Journal of Electronics and Electrical Engineering
https://ojs.wiserpub.com/index.php/JEEE/

UNIVERSAL WISER
PUBLISHER

Review

On MQTT Scalability in the Internet of Things: Issues, Solutions, and Future Directions

Marco Aurélio Spohn

Federal University of Fronteira Sul, Chapecó, SC, Brazil
E-mail: marco.spohn@uffs.edu.br

Received: 13 July 2022; **Revised:** 15 September 2022; **Accepted:** 28 September 2022

Abstract: The Publish/Subscribe (P/S) paradigm plays an essential role in developing Internet of Things (IoT) applications. Among the most representative P/S protocols, there is Message Queuing Telemetry Transport (MQTT). Standard implementations employ a single server acting as a broker for client-to-client communication: publishers send messages to the broker, which forwards them to the subscribers. A single server is a single point of failure and a potential bottleneck. Most IoT applications require a reliable and scalable communication system. MQTT systems can evolve in such requirements through clustering or federation of brokers, resulting in more complex communication architectures. This work presents an overview of current issues and solutions for addressing MQTT scalability in the IoT context.

***Keywords*:** MQTT, scalability, Internet of Things

1. Introduction

The Publish/Subscribe (P/S) paradigm is a standard method for client-to-client communication: producers of information (publishers) send messages to a server/broker that relays the data to consumers (subscribers). This approach's main disadvantage is that the broker is both a bottleneck and a single point of failure. Among the most representative P/S protocols, there is Message Queuing Telemetry Transport (MQTT) [1]. Due to its low complexity, the protocol is available in most Internet-of-Things (IoT) development platforms [2].

MQTT scalability is paramount for many IoT applications. Communication between IoT devices and their surrounding entities (e.g., gateways, local servers/orchestrators) depends on low overhead protocols. IoT systems must accommodate a growing number of clients with limited computing capabilities. When devices run on batteries, lowering communication costs is mandatory.

Adding more resources is a usual strategy to cope with increasing demand. Thus, scalability is more ready to address in standard client-server systems: once demand increases, more servers can come online to receive requests from the load balancer. On the other hand, MQTT is a protocol for client-to-client communication, having a server acting as a broker. Therefore, one can not make the same assumptions when analyzing performance results from scaling the system's number of brokers because clients may connect to different brokers. Whenever clients connect to various brokers, broker-to-broker communication might be needed to close the loop between clients. A publication for which there is also a remote subscriber requires sending the message to the destination broker. Such extra computing and communication overhead does not occur in a typical client-server system because it restrains the communication between the client and the server.

Brokers' orchestration can occur through clustering or the federation of servers. Clustering can be vertical or horizontal. In vertical clustering, also known as scaling up, the same physical machine can host many virtual machines (or containers), running a broker instance. Meanwhile, in horizontal clustering, brokers are placed in separate physical machines. In turn, the federation of brokers extends the deployment beyond every single domain: an overlay network can virtually connect any set of brokers, creating an environment for exploring redundancy exceeding clustering capabilities. Notwithstanding, the orchestration can mingle both approaches resulting in hybrid architectures.

This work presents an overview of the issues and solutions related to the scalability of MQTT specifically. First, the focus is on single broker enhancements, showing approaches for developing the broker's capability. Such improvements can potentially address the expected scalability depending on the target applications. However, a single server has resource limits, regardless of how we scale it vertically. Second, the attention is on horizontal scalability strategies, usually based on clustering within a single administrative domain. Thereon, we switch gears to the available methods for brokers' federation to extend the scalability beyond a particular domain.

2. Vertical Scalability

Vertically scaling an MQTT system implies adding more computing power to the broker-hosting machine. By increasing processing and memory capacity, the broker can serve a rising number of clients. A single machine can also host multiple broker instances, expanding the overall capacity, especially when brokers are single-threaded.

Computing and communication capabilities are scarce for many IoT devices. When the devices rely on batteries, processing data and performing the required communication must be planned so that it allows for extending the system lifetime. One way to comply with that is to include client scalability in the system design. The broker could accommodate some of the client processing load or, at least, not send unnecessary messages to the clients (e.g., clients could instruct the broker to send pre-processed sets of data instead of every single sample). Such instructions could be part of the subscription process, including condition statements, functions, and other sophisticated tasks to scan the data publishers send to the broker [3].

2.1 *Server's Scalability*

When the broker runs its services on a single thread (e.g., Mosquitto [4]), only one CPU/Core is necessary. Handling all the publication and message forwarding tasks must pass through the same thread, not allowing to take advantage of multiple cores when available. In order to use a single machine's total processing capacity, it would be necessary to run multiple broker instances.

As MQTT is TCP based, the overall system performance relies on how TCP handles all active connections. Higher TCP contention hinders MQTT performance, mainly when the in-kernel TCP stack does not operate over multiple cores, as is usually the case.

One way to address the TCP stack limitation issues is to deal with the stack at the user level. Pipatsakulroj et al. [5] proposed muMQ, a scalable MQTT broker based on mTCP [6]), that runs the TCP stack in the user space capable of handling many connections simultaneously over multiple CPU cores. The solution explores an event-driven technique based on *epoll*, which is a Linux kernel system call for handling scalable I/O events through a notification mechanism. A single thread can handle multiple TCP connections, channeling TCP streams through a parallelized event loop.

Under the hood, mTCP bypasses the in-kernel TCP stack employing the Receive Side Scaling (RSS) [7] network driver technology for distributing network traffic over multiple cores. Because the cores share the execution engine, hyper-threading is out of an option when it is available. A Poll Mode Driver (PMD) interacts directly with the Network Interface Card (NIC) hardware, directing packets' streams to the corresponding NIC-related threads.

Therefore, in muMQ, a high scalability degree is achieved through a direct relationship between every TCP thread and the corresponding application thread running on the same CPU core. Communication between the threads happens through a shared buffer assigned to an mTCP context and its related polling descriptor. Race conditions among applications' threads are avoided by employing message queues in the application context and assigning a dedicated

FIFO list to every user-level socket.

In standard TCP, a packet ordering gap requires the receiver to wait until the sender retransmits the missing packets, what is usually called the *head-of-line blocking* [8]. The more often the situation arises, the more it impacts the connection throughput. TCP can also face connection issues leading to wasted CPU time and memory usage. One classic example is the SYN flood attack that fills the target machine's TCP pending connections table preventing regular clients from connecting.

Given the inherent TCP disadvantages, Google introduced QUIC, a new protocol for streaming [9,10,11]. It is not a new transport layer protocol, as it runs on top of UDP and provides user-level multiplexing mechanisms. QUIC's reliance on UDP allows a straightforward way to provide connection migration besides not incurring the intrinsic TCP connection overhead.

MQTTw/QUIC [12] is an MQTT approach based on QUIC. Two software agents provide the integration between MQTT and QUIC, one located between the broker and the QUIC server, and one other sitting between MQTT clients and the QUIC client. Under three IoT connection scenarios (i.e., local wired, local wireless, and long-distance networks), performance results show gains in control overhead, processor and memory utilization, and delivery delay compared to MQTT. Nonetheless, the traffic pattern used in the tests did not allow the evaluation of QUIC's flow control mechanism. However, the authors mention that such an issue may be relevant to some IoT scenarios, demanding proper attention in future works.

An MQTT QUIC variant (client and broker developed in Go language, https://golang.org/) is proposed by Fernández et al. [13]. The authors, motivated by QUIC's low latency, evaluated their solution for diverse IoT scenarios based on WiFi, 4G/LTE, and satellite connection, with RTT going from 25 ms to 600 ms. The QUIC variant surpasses MQTT, especially for lossy channels with short RTT. Network bandwidth limitation is not crucial when IoT communication follows a few packet transmission patterns. In future work, the authors mention exploring multi-messages transmission in the same QUIC packet.

2.2 *Clients' Scalability*

MQTT+ [3] is an MQTT compatible broker with an extended subscription syntax for topic-related rules such as data filtering, processing, and advanced tasks. MQTT+ main features are:

- Definition of conditions/rules for sending publications to subscribers. For example, the client could specify to receive publications only if their assigned values fall within a range of values.
- Definition of an expiration time (Data TTL) for publishing messages: let the user define a lifetime for published messages, leaving the decision problem and the extra processing time to the broker.
- The broker lets the aggregation of several topics (spatial data) in a single message, reducing the number of required transmissions.
- Definition of temporal aggregated measurements (e.g., sum, count, average, minimum, maximum) for a specified time interval.
- The broker implements specialized functions for processing data before handling them to the subscribers. Processing functions, such as video and data compression, and signal processing procedures, are advertised through the MQTT system topic (i.e., $SYS topics).
- As an extra capability, all the previous traits can be mixed in elaborate subscriptions.

A heterogeneous IoT environment comprises devices with contrasting hardware capacities. Devices with limited resources might need more stringent flow control when playing the subscriber role. For such scenarios, Hwang et al. [14] propose the Reception Frequency Control (RFC) algorithm for letting the subscriber inform the broker of its maximum message receiving rate (called Maximum Reception Period, MRP). A subscriber reports its MRP information right after the packet's QoS field in its header. The broker then keeps track of all subscribers' MRPs, controlling the message transmission rate. Results show that RFC is adequate to its initial purpose, besides reducing the broker's network traffic.

Sadeq et al. [15] propose a QoS and flow control mechanism in which subscribers send feedback to the publishers, through the broker, regarding the subscribers' acceptable flowrate observing the following instructions:

- Pause sending: instructs the publisher to pause new publications until further notice.
- Un-pause sending: let the publisher restart sending new publications.
- Send faster: instructs the publisher to allow a faster publication rate whenever the subscriber's buffer usage is at

20% or lower.

- Send slower: inform the publisher to reduce the publication rate when the subscriber's buffer usage reaches 90%.

Unlike Hwang et al. solution, the flow control happens at the publisher, meaning it is more appropriate when a larger set of subscribers connect to unique topics. Their solution also addresses traffic classification by combining standard MQTT QoS and flow control according to the following traffic profiles:

- Regular data: it is the standard best-effort MQTT approach (i.e., QoS 0) with no flow control;
- Real-time traffic: it is the same as regular data but with the addition of flow control;
- Critical and time-sensitive traffic: it relies on the first type of reliable transmission in MQTT (i.e., QoS 1) plus the flow control;
- Critical data: this is virtually the same as the previous profile, but with QoS 2.l

Jo and Jin [16] tackle the problem when a single subscriber receives periodic publications from several publishers (i.e., N-to-1 communication). Applications within this domain include the systematic data acquisition process in cyber-physical systems. Clients provide their timeliness constraints by letting the corresponding publishers adjust their transmission rates according to the available clients' communication and computing resources. Clients communicate such information through adaptation request messages, but it is unclear how such exchange occurs without changes to the broker. Nevertheless, performance results point to overall improvements in messages' timeliness.

MQTT standard has evolved, including enhanced client support to help scale the system. Even though some character-istics help enhance a broker's performance, they are specific to a single broker instance. On its current version (version 5, v5) [1], we can pinpoint the following features in that direction:

- During the connection stage, the broker can inform the client regarding services' restrictions because some features may push the server beyond its limits. For example, the server may tell it does not support retained messages.
- The client can also decide if it desires or not to receive any retained message.
- The broker may redirect the client to another server during connection.
- The client can define an expiration time for non-clean sessions (for QoS 1 and 2). The broker can delete all retained data after expiration, freeing up the broker's resources.
- The client can inform a maximum message size: the broker automatically discards any message bigger than that.
- The client can announce the maximum number of simultaneous transmissions of messages with QoS 1 and 2, allowing some flow control.
- Introduction of topic aliases, reducing the topic field size; consequently, reducing the packet size.
- Non-local publishing: a client may choose not to receive its publications when it is also a subscriber to the same topic.
- Shared subscriptions: a set of subscribers can join a virtual group corresponding to the same topic. The broker forwards a new published message to just one group member, usually following a round-robin process among group members.
- The publisher can transmit metadata (e.g., timestamps, message sequence numbering, content specification) outside the payload directly to the subscribers.
- Even though the P/S paradigm remains central to MQTT, there is the possibility to mimic the request/response model typical to the client-server model. It employs a response topic, which allows an MQTT client to act as a server, though still passing all the communication through the broker.

3. Horizontal Scalability

As a form of vertical broker clustering, one might choose to run several broker instances in the same machine. The overall system capacity increases depending on the machine's computing and communication capabilities. MQTT services remain available as long as at least one broker instance is working and reachable. Nonetheless, the system is prone to general hardware failures, communication problems, and the like, returning to the inconvenient single point of failure.

One may aim at a horizontal approach based on clustering or brokers' federation to advance scalability to a broader level. Next, we explore both, presenting the basics, challenges, and some of the most representative proposals in the area.

3.1 *Clustering*

A set of brokers can act together, forming a cluster and keeping it transparent to the clients. A load balancer (LB) acts between clients and the brokers, striving to spread the working load evenly along the servers. An elastic service capacity is possible through brokers' dynamic activation and deactivation. Clustering allows scaling up the system while addressing reliability and availability as well.

The LB acts as a reverse proxy at the application layer, acting as a single entry point and dynamically redirecting the client to one of the available brokers. A Destination Network Address Translation (DNAT) table stores the necessary information to keep the traffic streaming between clients and their corresponding brokers. Clustering brokers can run on dedicated machines or in virtualized environments such as virtual machines (VMs) or containers.

Round robin and random selection are the usual choices for picking a broker for a new client. When publishers and their subscribers are with different brokers, they must route the messages between them. Therefore, clustering entails learning where and how to reach all the intended targeted clients. Transparency allows keeping clients working as usual, at the expense of additional control overhead at the brokers' side, which is the most impending factor in enhancing cluster performance.

Distributing clients randomly over the clustering brokers may be the fairest approach, but Detti et al. [17] show that it presents not-so-proportional performance results. For an increasing number of brokers, the cluster scales up sublinearly. Results show that the cluster consumes approximately half of all available computing and communication resources, mainly due to the control and intra-cluster message routing.

As a result of their findings, Detti et al. [17] studied mechanisms based on analytical and simulation analysis to optimize load balancing. Instead of letting clients connect to the cluster over a single connection, the load balancer chooses a set of brokers to connect simultaneously, given that the brokers match the target topics. The load balancer connects the client to m brokers out of the total available in the cluster, associating each session to a set of topics. Results show that choosing multiple connections does not impact or limit the fairness constraint, primarily when the number of topics per client increases or the cluster expands.

The sublinearity is deeply conditional to applications' profiles. The harshest situation occurs when brokers need to send any new publication to all the remaining clustering brokers. Therefore, sublinearity is more evident when there are fewer subscribers per topic while being less pronounced when there is a large set of subscribers for a reduced number of topics. The proposed greedy load balancing solution performs nearly linearly for IoT and social network systems.

In short, Detti et al. show it is possible to improve MQTT cluster load balancing, but there is no clear line between performance improvements while keeping fair broker load allocation. It is also a fact that applications vary widely regarding topics, the number of clients, and topic density. Intra-cluster communication is costly mainly due to MQTT statefulness and extra complexity in brokers and clients.

Jutadhamakorn et al. [18] employed NGINX [19], a web server with integrated load balancing, for an MQTT clustering system. For orchestrating an elastic number of brokers in the cluster, they use Docker Swarm [20], which employs containers as a basic execution unit. The load balancer selects a broker by employing a hash function with the client ID as input. Even though performance results are promising, it is unclear how intra-cluster routing works.

The devices connect to the cloud through gateways in a fog IoT model. Rausch et al. [21] propose a QoS-aware middleware for orchestrating clients' connections to a pool of MQTT brokers through the gateway devices. Their work's main contribution is taking client proximity and latency instabilities (mainly due to link particularities) for optimizing QoS. Since gateways are closer to a new connecting device, they are better prepared to pick an available broker to connect it. The routing among brokers follows a bridging scheme, with a centralized controller operating as the registry and monitoring hub for the whole orchestration process. The controller shares with the brokers bridging tables specifying the association between subscribers and brokers. A proximity engine runs in the controller to monitor the latency between gateways and brokers and reconnect them dynamically to improve QoS.

Communications between clients and the broker are unicast by default. Direct Multicast-MQTT (DM-MQTT) [22] is a Software Defined Network (SDN) system that builds a multicast tree connecting publishers to their subscribers. The fundamental idea behind this approach is to skip the broker and lower the end-to-end communication delay between clients.

Al-Fuqaha et al. [23] propose an improved MQTT architecture with the following characteristics:

- Similar to DM-MQTT, there is a Machine-to-Machine (M2M) communication mode performing on a multicasting way without passing through the broker, with the intent to provide increased reliability.
- Supports subscribers' broker migration to enhance the QoS requirements.
- Supports multiple messages' dynamic priority queues: QoS control allows removing or moving messages between queues for single or multi-broker scenarios.
- Allows more complex client behavior by providing brokers' traffic statistics.

Even though some simulation results point to improvements in reducing queueing delays, there is a lack of a full feature implementation of the proposed architecture to compare it to the standard MQTT. Besides that, the authors mention a management node without further explanation about its role. The inter-broker routing process is not present in their work either.

3.2 *Federation*

A set of brokers can work together by adopting a Peer-to-Peer (P2P) orchestration practice. We can refer to such an assemblage as a federation of brokers. It is also desirable that it happens in a self-managed way, mainly because there is not a single point of entry as is the case for the load balancers in clustering.

The federation can span multiple administrative domains, building upon an application overlay network. Virtual links connect broker nodes, allowing any topology. Federation and clustering could combine to extend their inherent scalability and reliability properties.

For a self-managed federation to work, broker nodes play a central role in building and maintaining the infrastructure as follows:

- A broker node needs to find itself in the overlay net-work, its neighbors, and how to reach them. The virtual topology may be static or dynamic.
- Clients must be able to access brokers as usual, and the brokers must handle all the message routing to reach the subscribers wherever they are.
- Control overhead must be minimal due to topic messages' routing.

Besides the intended aggregation of brokers' capacity to provide scalability, the federation lets exploring reliability due to the many possible ways of interconnecting the broker nodes. Multi-path routing can address broker reachability in extensive ways, which are impossible when employing a cluster running in a single administrative domain.

A self-organizing federation of autonomous brokers is presented by Spohn [24,25,26] and by Ribas and Spohn [27]. The solution builds a mesh connecting subscribers to the same topic, having minimal control overhead. Each mesh has a single-core broker node as a reference for building and maintaining the mesh. Publications can initiate from inside a mesh member or any other node in the overlay network (every node learns about any existing core). When coming from outside the mesh, the brokers route the publication toward the mesh core: it first reaches the core or the mesh; whatever happens first makes the publication spread just inside the mesh. However, there is no extensive performance analysis for the proposed solution, including no such comparison to clustering.

Longo et al. [28] propose MQTT-ST, which works by interconnecting a set of brokers through a spanning tree structure along the federation model. Message routing happens by flooding the tree, which is essentially loop-free. However, compared to the previous solution, there is no path redundancy (i.e., any link failure partitions the overlay network).

4. Available Implementations

RabbitMQ [29] is an open-source message broker supporting multiple protocols, including MQTT version 3.1 and a distributed deployment for brokers' clustering and federation. Clustering nodes are equal peers, having every required state for running a broker replicated across all nodes. As an exception to this rule, by default, message queues are stored just on one node while accessible to all peers. However, a queue type supporting replication is available: quorum queues implement a long-lasting replicated FIFO queue based on a consensus algorithm.

In their turn, clients connect to just one RabbitMQ broker at a time. The connecting broker handles any required routing to other peer brokers. When using queue mirroring, some node plays the queue master/leader (called quorum queue leader).

HiveMQ [30] is a Java-based MQTT implementation supporting clustering. Even though there is a limited open-source Community Edition, it does not include clustering features. On the other hand, an evaluation version with full capabilities is available but allows only a few connections.

VerneMQ [31] and eMQTT [32] are open-source MQTT compliant implementations based on the Open Telecom Platform (OTP), a collection of middleware and libraries written in Erlang (Erlang is a programming language developed initially at the Ericsson Computer Science Laboratory). Both implementations include clustering, with Erlang natively supported real-time messaging and distributed capabilities.

SwiftMQ™ [33] builds around the concept of Federated Router Network, where each broker is a SwiftMQ router running a routing engine. Besides MQTT, it supports other protocols over an integrated microservice platform responsible for intercepting, processing, and analyzing messages sent over queues or topics. Even though the routing process is dynamic, the solution is not self-organized.

5. Testing Scalability

Testing an MQTT system (single or multi-broker) requires varying a broad range of configuration parameters [34]. Each client requires a TCP connection to the broker, allowing us to test how many clients the system can sustain without degrading performance. Connection requests can eventually burst due to higher demand or an intentional denial-of-service attack. Bursts can be tested separately or in combination with various active client connections.

The system's limits for topics and subscriptions are a test achievable by varying the number of clients subscribed to the same topic and the number of subscriptions a single client can hold. Jointly, we can test message throughput: i) *fan-in*, for situations with a large number of publishers and a smaller set of subscribers; ii) *fan-out*, for circumstances with a few publishers and a large set of subscribers for the same topic; and iii) *one-to-one*, for stressing throughput when there is a large set of single pairs of publishers and subscribers, which is also limited to the maximum number of TCP connections a broker can hold without degrading performance.

Message size and publication periodicity are test parameters that fit all possible scenarios. Most IoT applications require topics with tiny messages, with just a few bytes. Nevertheless, the supposed possible message size range requires testing. In addition to that, the broker's incoming traffic will vary with the frequency publishers send messages to the broker. In turn, the outgoing traffic is also a function of the remaining parameters (e.g., number of subscribers).

The three available QoS levels impose additional control overhead upon clients and brokers. Therefore, any configuration scenario can be categorized accordingly to the required QoS level, including situations with any possible QoS combination among clients.

For persistent sessions and QoS 1 and 2, published messages remain in queues for future delivery to offline subscribers. Queuing storage capacity is limited, requiring the control of queued messages for every single disconnected client. The overall queuing capacity must endure the expected load, and the predicted on and off periods of all clients, guaranteeing delivery of all retained messages to the clients once they return online.

The last published message for any topic is stored in the broker whenever the retained message flag is active in the publication. The broker will always send the retained message when a new client connects, or an existing client reconnects to the broker. Thus, one can test how the increasing number of publishers with retained messages affects performance. On the other side of the broker, we can test the impact of varying the number of new subscriptions and reconnections, affecting the number of retained message transmissions.

As mentioned earlier, MQTT v5 introduces many new features, which, among all, testing shared subscriptions extends scalability beyond brokers' boundaries. In addition, all the new user-defined controls allow testing a greater variety of scenarios.

5.1 *Tools*

Some open-source MQTT performance tools allow stressing out the main protocol features. Malaria [35] is a set of tools that enables mimicking multiple simultaneous clients. A specified number of separate processes can run to publish to a particular broker a given number of messages of varying sizes at a given rate. The subscribers' side supports specifying the number of expected publishers and the total number of messages assumed for each publisher. The application captures message success rate, messages per second, and timing measures (i.e., message timing mean, standard deviation, minimum, and maximum values).

Rausch [36] authored a tool suite for benchmarking P/S systems. Even though the tool provides complete logging of messages sent/received, it does not compute any statistics, leaving the logging file's post-processing to the user. The tool provides an interactive shell interface for starting a set of clients (i.e., publishers and subscribers) with a specific behavior based on the following options:

- Topic name.
- Number of publishers and subscribers.
- Logging/recording of messages.
- Publishing frequency (messages/s): to stop publishing, the frequency can be set to zero in the shell command line.
- Dump of recorded messages for post-processing.

Hiep [37] presents a plugin for Apache JMeter™ (JMeter is an open-source Java application designed to test load behavior and measure performance) for testing MQTT brokers. The tool provides a graphical interface allowing the definition of the following test options:

- Topics' list: The application creates one connection to the broker for each topic.
- Publishing strategies: *round robin* (publishes topics uniformly following a circular order) and *random* (chooses a topic randomly).
- Message type: text message, generated value (integer, long, float, or double both within a given range), a fixed value (same types as previously, with the addition of string type), and a random byte array with a given size.
- Message format: defines the encoding type (binary, base64, binhex, or plain text).
- Message timestamp: adds an eight-byte timestamp to each message.
- Retained messages and QoS options: allows setting the retain flag and any of the three available QoS options.

We can define the number of samples to aggregate at the subscriber's side, leaving JMeter the measuring task. Performance results are available as an aggregate graph (including average, median, and deviation values for the metrics) or table format (with all particular samples) for exporting and post-processing.

Jianhui and Xiang [38] present a tool in Go language for measuring the broker's forwarding latency. The tool allows setting the following parameters:

- Authentication parameters (i.e., username and password);
- QoS of published messages;
- Subscription QoS level;
- Messages payload size (in bytes);
- Number of client pairs (by default, the tool instantiates ten pairs);
- Number of messages per publisher (defaults to 100 messages);
- Keepalive period;
- Benchmark results format (allows text and JSON formats).

Each published message carries a timestamp in its payload, allowing computing of the following statistics:

- For publishers: success ratio, publication delay (minimum, maximum, mean, and standard deviation), and publication bandwidth (i.e., messages per second);
- For subscribers: forward success ratio and latency (minimum, maximum, mean, and standard deviation).

6. Open Directions for Research

Scaling a single broker means achieving the most significant number of simultaneous client connections. Therefore, there is a need to understand all the available services and mechanisms to use all available CPUs/Cores. For example, as seen in previous works, TCP performance improves when connection handling happens in user space with a unique low-level network interface polling service. Likewise, it shows space for improvements if the TCP stack ever has multi-core support at the kernel level.

Vertically scaling a single machine can improve a single broker and a vertical cluster of individual brokers. In the former case, we still have the single point of failure drawback, whereas we grant some fault tolerance level to the system in the latter case. Both situations are equally prone to general failure if the host machine breaks apart or entirely disconnects from the network.

Horizontal clustering is the preferred scalability approach. A load balancer runs as the system's gateway for guiding clients to one of the available brokers. The decision is usually a random selection to achieve a uniform work distribution among brokers. As shown in previous work, such a standard load balancing procedure results in a sub-linear behavior, mainly due to the extra control overhead resulting from intra-cluster routing. Recent proposals show space for improving broker selection, ordinarily conditioned to applications' characteristics (e.g., number of clients, topics' subscriptions distribution, number of brokers).

Clustering regularly is subject to one particular administrative domain, whereas brokers' federation can span beyond borders. A federation approach should increase scalability and availability, exceeding clustering capabilities. However, both methods can combine, resulting in richer architectures. There is much to learn from possible improvements attainable through the federation, mainly related to the inter-broker routing process.

There are many possible ways to improve MQTT scalability conceivably by combining the already known issues and solutions, as follows:

- Brokers can take plenty of advantage of multi-threading architectures jointly with a parallelized TCP stack. An analysis [39] shows many ways to provide better Linux scalability support for multi-core systems. It is possible for multi-core packet processing in the kernel by taking advantage of network cards with multiple hardware queues. It is also feasible to process each connection entirely on one core, mainly beneficial for fast connections such as those coming from publishers.
- Alternative transport protocols, such as QUIC, pave a promising path to enhance current broker standards while allowing a rich environment for proposing new services and mechanisms, given the transport protocols' specificities.
- Give proper attention to clients' scalability, providing mechanisms extending de protocol syntax to include richer filtering and processing options. Of course, some of the processing load goes to the server. However, it probably allows having many more connected IoT clients with the same physical machine with the appropriate vertical tuning.
- Explore dynamic load balancing mechanisms by constantly analyzing clients' traffic profiles and adjusting broker selection, incurring the lowest impact on intra-cluster routing.
- For some IoT applications, improved QoS and flow control are paramount. Proper protocol syntax extensions could address such requirements jointly with the traffic profiling extensions to the load balancer and the broker.
- Designing new orchestration architectures with combined clustering and federation capabilities, exploring scalability and availability to a broader extent.
- There is a need for better testing tools to address the specificities of a wider variety of applications and systems architectures.

7. Conclusions

The MQTT protocol presents low complexity for clients (publishers and subscribers). Nevertheless, its default configuration relies on a single broker, a potential bottleneck besides a single point of failure. IoT systems can grow exponentially in size, requiring a scalable communication infrastructure. For non-critical IoT applications, vertically scaling a single broker might cope with the expected growing client demand. Nonetheless, when more than one broker is required, clustering or the federation of multiple brokers is usually the way to address scalability.

Vertically scaling a cluster of brokers (i.e., running in the same physical machine) might perform better than beefing up a single broker for the same physical machine. However, even though some degree of fault tolerance is present (i.e., if individual brokers fail), the whole system collapses when the physical machine wreaks havoc. In such cases, we must put forward horizontal scalability through clustering or brokers' federation.

Current solutions show that the broker can increase its capacity by taking advantage of all available computing resources, mainly by processing simultaneous client connections over all possible CPUs/Cores. On the clients' side, enhanced protocol syntax rules (e.g., data filtering, processing, and aggregation tasks) can lower the client load and provide adaptive flow control mechanisms addressing the inherent IoT devices' heterogeneity.

For horizontal scalability, load balancing is the main point to focus on for improving clustering. Brokers' federation allows breaking the single administrative domain barriers, spanning any reachable Internet destination. Even though the federation warrants potential enhanced availability, it is unclear how the federation capacity improvements compare to clustering.

Conflict of Interest

There is no conflict of interest for this study.

References

[1] Standard, O. Mqtt version 5.0. Available online: http://docs.oasis-open.org/mqtt/mqtt/ (accessed on 01 July 2022)

[2] Al-Fuqaha, A.; Guizani, M.; Mohammadi, M.; Aledhari, M.; Ayyash, M. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Commun. Surv. Tutor*. **2015**, *17*, 2347–2376. https://doi.org/10.1109/COMST.2015.2444095

[3] Giambona, R.; Redondi, A.E.C.; Cesana, M. MQTT+: Enhanced Syntax and Broker Functionalities for Data Filtering, Processing and Aggregation. In Proceedings of the 14th ACM International Symposium on QoS and Security for Wireless and Mobile Networks (Q2SWinet'18), Montreal, QC, Canada, 28 October 2018–2 November 2018. https://doi.org/10.1145/3267129.3267135

[4] Foundation, E. Eclipse mosquitto: An open source mqtt broker. Available online: https://mosquitto.org (accessed on 01 July 2022)

[5] Pipatsakulroj, W.; Visoottiviseth, V.; Takano, R. muMQ: A lightweight and scalable MQTT broker. In Proceedings of 2017 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN), Osaka, Japan, 12–14 June 2017. https://doi.org/10.1109/LANMAN.2017.7972165

[6] Jeong, E.; Wood, S.; Jamshed, M.; Jeong, H. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In Proceedings of 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), Seattle, WA, USA, 2–4 April 2014. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong.

[7] Choudhary, A. Introduction to Receive Side Scaling (RSS). Available online: https://medium.com/@anubhavchoudhary/introduction-to-receive-side-scaling-rss-7cd97307d220 (accessed on 01 July 2022)

[8] Bziuk, W.; Phung, C.V.; Dizdarević, J.; Jukan, A. On HTTP performance in IoT applications: An analysis of latency and throughput. In Proceedings of 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, Croatia, 21–25 May 2018. https://doi.org/10.23919/MIPRO.2018.8400067

[9] Roskind, J. Quic - quick udp internet connections. Available online: https://docs.google.com/document/d/1RNHkxVvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit (accessed on 01 July 2022)

[10] Quic, a multiplexed stream transport over udp. Available online: https://www.chromium.org/quic (accessed on 01 July 2022)

[11] Iyengar, J.; Thomson, M. Quic: A udp-based multiplexed and secure transport. Available online: https://tools.ietf.org/html/draft-ietf-quic-transport-34#section-1 (accessed on 01 July 2022)

[12] Kumar, P.; Dezfouli, B. Implementation and analysis of QUIC for MQTT. *Comput. Networks* **2018**, *150*, 28–45, https://doi.org/10.1016/j.comnet.2018.12.012

[13] Fernández, F.; Zverev, M.; Garrido, P.; Juárez, J.R.; Bilbao, J; Agüero, R. And QUIC meets IoT: performance assessment of MQTT over QUIC. In Proceedings of 2020 16th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), Thessaloniki, Greece, 12–14 October 2020. https://doi.org/10.1109/WiMob50308.2020.9253384

[14] Hwang, K.; Lee, J.M.; Jung, I.H. Extension of Reception Frequency Control to MQTT Protocol. *Inter J. Innov. Tech. Explor. Engineer*. **2019**, *8*, 215–219. https://www.ijitee.org/wp-content/uploads/papers/v8i8s2/H10390688S219.pdf
[15] Sadeq, A.S.; Hassan, R.; Al-Rawi, S.S.; Jubair, A.M.; Aman, A.H.M. A Qos Approach For Internet Of Things (Iot) Environment Using Mqtt Protocol. In Proceedings of 2019 International Conference on Cybersecurity (ICoCSec), Negeri Sembilan, Malaysia, 25–26 September 2019. https://doi.org/10.1109/icocsec47621.2019.8971097
[16] Jo, H.-C.; Jin, H.-W. Adaptive Periodic Communication over MQTT for Large-Scale Cyber-Physical Systems. In Proceedings of 2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications. Negeri Sembilan, Malaysia, 25–26 September 2019. https://doi.org/10.1109/cpsna.2015.21
[17] Detti, A.; Funari, L.; Blefari-Melazzi, N. Sub-Linear Scalability of MQTT Clusters in Topic-Based Publish-Subscribe Applications. *IEEE Trans. Netw. Serv. Manag*. **2020**, *17*, 1954–1968. https://doi.org/10.1109/tnsm.2020.3003535
[18] Jutadhamakorn, P.; Pillavas, T.; Visoottiviseth, V.; Takano, R.; Haga, J.; Kobayashi, D. A scalable and low-cost mqtt broker clustering system. In Proceedings of 2017 2nd International Conference on Information Technology (INCIT), Nakhonpathom, Thailand, 2–3 November 2017. https://doi.org/10.1109/INCIT.2017.8257870
[19] Sysoev, I. nginx [engine x]. Available online: https://nginx.org/en/ (accessed on 01 July 2022)
[20] Swarm mode overview. Available online: https://docs.docker.com/engine/swarm/ (accessed on 01 July 2022)
[21] Rausch, T.; Nastic, S.; Dustdar, S. EMMA: Distributed QoS-Aware MQTT Middleware for Edge Computing Applications. In Proceedings of 2018 IEEE International Conference on Cloud Engineering (IC2E), Orlando, FL, USA, 17–20 April 2018. https://doi.org/10.1109/ic2e.2018.00043
[22] Park, J.-H.; Kim, H.-S.; Kim, W.-T. DM-MQTT: An Efficient MQTT Based on SDN Multicast for Massive IoT Communications. *Sensors* **2018**, *18*, 3071, https://doi.org/10.3390/s18093071
[23] Al-Fuqaha, A.; Khreishah, A.; Guizani, M.; Rayes, A.; Mohammadi, M. Toward better horizontal integration among IoT services. *IEEE Commun. Mag*. **2015**, *53*, 72–79, https://doi.org/10.1109/mcom.2015.7263375
[24] Spohn, M.A. Publish, subscribe and federate! *arXiv* 2020. https://doi.org/10.48550/arXiv.2006.03675
[25] Spohn, M.A. An Endogenous and Self-organizing Approach for the Federation of Autonomous MQTT Brokers. In Proceedings of 23rd International Conference on Enterprise Information Systems (ICEIS), Prague, Czech Republic, 26–28 April 2021. https://www.insticc.org/node/TechnicalProgram/iceis/2021/presentationDetails/104088
[26] Spohn, M.A. Self-organizing Federation of Autonomous MQTT Brokers. In *Enterprise Information Systems*, Filipe, J.; Śmiałek, M.; Brodsky, A.; Hammoudi, S. Eds. Springer: Cham, 2022, pp. 369–387. https://doi.org/10.1007/978-3-031-08965-7_19
[27] Ribas, N.K.; Spohn, M.A. A New Approach to a Self-Organizing Federation of MQTT Brokers. *J. Comput. Sci*. **2022**, *18*, 687–694, https://doi.org/10.3844/jcssp.2022.687.694
[28] Longo, E.; Redondi, A.E.C.; Cesana, M.; Arcia-Moret, A.; Manzoni, P. MQTT-ST: a Spanning Tree Protocol for Distributed MQTT Brokers. In Proceedings of ICC 2020 - 2020 IEEE International Conference on Communications (ICC), Dublin, Ireland, 7–11 June 2020. https://doi.org/10.1109/ICC40277.2020.9149046
[29] RabbitMQ message broker. Available online: https://www.rabbitmq.com (accessed on 01 July 2022)
[30] HiveMQ mqtt broker. Available online: https://www.hivemq.com (accessed on 01 July 2022)
[31] VerneMQ mqtt broker. Available online: https://www.vernemq.com (accessed on 01 July 2022)
[32] The massively scalable mqtt broker for iot and mobile applications. Available online: https://emqtt.io/ (accessed on 01 July 2022).
[33] SwiftMQ platform. Available online: https://www.swiftmq.com (accessed on 01 July 2022)
[34] Top 10 iot scalability tests for an mqtt broker. Available online: https://www.hivemq.com/blog/mqtt-broker-scalability-tests/ (accessed on 01 July 2022)
[35] Palsson, K.; Hankinson, E.; Peuster, M. mqtt-malaria. Available online: https://github.com/etactica/mqtt-malaria (accessed on 01 July 2022)
[36] Rausch, T. Tool suite for benchmarking pub/sub systems. Available online: https://git.dsg.tuwien.ac.at/emma/pubsub-benchmark (accessed on 01 July 2022)
[37] Hiep, T. mqtt-jmeter: plugin for jmeter to test mqtt protocol. Available online: https://github.com/tuanhiep/mqttjmeter (accessed on 01 July 2022)
[38] Mqtt broker latency measure tool. Available online: https://github.com/hui6075/mqtt-bm-latency (accessed on 01 July 2022)
[39] Boyd-Wickizer, S.; Clements, A.T.; Mao, Y.; Pesterev, A.; Kaashoek, M.F.; Morris R.; Zeldovich, N. An analysis of linux scalability to many cores. In Proceedings of 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10), Vancouver, BC, Canada, 4–6 October 2010.