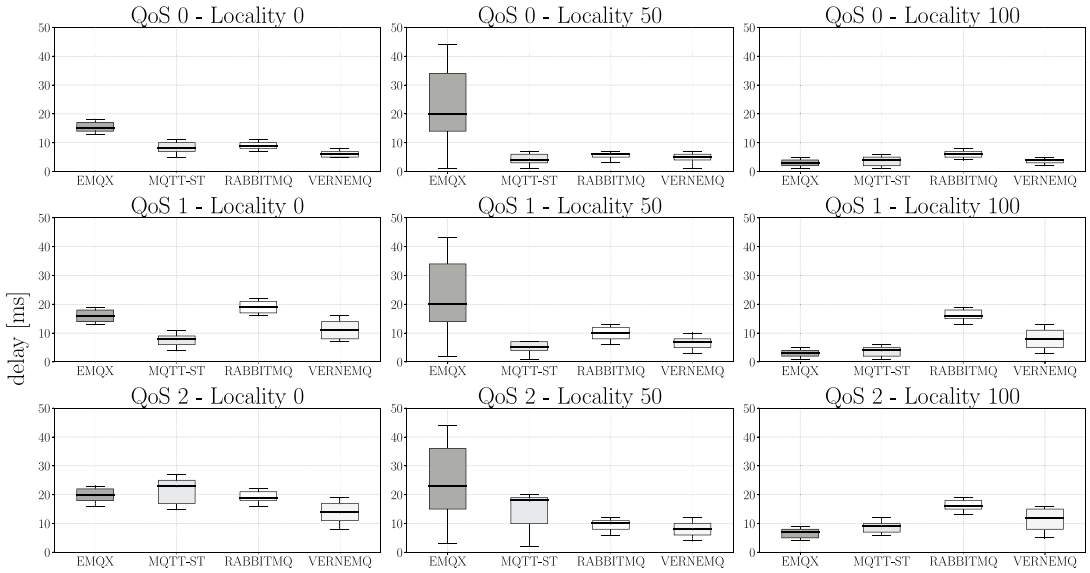
2) Traffic Overhead: In a distributed scenario, rather than publisher/subscriber traffic which in our experiments is constant, is important to take into account the traffic generated by the intrabroker communication. Our test shows that distributed MQTT brokers can become extremely talkative and this might cause network resource depletion. Brokers in an MQTT cluster have to communicate continuously to advertise their presence to the other counterparts, disseminate clients connections and subscriptions. As explained in Section III, intrabroker communication may be achieved in different ways: clustering approach with or without master nodes, or bridging brokers.

In order to deeply analyze the traffic in distributed brokers, we split the analysis into three different broker phases.

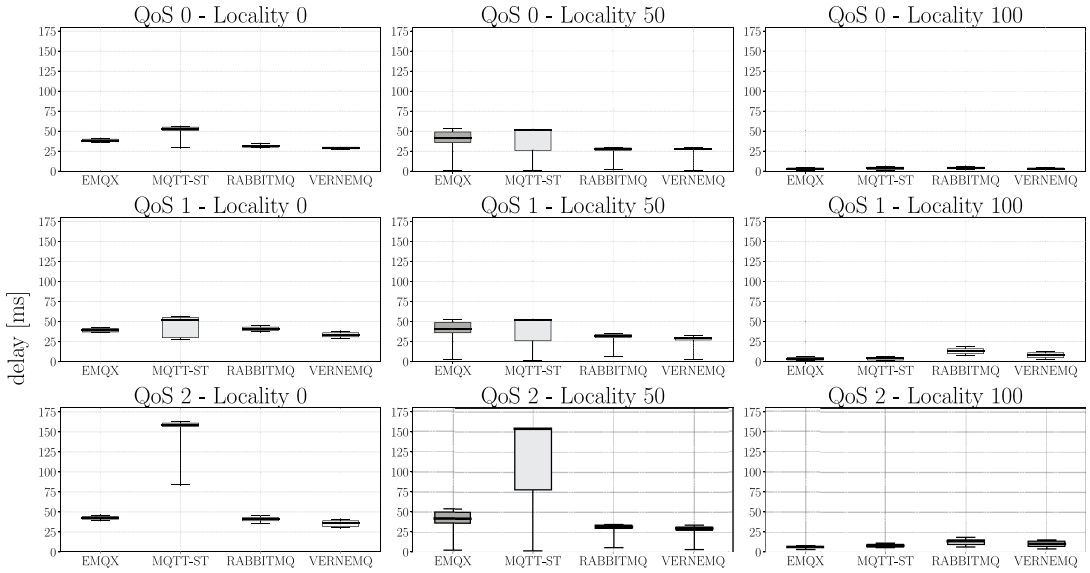
1) *Control-Communication:* It corresponds to the overhead traffic flowing after the ensemble formation process. In this phase, the brokers are still in an idle state, without any clients connected, nor any messages exchanged.
2) *Subscriptions-Forwarding:* It is the traffic originated by the subscribers connection and subscriptions; at this moment, cluster-based brokers disseminate to the other brokers clients connections and clients subscriptions containing among the others topic, will message, keep alive, and retain message.
3) *Publish-Exchange:* It is the traffic generated by the dissemination of the publish messages between the brokers.

Traffic data is captured through the `tcpdump` tool, listening on the routers inbound network interfaces, thus capturing only the intrabrokers traffic. Moreover, to make the comparison fair, the byte amount is normalized over the duration of the simulation. As one can see from Fig. 5, HiveMQ and RabbitMQ[11] result in the worst in terms of traffic overhead,

[11]RabbitMQ supports compression for internode traffic in the commercial VMware Tanzu RabbitMQ version. RabbitMQ claims that bandwidth usage might be reduced by 16 times.



(a)



(b)

Fig. 4. Average message E2E delay for EMQX, VerneMQ, RabbitMQ, and MQTT-ST. HiveMQ and HiveMQ bridging are excluded because, in our simulation, they did not satisfy the minimal hardware requirements. (a) Average message E2E delay when the RTT between the brokers is 5 ms. (b) Average message E2E delay when the RTT between the brokers is 50 ms.

consuming roughly more than ten times compared to EMQX and VerneMQ in most of the configurations. In clustered brokers, the control-communication consumes a nonnegligible amount of channel, around the 15% of the overall. Such communication of control messages is mainly used for maintaining the cluster active and inform nodes about each other's presence. Moreover, since each broker has to notify the others for clients connection and clients subscription, most of the traffic comes from the subscriptions-forwarding, while the publish-exchange contributes only for the 7% of the total, on average.

With regards to the bridging extension (MQTT-ST and HiveMQ briding-version), we can notice an extremely reduced traffic overhead compared to the native-cluster approach of the other brokers. The main reason behind this lies in the naive message exchange of the bridging. In this case, no clients connection and subscriptions are notified to the others and, for MQTT-ST, new brokers joining or leaving the ensemble are advertised with a custom PING message with a minimal packet overhead. Conversely, pure MQTT publish-exchange increase considerably the traffic overhead in cases where multiple ACKs are needed, e.g., QoS > 0.

3) Physical Resources: Other authors have already provided scalability tests with a high number of clients connected to a single broker or a cluster of MQTT brokers,

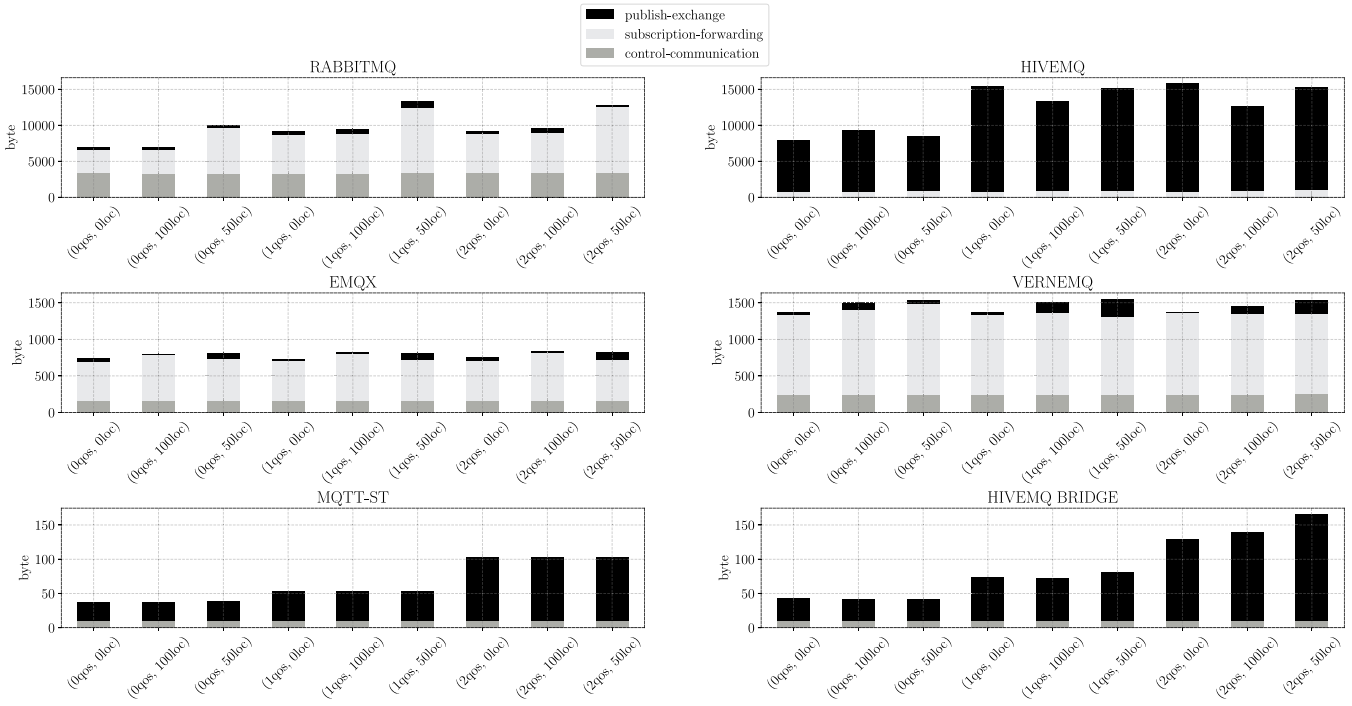


Fig. 5. Traffic overhead: average bandwidth used by the brokers ensemble communication.

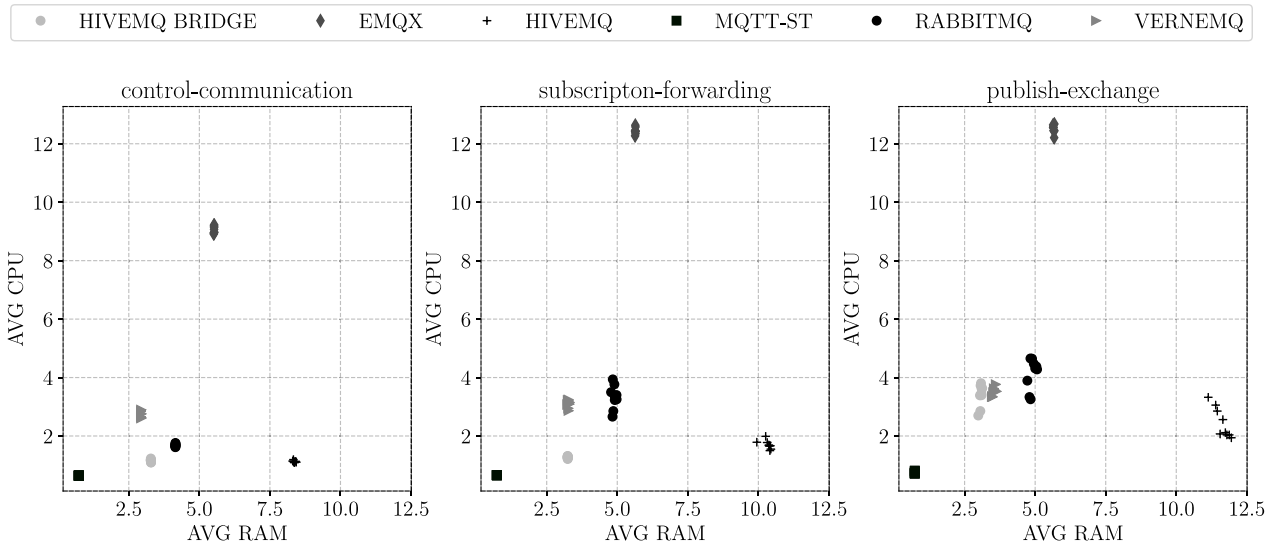


Fig. 6. Physical resources: average RAM and average CPU consumed by the ensemble of brokers.

demonstrating theoretical unlimited scalability [9], [10]. However, in scenarios where server resources are a concern (e.g., edge gateways), unlimited horizontal scalability over the cluster is not possible.

In our test, we focus on resources consumed by the broker on the host machine, assigning limited resources to our cluster (2-GB RAM, 2 CPU per node). Resources are logged using the built-in `docker stats`[12] tool obtaining the resource usage statistics of the running MQTT brokers in a live stream fashion. As for the traffic overhead, we split the analysis into the same three different phases: 1) control-communication; 2) subscriptions-forwarding; and 3) publish-exchange.

Fig. 6 shows the CPU utilization (*y*-axis) over the RAM utilization (*x*-axis), having the *best* candidate on the lower left side of the graph. As one can see, just handling the cluster is the most resource-consuming task, while the other two phases only add a slight increase in the resources utilization. As expected, MQTT-ST shows the best resource performance. First, it is built on top of Mosquitto, fully developed in C language, guaranteeing a lightweight code; second, it relies on the bridging for the distribution and only on MQTT primitives for the brokers' ensemble creation, discarding advanced features

[12]See https://docs.docker.com/engine/reference/commandline/stats/. Last accessed: January 12, 2022.

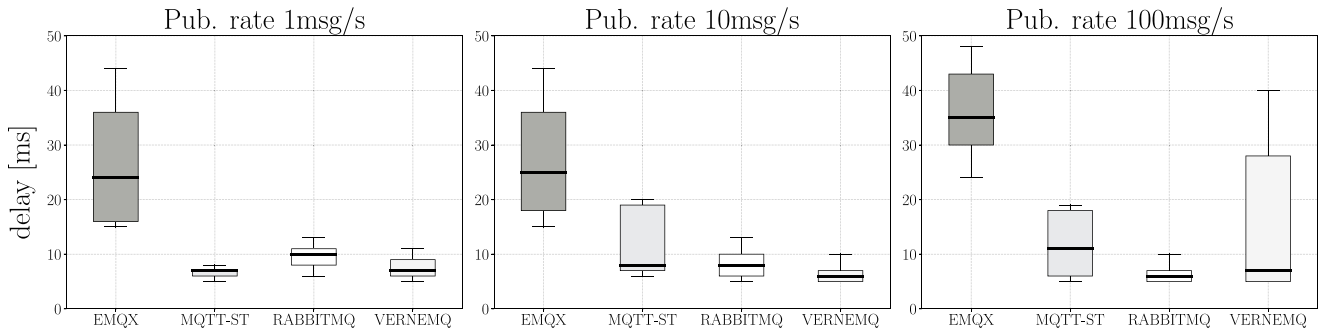


Fig. 7. Scalability performance for average message E2E delay with increasing publication rate, from 1 to 100 msg/s for the scenario of QoS 1 and 50% locality.

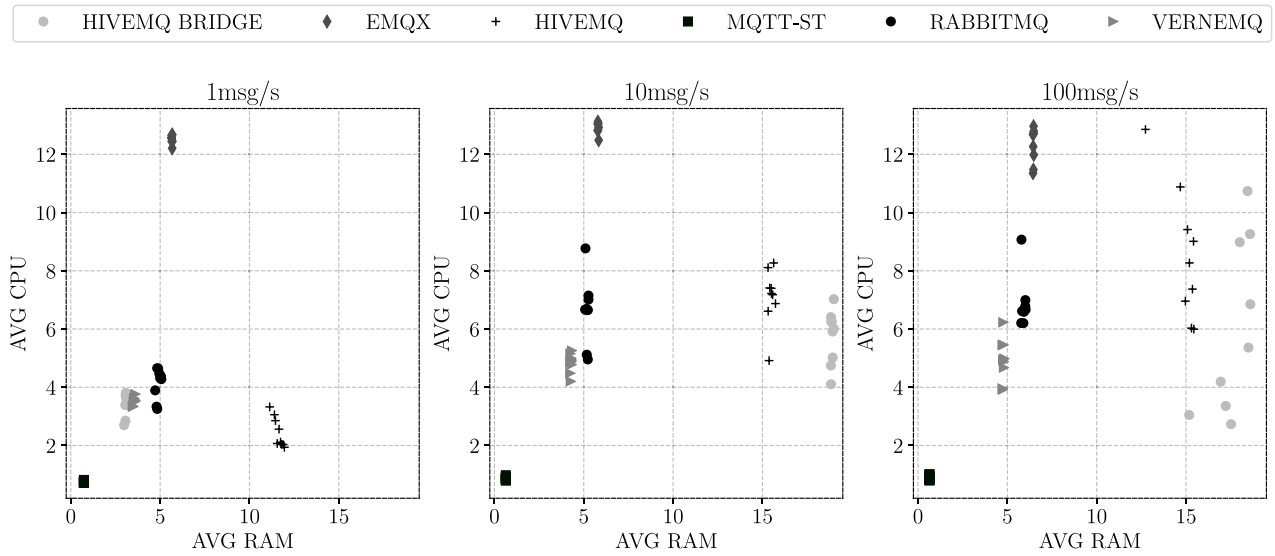


Fig. 8. Scalability performance for the machine resource usage, increasing the publication rate from 1 to 100 msg/s. The results refer to the publish-exchange phase.

implemented by the other competitors. Similar results are also visible for HiveMQ with the bridging extension. Erlang-based brokers (RabbitMQ, EMQX, and VerneMQ) behave similarly having a reasonable amount of resources consumed. The only exception is EMQX with a high use of CPU compared to the others two rivals. HiveMQ, which is the only one implemented in Java, is reasonably resources eager when the clustering comes into play, especially regarding RAM consumption. The reason behind this may reflect the Java Garbage Collector behavior.

D. Scalability Performance

In addition, BORDER also allows to easily analyze the scalability performance of different distributed MQTT broker implementations. For the task at hand, we increased the publication rate from 1 msg/s to 10 and 100 msg/s. Figs. 7 and 8 show the obtained E2E delay for the scenario of QoS 1 and 50% locality, and the machine resources usage. As one can see, even when the publication rate is increased by two orders of magnitude, the performance metrics monitored decrease minimally. Clearly, the obtained behavior is specific of the underlying resource configuration used by the broker (i.e., 2 cores and 2 GB of RAM in this case).

E. Discussion

From the analyzes of the experimental results, it is clear that cluster-based approaches and broker bridges have significant differences. Such dissimilarities are mainly caused by the distribution approach they provide: in-band MQTT for the bridging and out-band for proper brokers' clusters. MQTT brokers bridging outperforms the clustering approach in terms of traffic overhead and generally for machine resources consumption. Thus, the bridge approach is suitable for IoT-constrained and frugal environments, where network traffic and broker resource/energy consumption are a concern. However, the bridging fails in terms of robustness, for instance, in scenarios needing extreme low latency with high message QoS (e.g., QoS $= 2$), high availability of the node, and message replication between the nodes. Such use cases are ideal for an MQTT broker that provide clustering. In our scenarios, among the analyzed cluster-based brokers, we found minimal differences: EMQX and VerneMQ showed a minor network consumption, while RabbitMQ and VerneMQ had a slightly lower E2E delay than the other candidates.

The Mosquitto broker (with the MQTT-ST enhancement) shows adequate performances, especially in low resources scenarios. Moreover, since the Mosquitto project is released as an open-source license, it is suitable for innovative protocols integration, testing, and starting point for future works. Similarly, HiveMQ and EMQX provide bridge extension support and additional customizable plug-ins.

VII. Conclusion

This article proposed BORDER, a Python-based framework for easily creating and benchmarking topologies of distributed MQTT brokers. The framework exploits Docker containers for the broker deployments and the Containernet tool for the network emulation. We have tested five of the most popular MQTT brokers that allow distribution in various scenarios, in particular showing the differences between cluster-based and bridging-based MQTT distribution approaches. Results showed that the bridging approach might significantly reduce the traffic overhead and resource consumption compared to some popular MQTT clusters. However, MQTT bridging fails when robustness and E2E delay are a concern. Future work direction will target the analysis of more complex MQTT broker architectures (e.g., multilevel trees) and the introduction of new metrics, such as failure recovery time and reliability. The BORDER project is available for download at *https://github.com/ANTLab-polimi/BORDER*.

References

[1] D. Soni and A. Makwana, "A survey on MQTT: A protocol of Internet of Things (IoT)," in *Proc. Int. Conf. Telecommun. Power Anal. Comput. Tech. (ICTPACT)*, vol. 20, 2017, pp. 1–5.

[2] A. Banks, E. Briggs, K. Borgendale, and R. Gupta. "MQTT Version 5.0," Mar. 2019. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html. Latest version: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

[3] A. E. Redondi, A. Arcia-Moret, and P. Manzoni, "Towards a scaled IoT pub/sub architecture for 5G networks: The case of multiaccess edge computing," in *Proc. IEEE 5th World Forum Internet Things (WF-IoT)*, 2019, pp. 436–441.

[4] B. Mishra and A. Kertesz, "The use of MQTT in M2M and IoT systems: A survey," *IEEE Access*, vol. 8, pp. 201071–201086, 2020. [Online]. Available: https://doi.org/10.1109/access.2020.3035849

[5] T. Rausch, S. Nastic, and S. Dustdar, "EMMA: Distributed QoS-aware MQTT middleware for edge computing applications," in *Proc. IEEE Int. Conf. Cloud Eng. (IC2E)*, Apr. 2018, pp. 191–197. [Online]. Available: https://doi.org/10.1109/ic2e.2018.00043

[6] EMQ Technol. Co. Ltd., Zhengzhou, China, "EMQX, Version: 4.0.0," Jan. 19, 2020. [Online]. Available: https://www.emqx.io

[7] HiveMQ GmbH, Landshut, Germany, "HiveMQ Community, Version: 4," Jun. 18, 2020. [Online]. Available: https://www.hivemq.com

[8] Eclipse Foundation, Ottawa, ON, Canada, "Mosquitto, Version: 2.0.11," Jun. 8, 2021. [Online]. Available: http://mosquitto.org

[9] HiveMQ, Landshut, Germany, "10,000,000 MQTT Clients," Oct. 2017. [Online]. Available: https://www.hivemq.com/benchmark-10-million/

[10] H. Koziolek, S. Grüner, and J. Rückert, "A comparison of MQTT brokers for distributed IoT edge computing," in *Software Architecture*. Cham, Switzerland, Springer: 2020, pp. 352–368. [Online]. Available: https://doi.org/10.1007/978–3-030–58923-3_23

[11] M. Peuster, H. Karl, and S. van Rossem, "MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments," in *Proc. IEEE Conf. Netw. Funct. Virtualization Softw. Defined Netw. (NFV-SDN)*, Nov. 2016, pp. 148–153.

[12] E. Longo, A. E. Redondi, M. Cesana, A. Arcia-Moret, and P. Manzoni, "MQTT-ST: A spanning tree protocol for distributed MQTT brokers," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Jun. 2020, pp. 1–6. [Online]. Available: https://doi.org/10.1109/icc40277.2020.9149046

[13] I.-D. Gheorghe-Pop, A. Kaiser, A. Rennoch, and S. Hackel, "A performance benchmarking methodology for MQTT broker implementations," in *Proc. IEEE 20th Int. Conf. Softw. Qual. Reliabil. Secur. Companion (QRS-C)*, Dec. 2020, pp. 506–513. [Online]. Available: https://doi.org/10.1109/qrs-c51114.2020.00090

[14] B. K. Aichernig and R. Schumi, "How fast is MQTT? Statistical model checking and testing of IoT protocols," in *Proc. 15th Int. Conf. Quant. Eval. SysT. (QEST)*, 2018, pp. 36–52. [Online]. Available: http://www.qest.org/qest2018/

[15] Y. Sueda, M. Sato, and K. Hasuike, "Evaluation of message protocols for IoT," in *Proc. IEEE Int. Conf. Big Data Cloud Comput Data Sci. Eng. (BCD)*, May 2019, pp. 172–175. [Online]. Available: https://doi.org/10.1109/bcd.2019.8884975

[16] A. Shukla, S. Chaturvedi, and Y. Simmhan, "RIoTBench: An IoT benchmark for distributed stream processing systems," *Concurrency Comput. Pract. Exp.*, vol. 29, no. 21, p. e4257, Nov. 2017. [Online]. Available: https://doi.org/10.1002/cpe.4257

[17] Y. Sasaki and T. Yokotani, "Performance evaluation of MQTT as a communication protocol for IoT and prototyping," *Adv. Technol. Innovat.*, vol. 4, no. 1, pp. 21–29, Jan. 2019. [Online]. Available: https://ojs.imeti.org/index.php/AITI/article/view/2522

[18] F. Palmese, E. Longo, A. E. C. Redondi, and M. Cesana, "CoAP vs. MQTT-SN: Comparison and performance evaluation in publish-subscribe environments," in *Proc. IEEE 7th World Forum Internet Things (WF-IoT)*, Jun. 2021, pp. 153–158.

[19] D. Thangavel, X. Ma, A. Valera, H.-X. Tan, and C. K.-Y. Tan, "Performance evaluation of MQTT and coAP via a common middleware," in *Proc. IEEE 9th Int. Conf. Intell. Sens. Sens. Netw. Inf. Process. (ISSNIP)*, 2014, pp. 1–6.

[20] P. Sommer, F. Schellroth, M. Fischer, and J. Schlechtendahl, "Message-oriented middleware for industrial production systems," in *Proc. IEEE 14th Int. Conf. Autom. Sci. Eng. (CASE)*, 2018, pp. 1217–1223.

[21] D. L. de Oliveira, A. F. da S. Veloso, J. V. V. Sobral, R. A. L. Rabelo, J. J. P. C. Rodrigues, and P. Solic, "Performance evaluation of MQTT brokers in the Internet of Things for smart cities," in *Proc. 4th Int. Conf. Smart Sustain. Technol. (SpliTech)*, Jun. 2019, pp. 1–6. [Online]. Available: https://doi.org/10.23919/splitech.2019.8783166

[22] B. Mishra, "Performance evaluation of MQTT broker servers," in *Computational Science and Its Applications–ICCSA*, O. Gervasi *et al.* Eds. Cham, Switzerland: Springer, 2018, pp. 599–609.

[23] E. Bertrand-Martinez, P. D. Feio, V. de Brito Nascimento, F. Kon, and A. Abelém, "Classification and evaluation of IoT brokers: A methodology," *Int. J. Netw. Manag.*, vol. 31, no. 3, p. e2115, Jun. 2020. [Online]. Available: https://doi.org/10.1002/nem.2115

[24] Scalagent, Échirolles, France, "Benchmark of MQTT servers," Jan. 2015. [Online]. Available: http://www.scalagent.com/IMG/pdf/Benchmark_MQTT_servers-v1–1.pdf

[25] R. Kawaguchi and M. Bandai, "A distributed MQTT broker system for location-based IoT applications," in *Proc. IEEE Int. Conf. Consum. Electron. (ICCE)*, Jan. 2019, pp. 1–4. [Online]. Available: https://doi.org/10.1109/icce.2019.8662069

[26] K. Terada, S. Ohno, H. Mukai, K. Ishibashi, and T. Yokotani, "An ICN system focusing on distributed MQTT brokers for IoT services," in *Proc. 7th Int. Japan-Africa Conf. Electron. Commun. Comput. (JAC-ECC)*, 2019, pp. 28–31.

[27] L. Stagliano, E. Longo, and A. E. C. Redondi, "D-MQTT: Design and implementation of a pub/sub broker for distributed environments," in *Proc. IEEE COINS IEEE Int. Conf. Omni-layer Intell. Syst. (COINS)*, Aug. 2021, pp. 1–6.

[28] J.-H. Park, H.-S. Kim, and W.-T. Kim, "DM-MQTT: An efficient MQTT based on SDN multicast for massive IoT communications," *Sensors*, vol. 18, no. 9, p. 3071, Sep. 2018. [Online]. Available: https://doi.org/10.3390/s18093071

[29] S. Abdelwahab and B. Hamdaoui, "FogMQ: A message broker system for enabling distributed, Internet-scale IoT applications over heterogeneous cloud platforms," 2016, *arxiv.org/abs/1610.00620*.

[30] A. Schmitt, F. Carlier, and V. Renault, "Dynamic bridge generation for IoT data exchange via the MQTT protocol," *Procedia Comput. Sci.*, vol. 130, pp. 90–97, 2018. [Online]. Available: https://doi.org/10.1016/j.procs.2018.04.016

[31] P. Manzoni, E. Hernández-Orallo, C. T. Calafate, and J.-C. Cano, "A proposal for a publish/subscribe, disruption tolerant content island for fog computing," in *Proc. 3rd Workshop Exp. Design Implement. Smart Objects SMARTOBJECTS*, 2017, pp. 47–52. [Online]. Available: https://doi.org/10.1145/3127502.3127511

[32] R. Banno, J. Sun, M. Fujita, S. Takeuchi, and K. Shudo, "Dissemination of edge-heavy data on heterogeneous MQTT brokers," in *Proc. IEEE 6th Int. Conf. Cloud Netw. (CloudNet)*, 2017, pp. 5–11.
[33] R. Banno, K. Ohsawa, Y. Kitagawa, T. Takada, and T. Yoshizawa, "Measuring performance of MQTT v5.0 brokers with MQTTLoader," in *Proc. IEEE 18th Annu. Consum. Commun. Netw. Conf. (CCNC)*, Jan. 2021, pp. 1–2. [Online]. Available: https://doi.org/10.1109/ccnc49032.2021.9369467
[34] Z. Y. Thean, V. V. Yap, and P. C. Teh, "Container-based MQTT broker cluster for edge computing," in *Proc. 4th Int. Conf. Workshops Recent Adv. Innovat. Eng. (ICRAIE)*, Nov. 2019, pp. 1–6. [Online]. Available: https://doi.org/10.1109/icraie47735.2019.9037775
[35] R. Banno, J. Sun, S. Takeuchi, and K. Shudo, "Interworking layer of distributed MQTT brokers," *IEICE Trans. Inf. Syst.*, vol. E102.D, no. 12, pp. 2281–2294, Dec. 2019. [Online]. Available: https://doi.org/10.1587/transinf.2019pak0001
[36] Apace Softw. Found., Forest Hill, MD, USA, "Apache ActiveMQ. Version: Artemis 2.17.0," Feb. 16, 2021. [Online]. Available: http://activemq.apache.org
[37] Emitter Studios, Sacramento, CA, USA, "B.V. Emitter. Version: 2.8," Jun. 4, 2020. [Online]. Available: https://emitter.io
[38] BeerFactory, Puchong, Malaysia, "HBMQTT. Version: 0.9.6," Jan. 25, 2020. [Online]. Available: https://github.com/beerfactory
[39] "MoscaJS. Aedes. Version: 0.45.0," Mar. 4, 2021. [Online]. Available: https://github.com/moscajs/aedes
[40] "Moquette. Version: 0.14," Jan. 22, 2021. [Online]. Available: https://github.com/moquette-io/moquette
[41] VMware, Inc. Palo Alto, CA, USA, "RabbitMQ. Version: 3.8.19," Jul. 7, 2021. [Online]. Available: https://www.rabbitmq.com/mqtt.html
[42] Octavo Labs AG, Zürich, Switzerland, "VerneMQ. Version: 1.12.1," Jul. 11, 2021. [Online]. Available: https://vernemq.com

Edoardo Longo received the M.S. degree in computer science and engineering from the Politecnico di Milano, Milan, Italy, in 2017, where he is currently pursuing the Ph.D. degree in telecommunications with the Department of Electronics, Information and Bioengineering, Advanced Network Technologies Laboratory.

From October 2021 to March 2022, he was a Visiting Researcher with the Computer and Communication Systems Labs, Technische Universität Berlin, Berlin, Germany. His research activities include distributed communication protocol for IoT systems, mobile-edge computing, and constraints devices.

Alessandro Enrico Cesare Redondi (Senior Member, IEEE) received the M.S. degree in computer engineering and the Ph.D. degree in information engineering from the Politecnico di Milano, Milan, Italy, in July 2009 and February 2014, respectively.

He is currently an Assistant Professor with the Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano. From September 2012 to April 2013, he was a visiting student with the EEE Department, University College of London, London, U.K. His research activities are focused on the design and optimization of IoT systems and on network data analytics.

Matteo Cesana (Senior Member, IEEE) received the M.S. degree in telecommunications engineering and the Ph.D. degree in information engineering from the Politecnico di Milano, Milan, Italy, in July 2000 and September 2004, respectively.

He is currently an Associate Professor with the Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano. From September 2002 to March 2003, he was a Visiting Researcher with the Computer Science Department, University of California at Los Angeles, Los Angeles, CA, USA. His research activities are in the field of design, optimization, and performance evaluation of wireless networks with a specific focus on communication technologies for the Internet of Things and future generation cellular networks.

Dr. Cesana is an Associate Editor of the *Ad Hoc Networks* (Elsevier).

Pietro Manzoni (Senior Member, IEEE) received the master's degree in computer science from the Università degli Studi Di Milano, Milan, Italy, in 1989, and the Ph.D. degree in computer science from the Politecnico di Milano, Milan, in 1995.

From November 1992 to February 1993, he did an internship with Bellcore Laboratories, Red Bank, NJ, USA, and from February 1994 to November 1994, he was a Visiting Researcher with the International Computer Science Institute, Berkeley, CA, USA. He is currently a Full Professor of Computer Engineering with the Universitat Politècnica de València, Valencia, Spain, where he is also a Co-Ordinator with the Computer Networks Research Group, Department of Computer Engineering. His research interests include networking and mobile systems and applied to intelligent transport systems, smart cities and the Internet of Things, and community networks (mesh).