Investigating the Reliability and Security of the MQTT Protocol

Yifeng Liu

A thesis

Submitted in partial fulfillment of the

Requirements for the degree of

Master of Science in Computer Science and Systems

University of Washington

Winter 2022

Committee:

Eyhab Al-Masri (Chair)

Wei Cheng

Hossam Fattah

Program Authorized to Offer Degree:

School of Engineering and Technology

University of Washington

Abstract

Investigating the Reliability and Security of the MQTT Protocol

Yifeng Liu

Chair of Supervisory Committee:

Eyhab Al-Masri

School of Engineering and Technology

The MQTT is a brokered, publish-subscribe protocol that offers varying quality of service levels, providing a medium for machine-to-machine communication that is lightweight, versatile, and loyalty-free. It has been adopted in agriculture, energy management, and factory automation, just to name a few applications. Industrial usage reports show that the MQTT has gained popularity during 2020 and 2021, surpassing that of HTTP. However, the reliance on the broker is often viewed as a drawback. This series of studies investigate an MQTT broker's resilience, in particular, the temporal response to various testing parameters, including quality of service levels and transport layer security. An ideal linear relation is found between the mean response time and the payload size which allows performance measurement through a ranking score. Recommendations are made on the suitability of tested brokers for edge-side and cloud-side deployments. This study also discovered Slow Subscribers, a Slow Denial-of-Service attack against MQTT. The attack can be carried out with little resources and is shown to disrupt critical guarantees of message delivery. We documented the implementation detail of this attack and proposed a detection method based on probability distribution. Two MQTT messaging broker products are evaluated based on their responses to the attack. Finally, based on observations from service logs and collected datasets,

we propose Remistry, a multi-processing-based software architecture for MQTT brokers. To our knowledge, multi-processing has not been extensively adopted by MQTT brokers.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to express sincere gratefulness to Dr. Al-Masri for his input. This project would not be possible without his assistance.

I would like to thank Dr. Pan as well as his research colleagues for attending my thesis presentation.

Dedication

This work is dedicated to the package maintainers at the Debian project, for their unsung contributions keeping the Linux ecosystem secure.

Declaration

I declare that I am the sole author of this thesis. Parts of this thesis are based on existing publications and/or have been submitted for consideration for possible conference and/or journal publication. Copyright may be transferred in the future without notice. A digital copy of this thesis may be available at the University of Washington ResearchWorks database.

The contributions of this thesis are based on the following publications:

1. Y. Liu and E. Al-Masri, "Evaluating the Reliability of MQTT with Comparative Analysis," *2021 IEEE 4th International Conference on Knowledge Innovation and Invention (ICKII)*, 2021, doi: 10.1109/ICKII51822.2021.9574783;
2. A journal paper which, at the time of writing, is under peer review;
3. A second journal paper which, at the time of writing, is in progress.

Chapter 1: Introduction

In the context of the Internet of Things (IoT), Message-Oriented Middleware (MOM) provides a common communication medium among sensors, gateway devices, and cloud services [30]. The Message Queuing Telemetry Transport (MQTT) is an open, publish-subscribe protocol that is widely supported by cloud providers and has been implemented in a variety of programming languages [31] [32]. According to recent reports [18] [19] [43] [44] [45], the MQTT protocol gained attention in the industry between 2020 and 2022, reaching and sometimes surpassing the popularity of HTTP. OASIS Open published two key protocol standards, MQTT v3.1.1 and MQTT v5 [33]. Both versions of the protocol provide three different levels of Quality of Service (QoS) (0: at most once, 1: at least once, and 2: exactly once), each of which necessitates a different sequence of interactions between publishers, brokers, and subscribers. HTTP does not include, for example, a similar Quality of Service functionality.

Consider the following scenario for implementing MQTT in an industrial IoT environment for the purpose of monitoring an assembly line. Figure 1 illustrates the detailed architecture for

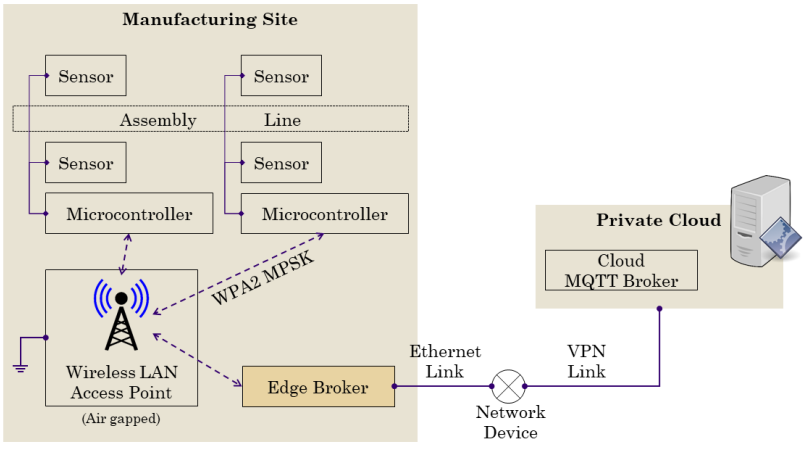


Figure 1. A high level network topology of an IIoT-enabled manufacturing plant that integrates sensors and actuators communicating via the MQTT protocol

this case. Assume that the administrators of a manufacturing or factory automation plant would like to determine the productivity impact of a personnel deficit. As a result, IR proximity sensors are used to monitor an assembly line. When an object is detected, these sensors generate a reading and can transmit the reading electronically through the MQTT protocol. To receive telemetry data, an edge device running MQTT broker software is placed on-site. Wireless communication between machines is possible via an air-gapped LAN access point secured by Multiple Pre-Shared Key authentication. Only the edge device has access to the outside world via an Ethernet link in this network design. By employing MQTT protocol characteristics, the edge broker is bridged to the cloud broker. As seen in Figure 1, the edge broker constitutes a single point of failure because a denial-of-service (DoS) attack on the edge broker will prevent all sensors from uploading data. The reliance on a broker is sometimes noted as a drawback of MQTT [34]. As such, we are interested in the fault tolerance of the edge broker under such an attack. That is, we would like to investigate how MQTT middleware implementations respond to system resource over-commitment in particular.

The Slow Subscribers attack, a denial-of-service vulnerability in the tested MQTT brokers, is due to a loophole in the MQTT v3.1.1 specification and the implementations' lack of granularity in access control. To carry out such an attack, the adversary must be granted subscribe access. We addressed the difficulty and impact of a successful attack. In response, we proposed Remistry (Resilient Middleware for Message Queue Telemetry Transport), a multi-processing-based software architecture, as a mitigation to the Slow Subscribers attack. To our knowledge, multi-processing has not been extensively adopted by MQTT brokers. The contributions of this paper are as follows:

1. We define Slow Subscribers, a DoS attack that overwhelms an edge broker device with undesired subscription requests.
2. We evaluate two lightweight MQTT broker implementations by comparing their responses to baseline and Slow Subscribers tests.
3. We propose Remistry, Resilient Middleware for Message Queue Telemetry Transport, as a mechanism for identifying and possibly mitigating this Slow Subscribers attack.
4. We develop a Python-based open-source stress testing package called Concurrent MQTT Evaluation Tool (Comet) for building our Slow Subscribers dataset [36].
5. We compile an open-source MQTT dataset for assisting researchers in detecting and mitigating the Slow Subscribers attack.

Our findings demonstrate that Slow Subscribers on a single compromised node are capable of disrupting an edge-side MQTT broker's capability if: (a) the authorization to subscribe to any active topic has been granted, and (b) sufficient subscriptions can be obtained. This implies that an attacker, for example, is not required to command massive botnets or to publish their own MQTT messages. As the name implies, a Slow Subscribers attack amplifies traffic through network congestion, which can result in high memory utilization on an edge device and, in some implementations, the total loss of QoS 1 and 2 messages.

To evaluate the Slow Subscribers attack, we simulate a scenario in which a variable number of IoT devices communicate telemetry payloads of varying sizes at a fixed sample rate. Then, we build a linear model to represent the ideal response times of a broker. Additionally, we design a ranking score for fault tolerance and a detection algorithm that employs it. We discuss the

implementation details of Remistry framework throughout the paper, while also investigating the framework's self-evaluation logic. To this extent, the objectives of this paper are as follows:

1. To develop a methodology for determining ideal response times of MQTT broker implementations and a mathematical model for predicting them.
2. To assess the appropriateness of several existing MQTT implementations for IoT systems at the edge and in the cloud.
3. To uncover the underlying causes of the performance degradation caused by the Slow Subscribers attack.
4. To inform the research community about how Slow Subscribers might disrupt IoT systems and to discuss attack mitigation measures.

Chapter 2: Related Work

2.1. Overview of MQTT

Communications in MQTT are optionally protected by Transport Layer Security which supports various authentication methods, including client certificates and Pre-Shared Key. In this paper, we study the performance impact of using TLS and TLS client certificates. On top of encryption, the MQTT defines [33] three service levels about message delivery. At QoS level 0, the publisher submits a PUBLISH packet to the broker, which the broker transmits to each subscriber; messages in this level are delivered at most once. At QoS level 1, the subscriber must acknowledge the recipient of each message by transmitting a PUBACK packet to the broker, resulting in a single roundtrip; messages in this level are delivered at least once. At QoS level 2, two roundtrips are required for each message: one for acknowledging PUBLISH and the other for acknowledging PUBACK; each message is delivered exactly once. Figure 2 depicts the MQTT protocol's flow diagram.

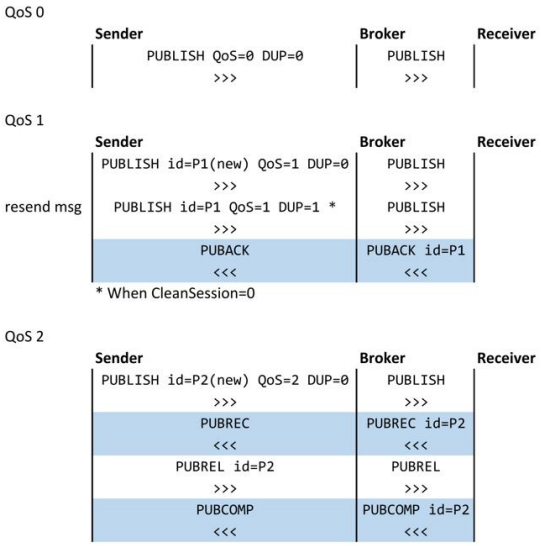


Figure 2. Flow diagram for MQTT Quality of Service Levels 0, 1, and 2.

In research literature, the performance of MQTT has been studied and compared extensively [46] [47] [48]. However, the selected papers did not propose a model for predicting a broker's performance under various input parameters. Additionally, the performance tradeoff from using transport layer encryption is not usually considered.

2.2. Architectures of Message-Oriented Middleware

Existing research studies have identified four common fault tolerance strategies within message-oriented middleware, including: **(a)** sharding [3] [4] [5] [6] [15], **(b)** replication [3] [15], **(c)** transaction logging [8] [22], and **(d)** abnormality detection [3] [7]. By distributing jobs across numerous server nodes, sharding minimizes disruption. It can be accomplished through topic hierarchy [3] [4] or client ID hashing [5]. Due to a lack of redundancy in these techniques, sharding is insufficient on its own to prevent messages from being lost. In replication, fault tolerance is achieved by synchronizing the state of a broker with failover nodes. In [6] [9], the authors present software architectures and implementation approaches for retaining broker states, while storing the information in non-persistent memory. State synchronization enables applications to execute transactions in a manner similar to that of Apache Kafka [22]. Our Remistry framework was developed such that it is compatible with sharding, replication, and logging while also integrating seamlessly within large distributed systems or data centers. Table 1 compares the features of existing MOM in terms of fault tolerance. As shown in Table 1, one essential feature that existing solutions lack is multi-processing (MP).

Multi-processing is an important aspect of task scheduling in operating systems, which can be defined as the execution of code in isolated, concurrent instances. It is distinguished from multi-threading by the isolation principle. Multi-processing is also quite different from containers in terms of optimization strategies due to the copy-on-write implementation of process generation.

Table 2 summarizes the distinctions among a thread, a process, and a container. For example, the well-known NGINX load balancer [20] distributes HTTP requests via multi-processing. The authors in [10] discuss in details the benefits and disadvantages of multi-processing. The authors in [6] delegated horizontal scaling to a container swarm.

Table 1.
Comparison of Fault Tolerance Techniques in Message-Oriented Middleware

	Sharding	**Replication**	**Logging**	**Detection**	**MP**
Resilient System [3]	Yes	Yes	No	Yes	No
GENOME [7]	No	No	No	Yes	No
muMQ [4]	Yes	No	No	No	No
Cluster SAGE [5]	Yes	No	No	No	No
Kafka [15] [22]	Yes	Yes	Yes	No data	No data
Nucleus [6]	Yes	Via Redis	Non-persistent	No	Via containers
Remistry (ours)	Compatible	Compatible	Compatible	Yes	Yes

Table 2.
Differences between Threads, Processes, and Containers

	Thread	**Process**	**Container***
Sharing memory	Yes	Manual	Manual, same pod
Passing open files	Yes, trivial	Yes	Same pod
Using channels	Yes	Yes	Yes
Isolation	Low	Moderate	High
Initialization	Fast	Moderate	Slow
Creation	Pthread API	Fork	Orchestration

** Using Kubernetes as an example*

While multi-processing would be beneficial for MQTT brokers, little to no research has been conducted to identify the characteristics of multi-processing approaches in MQTT. Further, existing studies do not distinguish between threads, processes, and containers or consider the distinctions are insignificant. Through the Remistry framework we proposed, a high level of granularity is maintained. To this extent, we design our fault tolerance technique which considers these distinctions particularly when considering DoS attacks.

Existing research has discovered several MQTT-related Denial of Service (DoS) attacks. The SlowITe [1] and its variant SlowTT [2] are identified as a class of slow DoS attacks against MQTT-based systems. Both types are based on the MQTT KeepAlive mechanism, which is used

to obtain and exhaust all TCP connections maintained by the message broker. These attacks are challenging to detect because they include a technique for masquerading anomalous behaviors: clients executing SlowITe and SlowTT attacks appear to the message broker as idle publishers.

There are, arguably, fewer research studies on MQTT Slow DoS detection than there are for HTTP and TCP [35]. In the context of HTTP and TCP, studies such as [11] [12] have developed traffic analysis-based detection methods. In [14], the authors propose a method for bridging MQTT and HTTP in both directions, enabling interoperability between the protocols. To this extent, load balancers developed for HTTP may be repurposed for MQTT. However, this strategy may also introduce vulnerabilities to both protocols, amplifying potential threats against systems that use this bridging mechanism.

2.3. Dependency on the Operating System

Inter-Process Communication (IPC), a non-standardized but widely available feature in modern operating systems, is required for communications between processes and between containers due to the increased isolation enforced by the operating system. Messages are transmitted between a pair of processes through a channel.

To this extent, we designed Remistry's architecture such that it is composed of numerous sub-processes, all of which rely on bidirectional IPC channels to function. Another essential feature is the ability to specify the users of a channel. The authors in [16] [25] provide detailed descriptions of the corresponding system APIs. In cases a target operating system does not support bi-directional channels or pairing, memory mapping may be employed. The authors of [10] examined alternative IPC methods including shared memory mapping. The availability of pair-based IPC is assumed thereafter.

An advantage of running multiple sub-processes is fast recovery. Linux contains a memory protection mechanism called the Out-of-Memory (OOM) manager, which is critical for Remistry's functionality. When the OOM manager is triggered, it terminates any process that could potentially free up a large amount of resources. Developers have considered alternative out-of-memory daemons [27], which are designed to evaluate the system's performance using more advanced metrics and to terminate any undesirable processes prior to invoking the kernel protection. The authors in [13] provide a benchmark for out-of-memory (OOM) daemons. In the architecture section, we describe the details of Remistry framework.

Chapter 3: Performance Model

Linear regression is incorporated as the method to predict the mean response time with regards to an arbitrary payload size. A normalization formula called Signed Relative Error (SRE), explained later, is applied to the residuals of linear regression so that error values become comparable regardless of payload size. By finding the standard deviation of SRE, one can determine the broker's timeliness of message delivery. This standard deviation value derives the ranking score for a broker implementation.

3.1. Comparing Linear Regression Models

Two linear models are by their limits as the input variable x approaches infinity. Let x represent the payload size in bytes and $m > 0$ be the number of publishers; with other parameters unchanged, the end-to-end response time $f(x)$ in milliseconds are modeled after a first-degree polynomial, given in Equation (1).

$$f(x; m) = A_0 + A_1 x \tag{1}$$

On the right hand side, A_0 and A_1 are the intercept and coefficient outputs of R's Ordinary Least Squares (OLS) optimization, respectively [26]. As we increase the number of publishers to some $n > m$, we again model the response time $g(x)$ after a first-degree polynomial related to payload size x, which is given in Equation (2).

$$g(x; n) = B_0 + B_1 x \tag{2}$$

On the right hand side, B_0 and B_1 are obtained from the OLS optimization as stated earlier for Equation (1). For the purpose of our study, we strictly assume the coefficients are positive (this is equivalent to assuming $A_1,\ B_1 > 0$). Subsequently, we define $p_{m,n}$ by Equation (3) with respect

to payload size x in order to compute how much slower the response time will become after changing the number of publishers from m to n. This equation returns a "slowdown percentage" value with respect to any payload size.

$$p_{m,n}(x) = \left[\frac{g(x)}{f(x)} - 1\right] \cdot 100\% \ \ (f(x) \neq 0) \tag{3}$$

At this stage, our goal is then to derive a real number quantifier from the real-valued function. By visualization, we observe that $p_{m,n}(x)$ converges to a horizontal asymptote as x approaches infinity. The horizontal asymptote of $p_{m,n}(x)$ can be obtained by computing a limit. Formally, the limit satisfies Equation (4).

$$\lim_{x\to\infty} \frac{g(x)}{f(x)} = \lim_{x\to\infty} \frac{B_0 + B_1 x}{A_0 + A_1 x} = \frac{B_1}{A_1} \tag{4}$$

Under ideal circumstances, the response time slows down by the value signified in Equation (5).

$$\text{Slowdown} = \left(\frac{B_1}{A_1} - 1\right) \cdot 100\% \tag{5}$$

To put it in another way, a slowdown percentage is computed from "the quotient of slopes minus one, times 100%". Based on Equation (5) and given that the resulting publisher count increased from m to n, the slowdown percentage is valid for sufficiently large payload sizes, which will be used for subsequent analysis throughout this study.

3.2. Comparing Observed and Predicted Values

Although the absolute value function can be used to evaluate the goodness of fit of regressions, it is impractical for the present study requirements because the mean response time is

also dependent on payload size. As defined in [17], the relative error equation is a mathematical tool for computing normalized residuals. It provides a ratio whereas the absolute value function generates a milliseconds-based result. To this extent, for a regression model f and a payload size x, we compute the relative error $E(x, y; f)$ with respect to the observed response time $y > 0$ and predicted response time $f(x)$ based on Equation (6).

$$E(x, y; f) = \frac{|f(x) - y|}{|y|} \quad (y \neq 0) \tag{6}$$

Based on the relative error calculation from Equation (6), we designate a Signed Relative Error (SRE) function as presented in Equation (7) to be used for generating the density plots of residuals.

$$\mathrm{SRE}(x, y; f) = \begin{cases} -E(x, y; f) & (\text{if } f(x) \geq y) \\ E(x, y; f) & (\text{otherwise}) \end{cases} \tag{7}$$

The standard deviation of signed relative errors will be turned into a numeric indicator of fault tolerance. Let us suppose an averaged dataset (which is described in more details in Chapter 5: Setup of Experiments) is a mapping function from payload sizes X to response times Y. We extract a sub-dataset $d: U \to Y$ $(U \subseteq X)$ such that the sub-dataset bears a constant publisher count n, for which the corresponding linear model is labeled f_n. Then, Equations (8) and (9) are applied to calculate the non-positive fault tolerance level T. If an implementation is shown to deliver messages on time in more occasions, the resulting score will be higher.

$$T = -\mathrm{stdev}(S) \tag{8}$$

Where

$$S \coloneqq \{\mathrm{SRE}(x, d(x); f_n) \mid x \in U\} \tag{9}$$

Chapter 4: Proposed Architecture

The Slow Subscribers is a Denial-of-Service attack having the potential to disrupt edge-side MQTT brokerage services. We will investigate the characteristics of this attack in a further chapter. An effective strategy to mitigate Slow Subscribers can be achieved through a combination of methods, including: limiting the number of unique client identifiers, the number of subscribers to a topic, and the number of topics a subscriber may register for. The challenge is, however, it is not easy to determine the values of these configuration variables. Therefore, it is reasonable to conclude that adding configuration variables is unlikely to yield a usable mitigation strategy to the Slow Subscribers attack and the impact this attack may cause.

4.1. Overview of Remistry

Because the Slow Subscribers attack results in extraordinary memory usage on brokers, we introduce Remistry, a resilient middleware framework for Message Queue Telemetry Transport (MQTT) that is capable of providing a granular handling of resource commitment errors, in particular the out-of-memory (OOM) problems. The Remistry architecture closely resembles that of HTTP load balancers. Figure 3 illustrates the building blocks of Remistry, consisting of multiple sub-processes that work collaboratively, including: (a) a control daemon, (b) a process manager, and (c) several worker processes forming a process pool.

Control daemon is responsible for monitoring the load and resolving any component failures. In addition, the control daemon maintains an IPC channel for each of its direct and indirect child processes. As a communication hub, the control daemon acts as a reporting agent that informs system administrators the health of the middleware using simple RESTful service endpoints.

Process manager is loaded thereafter. It is a sub-process with necessary access control privileges for coordinating the worker process pool. However, the process manager contains no analysis reasoning or logic. That is, it mainly responds to the control daemon's remote procedure calls. Preventing the process manager from becoming overloaded is a significant benefit of this architecture.

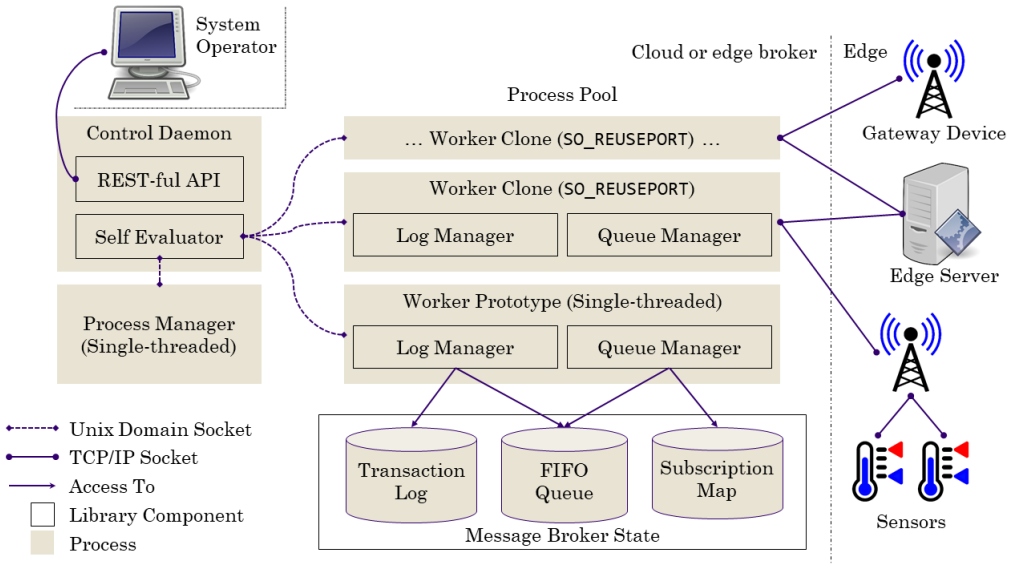


Figure 3. Architecture of Remistry: a Resilient Middleware Framework for Message Queue Telemetry Transport

Worker prototype is a process manager-created, partially initialized sub-process. It preloads dynamic libraries in order to implement process spawning taking advantage of Linux's copy-on-write (COW) feature, also referred to as implicit sharing or shadowing. Assuming the shared libraries' memory pages remain identical after initialization, worker clones do not need to reload the libraries. This dramatically enhances Remistry's performance and resource management.

Worker clone is a forked sub-process from the worker prototype. It is responsible for accepting TCP connections from edge servers, gateway devices, etc. All worker clones listen on

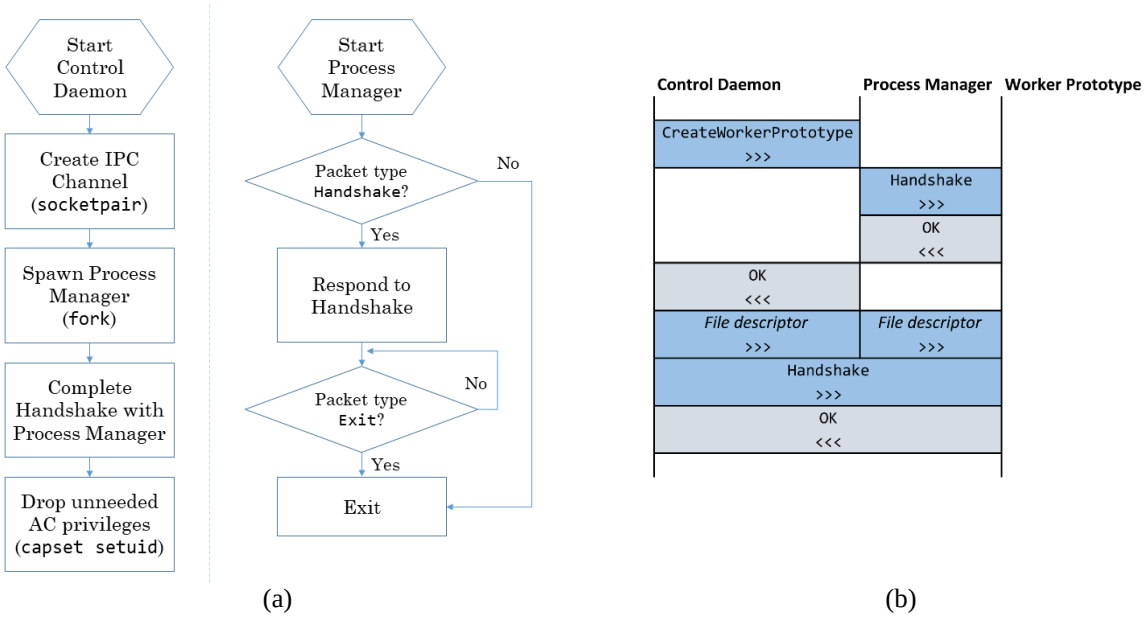
the same MQTT port. Listening on a shared TCP port is generally supported on Linux [23] and FreeBSD [24] operating systems.

We shall start analyzing the suitability for attack detection and fault tolerance on this architecture. To begin, the out-of-memory (OOM) daemon delivers a signal to one of the sub-processes, resulting in a partial service failure. Assuming one or more of the worker sub-processes had been terminated, recovery will be possible. The strategies are: if one of the worker clones failed, recreate it by performing a fork operation at the worker prototype; if the worker prototype crashed, respawn it from the process manager. The detection of component failures may be achieved by polling. In case of a failure, the control daemon will receive an error code. Through this process, Remistry maintains a high level of fault tolerance in mitigating Slow Subscribers attacks particularly across large distributed system environments (e.g., data centers or IIoT plants).

4.2. Practical Considerations

It is possible to implement Remistry on UNIX operating systems. On UNIX, the `init` process is responsible for launching persistent background services known as daemons. In Remistry, the control daemon is the entry point for `init`. When started, control daemon inherits root privileges which will be transferred to a process manager. After it has established an IPC handshake, the control daemon will relinquish unnecessary privileges and offload future operations to the process manager. This process is illustrated in Figure 4(a). Furthermore, by transmitting messages and file descriptors first to the process manager and then to the worker pool, it is feasible to initialize a worker prototype indirectly from the control daemon. Figure 4(b) demonstrates a proposed inter-process communication flow which establishes a direct channel between the control daemon and the worker prototype. Thereafter, it is trivial to initialize several

multi-threaded worker clones. As previously mentioned, each worker clone is responsible for handling MQTT connections and reporting statuses to the control daemon.



(a) (b)

Figure 4. (a) Control flow for initializing control daemon & process manager; (b) Inter-process flow diagram for initializing a worker prototype from the control daemon

Chapter 5: Setup of Experiments

We introduce a lab setup that will later become Comet, a customized software package developed to accurately measure the response timing of any broker implementation. The lab setting consists of two Google Compute Engine (GCE) virtual machines, one as the tester and the other one as the broker. In order to simulate edge device capabilities, we selected the "e2-small" machine type with two vCPUs and 2 GB of total memory. Additionally, we integrated geographic variance into our testing. For the many-to-one baseline and Slow Subscriber scenarios, we deployed a testing instance for the Mosquitto test cases in North America, while the broker instance is located in Europe; we deployed the testing and broker instances for NanoMQ at two separate regions in North America. For TLS tests, virtual machines were deployed in the same region.

In this study, we tested Eclipse Mosquitto [28] and NanoMQ [29] messaging broker implementations. Mosquitto is a lightweight and open source broker that is suitable for all types of IoT devices (e.g., from constrained gateways to full servers). NanoMQ is also a lightweight MQTT message broker designed for edge computing scenarios. It is built on the NNG lightweight brokerless messaging library, which supports asynchronous input/output multiplexing. This makes NanoMQ an ideal messaging platform for a wide variety of Internet of Things applications, including industrial control, factory automation, environment monitoring, and smart cities. Additionally, NanoMQ supports a broad range of 5G Multi-Access Edge Computing (MEC) deployments [29].

The features of both messaging brokers are summarized in Table 3. As shown, both projects share many similarities: they are written in the same programming language and support all three QoS levels. They are both lightweight and proper for edge environments. We believe that,

based on the commonalities these two messaging brokers share, it is relatively reasonable to use them in our study when evaluating the Slow Subscribers attack. In addition, because NanoMQ has insufficient compliance with MQTT 5, the MQTT protocol version that we employed was version 3.1.1 for all examined tests.

Table 3.
Feature Comparison of Eclipse Mosquitto and NanoMQ Messaging Broker Implementations

	Mosquitto [28]	**NanoMQ** [29]
Version	1.6.9	0.5.9
Support and compliance		
QoS levels	0, 1, 2	0, 1, 2
MQTT v5	Yes	Partial
MQTT v3.1.1	Yes	Yes
TLS	Yes	Yes
WebSocket	Yes	Yes
Bridging	Yes	Yes
Implementation		
Edge deployment	Yes	Yes
Multi-threading	Yes	Yes
Async I/O	No data	Yes
TLS library	OpenSSL	Mbed TLS
Main language	C	C
Maturity		
Year project started	2009	2020
Documentation	Complete	In progress

The lab setup and its network data flow are depicted in Figure 5. We employed a single tester instance for simulating multiple MQTT clients, including publishing clients and subscribing clients. Each publishing client is transmitting QoS 0, 1, and 2 messages of certain payload sizes (measured in bytes) to an MQTT topic `/test/{SensorName}` with a fixed sample rate (measured in messages per second). The placeholder `{SensorName}` represents an IoT sensor device that is registered as part of an IoT system.

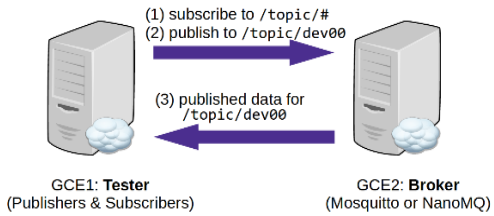


Figure 5. The lab setup and data flow between tester and broker instances

The TLS dataset records the Mosquitto broker's response timing at 3 security levels, from lowest to highest: (a) plaintext, (b) TLS v1.3, and (c) TLS v1.3 with mutual authentication using client certificates ("TLSMA" thereafter). In this dataset, the publisher count and subscriber count are always 1. For each payload size, a total of 7 messages are transmitted, and a mean response time value is subsequently computed for that payload size.

The baseline dataset simulates a "many-to-one" scenario with many publishers and one subscriber. For the baseline data generation, one subscriber receives all telemetry data by using the `/test/+` subscription wildcard. We employed a fixed sample rate that is 5 messages/second across all tests, meaning each device sends 5 messages of the same size within 1 second before switching to the next payload size. We believe this is a reasonable sample rate for monitoring devices in industrial use cases and settings. For each message identified by a unique in-payload ID number, only the earliest response time is recorded. A minimum delay cutoff of 5 seconds is applied between each payload size setting to account for potential messages in-transmission from the broker.

In the Slow Subscribers scenario, which is internally called "many-to-many", we simulated a large number of subscribing clients and publishing clients on one VM instance. Each subscriber announced a subscription to the same topics. The number of subscribers are the same as the number of publishers. Other configuration parameters remain unchanged.

We developed a stress testing package named Comet which stands for Concurrent MQTT Evaluation Tool. It assembles the ground truth or baseline measurements and the Slow Subscribers datasets [36]. We employed an MQTT client using Eclipse Paho (version 1.5.0) written in Python [37]. In tests that involve multiple publishers, the scheduling of test cases has been adjusted to minimize the spillover phenomenon due to in-flight messages from preceding tests. In this paper, we measure 5 variables' effect on the average response time: payload size, publisher count, quality of service level, security level, and broker implementation.

5.1. Measurement Error due to Spillover

The MQTT's quality of service (QoS) characteristics may interfere with data measurement due to the fact that the broker and publishers may attempt to retransmit unacknowledged MQTT packets from previous tests. As a result, the current test's response times may be larger than the ground truth due to the increased server load. We define this behavior as a "spillover."

Furthermore, a client library implementation is not required to report duplicated messages, as specified in the MQTT v3.1.1 specification [21]. The Eclipse Paho client library, for example, is not required to report the toggling of the DUP flag in a PUBLISH packet (see Figure 2). As a result, there is no accurate way for detecting MQTT message retransmission at the application layer, although we can identify any spillover by counting the number of lost messages.

To mitigate potential spillover situations, the test runner will wait for at least 5 seconds before changing to the next set of parameters. The waiting duration is chosen such that it is approximately 100% more than the maximum average response time in the baseline dataset. Following that, if any sent packet is not received, the test program will wait for any late packets in

increments of 5 seconds (a primary timeout), for a maximum of 120 seconds (a secondary timeout). Any message not received within the secondary timeout will be logged for further reporting.

5.2. Implementation of the Slow Subscribers Attack

Similar to SlowTT, the Slow Subscribers attack only requires sending control packets to maintain MQTT connections. Because the attacker only needs to send a single announcement for each subscription request, this novel MQTT attack results in amplification which is unlike SlowTT. This is a fundamental difference between prior MQTT attacks and the novel Slow Subscribers attack.

We identified that tested MQTT brokers support numerous connections from the same IP address, which we believe is required to accommodate devices behind NAT. Instead of IP addresses or ports, MQTT uses a 23-byte string chosen by the client implementation. To simulate several MQTT clients on one cloud VM, we kept a collection of unique ID strings. This allowed us to run Slow Subscribers without a computer cluster. Because the access control lacks granularity, Mosquitto and NanoMQ allow a single client with “subscribe access” to create an unbounded number of spoofed devices.

Chapter 6: Results and Evaluation

Our evaluation's objective is to demonstrate the extent to which a Slow Subscribers attack disrupts the communication across IoT systems. Additionally, based on our evaluation and analysis, we provide recommendations on mitigating the effects of this novel attack and ways on minimizing the risks associated with it. As part of our contribution, we created a public MQTT dataset for scholars to further their research efforts on MQTT [36]. We believe it is imperative to maintain a high level of transparency on this novel attack, ways to detect it and mechanisms for preventing to improve the security and resilience of existing IoT systems, particularly those involving edge or fog devices. For our experiments, we deployed the Eclipse Mosquitto and NanoMQ messaging broker implementations. Due to their compatibility with edge devices, we identified those brokers as viable platforms for evaluating our detection and mitigation strategies. We described in more details the rationale for choosing these two types of brokers in Chapter 5: Setup of Experiments.

The strategy for evaluation and data generation is as follows. Raw data frames collected from Google Compute Engine are averaged by payload size and interpolated using linear regression. We used Comet to assemble test results such that each test file has a fixed QoS level, security level, publisher count, and subscriber count; a result file contains multiple test cases, and each test case logs a packet ID, payload size, and response time. For each test file, we load it as a data frame, group the data frames by payload size, and compute averages response time. These steps will generate an "averaged dataset" consisting of the payload size column and the mean response time column.

A combination of methods are employed to maintain the accuracy of linear models. Prior to performing linear regression on the averaged data, we need to remove outliers or noisy data. In

cases that a series appears to have super-linear response time growth, we eliminate the tail and performed Ordinary Least Squares (OLS) on the pseudo-linear part of the series. In some other cases, we eliminated spontaneous increases of the response time in order to prevent biasing the regression algorithm.

Data from the evaluation are provided in several tables. The linear regression parameters computed in the tables are defined in Equation (1). The "slowdown percentage" values identified are formally given by Equation (10), which is derived from Equation (3) after setting $m = 1$. It computes an implementation's theoretical percentage-increase of response time at $n \geq 1$ publishers compared to the case of having only 1 publisher, assuming the payload size x is sufficiently large.

$$\lim_{x \to \infty} p_{1,n}(x) \tag{10}$$

6.1. Security-Performance Tradeoff

The performance tradeoff from using plaintext, TLS v1.3, and TLS mutual authentication (TLSMA) is studied using linear regression. The payload size range is partitioned into non-

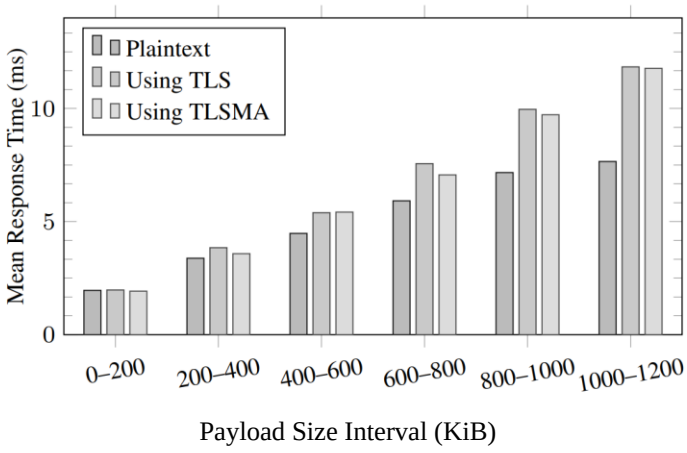


Figure 6. Mosquitto's QoS 0 response timing with regard to payload size, using plaintext, TLS, and TLSMA

overlapping intervals of 200 KiB, and the mean response time in each interval is computed. At a first glance using aggregated data from QoS level 0, plotted Figure 6, the mean response time differences between TLS and TLSMA are not significant. In particular, there is a 3% increase in the 0 to 200 payload size interval. Computed from Table 4, the corresponding percentage increases for QoS levels 1 and 2 are 11% and 2%, respectively. For larger payloads, it may be more useful to model the mean response times using a first-degree polynomial, whose coefficients of have been obtained using Ordinary Least Squares and provided in Table 5. The slowndown percentages demonstrate there is a marginal response time difference between TLS and TLSMA in Mosquitto for sufficiently large payload sizes. In our testing environment, using TLS or TLSMA at QoS levels 1 and 2 resulted in a seventy percent increase of mean response time compared to the plaintext configuration. The worst performance was measured at QoS level 2: using TLS or TLSMA at QoS level 2 is eighty to ninety percent slower than using plaintext at QoS level 2. Note well that individual percentage values given in Table 5 and Table 6 are not comparable across QoS levels.

Table 4.
Mosquitto's Mean Response Time (ms) in Non-Overlapping Payload Size Intervals (KiB)

QoS	Security	0--200	200--400	400--600	600--800	800--1000	1000--1200
0	PT	1.9575	3.3773	4.4756	5.9152	7.1644	7.6543
0	TLS	1.9738	3.8433	5.3904	7.5574	9.9594	11.836
0	TLSMA	1.9233	3.5789	5.4169	7.0627	9.7204	11.7716
1	PT	1.8276	2.8415	3.6975	5.1162	6.8120	7.8951
1	TLS	1.9451	3.5139	5.5778	7.9114	10.1787	11.8742
1	TLSMA	1.7534	3.0070	4.4488	6.8328	9.9884	11.7456
2	PT	2.8024	3.8195	4.5763	6.1935	7.6070	8.2785
2	TLS	2.5047	3.7768	5.3041	7.9585	11.4725	12.8641
2	TLSMA	2.4494	3.7117	5.2145	7.7037	10.6853	12.0606

Table 5.
Mosquitto's temporal response to different security transports, modelled by the first-degree polynomial

QoS	Security	Intercept (ms)	Payload size coef. (ms/KiB)	Slower (%) From Eqn. (10)
0	Plaintext	1.5520	0.0059	**0.0000**
0	TLS	0.7749	0.0100	69.1328
0	TLSMA	0.6380	0.0099	67.8866
1	Plaintext	0.9554	0.0062	**0.0000**
1	TLS	0.6644	0.0103	64.8209
1	TLSMA	0.0141	0.0105	67.8339
2	Plaintext	2.0868	0.0058	**0.0000**
2	TLS	0.6673	0.0111	92.1162
2	TLSMA	0.8452	0.0102	77.0705

Table 6.
Mosquitto's Response to Baseline Tests

QoS Level	Publishers	Intercept (ms)	Payload Size Coeff. (ms/byte)	Slower (%) From Eqn. (10)
0	1	92.229	0.00171	**0.000**
	2	109.725	0.00191	11.258
	4	119.501	0.00226	31.922
	8	136.795	0.00260	51.366
	16	163.671	0.00300	74.691
	32	176.170	0.00407	137.487
	64	173.114	0.00624	263.849
	128	156.325	0.01080	530.181
1	1	94.374	0.00172	**0.000**
	2	110.853	0.00188	9.267
	4	120.695	0.00225	30.661
	8	138.961	0.00258	49.486
	16	164.571	0.00301	74.443
	32	183.974	0.00390	126.096
	64	187.737	0.00604	250.539
	128	163.060	0.01081	527.028
2	1	290.583	0.00235	**0.000**
	2	379.281	0.00208	-11.368
	4	406.205	0.00240	2.197
	8	441.758	0.00292	24.266
	16	480.014	0.00354	50.650
	32	512.204	0.00549	133.393
	64	481.531	0.00973	313.970
	128	514.177	0.01506	540.675

6.2. Many-to-One Baseline Data

In Table 6, the results from running baseline test cases for the Mosquitto broker are presented. As can be seen, the slowdown percentage for each QoS level is similar for the largest number of publishers or 128. For example, QoS level 0 with 128 publishers yields a smaller percentage of 530.181%, whereas QoS level 1 is somewhat similar with 527.028% and QoS level 2 with slightly higher value, being 540.675%. However, if we compare the intercepts in milliseconds, which predict the mean response time with regard to small messages, it is clear that QoS levels 0 and 1 perform similarly, whereas QoS level 2 on average takes three times as much time to respond as either of the former levels. Additionally, we observe that QoS levels 0 and 1 provide comparable results, both in terms of the slowndown percentages and the payload size coefficients, whereas QoS level 2 has higher values respectively. This is due to the fact that each message in QoS level 2 requires a total of two roundtrips: one for acknowledging PUBLISH and another for acknowledging PUBREL which increases the overhead significantly.

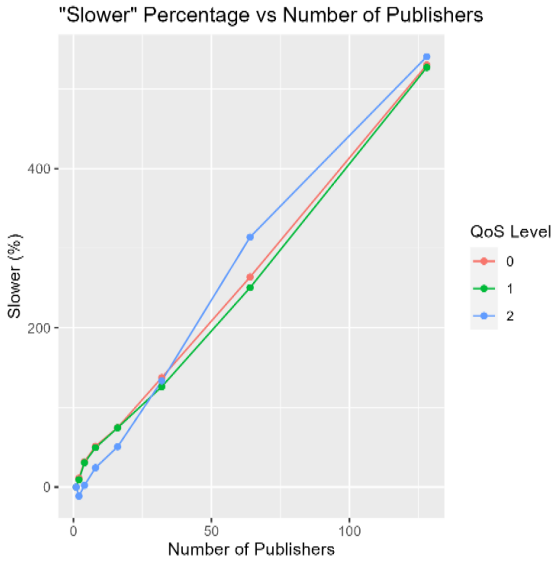


Figure 7. In baseline, the percentage value for which the average response time gets slower is proportional to the number of publishers.

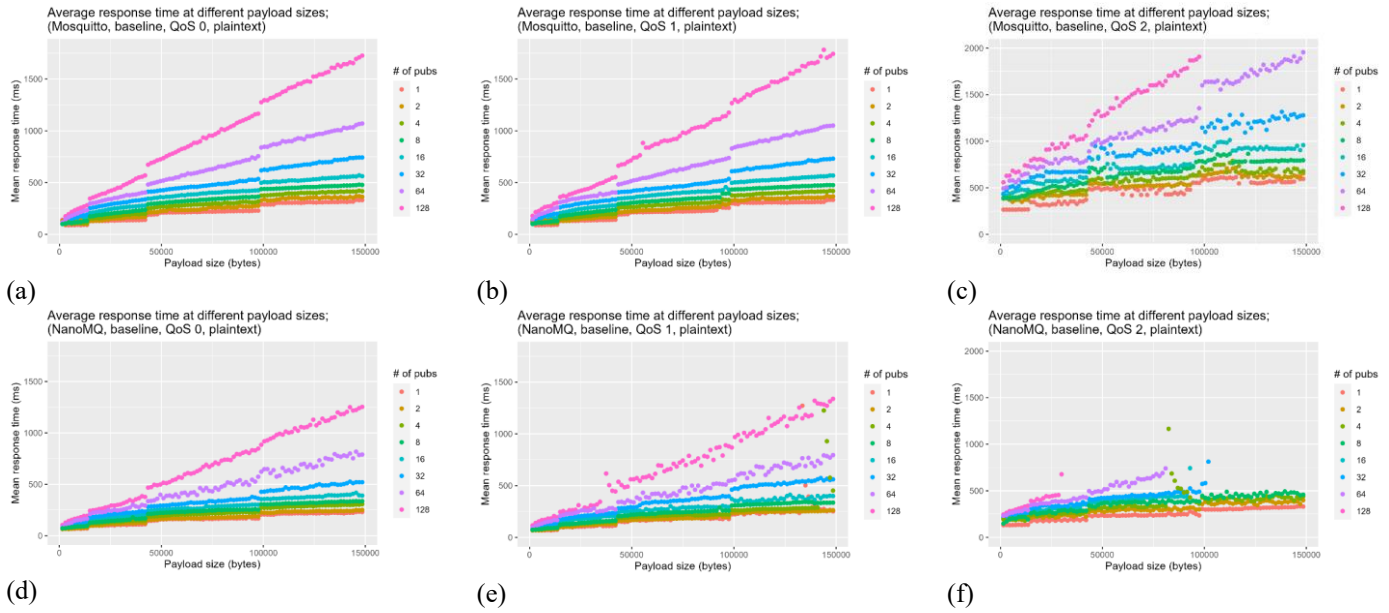


Figure 8. (a)-(c) Mosquitto's average end-to-end response times at different payload sizes, publisher counts, and QoS levels in the "many-to-one" baseline dataset; (d)-(f) NanoMQ's average end-to-end response times at different payload sizes, publisher counts, and QoS levels in the baseline dataset

In the baseline dataset, the slowdown percentage form a linear relation with the number of publishers, which is plotted in Figure 7. The OLS linear regression output for this relation is included in the MQTT dataset. In retrospect, the negative value in Table 6 for QoS level 2 with having two publishers is likely due to a measurement error.

As demonstrated in Figure 8, within the baseline dataset clearly exists a proportional relation between the payload size and the response time. To this extent, it is reasonable to utilize linear regression to model a broker's response time in the baseline dataset. As the publisher count increases, the standard deviations of signed relative error (SRE), as defined by Equation (8), do not change significantly when compared to the Slow Subscribers dataset. This is an important feature that distinguished legitimate messages from Slow Subscribers messages.

Beside the payload size, Quality of Service levels have a visible impact on the response time pattern. We demonstrate in Figure 8(a) the results of executing baseline performance

experiments on Eclipse Mosquitto using QoS level 0. As can be seen, the number of publishers ranges from 1 to 128. Each point's x coordinate records the payload size setting used by every publisher, and the y coordinate records the average time for a message to completely arrive at the broker and completely return back to the tester instance. The linear nature of point distribution implies that Mosquitto's average response time is linearly proportional to the payload size. Results from Figure 8(d) show that the NanoMQ broker produced a similar linear response but with greater variances, especially in the cases having 64 and 128 publishers. Despite the QoS level 0 specification requiring no guarantee of reliability, we did not observe a message loss within the test data that produced both subfigures.

In Figure 8(b) and (e), we present the baseline results for QoS 1. As shown in both subfigures, the variance for QoS 1 is visibly larger than that of QoS 0. In Figure 8(f), series 4, the response times are shown to have a spontaneous jump for payload sizes between 125 KiB and 150 KiB. This could be due to MQTT message duplication as the broker was trying to retransmit QoS level 1 messages. Compared to Figure 8(e), Mosquitto's response time pattern in Figure 8(b) demonstrates fewer spontaneous increases in average response time and a smaller variance.

The most significant variance can be observed at QoS 2, with the results shown in Figure 8(c) and (f). NanoMQ's responses in Figure 8(f) implied that the broker could not complete test cases for payload sizes greater than 100 KiB with the presence of 32, 64, and 128 subscribers. In the case of 4 subscribers, NanoMQ is shown to experience a spontaneous increase in response time between payload sizes 75 KiB to 100 KiB. Mosquitto's QoS 2 responses, plotted in Figure 8(c), are much more stable and complete than the NanoMQ counterpart.

Overall, QoS 0 tests demonstrated the best performance in terms of response time variance, and QoS 2 is shown to have the worst performance especially for large payload sizes. Therefore, large messages in QoS 2 are not recommended for scenarios with a number of publishers.

We also observe a gap in mean response time at predicable x coordinates, and the phenomenon is consistent across both implementations and all QoS levels.

6.3. Impact of Slow Subscribers

In Figure 9(a) the distribution of points implies that the tests for 32, 64, and 128 publishers were not completed by Mosquitto. During the tests we also observed the termination of Mosquitto service as a result of excessive memory usage. However, it was able to self-recover by restarting the service. In contrast, the NanoMQ daemon did not crash. Figure 9(b) shows NanoMQ's response to Slow Subscribers at QoS 0. In contrast to the Mosquitto counterpart, more points are

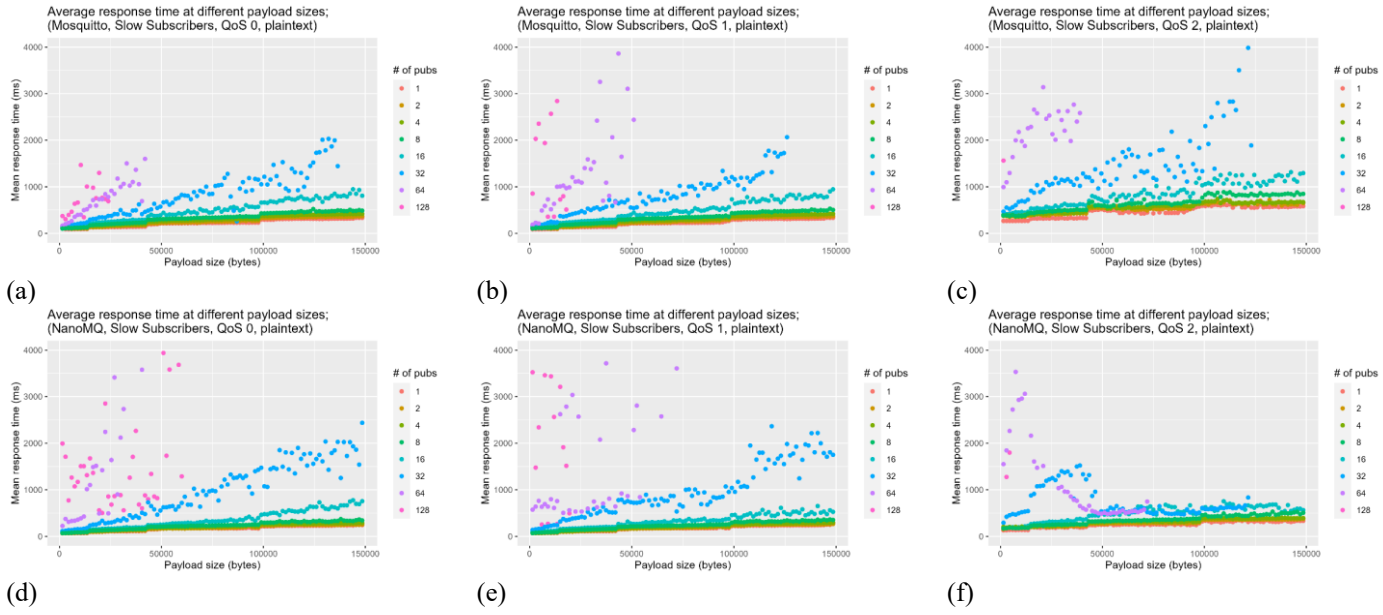


Figure 9. (a)-(c) Mosquitto's average end-to-end response times at different payload sizes, publisher counts, and QoS levels in the "many-to-many" or Slow Subscribers dataset; (d)-(f) NanoMQ's average end-to-end response times at different payload sizes, publisher counts, and QoS levels in the "many-to-many" or Slow Subscribers dataset.

present in Figure 9(e) series 128, demonstrating NanoMQ performed better than Mosquitto in response to the Slow Subscribers attack at this QoS level.

Mosquitto's response at QoS level 1, demonstrated by Figure 9(b), is similar to the broker's QoS 0 response. There is visible scattering of data points in Figure 7(e), implying NanoMQ was able to deliver messages to some of the subscribers, with response times comparable to that of the baseline dataset.

The Slow Subscribers attack at QoS 2 produced the worst response from Mosquitto shown in Figure 9(c). In particular, Series 32 demonstrated a super-linear growth, which is worse than the theoretical linear growth model. In Figure 9(f), a downward trend is observed in series 32 and 64. This can be due to two different reasons. Firstly, NanoMQ might be able to prioritize the delivery of messages to some brokers. Since we recorded each message's first arrival to any subscriber, an early, successful delivery to any subscriber would result in a smaller response time. Secondly, the decrease in response time could be caused by limitations of our measurement technique as described later.

6.4. Detecting Slow Subscribers with Probability Distribution

To evaluate linear regressions, we plotted for each publisher count the probability density functions of SRE, defined earlier in Equation (7). The density plots, generated after data cleaning, show the approximated mean values and standard deviations. Regarding Mosquitto's response to the baseline tests, Figure 10(a) shows normal distributions of SRE with a mean near zero, and Figure 10(b) shows SRE distributions in the presence of a Slow Subscribers attack. Under a Slow Subscribers attack, the distributions will obtain larger standard deviations than the baseline.

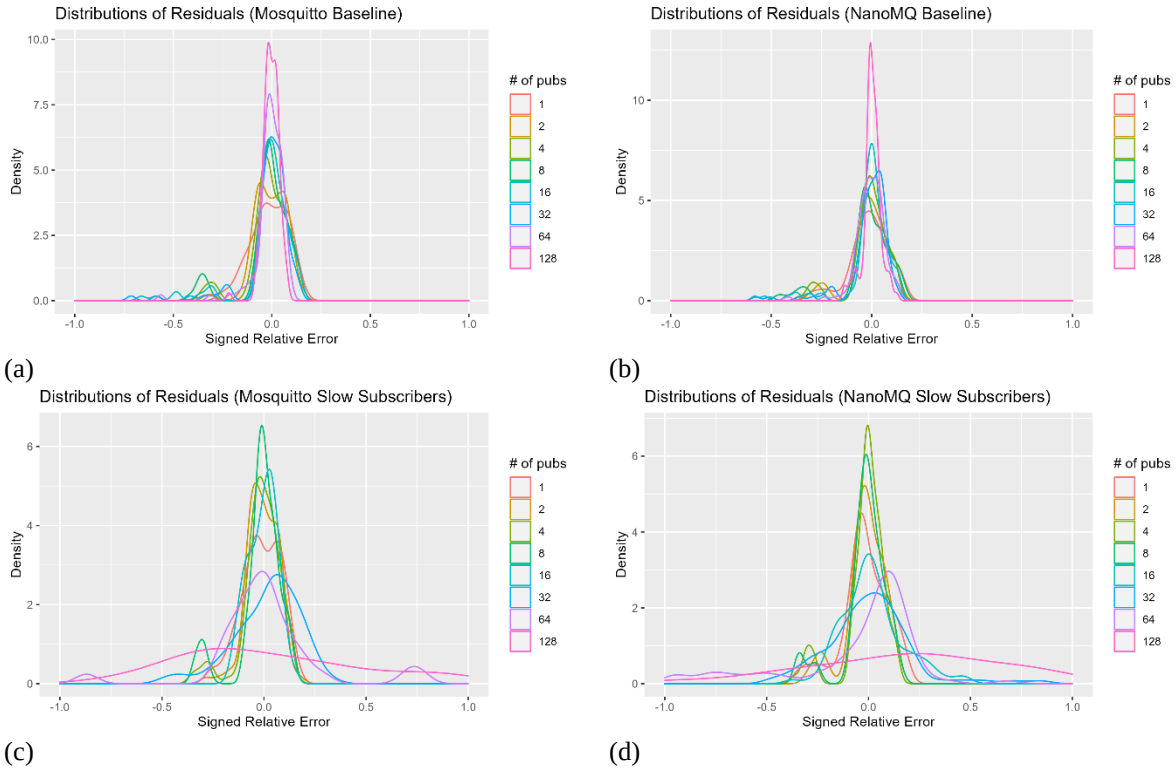


Figure 10. Distributions of linear regression residuals (computed with signed relative error on cleaned dataset)

(a), (b) are SRE distributions of the "many-to-one" or baseline dataset at QoS 0 for both brokers; (c), (d) are for the "many-to-many" and Slow Subscribers dataset at QoS 0 for both brokers.

NanoMQ's SRE distributions for the baseline and Slow Subscribers datasets are shown in Figure 10(c) and Figure 10(d), which clearly indicate that NanoMQ and Mosquitto exhibited similar behaviors under baseline tests and Slow Subscriber tests, respectively. This means a general attack detection algorithm can be adopted to both brokers.

To recap, we removed outliers from the MQTT dataset and then performed OLS linear regression for each publisher count, modeling the mean response time by payload size. Then, we obtained a sample of the signed relative error from a sub-dataset. Lastly, we plotted the probability density function from the signed relative error sample. Information about these probability distributions has been given in Table 7. For viewing convenience, applicable cell values have been multiplied by 100%. As an alternative, the OLS linear regression can be performed naively,

Table 7.
Distributions of Mosquitto's Linear Regression Residuals using Signed Relative Error (SRE)

Software & Publisher Count	QoS 0 Baseline		QoS 0 Slow Subscribers, From Cleaned Data		QoS 0 Slow Subscribers, From Raw Data	
	Stdev(SRE)	Mean(SRE)	Stdev(SRE)	Mean(SRE)	Stdev(SRE)	Mean(SRE)
M-1	9.23%	-1.16%	9.42%	-1.25%	//	//
M-2	8.20%	-0.81%	7.50%	-0.76%	//	//
M-4	10.31%	-1.60%	9.47%	-1.39%	//	//
M-8	12.16%	-2.03%	10.75%	-1.70%	//	//
M-16	14.41%	-2.37%	7.03%	-0.71%	//	//
M-32	12.93%	-1.93%	37.36%	-3.03%	//	//
M-64	8.38%	-1.09%	27.73%	0.96%	87.88%	4.22%
M-128	4.10%	-0.50%	50.70%	-3.41%	103.39%	-70.59%
N-1	10.23%	-1.48%	11.50%	-1.84%	//	//
N-2	9.64%	-1.38%	9.42%	-1.40%	//	//
N-4	10.31%	-1.56%	10.35%	-1.55%	//	//
N-8	12.88%	-2.18%	10.87%	-1.71%	//	//
N-16	12.94%	-2.06%	15.16%	1.77%	//	//
N-32	11.14%	-1.53%	23.14%	1.52%	//	//
N-64	7.42%	-0.75%	87.57%	-21.06%	//	//
N-128	4.50%	0.03%	180.25%	-119.81%	//	//

All standard deviations and means of SRE are multiplied by 100%. Abnormal values are shaded. Duplicate values which are the same as before are marked with "//"

without the data cleaning step. Results from such a process are listed in the "Raw Data" column. It can be seen that the Slow Subscribers attack is associated with large standard deviations or mean values far from zero. In this manner, the logistic regression model will be able to identify a Slow Subscribers attack.

Comparing the mean values of SRE at 64 publishers, we observe that the NanoMQ cell is negative, implying the linear model on average over-predicts the response slowdown by 21.06%. However, this value is not necessarily an indicator of fault-recovery because the mean SRE could have been skewed by messages that took longer to arrive, evident by the large standard deviation value.

Chapter 7: Discussion

We compared the temporal response of Mosquitto and NanoMQ. In the many-to-one test case, Mosquitto is more stable than NanoMQ due to the visibly smaller variance in average response times. In the Slow Subscribers test, our data suggest that NanoMQ is more resilient than Mosquitto.

The TLS dataset shows that using the client certificate for mutual authentication in TLS v1.3 results in a marginal increase of the average response time. Quality of Service levels 0 and 1 yielded similar performance metrics, whereas Quality of Service level 2 is the worst in terms of performance.

Under a Slow Subscribers attack, the response time may increase super-linearly as the payload grows. Contrary to baseline, the standard deviations of SRE in the attack may increase by more than one-fold, as consistently shown by both brokers in Table 7. Mathematically, Slow Subscribers responses are less appropriately fitted with a first-degree model as the publisher count increases. By using this mathematical relation, we are able to quantify a broker's performance score in the same way a linear regression's goodness of fit is evaluated. We designed a Slow Subscribers detection method based on the probability distribution of SRE, a key performance metric. By utilizing the Remistry framework, an implementation will be able to recover quickly by invoking a fork operation at the worker prototype process. Furthermore, Remistry is capable of reacting to signals from a user-space out-of-memory daemon. We also provided the flow of information in that scenario.

Tests involving a sufficiently large number of Slow Subscribers have led to unhandled exits of the broker service. However, tested implementations did not show any packet loss until broker

services completely failed, implying that an incremental error handling method was not present. In tests involving a sufficiently larger number of Slow Subscribers, it was observed that Mosquitto experienced an out-of-memory (OOM) error due to Linux's memory protection. We believe the tested version Mosquitto is more suitable for cloud deployments with its default configuration, due to a greater demand for memory vacancy. In comparison, NanoMQ is more suited for edge devices requiring QoS levels 0 and 1. Additionally, Mosquitto can be employed for task offloading from the edge to the cloud (or device-to-cloud cases), whereas NanoMQ is not reliable for cloud-based offloading and performs better in local, edge-based environments (or device-to-device use cases). This also suggests that brokers such as NanoMQ are unsuitable for MEC offloading or migration activities, whereas Mosquitto is much more dependable for cloud-based migration or task offloading to remote servers or data centers.

7.1. Slow Subscribers Attack Detection and Mitigation in Industrial IoT

We have demonstrated that the Slow Subscribers attack can be carried out with little computing resources and is capable of degrading the message delivery performance of edge-side MQTT brokers, disrupting critical communications of distributed systems or industrial systems such as factory automation. Data flows across IIoT from the sensors to the edge broker can be negatively impacted by such an attack. To illustrate the seriousness of the Slow Subscribers attack on industrial settings, let us reconsider the factory automation example from the Introduction section of this paper.

Assume that a manufacturing plant supervisor wishes to considerably reduce the amount of time required on system failures in the event of service disruptions or equipment breakdowns. As a result, the plant's administrators considered investing in predictive maintenance procedures, which will enable proactive data-driven techniques for monitoring the state of equipment

while predicting when maintenance is required [40]. By utilizing data analysis tools and techniques, it is then possible to detect anomalies in an assembly line or a factory process and identify potential system problems, neutralizing the risks of failure while also reducing costs associated with any manufacturing process disruptions [41]. However, predictive maintenance requires sensor placements across a large segment of equipment [42].

To this extent, a large number of suitable sensors have been placed for measuring the current state of health of nearly all the equipment that exist within the manufacturing plant. These sensors are programmed to continuously upload measurements using MQTT to edge gateways deployed across the plant. Examples of the sensors installed include temperature, humidity, water leak, vibration, motor current, tilt, sound pressure, magnetic field, and oil quality sensors.

Data generated from the dispatched sensors for monitoring purposes over a period of time can help identify patterns related to equipment failure. As the number of sensors increases, more data will need to be transmitted, which consequentially can result in making the Slow Subscribers attack more operational. If a malicious actor gained "subscription access" to parts of the MQTT network, he might be able to perform a Slow Subscribers attack, directly impacting the message transmission between sensors and edge gateways, which can potentially lead to an undesired inconsistency in data availability. The problem may get even more complex if the data is used for decision making involving manufacturing cycles or production strategies.

We believe that multi-processing can significantly help industrial systems reduce the risks of the Slow Subscribers attack. In addition to attack patterns demonstrated throughout the study, we evaluated 2 MQTT broker implementations for their edge deployment suitability. Recommendations detailed in Section "6.3. Impact of Slow Subscribers" suggest the avoidance of large messages at QoS level 2 in the use case of having a number of publishers. This can be very

helpful when deploying sensors and the type of MQTT broker within manufacturing, factory automation, or an industrial control environment.

7.2. Challenges and Limitations

We mentioned earlier that measuring a broker implementation's performance can be challenging due to interference or "spillover" from the MQTT protocol's Quality of Service [33] [38] [39] features. Our spillover mitigation algorithm is unable to detect lost, duplicated, or in-flight messages. This can significantly increase testing time and decrease accuracy. The temporary solution we adopted for minimizing precision errors is to wait for a fixed duration before switching to the next test case.

Presented in Figure 9(f), Series 32 and 64 exhibited a decreasing trend, but the same has not been observed in other test cases pertaining to NanoMQ, and we did not perceive any message loss (before the broker service became irresponsive). It is inconclusive what caused the decrease. It could be due to an implementation detail of the used MQTT client. For example, Paho may have buffered some incoming messages, resulting in a spontaneous decrease of measured values. Therefore, we suggest that future improvements to the current proposed methodology should: (a) investigate the use of a variety of MQTT client packages and (b) disable TCP cookies as well as background services.

Chapter 8: Conclusion and Future Work

We investigated the performance of MQTT brokers, the impact of a Slow Subscribers attack, and the methods to detect such an attack. Specifically, we primarily examined two messaging broker implementations: (a) Eclipse Mosquitto and (b) NanoMQ. Results from our experiments show that Eclipse Mosquitto achieves a higher reliability score compared to that of NanoMQ on cloud deployments with its default configuration. On the other hand, NanoMQ has shown to be well-suited for edge or fog environments particularly edge IoT devices that require QoS levels 0 and 1. Our data reveal that there exists a linear proportional relationship between the payload size of MQTT messages and the average response time, which is captured as a first-degree model. The use of client certificates in TLS v1.3 achieving mutual authentication does not degrade performance significantly compared to unilateral certificates in TLS v1.3. Furthermore, we provided recommendations on the broker's deployment suitability across cloud and edge or fog environments.

Throughout this paper, we identified the cause and potential impact of Slow Subscribers, which is a novel Denial of Service attack against MQTT brokers running within IoT environments. Based on the empirical data we collected through extensive experiments, we designed Remistry, a framework for detecting and recovering from resource commitment errors that aims to provide mitigation strategies as well as an acceptable level of fault tolerance in response to Slow Subscribers. As part of this study, we also published a public dataset for researchers to study and investigate the impact of the novel Slow Subscribers attack.

For future work, we plan to extend this study to include additional broker implementations developed in different programming languages, comparing their performance when dealing with

the Slow Subscribers attack. Beside MQTT brokers, future work would also select a variety of MQTT clients because the client implementation may cause discrepancies in measurements. Additionally, detection and mitigation are currently limited to receiving signals from the Out-of-Memory Daemon and terminating processes. What deserve more research are real-time attack detection and local migration.

With regards to learning experience, I was involved in a majority of the research process, including experiment design, software implementation, mathematical modeling, and data analysis. I heavily utilized Python for experiments and R for creating tables and graphics.

Bibliography

[1] I. Vaccari, M. Aiello, and E. Cambiaso, “SlowITe, a Novel Denial of Service Attack Affecting MQTT,” *Sensors*, vol. 20, no. 10, p. 2932, 2020.

[2] I. Vaccari, M. Aiello, and E. Cambiaso, “SlowTT: A Slow Denial of Service against IoT Networks,” *Information*, vol. 11, no. 9, p. 452, 2020.

[3] J. Wang, P. Jiang, J. Bigham, B. Chew, M. Novkovic, and I. Dattani, “Adding resilience to message oriented middleware,” *SERENE*, 2010, pp. 89–94.

[4] W. Pipatsakulroj, V. Visoottiviseth and R. Takano, “muMQ: A lightweight and scalable MQTT broker,” in *2017 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, 2017, pp. 1–6.

[5] P. Jutadhamakorn, T. Pillavas, V. Visoottiviseth, R. Takano, J. Haga and D. Kobayashi, “A scalable and low-cost MQTT broker clustering system,” *2nd International Conference on Information Technology (ICIT)*, 2017, pp. 1–5.

[6] S. Sen and A. Balasubramanian, “A highly resilient and scalable broker architecture for IoT applications,” in *COMSNETS ’18,* 2018, pp. 336–341.

[7] R. Savola, H. Abie, and J. Bigham, “Innovations and Advances in Adaptive Secure Message Oriented Middleware,” *ICDCSW*, 2010, pp. 288–289.

[8] Y. Jiang, Q. Liu, C. Qin, J. Su and Q. Q. Liu, “Message-oriented Middleware: A Review,” in *BIGCOM ’19*, 2019, pp. 88–97.

[9] J. E. Luzuriaga, J. C. Cano, C. Calafate, P. Manzoni, M. Perez and P. Boronat, “Handling mobility in IoT applications using the MQTT protocol,” *Internet Technologies and Applications*, 2015, pp. 245–250.

[10] X. Liu, L. Pan, C. Wang and J. Xie, “A Lock-Free Solution for Load Balancing in Multi-Core Environment,” *2011 3rd International Workshop on Intelligent Systems and Applications*, 2011, pp. 1–4.

[11] T. Hirakawa, K. Ogura, B. B. Bista and T. Takata, “A Defense Method against Distributed Slow HTTP DoS Attack,” *2016 19th International Conference on Network-Based Information Systems*, 2016, pp. 152–158.

[12] M. Sikora, T. Gerlich and L. Malina, “On Detection and Mitigation of Slow Rate Denial of Service Attacks,” in *ICUMT ’19,* 2019, pp. 1–5.

[13] S. K. Channabasappa, “Performance Analysis and Control of Latency Under Memory Pressure in the Linux Kernel for Edge Computing,” MS Thesis, University of North Carolina at Charlotte, Ann Arbor, 2019.

[14] M. Collina, G. E. Corazza, and A. Vanelli-Coralli, “Introducing the QEST broker: Scaling the IoT by bridging MQTT and REST,” in *2012 IEEE 23rd PIMRC*, 2012, pp. 36–41.

[15] T. Palino, N. Narkhede and G. Shapira, “Kafka Internals,” in *Kafka: The Definitive Guide*, O’Reilly, 2017.

[16] W. Gay, “Passing Credentials and File Descriptors,” in *Linux Socket Programming by Example*, Que, 2000.

[17] L. W. Johnson and R. D. Riess, “Errors in computations,” in *Numerical Analysis*. Reading, MA: Addison-Wesley, 1982, p. 3.

[18] IIoT World, “Survey Results: MQTT Widely Used in IIoT,” *IIoT World*, 2022. Accessed: Mar. 28, 2022. [Online]. Available: https://www.iiot-world.com/industrial-iot/connected-industry/survey-results-mqtt-widely-used-in-iiot/

[19] Eclipse Foundation, “2021 IoT & Edge Developer Survey Report,” 2022. Accessed: Mar. 28, 2022. [Online]. Available: https://iot.eclipse.org/community/resources/iot-surveys/

[20] *Tuning NGINX for Performance, NGINX HTTP Load Balancer.* Accessed: Mar. 28, 2022. [Online]. Available: https://www.nginx.com/blog/tuning-nginx

[21] OASIS Open. *MQTT Version 3.1.1, OASIS Standard.* Accessed: Mar. 28, 2022. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

[22] A. Mehta and J. Gustafson, “Transactions in Apache Kafka,” *Confluent*, Accessed: Mar. 28, 2022. [Online]. Available: https://www.confluent.io/blog/transactions-apache-kafka

[23] *socket – Linux socket interface.* Accessed: Mar. 28, 2022. [Online]. Available: https://manpages.debian.org/bullseye/manpages/socket.7.en.html

[24] *getsockopt, setsockopt – get and set options on sockets.* Accessed: Mar. 28, 2022. [Online]. Available: https://www.freebsd.org/cgi/man.cgi?setsockopt

[25] *Access ancillary data.* Accessed: Mar. 28, 2022. [Online]. Available: https://manpages.debian.org/bullseye/manpages-dev/cmsg.3.en.html

[26] *R: Fitting Linear Models.* Accessed: Mar. 28, 2022. [Online]. Available: https://stat.ethz.ch/R-manual/R-devel/library/stats/html/lm.html

[27] *OOMD: A new userspace OOM killer, Facebook.* Accessed: Mar. 28, 2022. [Online]. Available: https://facebookmicrosites.github.io/oomd

[28] *Eclipse Mosquitto*. Accessed: Mar. 28, 2022. [Online]. Available: https://mosquitto.org

[29] *NanoMQ*. Accessed: Mar. 28, 2022. [Online]. Available: https://nanomq.io

[30] M. Albano, L. L. Ferreira, L. M. Pinho, and A. R. Alkhawaja. “Message-Oriented Middleware for Smart Grids,” *Computer Standards & Interfaces*, 2015, vol. 38, pp. 133–143.

[31] U. Hunkeler, H. L. Truong and A. Stanford-Clark, “MQTT-S — A publish/subscribe protocol for Wireless Sensor Networks,” *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE '08)*, 2008, pp. 791-798, doi: 10.1109/COMSWA.2008.4554519.

[32] M. Singh, M. A. Rajan, V. L. Shivraj and P. Balamuralidhar, “Secure MQTT for Internet of Things (IoT),” *2015 Fifth International Conference on Communication Systems and Network Technologies*, 2015, pp. 746-751, doi: 10.1109/CSNT.2015.16.

[33] *OASIS MQTT Specification.* Accessed: Mar. 28, 2022. [Online]. Available: https://oasis-open.org/committees/tc_home.php?wg_abbrev=mqtt

[34] T. Yokotani and Y. Sasaki, “Comparison with HTTP and MQTT on required network resources for IoT,” *2016 International Conference on Control, Electronics, Renewable Energy and Communications (ICCEREC)*, 2016, pp. 1-6, doi: 10.1109/ICCEREC.2016.7814989.

[35] N. F. Syed, Z. Baig, A. Ibrahim, and C. Valli. “Denial of service attack detection through machine learning for the IoT,” *Journal of Information and Telecommunication*, 2020, no. 4, pp. 482–503.

[36] *MQTT Dataset, Internet of Things Research Group (INTRES), University of Washington Tacoma.* Accessed: Mar. 28, 2022. [Online]. Available: https://github.com/uwtintres/MQTT-Dataset

[37] *Eclipse Paho Python Client.* Accessed: Mar. 28, 2022. [Online]. Available: https://github.com/eclipse/paho.mqtt.python

[38] Y. Liu and E. Al-Masri, "Evaluating the Reliability of MQTT with Comparative Analysis," 2021 4th International Conference on Knowledge Innovation and Invention (ICKII), 2021, pp. 24–29.

[39] E. Al-Masri et al., "Investigating Messaging Protocols for the Internet of Things (IoT)," in IEEE Access, vol. 8, pp. 94880-94911, 2020.

[40] R. K. Mobley, *An introduction to predictive maintenance.* Elsevier, 2002.

[41] H. M. Hashemian and W. C. Bean, "State-of-the-Art Predictive Maintenance Techniques," in *IEEE Transactions on Instrumentation and Measurement*, vol. 60, no. 10, pp. 3480-3492, Oct. 2011, doi: 10.1109/TIM.2009.2036347.

[42] C. Scheffer and P. Girdhar, *Practical machinery vibration analysis and predictive maintenance*. Elsevier, 2004.

[43] R. A. Atmoko and D. Yang, "Online Monitoring & Controlling Industrial Arm Robot Using MQTT Protocol," *2018 IEEE International Conference on Robotics, Biomimetics, and Intelligent Computational Systems (Robionetics)*, 2018, pp. 12–16.

[44] C. Sun, K. Guo, Z. Xu, J. Ma and D. Hu, "Design and Development of Modbus/MQTT Gateway for Industrial IoT Cloud Applications Using Raspberry Pi," *2019 Chinese Automation Congress (CAC)*, 2019, pp. 2267–2271.

[45] E. Shahri, P. Pedreiras and L. Almeida, "Enhancing MQTT with Real-Time and Reliable Communication Services," *2021 IEEE 19th International Conference on Industrial Informatics*, 2021, pp. 1–6.

[46] B. Safaei, A. M. H. Monazzah, M. B. Bafroei and A. Ejlali, "Reliability side-effects in Internet of Things application layer protocols," *2017 2nd International Conference on System Reliability and Safety (ICSRS)*, 2017, pp. 207–212, doi: 10.1109/ICSRS.2017.8272822.

[47] D. Thangavel, X. Ma, A. Valera, H. -X. Tan and C. K. -Y. Tan, "Performance evaluation of MQTT and CoAP via a common middleware," *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, 2014, pp. 1–6, doi: 10.1109/ISSNIP.2014.6827678.

[48] S. Lee, H. Kim, D. Hong and H. Ju, "Correlation analysis of MQTT loss and delay according to QoS level," *The International Conference on Information Networking 2013 (ICOIN)*, 2013, pp. 714–717, doi: 10.1109/ICOIN.2013.6496715.