Towards Resilient IoT Messaging: An Experience Report Analyzing MQTT Brokers

Sten Gruener
ABB Corporate Research
Ladenburg, Germany
sten.gruener@de.abb.com

Heiko Koziolek
ABB Corporate Research
Ladenburg, Germany
heiko.koziolek@de.abb.com

Julius Rückert
ABB Corporate Research
Ladenburg, Germany
julius.rueckert@de.abb.com

***Abstract*—Many Internet-of-Things (IoT) applications for smart homes, connected factories, or car-to-car communication utilize broker-based publish/subscribe communication protocols, such as the MQTT protocol. Commercial IoT applications have high reliability requirements for messaging, as lost messages due to unstable Internet connections or node failures can harm devices or even human beings. MQTT brokers implement numerous architectural availability tactics, but former analyses of MQTT communication have mainly focused on performance measurements under stable conditions. We have created the MAYHEM resilience testing tool for MQTT brokers and applied it in various resilience experiments on different MQTT brokers (VerneMQ, Mosquitto, HiveMQ, EMQ X). We found that MQTT QoS level 0 is already robust against minor packet loss, that selected broker message persistency solutions can lead to lost messages, and that most clustered MQTT brokers favor availability and performance over communication integrity. The results can support IoT practitioners in architectural decisions and researchers as well as broker vendors in optimizing designs and implementations.**

***Index Terms*—Software Architecture, Message Brokers, Message-oriented Middleware, Availability, Resilience, Docker, Kubernetes**

I. Introduction

The market for IoT applications and devices is growing fast, with dozens of billion devices expected to be connected to the Internet in the next few years [1]. IoT devices support for example smart home scenarios to automate heating and lighting, smart cities to optimize traffic, factories and production plants to minimize failures and optimize production and smart grids to cope with fluctuating renewable energy production [2]. IoT applications require lean and robust communication protocols, such as MQTT, COAP, or OPC UA. MQTT is a broker-based publish/subscribe protocol used in many IoT applications [3]. MQTT messages may for example include device telemetry data, status information, and configuration data.

Many commercial IoT applications require safe and reliable message transfers, since lost messages can lead to catastrophic failures. However, due to their distributed nature, IoT systems have many potential sources of failure. Individual devices may run out of battery power, Internet connections may be unstable, and backend applications executed in cloud computing environments may suffer from node failures or competing workloads. MQTT brokers prepare for these failure sources with capabilities for message retransmission, persistency, and cluster replication. Whether these capabilities work as expected in a given application scenario is largely unknown. There are no known investigations on the resilience of clustered MQTT brokers.

Existing analyses of MQTT brokers often focus on performance evaluation in failure-free test environments (e.g., [4]–[10]). Several works have tested the resilience behavior of selected MQTT brokers, but did not included clustered MQTT brokers (e.g., [11]–[13]). Other works proposed novel clustered MQTT brokers and tested their behavior (e.g., [14]–[16]). However, there are no systematic investigations in literature analyzing the resilience behavior of clustered commercial and open-source MQTT brokers. Developers with high dependability requirements need to test the available brokers and carry out own resilience experiments.

The contributions of this paper are experiences from testing different resilience variants of MQTT brokers under failure conditions. We have designed and implemented the MAYHEM MQTT workload driver and failure simulator to carry out experiments provoking message loss. MAYHEM was used to test the MQTT brokers Mosquitto, EMQX, HiveMQ, and VerneMQ running as software containers in a high availability Kubernetes cluster, which is representative for commercial IoT edge gateway clusters. We were able to detect message loss in all described resilience variants, although broker persistency and cluster replication of selected brokers can successfully be used to prevent message loss. We report on differences detected from the multiple brokers and discuss the tradeoffs in the different variants.

The remainder of this paper is structured as follows. Section 2 describes the architecture for the messaging scenarios as well as potential failure sources in more detail. Section 3 introduced four practical resilience variants that are rooted in classical availability tactics and implemented by the available brokers. Section 4 details our testing setup and the MAYHEM workload driver. Section 5 reports experiment results on message loss and other properties for the different resilience variants. Section 6 summarizes lessons learned and Section 7 surveys related work.

II. Broker Communication Architecture

Fig. 1 shows a generic, clustered MQTT broker architecture common for IoT applications. **Publisher** clients may

be IoT devices that send MQTT messages (e.g., telemetry data, status information) to MQTT broker instances via the MQTT protocol. They may queue unacknowledged messages for re-sending. Small IoT applications in a smart home may have dozens of publishers, while large IoT applications in a smart grid or smart factory may comprise 100,000s of publishers. Fleets of connected cars or smart appliances may send 1000s of messages per second over the Internet to the broker instances. There are many libraries to create MQTT clients (e.g., Eclipse Paho, MQTT-C, wolfMQTT).

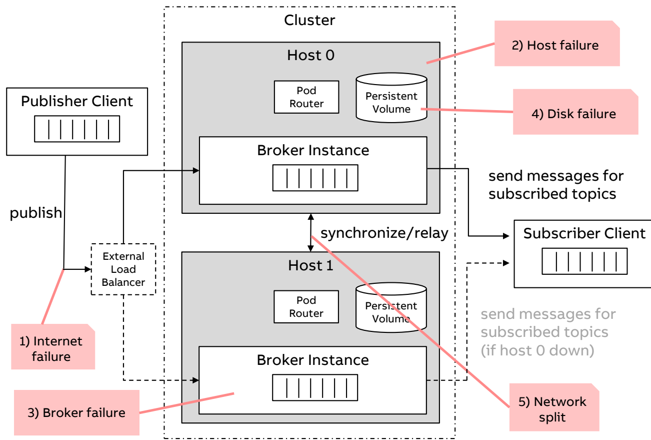

Fig. 1. Broker-based publish/subscribe architecture in a cluster (conceptual architecture). Brokers need to cope with different failure sources.

Broker instances decouple publishers from subscribers and relay messages between them (n-to-m communication). Brokers group messages into topics to which clients can subscribe. They manage live message queues for topics and can persist session data to allow for consistency after broker restarts. Popular MQTT broker implementations are Mosquitto, HiveMQ, VerneMQ, SwiftMQ, and EMQ X. Several AMQP brokers provide MQTT plugins to serve as MQTT brokers, but often they only offer limited support of the MQTT specification.

In larger enterprise scenarios, multiple MQTT broker instances may be deployed within a **cluster** of computing nodes, to allow for high scalability and availability supported by virtualization and containerization. External load balancers (e.g., MetalLB, HAProxy, or specialized solutions offered by a cloud provider) and pod routers (e.g., kube-proxy, Cilium) distribute MQTT client sessions to specific broker instances running as software container in the cluster. During normal operation (no failures), sessions are routed consistently in these two steps, so that traffic for a specific client session is directed to the same broker instance. In the setup depicted in Fig. 1, a client might be first directed to host 0 by the external load balancer and from there to a broker instance on host 1. Broker instances may relay MQTT messages for topics to other broker instances to serve additional subscribers of these topics.

Subscribers connect to MQTT brokers and register themselves for updates to specific topics. A client can request a persistent session, in which case the broker stores all subscriptions and unconfirmed messages for a client, even if the client is offline. This makes communication across unreliable Internet connections more robust and allows the client to access the information upon reconnection. Typical subscribers are other IoT devices, mobile application, cloud applications, dashboards, analytical applications, or time-series databases.

Critical enterprise IoT applications may have demanding **dependability requirements**. A client shall always be able to connect to a running broker instance (i.e., availability). A published message shall not be lost before being sent to the subscribers (i.e., reliability). Subscribers shall receive messages in the same order as they have been published and not receive duplicates (i.e., consistency). However, many applications only require a subset of these quality attributes and IoT applications are typically not safety-critical.

The architecture sketched in Fig. 1 includes several potential **sources of failure**, which may complicate fulfilling dependability requirements:

1) Internet failures: the connection between clients and the broker may be unreliable, since routing paths might change over time, middleboxes interfere with the traffic, or high network loads along the routing path throughout the Internet. IoT devices may be mobile and rely on wireless communication, therefore experience packet loss or connection drops.
2) Host failures: broker instances often execute in parallel to other workloads and may suffer from other crashing applications on the same node or host hardware failures. Operating systems may crash or fail due to other reasons.
3) Broker failures: bugs in broker implementations and overloads due to denial of service attacks may make brokers unresponsive and require restarts.
4) Disk failures: all data persisted to disk is subject to potential disk failures, which is typically addressed with redundant or distributed storage solutions.
5) Network splits: failing network equipment may lead to network partitions in clusters.

III. Practical Resilience Variants

This section provides resilience variants for demanding dependability requirements in a setting as sketched in Fig.1. Deeper performance, security, maintainability, extensibility, and usability investigations are out-of-scope for this paper and handled in other papers [17]. However, the fulfillment of dependability requirements always involves trade-offs with other quality attributes, which will be discussed in the following.

A. Variant #0: QoS 0 and TCP Retransmissions

By default single, non-clustered MQTT brokers as well as clients maintain QoS 0 MQTT sessions. QoS 0 provides an *at most once* delivery semantic, where a publisher sends a message to the broker and forgets about it as soon as it has transferred it to the network stack. Using TCP, the network stack ensures a transmission of the message, as it issues retransmissions after a time-out. However, no other guarantees for the delivery are provided. In case the broker crashes or

the network stack fails after receiving the message but before delivering it to the broker application, the message is lost.

For QoS 0 MQTT subscriber sessions, the delivery of messages to the subscribers is not guaranteed but already protected from minor problems such as packet loss due to the use of a reliable transport protocol. This variant is simple and fast in terms of MQTT mechanisms as no application-layer acknowledgments and message buffering are required.

B. Variant #1: QoS 1 with Publication Acknowledgments

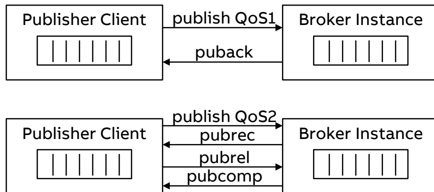

Fig. 2. Variant #1: QoS 1 (two-way handshake) and QoS 2 (four-way handshake) to handle unstable Internet connections on the application layer.

MQTT provides levels QoS 1 (at least once) and QoS 2 (exactly once) to deal with unstable network connections and overload scenarios (Fig. 2). If a publisher sends a QoS 1 message, the broker acknowledges the successful reception with a PUBACK message. The publisher queues sent messages until receiving corresponding PUBACKs and re-sends them after a timeout. The same mechanism is used between broker and subscribers, however there is no end-to-end acknowledgement from publisher to subscriber to keep them decoupled. The re-sending can lead to duplicated messages.

QoS 2 messages require two request/response flows between sender and receiver (four-way handshake). On message arrival, the receiver sends a PUBREC message to the sender. The sender answers with a PUBREL message that is finally acknowledged with a PUBCOMP message. Only after this last message, it is ensured that the data message was transferred successfully and exactly once. This is the most safe way of messaging but implies a latency penalty due to the additional network round-trips. For many applications, QoS 1 guarantees are sufficient as duplicated messages can be tolerated.

This variant is a realization of the "transaction" availability tactic and only requires proper configuration but no additional hardware. Besides unstable Internet connections, this variant can potentially also handle short broker outages (e.g., restarts or short overloads). However, if a broker fails after sending a PUBACK, but before delivering the message to subscribers, messages can get lost. Configuring persistent sessions instructs the broker to queue QoS 1/2 messages that have not yet been delivered to clients, for example if these clients are temporarily offline. This does not automatically imply that the session data is stored to disk, however brokers such as VerneMQ and HiveMQ by default activate disk persistence in case of QoS 1/2 messages, which carries an additional performance penalty besides the network round-trips.

C. Variant #2: Single-Instance Broker / QoS 1 / Persistence

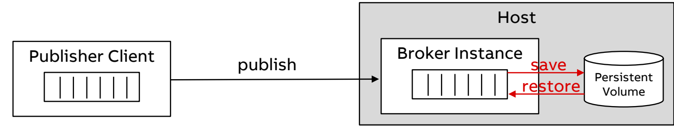

Fig. 3. Variant #2: Brokers can store messages to disk to survive host failures. Distributed storage mechanisms in a cluster can further increase availability and shorten outage periods.

In this variant (Fig. 3), a single, non-clustered MQTT broker relies on a file-system persistence mechanism. It stores the clients' session data including subscribed topics, unacknowledged QoS 1 messages etc. When operating as a container, the file-system persistence of the MQTT broker may be mounted into the container context and synchronized by means of replication on the layer of file storage, which is transparent for the container orchestration system.

While persistent storage addresses broker failures and host shutdowns/reboots, additional failure scenarios like disk-failure or even host failure require a redundancy mechanisms for the persistent volume. This may be achieved by mounting a network attached storage into the container context or having a more sophisticating replication technique for persistent volumes, e.g., distributed storage replication (e.g., ceph[1]). In K8s persistent storage is mapped through the concept of Persistence Volume Claims (PVCs) which we used to map file-based persistent storage of the brokers during experiments.

Advantages of this variant are the usage of file-system replication features of brokers which is widely supported, allowing this variant also for brokers that are not cluster-capable by themselves. Disadvantages are a downtime for switch-over in case of a host failure and also an additional complexity of storage synchronization techniques.

D. Variant #3: Clustered Broker without Queue Mirroring

For this variant, multiple broker instances run in a cluster as depicted in Fig. 4. The simplest cluster setup (this variant) is used to scale broker resources and increase service availability by running individual broker instances on separate hardware nodes. Broker instances coordinate with each other on topics of their individual clients and establish a message forwarding for shared topics. As a result, publishers and subscribers can be connected to different broker instances of the cluster. To the clients, the cluster appears like a single broker. When deployed in a container orchestration environment, such as Kubernetes (K8s), the cluster can be exposed as service to clients using a single service IP address.

Resilience and availability consideration: This variant optimizes for availability. As long as at least one broker instance is available, clients can connect to the cluster, e.g. via a common service IP address. Also, in case of broker instance failures, clients can quickly reconnect to another instance, without having to wait for restart of a broker instance or host. The

[1]https://ceph.io/

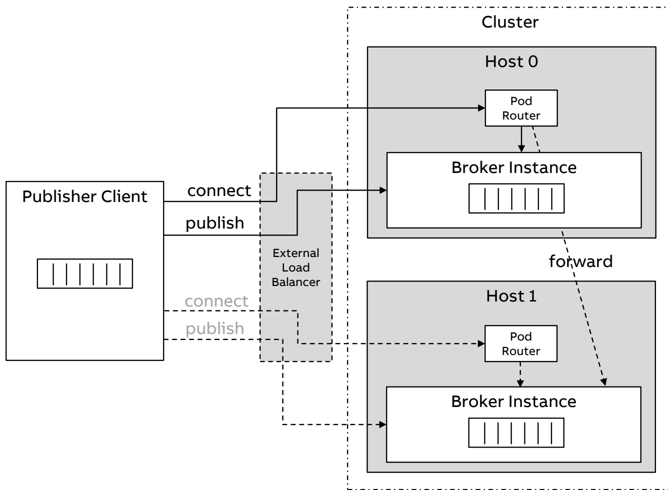

Fig. 4. Variant #3: Cluster without Queue Mirroring: multiple broker instances receive and send messages, but do not fully synchronize sessions among each other. Messages are only forwarded between instances if required for an already connected subscriber.

dynamic dispatching from service IP address to instances is done on (re-)connect by a load balancing mechanism.

This variant does not guarantee communication integrity, i.e., QoS 1/2 MQTT messages may get lost in case of broker instance failures. This can happen if a broker instance acknowledges a received message but fails before delivering it to all subscribers or forwarding it to other broker instances. A solution to this problem is available with more sophisticated cluster variants using queue mirroring as described next.

Examples for cluster variants without queue mirroring are VerneMQ[2] and the open-source version of EMQ-X[3]. By avoiding queue mirroring, this variant is optimized for performance and low delays as it avoids an additional coordination to achieve integrity across broker instances. This also reduces the network traffic inside the cluster.

This variant is a realization of the "Cold Spare" availability tactic: available broker instance act as cold spare instances that can immediately take over but do not carry the session state of the failed broker. However, as the instances are already running, they do not suffer from the typical long start-up times of cold spares. Thus, this variant can be already considered optimal regarding availability, but not regarding integrity.

Load balancing considerations: To effectively use cluster resources, load balancing mechanisms are applied to distribute client MQTT sessions to broker instances. In addition, load balancing is important in case of broker instance failures, where clients need to be directed to one of the remaining broker instances. A number of load-balancing mechanisms and implementations exist that can be used to address these needs that are distinguished in external and internal mechanisms.

In a cloud environment, external load balancing is a typical service provided by the cloud platform provider. External load balancing might be implemented using existing routing infrastructure (e.g. using BGP and equal-cost multipath (ECMP) routing[4]), specialized load-balancing hardware appliances, software services (e.g. HAProxy[5]) running as dedicated service, or simple software-based solutions (e.g. MetalLB[6]).

The purpose of the external load balancer is to redirect network traffic of clients to hosts of the cluster. At the host, an internal load balancing by a pod router might happen to forward and distribute client sessions to service instances. This can be essential because the external load balancer typically does a simple equal-share distribution of sessions, while the internal load balancer can take more fine-grained decisions depending on cluster load and actually running service instances.

In case of K8s, the internal load balancing is done by kube-proxy, which randomly assigns clients to available broker instances. Kube-proxy establishes client-specific iptables rules for the traffic forwarding, which remain in place throughout the client session. In case an assigned broker instance fails, kube-proxy takes a new forwarding decision to an alternative broker instance. In case the entire forwarding host fails, all TCP sessions forwarded by the host fail, not only those to broker instances on this host.

E. Variant #4: Clustered Broker with Queue Mirroring

This variant extends the simple cluster implementation in that it implements a queue mirroring concept as depicted in Fig. 5. Out of the brokers we analyzed only HiveMQ[7] supported this feature.

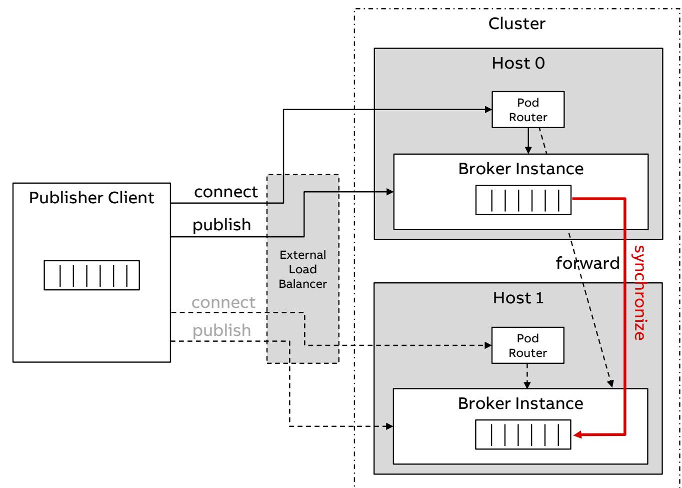

Fig. 5. Variant #4: Cluster with Queue Mirroring: brokers replicate messages to other instances in the cluster before acknowledging them to the publishers.

In addition to the special forwarding of messages of topics to other broker instances with connected subscribers to these topics and the cold standby failover provided by the simple cluster variant, here, broker instances also ensure that received

[2]https://vernemq.com/
[3]https://www.emqx.io/
[4]https://tools.ietf.org/id/draft-lapukhov-bgp-ecmp-considerations-05.txt
[5]https://www.haproxy.com/
[6]https://metallb.universe.tf/
[7]https://www.hivemq.com/

QoS 1/2 messages have been successfully replicated before acknowledging the delivery with a PUBACK message to a publisher.

Replication can cover one or more instances, depending on the communication integrity requirements and broker configuration. In addition to the messages, also other session data of clients may be replicated. As a result, clients can reconnect to the cluster after a broker instance failure and immediately pick up their session. In this setup, integrity of QoS 1/2 messages is guaranteed as long as at least one replica of the session and queue survives in a failure scenario.

As it cannot be ensured that clients connect to a broker instance that actually maintains a replica of their session and queue, additional mechanisms are implemented to migrate sessions to the right instance on client reconnects. One key implication of this variant is that the message throughput for QoS 1/2 messages is reduced, especially if the number of in-flight messages are limited. Before a message is acknowledged, the broker instance needs to coordinate with other instances and replicate the message, which decreases the messages that can be processed by the cluster in a given time.

IV. Testing Setup

MAYHEM (MQTT AvailabilitY HEdging Module) is written in Python 3.9 and uses Eclipse Paho 1.5.0 libraries for MQTT communication. The main components of MAYHEM are publisher and subscriber instances which are connected to the MQTT broker under test.

The generic test setup is simple: an instance of the publisher publishes incrementally numerated messages to a single topic. Multiple subscriber instances receive the messages from the brokers due to a subscription for the topic. All MQTT clients log their sent or received messages to log files which are evaluated according to the following consistency criteria:

- Missing messages, i.e., messages that have been published, but not received by subscribers.
- Duplicated messages, i.e., messages which are received by subscriber instances multiple times.
- Out of order messages, i.e., enumerated messages that are received out of sequence by subscribers. Note that duplicates are not counted as out of order messages if their number are lower than the highest message number received by respective subscriber since those ordering issues can be detected easily by the subscriber application.

One design decision for our test setup was to use as many default settings for MQTT brokers and clients as possible. Still, some tweaks were introduced, e.g., for QoS-1-based experiments we enforced in-flight-window size of 1 for publisher. This means that the publisher explicitly waits for a PUBACK after sending a message. Increasing the in-flight window size resulted in a vast number of out of order messages which was not compatible with our use cases.

Additionally MAYHEM can observe broker instances to which MQTT clients are connecting during experiments (cf. Section V-C and V-D). The actual broker instance within the cluster is not detectable via MQTT protocol due to used load balancer setup. Therefore, broker-specific management APIs are used to relate MQTT client ID to the cluster instance name. A typical test run consists of one publisher and two subscriber instances, publishing 5000 and receiving 10000 messages on one topic, respectively. Messages contain a sequence number and a timestamp resulting in a few bytes of payload.

	EMQ X	HiveMQ	Mosquitto	VerneMQ
Language	Erlang	Java	C	Erlang
Version	4.2.1	4.4.2	1.6.12	1.11.0
Multi-threading	yes	yes	no	yes
Persistence	ext. DB (enterprise plugin)	file	file	int. DB
Cluster support	free	enterprise	no	free
Queue mirroring	no	enterprise	no	no

TABLE I
Evaluated MQTT brokers and their features

To test broker resilience, MAYHEM performs deletion of K8s pods via K8s RESTful APIs. Per default, pod deletion is performed with a grace period of 0 to simulate catastrophic failures, e.g., node failures. Applications running in a pod's containers are terminated by a SIGKILL signal preventing them to perform any graceful shutdown actions like extra pre-termination synchronization or data persisting. Further failure scenarios, e.g., partial network packet loss or complete network isolation, were simulated using Chaos Mesh[8].

The evaluated MQTT brokers are summarized in Table I:

- **EMQ X** is an Erlang-based multi-treaded broker supporting clustering without queue mirroring even in the free version. Persistency requires storage backend plugins available for enterprise edition (EMQ X EE) only, which then transfer messages to separately deployed storage solutions. Plugins are available for example for Redis, PostgreSQL, MongoDB, or InfluxDB. We used the Redis integration in our experiments.
- **HiveMQ** is a Java-based broker with cluster support in the enterprise version. It is the only broker in the list supporting queue mirroring in cluster mode and stores persistent data to the file system.
- **Mosquitto** is a C-based single-threaded broker with no cluster support. For persistence, it has an internal database called "mosquitto.db".
- **VerneMQ** is an Erlang-based cluster-capable broker. For persistence, it uses an internal LevelDB key/value store for QoS 1/2 messages, which in our experiments was mapped to a persistent volume claim in K8s.

For all brokers, we tried to use the maximal persistence settings, e.g., Mosquitto is configured to persist on every network event.

Additionally we tried out emitter[9] and SwiftMQ[10] MQTT brokers, however initial tests failed due to emitter-specific time

[8]https://www.chaos-mesh.org/
[9]https://emitter.io/
[10]https://www.swiftmq.com/

to live and messages recall settings and random transmissions of null-bytes in SwiftMQ. We leave investigation of these brokers to future work.

Our hardware setup is based on a Starling X R3.0 Bare Metal All-in-one Duplex installation running on two servers each with a dual-CPU Intel Xeon CPU E5-2640 v3 2.60 Ghz, 2x 8 Cores, 16 Threads, L3-Cache 20 MByte, Dual Processor (i.e., 64 cores overall). Each server has 256 GB RAM (1866 Mhz) and run CentOS 7 Linux with a K8s installation based on Docker with persistence realized via ceph. The MAYHEM tests are run on a Windows 10-based server having a Xeon E5-2660 v4 2.0 Ghz CPU with 16 GB of RAM. Servers are interconnected via Gigabit Ethernet.

V. Resilience Experiment Results

A. *Variant #1: QoS 1/2 with Publication Acknowledgements*

The research question for variant #1 was: What is the effect of MQTT QoS 0/1/2 levels in case of network packet loss due to unstable connections? Former benchmarks analyzed the latencies of QoS 0/1/2, but did not simulate packet loss [4], [5]. Lee et al. [11] concluded from experiments that end-to-end latency and message loss are closely related. Thangavel et al. [12] measured MQTT latency for QoS 1 under network packet loss and showed that the average end-to-end latency can go up to 10 seconds for 25 percent packet loss. Roy et al. [13] showed in experiments that larger message payloads can lead to higher message losses in case of network disturbances in wireless networks when using MQTT-SN over UDP. Non of these works considered the effect of TCP retransmissions.

Our hypotheses for the research question were: 1) QoS level 0 shows higher message loss under packet loss, whereas QoS level 1 and 2 show no message loss. 2) QoS level 1 and 2 have higher publisher-to-subscriber latencies due to network retransmissions. We designed an experiment setup using MAYHEM with one publisher sending 1000 messages to the Mosquitto MQTT broker (at 100 msg/second), which relayed the messages to two subscribers. In each experiment all clients and the broker used the same QoS level configuration, i.e., 0, 1, or 2. We simulated packet loss using Chaos Mesh (based on netem) on outgoing messages of the broker network interface. Before the experiments the configured packet loss percentage was validated by sending 100 ping messages to the broker network interface.

Fig. 6 shows the successfully transferred messages (e.g., 99%) for different percentages of simulated packet loss. Most of our experiments showed no message loss. Only for a simulated packet loss of 30% the successfully transferred messages dropped to 83% for QoS 0. Simulating packet loss above 30% led to many connection problems between clients and brokers and was therefore excluded. Wireshark analyses showed that each experiment under packet loss included a number of TCP retransmissions.

As a second y-axis the chart shows the mean publisher-to-subscriber latency in seconds. The mean latencies for QoS 0 and 1 are close for up to 20% packet loss. Latencies for QoS 2 rapidly increase for higher packet loss percentages, as the entire four-way handshake needs to be redone in case of packet loss. The worst-case publisher-to-subscriber latencies for QoS 1 and 2 actually went up to more than 30 seconds. This may be problematic for many practical applications.

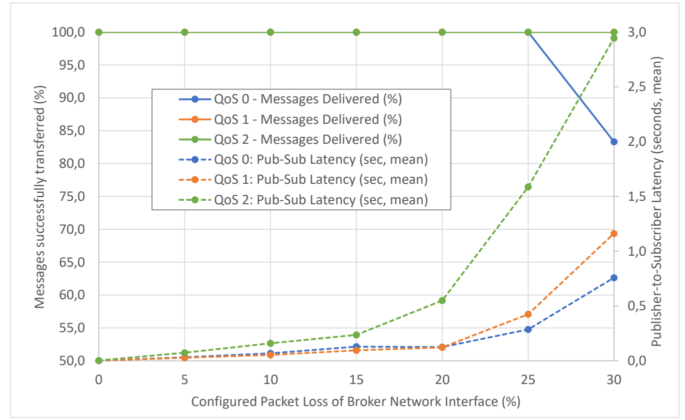

Fig. 6. Message loss and latencies of different QoS levels for simulated packet loss: almost no messages get lost, but publisher-to-subscriber latencies rise steeply for more packet loss.

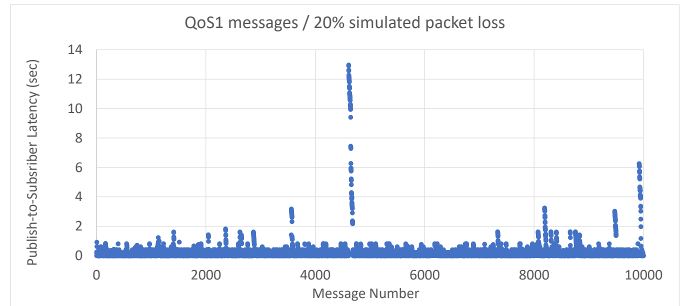

Fig. 7. Publisher-to-subscriber latency outliers: MQTT brokers sporadically combine multiple messages into a single package. If lost due to simulated packet loss, TCP retransmission lead to high latencies.

The latencies includes a few high outliers, whereas the median value for all experiments was comparably low. We analyzed these outliers deeper using Wireshark and found that the broker sporadically bundles multiple small MQTT messages into a single packet when sending them to the subscribers. Fig. 7, from an experiment run with 10000 messages, shows the effect of several multi-message packets, whose retransmission lead to high latencies. If such multi-message packets get lost due to network problems, very high publisher-to-subscriber latencies can occur in retransmission. In further analyses we found that the number and size of multi-message packets increases with higher messaging frequencies.

In conclusion, in our experiments QoS level 0 turned out to be quite robust against simulated packet loss up to 20%. TCP retransmissions were sufficient in our experimental setup to avoid message loss. Avoiding extra roundtrips from higher QoS levels can lower latencies and increase performance. High message frequencies should be avoided by throttling in case of unstable network connections to prevent bundling of many

messages into single packets.

B. Variant #2: Single-Instance Broker / QoS 1 / Persistence

The research question for variant #2 was: does broker persistency prevent message loss? Our hypothesis was that brokers can save queued messages to disk and retrieve these messages even after a broker failure and restart to correctly transfer them to all subscribers. Other researchers have investigated different aspects of message persistency, but did not evaluate popular MQTT brokers. For example, Sen et al. [14] implemented their own cluster MQTT broker called “Nucleus”, which used a distributed Redis backend for persistent storage. They emulated broker failures and showed that the Redis backend could limit message loss to less than 1% in their experimental setup. Petnik et al. [18] used Apache Kafka on top of MQTT to stream messages into a storage backend, but did not measure message loss under failure conditions. Geier [16] implemented an own message storage and migration solution for MQTT brokers.

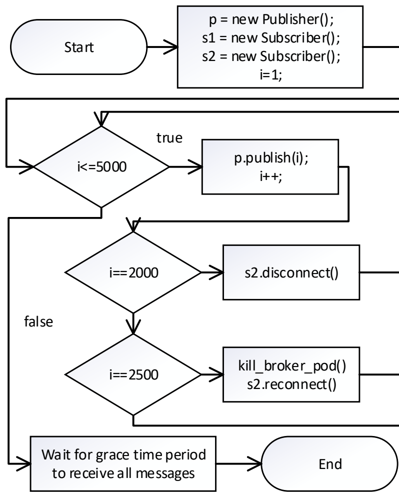

Fig. 8. Flow chart of a generic MAYHEM experiment.

Our experimental setup included different commercial and open source brokers with enabled message persistency. Each broker had the persistent message queue length configured to 10000 and used persistent volume claims in K8s on top of the ceph file system. Fig. 8 shows the experiment procedure. One publisher sends 5000 messages (payload: 4 bytes) using QoS 1 to be received by two subscribers. MAYHEM disconnected subscriber 2 at message 2000 to fill up the broker queue for disconnected clients and to trigger disk persistency. After publishing message 2500, MAYHEM “hard-killed” the broker pod (SIGKILL) without a grace period to emulate a node failure. Thus, K8s automatically restarted the broker pod and all clients reconnected once the broker was back online. Due to QoS 1, subscriber 1 should receive all messages, but may get duplicates. Also subscriber 2 should receive all messages, including message 2000-2500 when it was disconnected, since they were to be restored from the persistent storage. After the publisher had sent all messages, MAYHEM waited 20 seconds to allow the broker to deliver late messages to the subscribers. Each experiment was repeated 50 times to make temporary glitches from the K8s cluster visible. In case of using persistency, we tried to used maximal persistence settings for all brokers, e.g., in the configuration of mosquitto we persist on every network event.

	EMQ X EE (Redis)	HiveMQ (int. DB)	Mosquitto (int. DB)	VerneMQ (LevelDB)
Average Message Rate (msg/sec)	400.12	391.24	282.78	216.06
Average Message Loss	1.75%	0%	0%	5.34%
Average Message Duplicates	8.52%	0.14%	0.03%	0.02%
Average Message Out-of-Orders	1.07%	0%	0%	0%

TABLE II
MESSAGE LOSS FOR MQTT BROKERS WITH ENABLED PERSISTENCY: VERNEMQ AND EMQ X LOST MESSAGES IN OUR EXPERIMENTS, MOSQUITTO AND HIVEMQ NOT.

Table II summarizes our experiment results. In 50 runs each, Mosquitto and HiveMQ showed no message loss at all, despite killing the broker in the middle of each experiment. Both Mosquitto and HiveMQ had very few message duplicates and almost no out-of-order messages (Mosquitto had 2-3 out-of-order messages in a few experiments). Although our experiments showed no message loss, we for example did not analyze larger payload sizes or higher message frequencies, which could challenge the persistency mechanisms.

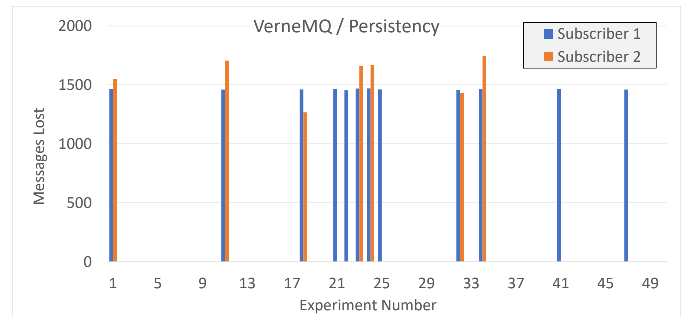

Fig. 9. Message loss for VerneMQ with persistency.

For VerneMQ, 81% of the experiments showed no message loss, validating that the persistency was correctly configured. For the remaining 19% of the experiments we experienced subscribers unable to reconnect to the restarted brokers, which led to a loss of messages at the end of the experiment runs. This led to an overall average message loss of 5.34%. Fig. 9 shows the message loss for subscriber 1 and 2 in each experiment (50 repetitions). In case of message loss, the subscribers lost the last 1500 messages (message 3500-5000). We analyzed the connection problems with Wireshark and found that the broker failed to immediately send CONNACK messages to subscriber CONNECT messages, although the lower level TCP

sessions were correctly established. We did not experience such sporadic connection problems for the other brokers, and thus assume an issue in the broker implementation.

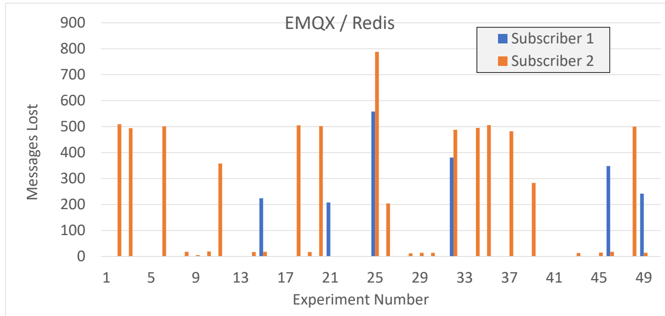

Fig. 10. Message loss for EMQ X EE with Redis persistency backend.

For EMQ X, we first tested a MySQL storage backend, which proved to be too slow (around 50 msg/sec). Thus, we configured a Redis storage backend that allowed much faster publishing (400 msg/sec). In 67% of the experiments, the subscriber correctly received all 5000 published messages, but in 33% of the experiments subscribers showed message loss, leading to an overall average message loss of 1.75%. Fig. 10 indicates that in most cases with message loss subscriber 2 lost around 500 messages, i.e., exactly the amount of messages supposed to be queued and persisted during the experiments. We however found all messages stored in Redis and conjecture that the transfer back from Redis to the subscribers did not work correctly in these cases. In some cases subscriber 2 lost only a dozen of messages, and in a few other cases also subscriber 1 showed message loss. EMQ X with Redis persistency also showed a number of out-of-order messages, which could be caused by the unordered key/value storage.

To summarize, variant #2 with a single instance broker persisting QoS 1 message to disk proved to be a reliable solution only for Mosquitto and HiveMQ. Both VerneMQ and EMQ X showed message loss, where our results rather indicate implementation problems than systematic conceptual problems. While persistency can prevent message loss as demonstrated, it always requires a full broker restart, which can take up to 30 seconds. During this time the publishers cannot deliver messages and need to build up internal queues. If an application however does not have high publishing frequencies and can deal with 30 second broker outages, variant #2 can be an easy to configure and handle solution for resilient IoT messaging. It avoids the complexity and intricates of a clustered configuration, where different broker instances need to be set up and synchronized. Especially in combination with a distributed file system that replicates the PVCs to different physical nodes, a single instance broker with persistency could also tolerate entire node failures. However, it should be noted that our small-scale experiments only provide a limited snapshot, as we did not analyze different payload sizes, message frequencies, nor very high client counts.

C. *Variant #3: Clustered Broker without Queue Mirroring*

The research question for variant #3 was: do clustered brokers without queue mirroring prevent message loss in case of broker failures? Our hypothesis was that the clustering could prevent message loss in case of graceful pod termination, because of the possible hand-over of session data to other cluster instances. However, in case of ungraceful pod termination, message loss should occur, since the killed broker instance has no chance of handing over its sessions and K8s restarts another pod instance without caring for persistently stored session data.

Most evaluations of clustered MQTT brokers in literature focus on performance [4], [6], [8], [10], but do not investigate resilience. Sen et al. [14] implemented their own clustered MQTT broker “Nucleus” and used a common storage backend for resilience. They measured package loss when emulating broker failures, which was however lower in case the storage backend was activated. Thean et al. [15] deployed multiple Mosquitto instances using Docker Swarm on five Raspberry Pis and used an HAProxy load balancer. While focusing on performance measurements, they also stopped one of the brokers nodes during the tests and showed that the broker cluster could recover without message loss and duplication. Authors of [17] investigated VerneMQ, HiveMQ, and EMQ X for performance and security and also found message loss in case of simulated broker failures. They however did not check graceful termination, connection problems and duplicates.

Our experiment setup included for each type of broker two broker pods running on one physical nodes to eliminate uncertainties regarding to load balancing. Furthermore, K8s was configured to always require two running instances, i.e., to restart failed pods. In these experiments, no queue replication happened between the broker pods, since the tested brokers do not support this feature except HiveMQ. To achieve a fair comparison, we disabled queue replication in HiveMQ which provoked a warning message from HiveMQ, because it is not a recommended configuration. We also disabled persistent volume claims in this setup, so that brokers could not retrieve message queues from disk.

Individual experiments were changed compared to Fig. 8: We connected the publisher and subscriber 1 to broker pod 1, and subscriber 2 to broker pod 2, and then sent 50000 messages (4 bytes payload) using QoS 1. We either killed (SIGTERM) or force-killed (SIGKILL) broker pod 2 at message 2500, upon which K8s started a new broker instance to restore the two instances required in the configuration. Graceful termination gives a broker instances 30 seconds to shut down in an ordered manner and potentially transfer stored session data to other instances. For ungraceful termination, this is not possible. Graceful termination can happen in case of node maintenance, but only ungraceful terminations correctly emulate node failures. Each client was configured with a keep-alive timer of 8 seconds and then immediately tried to reconnect in case of a detected connection failure.

Fig. 11 summarizes the results of about 20 experiments per broker. All brokers lose messages regardless graceful or

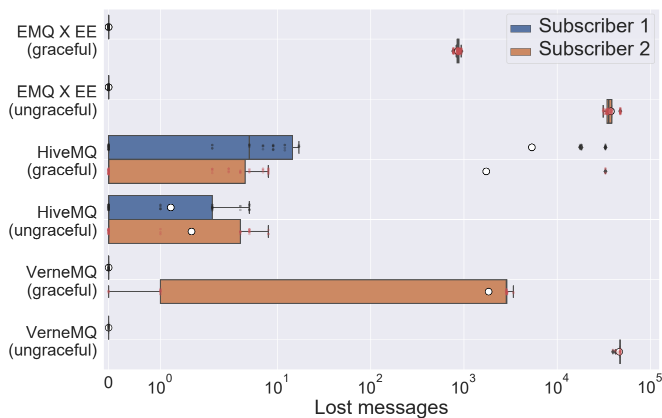

Fig. 11. Message loss for clustered MQTT brokers without queue mirroring comparing graceful and ungraceful termination. Average message rates are 575, 366 and 516 msg/sec for EMQ X, Hive MQ and VerneMQ, respectively.

ungraceful termination. EMQ X EE lost no messages for subscriber 1 (as expected, since it was always connected to broker pod 1) in both termination settings. During graceful shutdowns, subscriber 2 lost about 1000 messages on average due to the re-connection time. For ungraceful shutdowns, subscriber 2 showed reconnection problems (broker not answering to MQTT connect messages) resulting in large loss numbers.

HiveMQ did not demonstrate reconnection problems. As expected, for ungraceful termination subscriber 2 lost messages while being disconnected for several seconds. Surprisingly, we observed a loss of low number of messages for subscriber 1 which never experienced a reconnection to the broker.

VerneMQ lost no message for subscriber 1 in both settings. For graceful termination, however, we observed subscriber 2 reconnection problems where subscriber was often unable to reconnect at all and lost all messages after the disconnect. This behavior is similar to EMQ X EE, however technical details are different. Fig 12 shows an example reconnection attempt, where it takes the broker 72 seconds to answer a connect command. At that point, the client already initiated a TCP connection finalization, which the broker does not handle correctly, leading to a connection reset.

No.	Time	Source	Destination	Protocol	Info
22969	35.9	6.13.5	5.13.10	TCP	63492 → 1883 [SYN, ECN, CWR] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
22970	35.9	6.13.10	5.13.5	TCP	1883 → 63492 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1 WS=128
22971	35.9	6.13.5	5.13.10	TCP	63492 → 1883 [ACK] Seq=1 Ack=1 Win=2102272 Len=0
22972	35.9	6.13.5	5.13.10	MQTT	Connect Command
22973	35.9	6.13.10	5.13.5	TCP	1883 → 63492 [ACK] Seq=1 Ack=27 Win=29312 Len=0
30156	43.9	6.13.5	5.13.10	TCP	63492 → 1883 [FIN, ACK] Seq=27 Ack=1 Win=2102272 Len=0
30192	43.9	6.13.5	5.13.10	TCP	[TCP Retransmission] 63492 → 1883 [FIN, ACK] Seq=27 Ack=1 Win=2102272 Len=0
30196	43.9	6.13.10	5.13.5	TCP	1883 → 63492 [ACK] Seq=1 Ack=28 Win=29312 Len=0 SLE=27 SRE=28
36464	107.	6.13.10	5.13.5	MQTT	Connect Ack
36465	107.	6.13.10	5.13.5	TCP	1883 → 63492 [RST, ACK] Seq=5 Ack=28 Win=29312 Len=0
36474	107.	6.13.5	5.13.10	TCP	63492 → 1883 [RST, ACK] Seq=28 Ack=5 Win=0 Len=0

Fig. 12. Wireshark trace of unsuccessful reconnection attempt by VerneMQ.

D. Variant #4: Clustered Broker with Queue Mirroring

The final set of experiments was addressing the existence of a resilient MQTT broker which was especially providing no-loss and no-reordering guarantees for MQTT QoS 1 messages with a simultaneous high availability.

From the list of tested brokers, only HiveMQ falls into this category when using the default configuration of queue mirroring over broker instances. Our tests mostly confirmed those properties on a same experiment setup as in variant #3: only two messages were lost in one run with an ungraceful termination scenario. Some duplicates were present around message number where subscriber was reconnecting, still those could be simply filtered out on the application layer, since no reordering occurred. For graceful termination, we experienced non-systematic message loss of subscriber 1, similar to previous variant. We leave investigation of this behavior to future work.

To evaluate possible performance penalties for queue mirroring we tracked the average duration between sending and receiving a message on 500000 message runs without any disturbance. As expected, an increase was observed with a an average delivery duration of 5.86 ms without mirroring vs. 8.25 ms when mirroring was turned on. These numbers only provide a rough indication. For larger, higher-frequency messages replicated among more than only two pods, the expected latencies would likely increase much higher.

VI. Lessons Learned

We summarize experiences from the experiments as follows:

High availability does not prevent message loss: Broker vendors claim high reliability of their clustered implementations. However, "high availability" simply means that clients can always (re-)connect to a running instance and keep sending or receiving messages even if one or more of the instances fail. It does not necessarily guarantee absence of message loss, as clustered brokers may acknowledge messages that are not replicated or persisted. VerneMQ and EMQ X do not replicate all messages among cluster instances automatically, but instead rely on disk persistence for preventing message loss. This can be a good compromise between integrity and performance, but is not sufficient for applications that do not tolerate any message loss.

Load balancers are critical components for resilience: For a non-cloud cluster, a custom, external load balancer is required, which can be an additional single point of failure. Software solutions such as MetalLB provide different configurations (e.g., Layer 2 mode vs. Border Gateways Protocol) that have different benefits and drawbacks. BGP may require a special network router, while Layer 2 mode may trigger failovers caused by missing pods but not by node failures. While load balancers are not specific for MQTT communication, their configuration is an important factor contributing to the overall availability and reliability of the system and needs to be treated with special care. When we started our investigation, we underestimated this factor.

MQTT QoS 0 over TCP rather robust against packet loss: Our experiments showed that MQTT QoS 0 can still deliver messages reliably over TCP in case of limited packet loss. TCP retransmissions may thus be sufficient to prevent message loss in certain application scenarios, and higher level handshakes on the MQTT layer with QoS 1 / 2 could be

avoided to improve performance. However, our experiments only had few clients, small payloads, and low message frequencies. Additional experiments would be required to explore the utility of TCP retransmissions further. Our experiments showed that higher messaging frequencies can lead to bundling of messages into single packets, whose loss can lead to high latencies for retransmissions.

Large problem and solution space does not favor a single design: There are many different IoT application profiles, with vastly different numbers of clients (e.g., a dozen vs. several million), different message frequencies (e.g., 1 msg/sec vs. 1000 msg/sec), different payloads (e.g., 4 Bytes vs. 4 KBytes), different resilience requirements (e.g., no message loss, no duplicates, no out-of-order), different availability requirements (e.g., zero downtime vs. tolerable downtime). Accurate requirement specification is needed to design an appropriate solution. Reference scenarios are largely missing. We analyzed different variants implementing well-known availability tactics, mostly driven by the default offerings of the broker vendors. While QoS 1/2 is supported by almost any MQTT broker, persistency is activated by default by VerneMQ and HiveMQ. Only HiveMQ supported message replication to other instances before acknowledgement. If high availability and scaling is required, but minor message loss can be tolerated, then clustered MQTT brokers without queue mirroring may be a viable solution. If availability is of less concern, but no messages may be lost, then persistency on an non-clustered broker can be a more simple solution.

MQTT workload drivers challenged by failures: We started our resilience testing using existing MQTT workload drivers (e.g., JMeter, MZBench), since they provided the capabilities to run publishers and subscribers in configurations as described above. However, when we killed brokers or simulated other failures, we encountered different errors which complicated or even invalidated the measurements. MZBench workers ripped TCP connections apart or reported message transfers in their statistics that actually did not happen, since the broker had been killed. These experiences led to the implementation of our own MAYHEM tool using Eclipse Paho.

VII. Related Work

The Software Engineering Institute (SEI) in Pittsburgh, US has collected a catalog of availability tactics that could be considered as templates for the different resilience variants investigated in this paper [19]. Rozanski and Woods [20] describe the availability and resilience perspective of software architecture, including architectural tactics and checklists for architects. Our work can be considered an evaluation of technology-specific implementations of these tactics.

Many authors discuss the architecture of IoT applications. Tsigkanos et al. [21] sketch a vision and roadmap for resilient IoT systems and favor decentralized control over central control to avoid a single point of failure. They also carry out experiments with coordinating resources at runtime [21]. Alkhabbas et al. [3] report from an industry survey with 66 participants that 73% of them are using the MQTT protocol in their IoT applications. They design a goal-driven approach to adapt IoT systems responding to changes in the topology and the status of their components [22]. De Sanctis et al. [23] propose a reference architecture for self-adaptive IoT systemsThese work do not deal with specifics of MQTT communication.

Other research is concerned with analyzing and comparing message brokers, including MQTT brokers. Sommer et al. [7] compare AMQP, MQTT, ZeroMQ, and Kafka for industrial applications, but do not investigate resilience. Based on experiments with the Mosquitto MQTT broker, Lee et al. [11] showed a correlation of message loss and end-to-end latency for each QoS level. Thangavel et al. [12] compared the MQTT and COAP protocols under message loss and measured the impact on latency. Roy et al. [13] focused on the MQTT-SN protocol (MQTT over UDP) and carried out experiments involving Mosquitto and message loss. These works did not yet consider broker persistency nor clustered MQTT brokers and their specifics.

Finally, there are proposals for clustered MQTT brokers in literature which involve resilience investigations. Sen et al. [14] implemented an own clustered MQTT broker called "Nucleus" using node.js. Thean et al. [15] used a bridge to deploy multiple Mosquitto brokers in Docker Swarm. Koziolek et al. [17] analyzed different open source and commercial clustered MQTT brokers. Our work contributes to these works by deeper investigating and comparing different variants for resilient IoT applications using MQTT. We have designed and implemented our own workload driver MAYHEM, which can be independently used be to simulate applications and failures.

VIII. Conclusions

We found through experimentation that MQTT traffic can be robust to a certain extent against low network packet loss even on the lowest configured QoS level 0. Non-clustered MQTT brokers combined with persistency can provide a simple solution for more resilient IoT messaging, in case broker restarts are tolerable in the application context and horizontal scalability is not needed. Clustered MQTT without queue mirroring can suffer from message loss in case of node failures, only clustered MQTT brokers with activated queue mirroring turned out to be mostly resilient, albeit slower, in our tests.

Our results target researchers designing new communication mechanisms for IoT applications, as well as practitioners with challenging dependability requirements. In future work, we aim at extending our experiments to additional MQTT brokers and test higher message frequencies, higher number of clients, and different load balancers. More long-time tests could reveal additional reliability issues with the clustered and virtualization execution environment. To lower the effort for setting up a testbed and carrying out experiments, it is conceivable to capture the empirically found resilience data into simulation models that could be parameterized for different application profiles to carry out quick resilience tests.

References

[1] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE communications surveys & tutorials*, vol. 17, no. 4, pp. 2347–2376, 2015.

[2] V. Lampkin, W. T. Leong, L. Olivera, S. Rawat, N. Subrahmanyam, R. Xiang, G. Kallas, N. Krishna, S. Fassmann, M. Keen *et al.*, *Building smarter planet solutions with mqtt and ibm websphere mq telemetry*. IBM Redbooks, 2012.

[3] F. Alkhabbas, R. Spalazzese, M. Cerioli, M. Leotta, and G. Reggio, "On the deployment of iot systems: An industrial survey," in *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2020, pp. 17–24.

[4] HiveMQ-Team, "Benchmarks hivemq 3.0.0 on aws," https://www.hivemq.com/benchmark-hivemq3/, Sep. 2015.

[5] ScaleAgent, "Benchmark of MQTT servers," https://bit.ly/2WsTw0Z, Jan. 2015.

[6] HiveMQ-Team, "10,000,000 mqtt clients: Hivemq cluster benchmark paper," https://www.hivemq.com/benchmark-10-million/, Oct. 2017.

[7] P. Sommer, F. Schellroth, M. Fischer, and J. Schlechtendahl, "Message-oriented middleware for industrial production systems," in *International Conference on Automation Science and Engineering (CASE)*. IEEE, 2018, pp. 1217–1223.

[8] D. O'Mahony and D. Doyle, "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.

[9] S. Profanter, A. Tekat, K. Dorofeev, M. Rickert, and A. Knoll, "OPC UA versus ROS, DDS, and MQTT: performance evaluation of industry 4.0 protocols," in *IEEE International Conference on Industrial Technology (ICIT)*, 2019.

[10] M. Chaudhari and P. Gupta, "Building pubsub for 50m concurrent socket connections," https://blog.hotstar.com/building-pubsub-for-50m-concurrent-socket-connections-5506e3c3dabf, Jun. 2019.

[11] S. Lee, H. Kim, D.-k. Hong, and H. Ju, "Correlation analysis of mqtt loss and delay according to qos level," in *The International Conference on Information Networking 2013 (ICOIN)*. IEEE, 2013, pp. 714–717.

[12] 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 *International conference on intelligent sensors, sensor networks and information processing (ISSNIP)*. IEEE, 2014, pp. 1–6.

[13] D. G. Roy, B. Mahato, D. De, and R. Buyya, "Application-aware end-to-end delay and message loss estimation in internet of things (iot)—mqtt-sn protocols," *Future Generation Computer Systems*, vol. 89, pp. 300–316, 2018.

[14] S. Sen and A. Balasubramanian, "A highly resilient and scalable broker architecture for iot applications," in *2018 10th International Conference on Communication Systems & Networks (COMSNETS)*. IEEE, 2018, pp. 336–341.

[15] Z. Y. Thean, V. V. Yap, and P. C. Teh, "Container-based mqtt broker cluster for edge computing," in *International Conference and Workshops on Recent Advances and Innovations in Engineering (ICRAIE)*. IEEE, 2019, pp. 1–6.

[16] M. Geier, "Ermöglichung von Mobilität und Nachrichtenübermittlungsgarantien in verteilten MQTT-Netzwerken," Master's thesis, Wien, 2019.

[17] H. Koziolek, S. Grüner, and J. Rückert, "A Comparison of MQTT Brokers for Distributed IoT Edge Computing," in *Software Architecture - 14th European Conference, ECSA 2020, L'Aquila, Italy, September 14-18, 2020, Proceedings*, ser. Lecture Notes in Computer Science, A. Jansen, I. Malavolta, H. Muccini, I. Ozkaya, and O. Zimmermann, Eds., vol. 12292. Springer, 2020, pp. 352–368. [Online]. Available: https://doi.org/10.1007/978-3-030-58923-3_23

[18] J. Petnik and J. Vanus, "Design of smart home implementation within iot with natural language interface," *IFAC-PapersOnLine*, vol. 51, no. 6, pp. 174–179, 2018.

[19] J. Scott and R. Kazman, "Realizing and refining architectural tactics: Availability," Carnegie-Mellon University Pittsburgh, PA, Software Engineering Institute, Tech. Rep., 2009.

[20] N. Rozanski and E. Woods, *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley, 2012.

[21] C. Tsigkanos, S. Nastic, and S. Dustdar, "Towards resilient internet of things: Vision, challenges, and research roadmap," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 1754–1764.

[22] F. Alkhabbas, I. Murturi, R. Spalazzese, P. Davidsson, and S. Dustdar, "A goal-driven approach for deploying self-adaptive iot systems," in *2020 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2020, pp. 146–156.

[23] M. De Sanctis, H. Muccini, and K. Vaidhyanathan, "Data-driven adaptation in microservice-based iot architectures," in *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2020, pp. 59–62.