A Comparison of MQTT Brokers for Distributed IoT Edge Computing

Heiko Koziolek, Sten Grüner, and Julius Rückert

ABB Corporate Research Center Germany, Ladenburg, Germany
{heiko.koziolek, sten.gruener, julius.rueckert}@de.abb.com

Abstract. Many enterprise IoT application scenarios, such as connected cars, smart cities, and cloud-connected industrial plants require distributed MQTT brokers to achieve high scalability and availability. With a market of over 20 MQTT brokers, it is hard for software architects to make a good selection. Existing MQTT comparisons often include only non-distributed brokers, focus exclusively on performance, or are difficult to generalize. We compared three distributed MQTT brokers for performance, scalability, resilience, security, extensibility, and usability in an enterprise IoT scenario deployed to an edge gateway cluster. We found that EMQX provided the best performance (28K msg/s), while only HiveMQ showed no message loss in our test scenario. VerneMQ offers similar features as the other brokers but is fully available as open source. The paper includes decision guidance for software architects, listing six major decision points regarding MQTT brokers.

Keywords: IoT, MQTT, Distributed Messaging, Edge Computing, Virtualization, Software Containers, Benchmarking, GQM, Performance

1 Introduction

The global Internet-of-Things (IoT) market has an estimated volume of 190 BUSD and is expected to grow to more than 1100 BUSD by 2026 [6]. There are many application areas where connected devices provide value-adding functions: smart cities, industrial plants, smart home, connected cars, smart energy grids, etc. These devices often send telemetry data to edge gateways and cloud platforms, where the data is used for monitoring, supervision, predictive maintenance, and optimization. One of the most popular protocols for this type of communication is MQTT (Message Queuing Telemetry Transport, ISO/IEC 20922), which implements a publish-subscribe pattern [11]. MQTT is specifically suited for IoT applications, since it is designed for unstable network connections and bandwidth saving [13].

There are more than 20 MQTT broker implementations available, making a selection hard for software architects. Software architects need to balance and prioritize different quality attributes of MQTT brokers to make an informed decision. There is a lack of evaluation criteria for such messaging brokers specifically in enterprise IoT scenarios. Such scenarios require scalable, high available

MQTT brokers deployed to a cluster, which brings special challenges for capacity planning and configuration.

Researchers and practitioners have studied different aspects of MQTT communication in the past. There are comparisons to other protocols, such as CoAP, AMQP, and Kafka [5, 22, 12, 21], as well as small-scale performance tests of different, non-clustered MQTT brokers [15, 21, 2]. However, there is no comprehensive comparison between *distributed* MQTT brokers available, which are deployed in highly scalable and redundant edge clusters for enterprise IoT. Practitioner experience reports have demonstrated impressive scalability of MQTT brokers on cloud platforms [17, 4], but are often difficult to generalize since they are geared towards specific contexts. Furthermore these tests often focus exclusively on performance, neglecting other quality attributes.

The contribution of this paper is a comparison of three representative, distributed MQTT brokers using evaluation criteria systematically defined using a Goal/Question/Metric (GQM) scheme [3]. We report on evaluation results for five quantitative metrics and provide additional qualitative analyses for security, usability, and extensibility. We found that EMQX showed the best throughput, while only HiveMQ achieved no message loss in our test scenarios. VerneMQ is fully available as open source, while providing similar features and quality as the commercial brokers. To obtain the previously defined metrics, we deployed the selected MQTT brokers in redundant edge gateway servers running the open-source edge virtualization platform StarlingX. This allowed analyzing the interplay with software containers and container orchestration using Kubernetes (K8s).

The remainder of this paper is structured as follows: Section 2 sets the context for Enterprise IoT messaging, for which Section 3 defines metrics and a representative experiment scenario. Section 4 provides a brief overview of distributed MQTT brokers to rationalize the selected candidates. Section 5 presents the analysis results for performance, scalability, resilience, security, extensibility, and usability. Section 6 summarizes the results and decision points as guidance for software architects. Finally, Section 7 investigates related work and Section 8 concludes the paper.

2 Background: Enterprise IoT Messaging

Fig. 1 shows an enterprise-scale, generic edge gateway cluster architecture that can be useful in different application domains. IoT **Devices** are for example sensors and actuators mainly publishing telemetry data to the edge gateway cluster and occasionally consuming control signals. Due to potential temporal network failures, possibly involving cellular connections and resource-constrained devices, the MQTT protocol [13] is well suited as it is resilient against temporal disconnects and has a low message size overhead thus saving bandwidth.

Message Broker Instances on the edge gateway cluster ingest messages from the IoT Devices and enable different applications to consume them. Distributed MQTT brokers with multiple instances, each residing on a separate

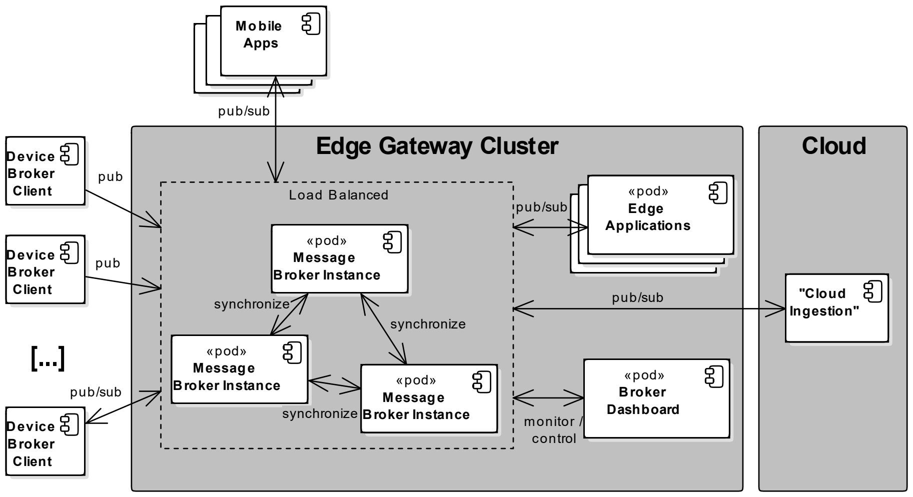

Fig. 1: Distributed Message Brokers on Edge Gateway Cluster

physical or virtual node, may scale horizontally (i.e., with the number of available nodes) to cope with a high number of connected devices and message workloads. The instances exchange messages and client session information, so that the overall system may survive crashes of individual instances of nodes and support high-availability scenarios. A load balancer provides network endpoints of available broker instances to clients interested in messaging, for example in a round-robin fashion. Many MQTT brokers provide a **Broker Dashboard** for monitoring and supervision of the clustered instances.

Mobile Apps subscribe for message topics, for example to display an alarm list to a field operator in an industrial plant or to provide car telemetry data to a car owner. **Edge Applications** may utilize messaging data to execute data analytics algorithms on premises, for example to enable predictive maintenance of individual devices or derive optimizations for the entire system. For larger analysis tasks or cross-site statistics, cloud applications in public data centers ingest the messages via Internet connections.

In enterprise IoT, the edge gateway cluster may contain multiple physical nodes that run multiple layers of virtualization. There are several platforms available supporting different aspects of such an edge gateways, e.g., EdgeX-Foundry, Fledge, KubeEdge, Azure IoT Edge, and StarlingX.

3 IoT Messaging Requirements

We define metrics and evaluation criteria for distributed MQTT brokers (Section 3.1) and specify the basic scenario used in later tests (Section 3.2).

3.1 Goal/Question/Metric

The goal of our study (according to GQM [3]) is to evaluate the quality attributes of distributed MQTT brokers in enterprise IoT scenarios from the software architect's perspective. The following questions with corresponding metrics to answer them shall support achieving this evaluation for each broker:

What is the performance? Metric **M1** is the maximum sustainable throughput (MST) [24] at which the broker is able to process all communicated messages. In this case, both publishers and subscribers are able to maintain stable message queues for an agreed reference workload. Metric **M2** is the average latency from publisher to subscriber in a given scenario. Short latencies are important for many IoT applications, where live monitoring of telemetry data is desired. Practical limits are set by network connections, which introduce latencies outside of the control of the broker.

What is the scalability? Metric **M3** is the maximum number of supported concurrent connections, each issuing a reference workload to the broker. Large-scale IoT scenarios involving smart cities, power distribution grids or fleets of connected cars may include millions of IoT devices. Metric **M4** is the time to start a new broker instance in case of a high load on the already running instances. This metrics pertains the dynamic scalability (elasticity) to cope with changing workloads without wasting computing resources.

What is the resilience? Metric **M5** is the message loss count in case of a broker instance crashing for a reference scenario. While losing individual sensor readings may be acceptable in some scenarios (e.g., temperature values in a smart home), it may be harmful in others (e.g., missing an emergency shutdown signal of a plant). This metric is influenced by the queue lengths configuration of a broker in relation to a particular workload.

What is the security? Security of a broker is largely determined by the user configuration and only to a lesser extent by the broker's security features. These include authentication and authorization mechanisms, as well as encryption support and overload protection procedures. We refrain from defining a potentially misleading, quantitative metric for security and instead provide a qualitative discussion in the evaluation section. Metric **M6** is only a side-aspect of security and measures the overhead of enabled TLS encryption on the maximum sustainable throughput (as percentage).

What is the extensibility? MQTT brokers offer plug-in mechanisms allowing third-party extensions, e.g., logging messages to a database. The evaluation section provides a qualitative discussion on the extensibility of the brokers.

What is the usability? The usability of a distributed MQTT broker includes both installation and operation. Easy deployment on container orchestration systems may be valued. We again refrain from defining a quantitative measure for usability, but instead provide a qualitative discussion.

3.2 Basic Experiment Scenario

MQTT performance tests can be categorized into "fan-in"-driven, "fan-out"-driven, and symmetric tests. Fan-in tests reflect typical IoT applications scenarios with a high number of IoT devices (e.g., 10,000s) acting as publishers, but only a few or a single subscriber (e.g., an analytics application). Fan-out tests are the opposite, e.g., a high number of mobile applications consuming data from few or a single publisher (e.g., weather station). We decided to use a symmetric test scenario with 10 publishers and 10 subscribers, as our goal was to assess quality differences of different brokers in a mostly representative scenario. This also avoids the need to optimize broker queue size configurations. We refer to other scalability tests ([9, 17, 4]) for specific fan-in / fan-out tests.

In our scenario, publishers try to send as many messages as possible to the broker instances and ultimately the subscribers. We tested in a range between 1,000 and 50,000 messages per second, which is higher than many real cases. For example, BMW's connected car platform processes 1,500 messages per second on HiveMQ, while Bose's messaging backend using VerneMQ ran up to 9,700 messages per second [17]. The workload expected for an industrial plant equipped with automation by ABB is within our experimentation range.

We used a fixed message payload size of 150 Bytes with random binary content. While a single telemetry datum (e.g., a temperature value) may be encoded with only 4 Bytes, we assume that messages provide additional meta data (e.g., identification, timestamps, etc.) in a realistic scenario.Payloads of 64 Bytes or 128 Bytes have been used in other benchmarks and a previous work [21] has found that payload sizes up to 4,096 Bytes have limited influence on the maximum sustainable throughput. Batching messages may improve overall throughput, but leads to more complexity on the consumer side, where the batches needs to be de-grouped as part of the application logic.

All publishers and subscribers use MQTT QoS 1 assuring no message loss, but requiring message acknowledgments (i.e., implying an extra network round trip). QoS level 2 would also exclude duplicated messages, but is considered to imply a too high overhead for most IoT scenarios, while QoS level 0 is risky in terms of message loss.

4 Distributed MQTT Brokers

Comprehensive feature comparison tables are available for more than 20 MQTT brokers[1]. There are also MQTT plug-ins available for message brokers originally designed for other protocols, such as RabbitMQ or Apache Kafka. However these plug-ins may be limited in their support of MQTT features. One of the most popular MQTT brokers is Eclipse Mosquitto (implemented in C). It supports MQTT versions 3.1 and 5.0 and has a low footprint, but provides no multi-threading and no native cluster support. AWS IoT and Microsoft Azure IoT provide basic MQTT support, but lack some features [10].

For our evaluation, we selected three representative, native MQTT brokers that provide cluster support and are available as open source (at least in feature-reduced "community-versions"). All of them support the full MQTT version 3.1 and 5.0 protocols, SSL/TLS, and all MQTT QoS levels.

[1] https://en.wikipedia.org/wiki/Comparison_of_MQTT_implementations

EMQX[2]**:** The Erlang/Enterprise/Elastic MQTT Broker (EMQX) started as an open source project in China in 2013. The developers created the company EMQ Technologies Co., Ltd. in 2017 for commercial support and services. The company claims having more than 5,000 enterprise users and customers from various application domains. EMQX is now available in multiple variants, as pure open source broker (1.5M docker pulls), as Enterprise broker, and as private cloud solution. There is also a lightweight variant (15 MB installation) called "EMQ X Edge" for resource-constrained IoT gateways, which may interface with KubeEdge[3]. The open-source variant is available under Apache License 2.0 for all major operating system and processor architectures.

HiveMQ[4]**:** The company dc-square started the development of the commercial MQTT broker HiveMQ in Germany in 2012. dc-square was renamed to HiveMQ in 2019 and created an open-source variant (Community Edition, Apache License 2.0, 0.5M docker pulls). The company claims having more than 130 customers for HiveMQ, among them BMW with a connected car platform and Mattenet with a platform providing the real-time flight status of drones. HiveMQ is implemented in Java and now available as community, professional, and enterprise edition, in addition to an IoT cloud platform variant with hourly subscription fees. The HiveMQ DNS discovery plug-in uses DNS service discovery to add or remove brokers instances to the cluster at runtime.

VerneMQ[5]**:** Octavo Labs AG from Switzerland is developing the VerneMQ MQTT broker since 2015. It is an open-source project (Apache License 2.0, 7.1M docker pulls) with two main developers that started after they had been working on an energy marketplace project. They discovered that AMQP and XMPP did not scale well enough for a large number of devices and started implementing VerneMQ using Erlang/OTP. There are no commercial variants with licensing fees, but the company offers commercial support around VerneMQ. There are several featured customers, among them Microsoft and Volkswagen.

5 Analysis of Distributed MQTT Brokers

5.1 Test Infrastructure

Our testbed is a StarlingX[6] all-in-one duplex bare metal installation running on two identical servers in a redundant, high-available fashion. Each server has a Dual Intel Xeon CPU E5-2640 v3 running at 2.60 GHz with 8x2 physical cores (32 threads), 128 GB of RAM and Gigabit connectivity.

StarlingX v3.0 is an open-source virtualization platform for edge clusters and runs on top of CentOS 7.6. All tested brokers run in Docker CE orchestrated by K8s. Prometheus monitoring tools measure CPU load among other metrics. For

[2] https://www.emqx.io/
[3] https://kubeedge.io/
[4] https://www.hivemq.com/
[5] https://vernemq.com/
[6] https://www.starlingx.io/

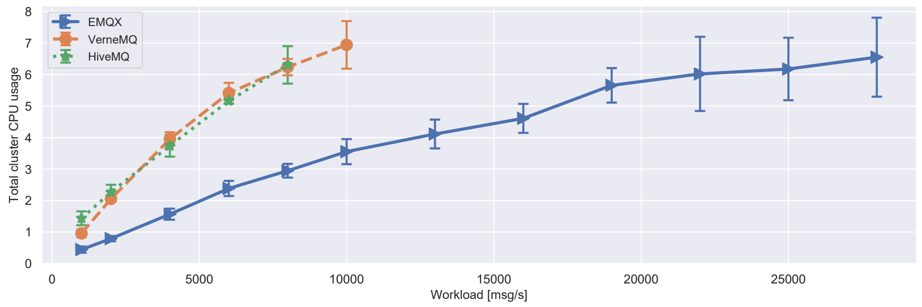

Fig. 2: Stable throughput compared to aggregated CPU usage of all broker pods

the broker installations we used helm charts (VerneMQ 1.10.2, EMQX 4.0.5) or public tutorials from broker vendors (Enterprise HiveMQ 4.3.2 evaluation). In K8s, the brokers use replication controllers (HiveMQ), stateful sets (VerneMQ, EMQX) and load balancer services (metalLB as Level 2 Load Balancer).

A dedicated node in the same Ethernet segment as the StarlingX controllers acts as load driver (CentOS 8.1, Intel Xeon CPU E5-2660 v4 @ 2.00 GHz, 16 cores (32 threads) and 8 GB of RAM). We evaluated different load driver applications, including mqtt-stresser, paho-clients, Locust/MQTT, JMeter/MQTT and MZBench. We decided to use MZBench[7] due to a low resource footprint, convenient Web UI allowing to monitor and export metrics, and also a possibility to define load scenarios in a simple Benchmark Definition Language (BDL). We utilized custom MZBench MQTT workers provided by VerneMQ[8]. We spawn MQTT workers locally in a Docker CE environment on the load driver node.

In addition to metrics from Prometheus and MZBench, we used broker dashboards provided by brokers to validate throughput measurements. A generic graphical MQTT client MQTTExplorer[9] was also used to validate topic lists.

5.2 Performance

To obtain the metrics **M1** (maximum sustainable throughput) and **M2** (average latency), we conducted experiments as described in Section 3.2.

During each experiment run, the publishers first established a defined publishing rate (e.g., 4,000 msg/s), held this for two minutes to assure stability, and then increased the publishing rate (e.g., by 2,000 msg/s) in two minute intervals. To avoid interference with background noise from other processes running on the edge gateway cluster, we configured each broker pod to utilize at most *four* CPU cores. This leaves other cores to execute edge analytics applications or broker dashboards and provides a fair comparison between the brokers.

Fig. 2 shows the aggregated CPU utilization (y-axis) over the messaging rate (x-axis) for the three analyzed brokers. We repeated each experiment three times

[7] https://satori-com.github.io/mzbench/
[8] https://github.com/vernemq/vmq_mzbench
[9] http://mqtt-explorer.com/

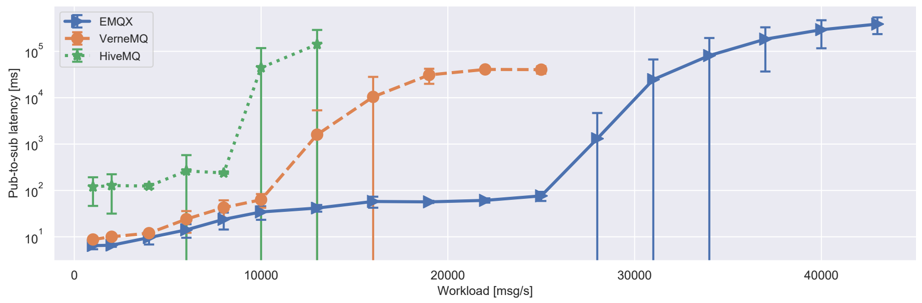

Fig. 3: Stable throughput compared to average publisher-to-subscriber latency

Table 1: Performance Metrics

Performance		EMQX	HiveMQ	VerneMQ
Maximum sustainable throughput (msg/s)	**M1**	28,000	8,000	10,000
Average latency at 1000 msg/s (ms)	**M2**	6.4	119.4	8.7

and report the average utilizations to exclude outliers. 95%-confidence intervals are shown as indication for variations across experiments. The figure shows the CPU utilization curves leveling out at around 4 CPU cores per pod (2 pods per broker). At this point the broker cannot sustainably handle the message load and the message queues run full. We defined an instability point where the average message consumption differs from the published messages by more than 100 msg/s. The plots only include the measurements before reaching this instability point after which, eventually, the message broker starts to drop messages.

Our measurement confirmed that the CPU was the bottleneck in this test scenario. Scenarios with substantially larger message payloads could however run into network bottlenecks, while fan-in and fan-out scenarios could overwhelm the publisher or subscriber queues. In our specific scenario, EMQX managed the highest MST with 28K msg/s, while VerneMQ managed 10K msg/s, and HiveMQ managed 8K msg/s. We confirmed these throughput numbers with independent measurements by MZBench and the respective broker dashboards. It should be noted that each broker allows for much higher message throughput in other scenarios if provided more CPU power (e.g., uncapped CPU assignment, and deployment to more nodes).

Fig. 3 shows the average publisher-to-subscriber latency for the same scenario. Before reaching CPU bottlenecks, the average latencies are below 150 ms for all brokers. At 28K msg/s for EMQX the bottleneck is reached, so that the average latency quickly increases beyond acceptable levels. A similar effect is visible for the other brokers when reaching their CPU bottlenecks.

Our scenario enables a rough performance comparison of the brokers. Table 1 summarizes the GQM metrics. The Erlang-based MQTT brokers outperform HiveMQ, which is implemented in Java. Each broker had equally configured message queues sizes. There may be additional configuration options to tune each broker's performance including broker specific and system-wide parameters.

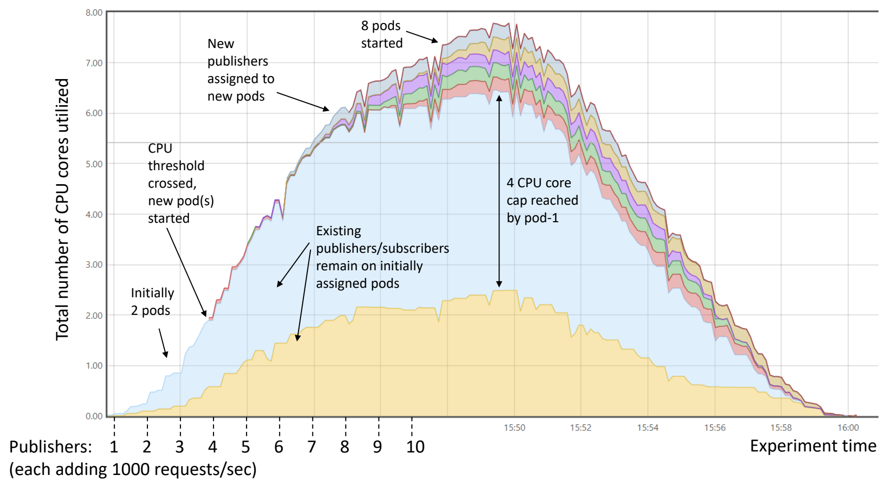

Fig. 4: Kubernetes autoscaling applied to broker pods (VerneMQ)

5.3 Scalability

Our tests showed that all evaluated brokers are multi-threaded and utilize as many CPU cores as available on a given host (tested up to 16 cores). Thus they support vertical scaling with more powerful CPUs. Software architects need to define their expected workload profile in the application scenario and can then perform capacity planning for the required number of nodes. Other authors have conducted MQTT scalability tests with millions of connections in larger clusters (see Section 7), demonstrating theoretically unlimited scalability (metric **M3**).

In an edge gateway cluster also horizontal scaling by creating additional broker instances is possible. We minimally tested horizontal scalability, since our testbed included only two physical servers. We configured K8s auto-scaling for a minimum number of 2 pods (1 per node) and a maximum of 8 pods. A CPU threshold was defined which triggered the instantiation of new pods.

Fig. 4 shows the CPU utilization per pod instance in a stacked line chart over the course of an autocaling experiment with VerneMQ. In the experiment, one new publisher connected every minute and added a publishing rate of 1000 msg/s to the overall publishing rate. At the peak, the experiment had 10 publishers with a total of 10,000 msg/s, and 10 subscribers consuming each message. Initially, 10 subscribers and 1 publisher are assigned by the load balancer to two active pod instances. Once three publishers have connected to the broker, the pre-defined CPU utilization threshold is crossed and K8s starts new pods.

We also observe that the load balancer assigns new connection requests to the newly started pods, but existing connections are not shifted between pods. Thus, the autoscaling is only effective if there are new connections. For a constant number of connections but a higher messaging rate, the cluster cannot benefit from autoscaling without disconnecting clients.

Table 2: Scalability Metrics

Scalability		EMQX	HiveMQ	VerneMQ
Maximum number of connections	**M3**	unlimited (tests up to 50 mio)	unlimited (tests up to 10 mio)	unlimited (tests up to 5 mio)
Time to start new broker instance (s)	**M4**	18.6	20.2	14.2
Container image sized (MB)		89.2	298.6	82.5

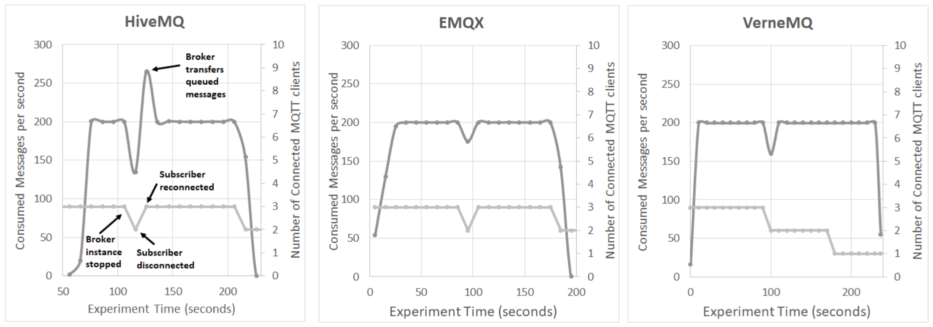

Fig. 5: Consumption rate and number of connections during resilience test

Metric **M4** is the start-up time of new broker pods, since load peaks below this time can only be handled by vertical scaling. We measured the duration of the transition between the "PodSheduled" and the "Ready" condition of the pod. Table 2 shows the average time of ten pod starts, excluding the time of downloading the container image when it is run on the node for the first time.

5.4 Availability/Resilience

We assessed resilience by modifying the scenario described in Section 3.2 to avoid high queue lengths. The modified scenario contains one publisher (100 msg/s), two subscribers, and two broker pods (B_1, B_2). B_1 had the publisher P and subscriber S_1 connected. B_2 had subscriber S_2 connected. Both subscribers consume all messages (total consumption rate of 200 msg/s). Furthermore, the scenario used QoS 1 and persistent sessions. We configured the broker queues to an in-flight message queue of 1,000 and on-/offline message queue of 50,000.

After a stabilization phase of each experiment, we forcefully stopped the broker process on B_2, i.e., Java VM or Erlang BEAM VM to simulate a crash. Subscriber S_2 was expected to reconnect to B_1 via the load balancer immediately, i.e., before B_2 pod is restarted by K8s. Furthermore, S_2 is supposed to resume its session, and receive messages that the broker buffered during the disconnect.

Fig. 5 shows that HiveMQ exhibits the expected behavior, resulting in zero message loss. The dark lines show the consumption rate of 200 msg/s that is temporarily disturbed due the subscriber disconnect upon broker pod B_2 stopping. We observe an immediate re-connection of S_2 and a temporary consumption rate *above* 200 msg/s for queued messages. Both EMQX and VerneMQ performed unexpectedly: the temporary decrease of the consumption rate is not equalized by a later increase over 200 msg/s. For VerneMQ also note the number of clients

Table 3: Average Message Loss during Resilience Test

Resilience		EMQX	HiveMQ	VerneMQ
Average messages loss in reference scenario	M5	580	0	82

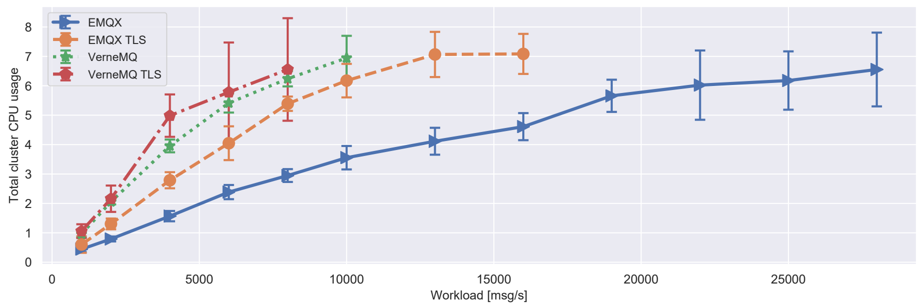

Fig. 6: CPU utilization with and without TLS encryption

constantly decreasing. We repeated each experiment three times to exclude temporary distortions but arrived at the same result summarized in Table 3. This unexpected behavior requires further investigations in future work.

5.5 Security

MQTT security can be tackled at the network level (e.g., using VPN), the transport layer (e.g., using TLS) and the application layer (e.g., authentication and authorization). In the following, we focus on the security at the transport layer.

We conducted tests using TLS encryption to measure CPU and bandwidth overhead. We configured each broker to use TLS v1.2, where encryption was terminated directly at the broker instance. Please note, we were not able to connect MZBench MQTT workers to HiveMQ due to reported SSL errors. Tests with other MQTT clients, e.g., MQTTExplorer, worked fine.

Fig. 6 shows the impact of TLS encryption on the CPU utilization for one representative broker (EMQX). The CPU utilization levels out at the cap of four CPU cores already at 16,000 msg/s when using TLS, compared to 28,000 msg/s without TLS. Installing certificates on the broker was similar between all the brokers and can be performed, e.g., by using K8s secrets mounted into the pod. An overview of additional security features of brokers can be found in Table 4.

5.6 Extensibility

All brokers offer plug-in mechanisms for developing extensions to the basic broker functionality. For example, plug-ins allow special authentication mechanisms or integration with monitoring frameworks.

VerneMQ provides hooks for changing protocol flow, events, and conditional events. Developers can write plug-ins in Erlang, Elixir, or Lua and load them during runtime. VerneMQ also provides webhooks, where a VerneMQ plugin

Table 4: Security Metrics

Security		**EMQX**	**HiveMQ**	**VerneMQ**
Authentication/authorization		files, database	files, database, OAuth, LDAP	files, database
Certificate-based authentication		yes	yes	yes
TLS version support		v1.1, v1.2	v1.1, v1.2, v1.3	v1.1, v1.2
Maximum sustainable throughput (TLS off)		28,000	8,000	10,000
Maximum sustainable throughput (TLS on)		16,000	?	8,000
Overhead of enabled TLS on MST	**M6**	43%	n/a	20%

dispatches an HTTP post request to a registered endpoint. This mechanism allows implementing extensions in any programming language.

HiveMQ plug-ins are Java JAR files and shall be integrated using dependency injection (using Google Guice). HiveMQ provides more than 30 callback types besides services to interact with the HiveMQ core (e.g., publish services to send new messages to clients). There is also a "RestService", which allows to create a REST API to be consumed by other applications. The HiveMQ marketplace provides a few open source plug-ins (e.g., Prometheus monitoring) and commercial plug-ins (e.g., HiveMQ for Kafka).

EMQX can also be extended with Erlang code, 25 plug-ins are already available from the vendor (e.g,. web dashboard, rule engine, Lua hooks, STOMP support). Plug-ins can be loaded at runtime, and there are also webhooks available. EMQ provides 15 hooks, chaining plug-ins on these hooks is possible.

In summary, the extensibility of all brokers is deemed good. HiveMQ has the most extensive developer guides and the most hooks, while being geared towards Java development. VerneMQ and EMQX may have more active communities due to their longer open source history, offer fewer hooks, and are geared towards Erlang development.

5.7 Usability

The installation of all brokers is smooth, which allows software architects to quickly perform experiments with their intended workloads configured in a load driver. All of them offer Docker containers, VerneMQ and EMQX provide helm charts for K8s. EMQX and HiveMQ are available as Amazon Machine Images. Users can configure the brokers via files and environment variables. All brokers provide a web-based dashboard for monitoring and troubleshooting, where connected clients and performance metrics can be investigated. The dashboards of HiveMQ and EMQX offer the most information. All brokers have command line interfaces. HiveMQ has the most comprehensive documentation and developer guides, including several MQTT tutorials, although the documentation of the other brokers is also good.

6 Architecture Decision Guidance

Fig. 7 shows a preliminary problem space modelled with ADMentor[10] and intended as architect decision guidance. In an enterprise IoT scenario, software architects first **(1)** need to decide whether the MQTT protocol is appropriate. This choice is beyond the scope of our paper (see [16, 21, 18]). To decide for a clustered broker **(2)**, a detailed specification of the expected workload profile should be created. This includes the number of publishers, subscribers, payload sizes, topics, expected QoS levels, publication/subscription rates,etc. Non-trivial scenarios likely benefit from a cluster.

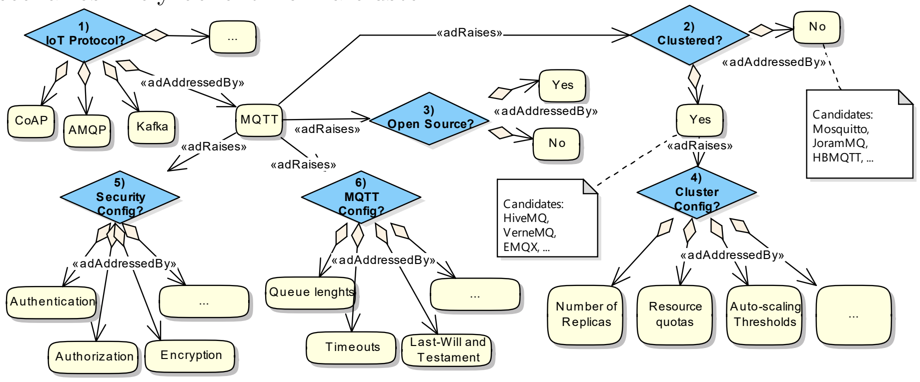

Fig. 7: Architecture Decision Guidance: IoT/MQTT Problem Space

A potentially business-driven choice **(3)** is the selection of an open-source or commercial MQTT broker, which may largely limit the available alternatives. Here, it needs to be traded-off whether licensing fees, commercial support, and advanced features are well invested compared to own development efforts and community support.

In a containerized edge cluster, the software architect may decide **(4)** on the number of required pod replicas, resource quotas, and auto-scaling parameters (if needed at all). Another decision point is the security configuration **(5)**. The MQTT configuration itself is a set of fine-granular decisions based the expected workload profile **(6)**.

The design of the MQTT topic space may also be in control of the software architect and there are guides available with best practices for topic spaces[11]. Here, the software architect does not have discrete options.

We found that the comparably easy installation of the brokers and the availability of powerful MQTT load drivers allow software architects to quickly evaluate MQTT brokers for a given application scenario. If the target deployment hardware is already available for testing, we recommend specifying the expected workload for a given load driver (e.g., in the simple Benchmark Definition Language of MZBench[12]) and then quickly running a few experiments to get a feeling on the performance and availability to expect. This exercise has been reported by

[10] https://github.com/IFS-HSR/ADMentor
[11] https://pi3g.com/2019/05/29/mqtt-topic-tree-design-best-practices-tips-examples/
[12] https://satori-com.github.io/mzbench/scenarios/tutorial/

others [17, 4] and also allows familiarizing with the usability and documentation of the brokers, which then supports making a final decision.

7 Related Work

A broad survey of IoT technologies, among them MQTT, was provided by Al-Fuquaha et al. [1]. Several books describe the protocol, applications, and usage scenarios in detail [13, 14, 8]. Naik [16] discusses criteria for selecting messaging protocols, such as MQTT, CoAP, AMQP, and HTTP. Several authors compared MQTT and CoAP [5, 22, 12].

Sommer et al. [21] specifically investigated MOM for industrial production systems, aiming at architectural decision support. They conducted performance tests with Mosquitto, RabbitMQ, Kafka, and JeroMQ and found the MST for Mosquitto with different payload sizes at around 1000 msg/s on an Intel i7 Windows-PC. These tests did not involve clustered brokers. Mishra [15] compared the throughput and latency of Mosquitto, BevyWiseMQTT, and HiveMQ in a small-scale, single broker instance scenario on a Raspberry Pi, but found little performance differences. Bertrand-Martinez et al. [2] qualitatively evaluated different MQTT brokers according to ISO 25010 quality criteria, among them EMQX and Mosquitto. In this scoring, Mosquitto received the highest rank due to simplicity and lightweightness.

There are also practitioner reports of performance tests with specific MQTT brokers: Mahony et al. [17] set up a Kubernetes cluster on AWS and deployed VerneMQ in up to 80 nodes to open 5 million messaging connections and more than 9500 msg/s (measured with Locust). The company Hotstar [4] evaluated the open source MQTT brokers VerneMQ and EMQX for distributing a social feed to mobile applications. They set the brokers up on up to 5 AWS extra-large node instances, ran performance tests with MZBench, and reached up to 50 million connections with EMQX. The HiveMQ team [9] demonstrated up to 10 million connections to HiveMQ deployed to 40 AWS EC2 instances.

Most broker vendors provide whitepapers on performance tests with their own brokers. HiveMQ conducted performance tests on AWS including fan-in and fan-out scenarios. For example, in a fan-in scenario with QoS1 they achieved up to 60K msg/s on an 8-core CPU. ScaleAgent [20] compared JoramMQ, Apollo, Mosquitto, and RabbitMQ at up to 44K msg/s and concluded that their JoramMQ broker performed best. HiveMQ provides several customer case studies on their website, for example BMW's connected car scenario with 1500 msg/s or a scenario involving 1000 connected air quality sensors with 1100 msg/s.

There are more general works related to our study: The SPECjms2007 benchmark [19] provided an agreed workload to test messaging systems (supermarket chain scenario), but has been retired as of 2016. Thean et al. [23] shows Mosquitto running in Docker Swarm. Architecture decision guidance models have been proposed for example for SOA [25], cloud computing [26], and microservices [7].

8 Conclusions

This paper analyzed distributed MQTT brokers deployed to an edge gateway cluster. We found that EQMX showed the highest throughput with 28K msg/s, while VerneMQ managed 10K msg/s and HiveMQ managed 8K msg/s, respectively. The test scenario was intentionally limited to a maximum of eight CPU cores). We found that the scalability of the brokers is potentially unlimited, since they are multi-threaded and can be horizontally scaled. Only HiveMQ managed our test scenario without message loss. All brokers have similar security features and offer extensions in any programming language using webhooks.

Our paper provides decision guidance for software architects in enterprise IoT scenarios. They can use the results in our paper as an orientation and quickly set up their own experiments using the tools referenced in the paper. Researchers can derive reference enterprise IoT scenarios from our paper, conduct additional tests, and build constructive models for IoT messaging.

As a next step, we intend to deepen our analysis with additional metrics and scenarios and broaden it by integrating additional messaging solutions. In addition to StarlingX, a complementary evaluation on more resource-constrained edge gateways is warranted. It is conceivable to construct predictive performance models for quick forecasting and to work an automated experiment generator as a software service utilizing cloud computing resources.

References

1. Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., Ayyash, M.: Internet of things: A survey on enabling technologies, protocols, and applications. IEEE communications surveys & tutorials **17**(4), 2347–2376 (2015)
2. Bertrand-Martínez, E., Feio, P., Nascimento, V., Pinheiro, B., Abelém, A.: A methodology for classification and evaluation of iot brokers. In: 9th Latin American Network Operations and Management Symposium, LANOMS. IFIP (2019)
3. Caldiera, V.R.B.G., Rombach, H.D.: The goal question metric approach. Encyclopedia of software engineering pp. 528–532 (1994)
4. Chaudhari, M., Gupta, P.: Building pubsub for 50m concurrent socket connections. https://blog.hotstar.com/building-pubsub-for-50m-concurrent-socket-connections-5506e3c3dabf (Jun 2019)
5. De Caro, N., Colitti, W., Steenhaut, K., Mangino, G., Reali, G.: Comparison of two lightweight protocols for smartphone-based sensing. In: Symposium on Communications and Vehicular Technology in the Benelux (SCVT). pp. 1–6. IEEE (2013)
6. Fortune Business Insights: Internet-of-things market research report. https://www.fortunebusinessinsights.com/industry-reports/internet-of-things-iot-market-100307 (Jul 2019)
7. Haselböck, S., Weinreich, R.: Decision guidance models for microservice monitoring. In: International Conference on Software Architecture Workshops (ICSAW). pp. 54–61. IEEE (2017)
8. Hillar, G.C.: MQTT Essentials-A lightweight IoT protocol. Packt Publishing Ltd (2017)

9. HiveMQ-Team: 10,000,000 mqtt clients: Hivemq cluster benchmark paper. https://www.hivemq.com/benchmark-10-million/ (Oct 2017)
10. HiveMQ-Team: Comparison of mqtt support by iot cloud platforms. https://www.hivemq.com/blog/hivemq-cloud-vs-aws-iot/ (May 2020)
11. Hohpe, G., Woolf, B.: Enterprise integration patterns: Designing, building, and deploying messaging solutions. Addison-Wesley Professional (2004)
12. Iglesias-Urkia, M., Orive, A., Barcelo, M., Moran, A., Bilbao, J., Urbieta, A.: Towards a lightweight protocol for industry 4.0: An implementation based benchmark. In: International Workshop of Electronics, Control, Measurement, Signals and their Application to Mechatronics (ECMSM). pp. 1–6. IEEE (2017)
13. Lampkin, V., Leong, W.T., Olivera, L., Rawat, S., Subrahmanyam, N., Xiang, R., Kallas, G., Krishna, N., Fassmann, S., Keen, M., et al.: Building smarter planet solutions with mqtt and ibm websphere mq telemetry. IBM Redbooks (2012)
14. Mesnil, J.: Mobile and Web Messaging: Messaging Protocols for Web and Mobile Devices. " O'Reilly Media, Inc." (2014)
15. Mishra, B.: Performance evaluation of mqtt broker servers. In: International Conference on Computational Science and Its Applications. pp. 599–609. Springer (2018)
16. Naik, N.: Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http. In: International systems engineering symposium (ISSE). pp. 1–7. IEEE (2017)
17. O'Mahony, D., Doyle, D.: Reaching 5 million messaging connections: Our journey with kubernetes. https://www.slideshare.net/ConnectedMarketing/reaching-5-million-messaging-connections-our-journey-with-kubernetes-126143229 (Dec 2018)
18. Profanter, S., Tekat, A., Dorofeev, K., Rickert, M., Knoll, A.: Opc ua versus ros, dds, and mqtt: performance evaluation of industry 4.0 protocols. In: IEEE International Conference on Industrial Technology (ICIT) (2019)
19. Sachs, K., Kounev, S., Bacon, J., Buchmann, A.: Performance evaluation of message-oriented middleware using the specjms2007 benchmark. Performance Evaluation **66**(8), 410–434 (2009)
20. ScaleAgent: Benchmark of MQTT servers. https://bit.ly/2WsTw0Z (Jan 2015)
21. Sommer, P., Schellroth, F., Fischer, M., Schlechtendahl, J.: Message-oriented middleware for industrial production systems. In: International Conference on Automation Science and Engineering (CASE). pp. 1217–1223. IEEE (2018)
22. Thangavel, D., Ma, X., Valera, A., Tan, H.X., Tan, C.K.Y.: Performance evaluation of mqtt and coap via a common middleware. In: International conference on intelligent sensors, sensor networks and information processing (ISSNIP). pp. 1–6. IEEE (2014)
23. Thean, Z.Y., Yap, V.V., Teh, P.C.: Container-based mqtt broker cluster for edge computing. In: International Conference and Workshops on Recent Advances and Innovations in Engineering (ICRAIE). pp. 1–6. IEEE (2019)
24. Tran, P., Greenfield, P., Gorton, I.: Behavior and performance of message-oriented middleware systems. In: International Conference on Distributed Computing Systems Workshops. pp. 645–650. IEEE (2002)
25. Zimmermann, O., Grundler, J., Tai, S., Leymann, F.: Architectural decisions and patterns for transactional workflows in soa. In: International Conference on Service-Oriented Computing. pp. 81–93. Springer (2007)
26. Zimmermann, O., Wegmann, L., Koziolek, H., Goldschmidt, T.: Architectural decision guidance across projects-problem space modeling, decision backlog management and cloud computing knowledge. In: Working IEEE/IFIP Conference on Software Architecture. pp. 85–94. IEEE (2015)