On the Latency of Secure MQTT over QUIC vs. TCP: An experimental evaluation

Sebastian Leclerc, Alessio Bucaioni, Mohammad Ashjaei
Department of Computer Science and Engineering
Mälardalen University, Västerås, Sweden
{sebastian.leclerc, alessio.bucaioni, mohammad.ashjaei}@mdu.se

***Abstract*—Smart-city services increasingly depend on IoT communication infrastructures that must provide secure connectivity without compromising latency-sensitive operation. This creates an engineering trade-off: security mechanisms protect urban sensing and service platforms, but may also increase communication latency, especially when devices connect under load. QUIC addresses part of this trade-off by integrating secure connection establishment into the transport layer, making it a relevant candidate for MQTT-based smart-city deployments. This study experimentally evaluates MQTT over QUIC and TCP under different security configurations, endpoint platforms, and traffic patterns. We separate three latency-relevant operating conditions: connection establishment, concurrent connection scaling, and sustained streaming in a multi-sensor, edge-to-cloud scenario motivated by smart-city applications. The results indicate that, in the evaluated configurations, the main latency cost of security appears during connection establishment, whereas steady-state encrypted data transmission adds limited overhead across the tested traffic patterns. For single connections, QUIC reduces per-connection latency by up to around 3.2 times compared to TCP with TLS. Under concurrent connection bursts, QUIC shows linear tail-latency growth in the measurements, and under high streaming load, it degrades more gradually than the TCP-based alternatives. The findings identify practical trade-offs among transport protocol choice, cryptographic algorithm, key type, and size, and endpoint computing capability. These trade-offs inform the design of event-driven smart-city services in which devices must establish or maintain secure communication under constrained latency and hardware conditions.**

***Index Terms*—Internet of Things, Edge, QUIC, TLS 1.3, Security, Authentication**

I. Introduction

Smart cities integrate large numbers of heterogeneous Internet of Things (IoT) devices across domains such as smart buildings, energy systems, urban mobility, public safety, and governance [1]. These systems depend on continuous data exchange among devices, edge nodes, and cloud services to support operational applications.

Several characteristics make this communication challenging: smart-city infrastructure is often mission-critical, privacy-sensitive, latency-constrained, and deployed at scale [1]–[3]. Prior work [1] identifies latency-critical scenarios, including smart-grid power distribution, emergency situation management, and city surveillance, that require end-to-end latencies on the order of 10 ms or below. Failures or delays may cause economic loss or endanger public safety, while the exchanged data can raise privacy concerns.

These requirements create a practical tension because stronger security mechanisms often introduce additional latency overhead [4]. At the same time, a May 2026 search of publicly exposed Message Queuing Telemetry Transport (MQTT) devices via Shodan[1] revealed more than 439,000 endpoints operating on the default unencrypted port 1883, indicating that unsecured MQTT deployments remain observable at Internet scale.

Properly configured TLS can mitigate common threats in smart-city IoT systems, including eavesdropping, man-in-the-middle attacks, and data integrity attacks [1], [4]. Architectural approaches can also reduce exposure and latency by using edge computing to process data on-premises before it traverses the public internet toward cloud infrastructure [2]. At the software-stack level, conventional deployments commonly use MQTT over TCP with a separate TLS layer. QUIC, originally introduced as "Quick UDP Internet Connections", has recently emerged as a candidate for secure low-latency communication because it integrates TLS 1.3 security over UDP within a single protocol stack [5].

However, QUIC remains underexplored in smart-city IoT settings, and limited evidence quantifies the latency cost of security under representative deployment conditions. While prior studies have primarily focused on protocol comparisons, architectural designs, or protocol tuning, less attention has been given to how security mechanisms affect latency across different deployment scenarios and operating conditions [4], [6], [7]. In this study, we experimentally compare MQTT over QUIC and TCP using different security configurations, endpoint platforms, and traffic patterns representative of smart-city workloads. We set up a multi-sensor, edge, and cloud environment and analyze connection establishment latency, concurrent connection scaling, and sustained streaming behavior.

The goal of this study is to investigate the latency cost of security in IoT communication systems for smart-city applications. The contributions are:

- An experimental comparison of MQTT over QUIC and TCP, analyzing connection establishment latency, concurrent connection scaling, and sustained streaming behavior under representative traffic patterns;

[1] www.shodan.io

- Insights into trade-offs between transport choice, authentication algorithm, key type and size, and endpoint computing capability.

The results indicate that the primary cost of security originates during the connection establishment, while steady-state encryption incurs negligible overhead over the evaluated traffic patterns. Key influencing factors include the choice of transport protocol, authentication algorithm, key type and size, and endpoint computing capability.

The remainder of this study is organized as follows: Section II provides background and reviews related work, Sections III describes the experimental setup, Section IV presents the experimental design and results, while Section V discusses the findings and validity considerations. Finally Section VI concludes the study.

II. Background and Related work

This section provides a brief background on the underlying technologies used in this study, followed by some of the related works it builds upon.

From an architectural perspective, edge computing refers to processing data on-premise, closer to where it is generated, before transferring it to the cloud [2], [8]. This approach enables functions such as data aggregation, filtering, and security enforcement at the network edge, thereby reducing both the volume of traffic sent to the cloud and the associated communication latency.

Within such edge-centric architectures, lightweight and efficient communication protocols are essential [4], [6]. One of the most prominent IoT communication protocols is MQTT, which follows a publisher-subscriber model in which information is organized into topics carrying associated data. Publishers transmit messages to a broker, which then distributes them to all interested subscribers. The protocol is lightweight, bidirectional, and designed to minimize bandwidth usage and power consumption, while enhancing reliability in lossy networks through multiple Quality of Service (QoS) levels. As IoT messages often carry sensitive information and traverse untrusted networks, securing communication channels is a fundamental requirement.

TLS 1.3, the latest IETF proposed standard, is a security protocol designed to provide a secure communication channel between two endpoints by ensuring authentication, confidentiality, and integrity [9]. During connection establishment, the endpoints negotiate how to set up the secure channel, commonly using Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) for key exchange. In TLS, the server is always authenticated, while client authentication is optional and commonly referred to as mutual TLS (mTLS). Authentication is typically achieved through asymmetric cryptography, such as RSA, ECDSA, or EdDSA (e.g., Ed25519), with the latter two being elliptic-curve-based. Most deployments use X.509 certificates, which bind an entity's identity to a public key through a trusted digital signature. Once the secure channel is established, application data is encrypted using computationally efficient symmetric encryption algorithms.

Building on TLS, QUIC is a natively secured transport protocol introduced by Google in 2017, initially aimed at improving HTTPS performance [5]. Since 2021, it has been a proposed standard by the IETF [10]. QUIC is built on top of UDP and eliminates head-of-line blocking, ensuring that packet loss in one stream does not block others. It further enables the exchange of application data as early as possible through tight integration with the TLS 1.3 handshake.

With this context established, we now review some of the related work this study is built upon.

Security remains a central challenge in smart-city IoT deployments, where large numbers of heterogeneous and resource-constrained devices are commonly used. Butun et al. [11] survey security threats in IoT systems, particularly wireless sensor networks, identifying numerous attacks across different system layers and highlighting the importance of encryption, authentication, and integrity protection.

From an architectural perspective, edge computing have been proposed to improve security by limiting data exposure before traffic traverses the public internet [12]. At the application layer, unsecured MQTT communication has been shown to enable unauthorized access and compromise of IoT infrastructures [13].

On cryptographic schemes, Aly et al. [2] argue that traditional asymmetric encryption developed for wired networks is unsuitable for small, battery-powered edge nodes. While this largely holds for ultra-constrained sensors, feasibility strongly depends on endpoint capabilities and usage patterns. Modern protocols such as TLS 1.3 together with elliptic-curve-based cryptography, are explicitly designed to minimize asymmetric operations, making public-key mechanisms possible for many contemporary IoT devices. Has et al. evaluate MQTT secured with TLS and symmetric AES on ultra-constrained NB-IoT devices and report higher overhead for TLS [14]. However, the TLS configuration is unspecified, and the AES-based approach provides confidentiality only via pre-shared keys, omitting authentication, integrity, and forward secrecy, which limits direct comparability.

QUIC, introduced by Langley et al. [5], integrates transport and cryptographic functionality to reduce connection establishment latency compared to TCP with TLS.

Prior studies have explored QUIC in IoT contexts. Study [6] compares QUIC to several IoT protocols, but focuses on protocols operating at different layers of the stack and leaves traffic patterns and measurement methodology largely unspecified. Another study [7] investigates tuning QUIC-based publish-subscribe architectures for IoT, focusing on transport-layer parameter optimization and resource-performance trade-offs rather than the latency impact of security mechanisms. The study in [4] proposes an architecture combining QUIC, WebSockets, and TLS session ticket sharing within an IoT cluster using a publish-subscribe mechanism. While this approach reduces reconnection latency via QUIC 0-RTT resumption, it assumes that cluster nodes already have a control-plane connection to receive shared session tickets and introduces

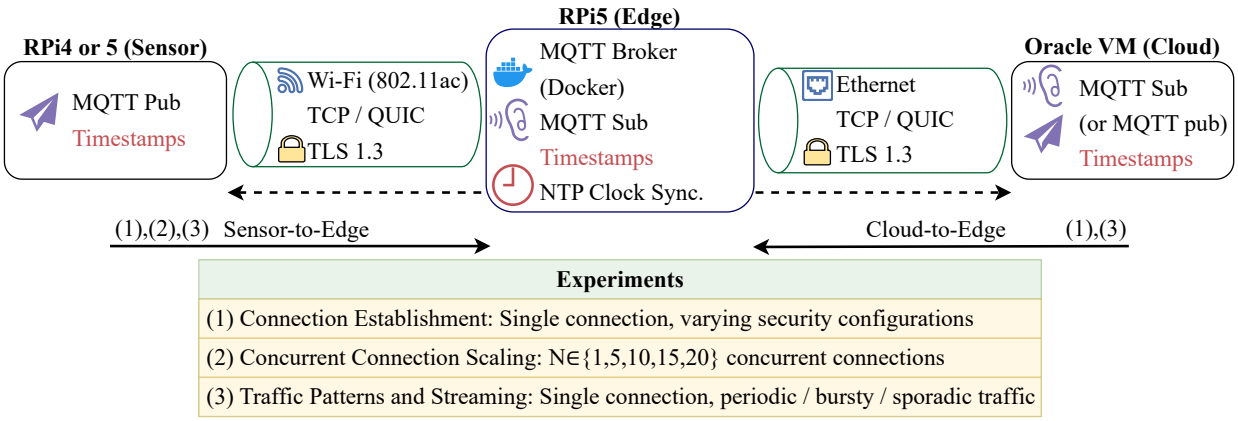


Fig. 1. Experimental sensor-edge-cloud architecture and evaluated experiments along each communication path.

additional security considerations, such as the impact of ticket compromise on the entire cluster.

III. Experimental Setup and Methodology

This section describes the experimental testbed, system configuration, and measurement methodology used throughout this study. We first outline the hardware platforms and network environment, followed by the OS configuration, software stack, time synchronization approach, and security mechanisms under evaluation. These components form the foundation for all experiments presented in the following section.

A high level view of the architecture and the conducted experiments is illustrated in Fig. 1. The experimental testbed consist of a sensor-edge-cloud architecture. The sensor node hosts multiple MQTT publishers, distributed across isolated CPU cores. The edge node hosts the MQTT broker within a Docker container, runs an MQTT subscriber, and serves as the local NTP time reference. The remote cloud node hosts an MQTT subscriber. Application-level timestamping is performed on all nodes. Communication between the sensor and edge uses Wi-Fi, while the edge-cloud path uses Ethernet over the public Internet, with transport based on plaintext TCP, TLS 1.3 over TCP, or QUIC. Experiments (1) and (3) are performed in both directions, while Experiment (2) is conducted in the Sensor-to-Edge path. The evaluation was split into segments to isolate access-network and endpoint effects. In particular, the Experiment (2) scenario is most representative for Sensor-to-Edge communication in real deployments.

For transparency and reproducibility purposes, all implemented code (publishers and subscribers), automation and evaluation scripts, data sets and supporting materials are included in the replication package[2].

A. Hardware platforms and Network

The sensor node consisted of either a Raspberry Pi (RPi) 4 using a quad-core ARM architecture at 1.8 GHz with 4 GB of RAM, or a RPi 5 using a quad-core ARM architecture at 2.4 GHz with 4 GB of RAM and hardware cryptography extensions. The reason for evaluating two different platforms was to see the impact of increasing endpoint computing capability. Another RPi 5 served as the edge device. The cloud node consisted of an Oracle Cloud [15] VM using a single virtual core (an "*OCPU*", corresponding to two hardware threads) operating at 2.0 GHz with 1 GB of RAM.

Communication between the sensor and edge nodes used Wi-Fi (IEEE 802.11ac, Wi-Fi 5), while the edge node was connected to the internet and the cloud via Ethernet through a consumer-grade router. This setup required port forwarding to allow inbound and outbound traffic, including SSH, NTP, and MQTT over TCP ports 1883 and 8883, as well as QUIC over UDP port 14567. QUIC connectivity was implemented using NanoSDK [16], which builds on NNG [17] as the messaging framework and MsQuic [18] as the QUIC transport implementation.

Unnecessary communication services, such as Bluetooth, avahi-daemon (DNS), and ModemManager (cellular communication), were disabled on all nodes, and Wi-Fi was disabled on the edge device to reduce background interference. Experiments were conducted in a typical indoor Wi-Fi environment with moderate interference, with the cloud endpoint located in the Oracle Cloud Stockholm region and accessed over the public internet.

B. Operating System Configuration and CPU Isolation

For the concurrent connection experiments marked as (2) in Fig. 1, CPU core isolation was configured on the sensor device [19]. The purpose was to reduce scheduler interference and OS-induced latency by isolating and distributing multiple MQTT publisher processes across dedicated CPU cores. This setup enabled the emulation of multiple concurrent clients using a single physical device, while improving timing control and experiment repeatability.

This approach does not fully eliminate non-deterministic effects originating from shared system components, such as the kernel network stack or I/O subsystems. However, it provides finer control over task placement and interrupt handling compared to unisolated execution. In this configuration, kernel housekeeping activities were primarily confined to core 0,

[2]https://github.com/SebastianLeclerc/quic_experimental

while the publishers under evaluation were pinned to isolated cores 1-3. The objective was not to achieve strict real-time execution, but to reduce cross-task interference and improve the consistency of latency measurements across runs.

C. Software Stack and MQTT Deployment

The edge device hosting the MQTT broker was deployed using the EMQX platform [20] inside separate Docker containers [21] per security configuration. It was configured to accept both plaintext and secured MQTT traffic on the aforementioned ports. Latency measurements were conducted in both communication directions and segmented across the sensor, edge, and cloud nodes. Depending on the measurement direction, the publisher was instantiated on the sensor node, the cloud node, or both. All experiments were based on either NanoSDK or Mosquitto [22], implementing lightweight Minimum Viable Product (MVP) publishers and subscribers. The specific implementations differed slightly across experiments (1), (2), and (3) in Fig. 1, depending on the experimental purpose, as further described in Section IV.

D. Time Synchronization and Timestamping

The edge node was configured as a local NTP reference, to which both the sensor and cloud nodes synchronized their clocks. Using Chrony, this resulted in an sub-millisecond edge clock accuracy relative to upstream NTP sources and a relative clock error of $\lesssim 2\,\text{ms}$ between the sensor and cloud nodes, which was sufficient for one-way latency comparison across nodes.

Timestamping was performed at the application level on each node using nanosecond-resolution timestamps referenced to the Unix epoch. For experiment (1), a timestamp was taken via the automation Bash script prior to transmission and upon reception of an MQTT message. For experiments (2) and (3), the application code injected a timestamp into the payload at message generation, which was compared upon reception at the subscriber. The measured interval therefore encompasses the complete end-to-end path, from a per-message perspective, including MQTT connection handling, TLS session establishment (where applicable), broker processing, and message delivery to subscribed clients.

E. Security Configuration

The same security protocol configurations were applied for both QUIC and TCP TLS-based experiments, following TLS 1.3 semantics as specified in RFC 8446 and RFC 9000 [9], [10]. In the QUIC-based setup, TLS 1.3 was integrated into the transport protocol, while for TCP-based communication, TLS was provided by OpenSSL (version 3.5.4) operating on top of TCP.

During connection establishment, key agreement was performed using ECDHE-based cipher suites. Server authentication was configured using X.509 certificates with the following (asymmetric) public-key algorithms: RSA (2048- and 4096-bit keys), ECDSA (256- and 384-bit elliptic curves), and Ed25519. The certificates were self-signed and deployed on the edge node using separate Docker containers per configuration. Client authentication (mTLS) was not used.

After connection establishment, application data was protected using symmetric AES-256 in Galois/Counter Mode (GCM) mode for all secured configurations. Consequently, the primary experimental variables related to security were: the presence or absence of transport security, the degree of integration between transport and security, and the server authentication certificate configuration. Additional security mechanisms, such as mTLS, authorization frameworks, or key management was not explored.

IV. Experimental Design and Results

This section presents both the design and the results of the three experiments conducted in this study. Each experiment isolates a different aspect of system behavior: connection establishment latency, concurrent connection scaling, and steady-state performance under varying traffic patterns. Together, these experiments provide a comprehensive experimental latency comparison of MQTT over TCP and QUIC across deployment scenarios and representative IoT payload sizes. Other aspects, such as CPU utilization and energy consumption were out of our scope.

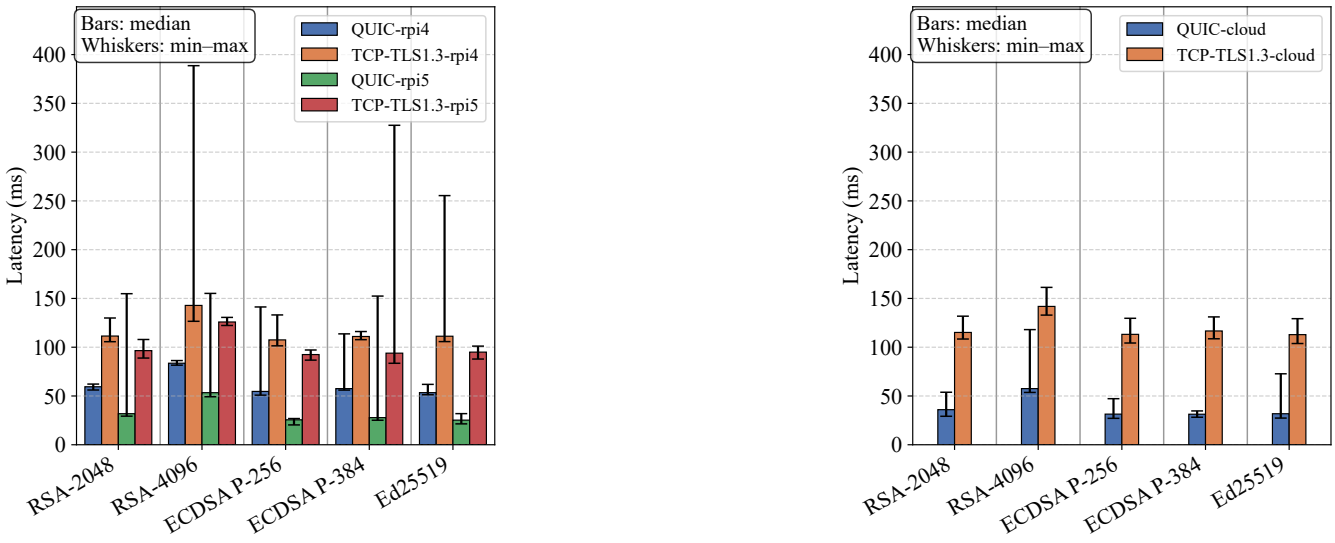
Experiment (1): Connection Establishment Latency

This experiment evaluated MQTT connection establishment latency over QUIC and over TCP with TLS 1.3. NanoSDK was used for QUIC-based communication and Mosquitto for TCP-based communication. Each measurement consisted of establishing a fresh MQTT connection, publishing a single 100 B message, and measuring the end-to-end latency from transmission at the publisher to reception at the subscriber. Both QUIC and TCP with TLS 1.3 employed server-only authentication with varying cryptographic configurations. For each combination of transport protocol, security configuration, and endpoint platform (RPi4, 5, or cloud), the experiment was repeated 50 times. Results were then aggregated and analyzed.

Experiment (1): Results

Beginning with the connection establishment latency, Fig. 2a shows results for the Sensor-to-Edge path, while Fig. 2b shows results for the Cloud-to-Edge path. Bars indicate median latency values, while whiskers denote the observed min and max. Each server-only authentication configuration was evaluated across multiple platforms and the two transport protocols. QUIC consistently achieved substantially lower connection establishment latency than TCP with TLS 1.3 across all evaluated configurations and platforms.

On the RPi4, QUIC latencies clustered around 53-83 ms, compared to approximately 107-142 ms for TCP with TLS 1.3, corresponding to roughly a 2× reduction. Elliptic-curve-based schemes (Ed25519 and ECDSA) outperformed RSA across both transports. RSA-4096 incurred the highest overall latency, while Ed25519 yielded the lowest, although the differences among the elliptic-curve schemes were small ($< 5\,\text{ms}$). This suggests that the latency is more sensitive to key size and



(a) Sensor-to-Edge.

(b) Cloud-to-Edge.

Fig. 2. Connection establishment latency for MQTT over TCP/TLS 1.3 and QUIC. Results show median (p50) end-to-end latency with minimum and maximum observed values for different endpoint platforms, including local RPI4/RPI5 nodes (left) and a remote cloud endpoint (right).

type. Upgrading to the RPi5 significantly reduced latency for all configurations, with QUIC clustering around 25-53 ms and TCP with TLS 1.3 around 92-125 ms. The relative improvement was up to 1.9× faster. QUIC benefited more from the increased computational capability, suggesting that its connection establishment is more sensitive to CPU performance.

For the Cloud-to-Edge path, similar trends were observed. QUIC again achieved markedly lower connection establishment latency than TCP with TLS 1.3 across all cryptographic configurations, and elliptic-curve-based schemes consistently exhibited the lowest latency. The relative improvement was more pronounced in this case, reaching approximately a 3.2× speed-up. Taken together, the choice of authentication algorithm, key size and type, and endpoint computing capability significantly influenced performance. Notably, the larger geographic distance to the cloud (while still within the same country) had a relatively small impact on latency.

Comparing the two paths, when an RPi5 was used in the Sensor-to-Edge path, the difference was approximately 5 ms for QUIC and 19 ms for TCP with TLS 1.3. This can plausibly be attributed to differences in network access characteristics, with higher contention on the wireless sensor side compared to the predominantly wired cloud path. In support of this interpretation, the Sensor-to-Edge measurements exhibited substantially higher variability, with larger spreads between min and max latency values and more pronounced outliers than observed on the Cloud-to-Edge path. Network factors such as contention and congestion were not explicitly controlled and therefore influence the observed latency measurements.

Experiment (2): Concurrent Connection Scaling

We evaluated connection-scale latency for MQTT over plaintext TCP and MQTT over QUIC by simulating $N \in \{1, 5, 10, 15, 20\}$ independent sensors. An MVP publisher and subscriber were used, intentionally reducing software complexity and overhead, while excluding non-essential functionality. QUIC used RSA-2048-based server authentication with AES-GCM encryption. Each sensor runs as a separate process spread out in isolated cores 1-3, establishes a fresh transport connection with a clean MQTT session, and publishes one timestamped message with a total payload of 100 B. A wildcard MQTT subscriber records receive timestamps from multiple publishers. For each N, the experiment was repeated 20 times, and the results were then aggregated and analyzed.

Experiment (2): Results

The results from Experiment (2) can be seen in Fig. 3, which evaluates connection-scale latency on the Sensor-to-Edge path as a function of the number of concurrently connecting sensors. The authentication algorithm was set to RSA-2048. Results show end-to-end median (p50) and tail (p99) latency for MQTT over QUIC and plaintext MQTT over TCP. Note that lightweight MVP clients are used in this experiment. Therefore, absolute latencies are not directly comparable to the connection establishment latencies reported in Experiment (1). Nevertheless, QUIC still remained faster than plaintext transmission by approximately 1.5× reduced latency when $N = 1$.

As the number of concurrent sensors increased, a linear growth can be observed for the tail latency, while the median remains relatively stable. At $N = 20$ concurrent clients,

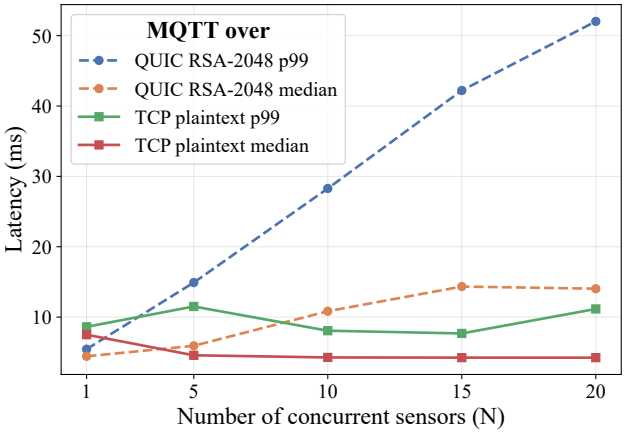


Fig. 3. Concurrent connection-scale latency for MQTT over QUIC and plaintext TCP. Median (p50) and tail (p99) end-to-end latency are shown for the sensor-to-edge path as a function of the number of concurrently connecting sensors.

the p99 latency for QUIC increased by nearly an order of magnitude compared to the single-client case. In contrast, TCP maintained relatively stable across all tested concurrency levels. The observed scaling behavior indicates that plaintext TCP handles bursts of concurrent connection establishments more robustly than QUIC in this experimental setup. QUIC applies cryptographic protection at the packet level, which may amplify per-packet processing cost under concurrent connection bursts. A clear limitation in this set up is the single sensor, hosting multiple publishers, having to share the computing resources.

Experiment (3): Traffic Pattern and Streaming Behavior

Finally, this experiment evaluated latency behavior for sustained MQTT message streams over plaintext TCP and secure MQTT over QUIC under representative traffic patterns. QUIC used the same security configuration as in Experiment (2), and the same MVP applications were employed. A single publisher, executed on the evaluated endpoints, transmits messages according to periodic, bursty, and sporadic traffic patterns. Periodic traffic uses fixed sending rates of 1–10,000 messages/s with payload sizes ranging from 32 to 1024 bytes over a duration of 30 s per run. Bursty traffic consists of bursts of 50 messages transmitted every 1 s, resulting in average rates of 50–400 messages/s. Sporadic traffic sends messages with exponentially distributed inter-arrival times at an average rate of 10 messages/s over a duration of 60 s. Identical workloads were applied to both transport protocols, and results were aggregated per configuration for comparison across traffic patterns and security settings.

Experiment (3): Results

Finally, the results of Experiment (3) can be seen in Table I, which evaluates end-to-end latency behavior under sustained traffic for plaintext MQTT over TCP and secure MQTT over QUIC across periodic, bursty, and sporadic traffic patterns. At low and moderate periodic message rates (up to approximately 1,000 messages/s), both transports exhibited stable median and tail latency, with no observable penalty attributable to transport-layer security.

At high periodic rates ($\geq$ 5,000 messages/s), TCP exhibited severe tail-latency amplification and collapsed, experiencing high loss rate of almost 25%. In contrast, QUIC showed more graceful degradation, with limited growth in tail latency under the same load conditions. This behavior suggests that QUIC handles sustained high-rate encrypted traffic more robustly than TCP in this setup. For bursty traffic patterns, both transports experienced moderate tail amplification, however, QUIC generally exhibited more predictable tail behavior. Under sporadic traffic, latency was dominated by idle periods between transmissions, and both transports showed stable behavior with no meaningful differences.

Disabling hardware cryptographic acceleration increased baseline latency for both transports but did not alter the relative trends observed across traffic patterns. Although plaintext does not use cryptography, it seems that changing the to the overall execution environment slightly affected both applications.

Cloud-to-Edge experiments at lower rates showed behavior similar to the Sensor-to-Edge path, indicating that under these streaming workloads the transport and endpoint characteristics dominated latency rather than the network path. Overall, the results indicate that the cost of securing data across different traffic patterns is negligible, that hardware acceleration benefits secure traffic streaming, and that hardware-induced limitations are reached more quickly when using TCP.

V. Discussion

Security comes at a cost, and this study set out to quantify that cost in order to provide practical insights for researchers and practitioners designing secure communication architectures for smart-city IoT deployments. To this end, we experimentally compared two transport protocols and multiple server authentication configurations under realistic operating conditions.

Across the experiments, the primary latency cost of security was observed during connection establishment. Experiments (1) and (2) show that authentication and key exchange dominate end-to-end latency differences, whereas Experiment (3) demonstrates that once connections are established, the performance gap between plaintext and secured communication becomes negligible. This indicates that the employed cryptographic mechanisms are efficient and deterministic, and that encryption does not meaningfully impact steady-state latency across different traffic patterns.

These findings have direct implications for smart-city IoT deployments, where large numbers of resource-constrained devices periodically establish connections to edge infrastructure. QUIC's lower per-connection establishment latency can improve responsiveness for sporadic or event-driven applications, such as mobile nodes, alarms, or on-demand sensing. However, in scenarios involving bursts of concurrent connections, such as after power outages, network disruptions, or in highly

TABLE I

SUMMARY OF OBSERVED END-TO-END LATENCY ACROSS TRAFFIC PATTERNS AND STREAMING BEHAVIOR USING PLAINTEXT MQTT OVER TCP AND SECURE MQTT OVER QUIC. ALL EXPERIMENTS USE RPI5 ENDPOINT PLATFORMS. QUIC EMPLOYS RSA-2048 SERVER AUTHENTICATION.

Direction	Traffic Class	TCP Behavior	QUIC Behavior	Key Observation
Sensor-to-Edge	Periodic, low rate ($\leq$100 msg/s)	Stable, low tail latency	Stable, low tail latency	No observable security-related latency penalty
Sensor-to-Edge	Periodic, moderate rate ($\leq$1000 msg/s)	Stable, low tail latency	Stable, low tail latency	Comparable p99 latency across transports
Sensor-to-Edge	Periodic, high rate ($\geq$5000 msg/s)	Collapsed (severe tail amplification)	Limited tail-latency growth observed	TCP collapses under load; QUIC degrades gracefully
Sensor-to-Edge	Bursty	Moderate tail amplification	Moderate tail amplification	QUIC generally exhibits more predictable tails
Sensor-to-Edge	Sporadic	Stable	Stable	Idle time dominates latency behavior
Sensor-to-Edge (Crypt. HW off)	Periodic (all rates)	Higher baseline latency	Higher baseline latency	Cryptographic hardware acceleration affects absolute latency
Cloud-Edge	As previous, but lower rates	Similar to Sensor-to-Edge	Similar to Sensor-to-Edge	Network path not dominant latency factor

dynamic environments, the connection scale experiments show that QUIC may incur higher tail latency. While such behavior may be acceptable for many IoT applications, expected connection patterns should be carefully considered when selecting a transport protocol.

Security configuration choices further influence latency. In particular, the server authentication key type and size had a significant impact, with elliptic-curve-based schemes consistently outperforming RSA-based alternatives. In addition, computing capability played a major role, with more powerful endpoints[3] yielding substantial latency reductions across all configurations. Taken together, these results highlight the need to balance endpoint computational resources, expected connection dynamics, timing requirements, and security when designing communication architectures for smart-city IoT systems.

The evaluated RPi5 ARM-based platform supports acceleration of symmetric AES and SHA operations [24]. As expected, this had no measurable impact on connection establishment latency in Experiments (1) and (2), where performance is dominated by asymmetric cryptographic operations. In contrast, a clear effect was observed in Experiment (3), where sustained encrypted traffic benefited from reduced per-message processing overhead. Hardware support for asymmetric cryptography also exists, which could be considered depending on constraints such as cost, performance requirements, and energy consumption.

It was further observed in Experiment (3) that increasing the message rate in combination with the payload size (i.e., throughput) beyond a certain point led to hardware-imposed limitations. Under such high loads, the system exhibited message loss, broker instability, memory pressure, and, in some cases, process termination. These observations indicate that achievable throughput in high-rate streaming scenarios is bounded not only by protocol behavior but also by the computational and memory resources of the underlying hardware platform.

Several factors were also evaluated and found to have negligible impact on the observed latency behavior. Real-time optimizations using a PREEMPT_RT kernel and standard system tuning did not yield meaningful improvements, as end-to-end latency was dominated by components outside strict real-time control, including the wireless network, the Linux networking stack, and software libraries. Likewise, no systematic latency difference was observed between TLS 1.2 and TLS 1.3. Although TLS 1.3 introduces a streamlined handshake intended to reduce latency, both versions negotiated equivalent cryptographic primitives under the tested configuration, resulting in comparable computational costs.

Since only server authentication was evaluated, unauthenticated clients could potentially connect to the system. While mTLS could mitigate this risk, it may introduce practical challenges related to device constraints and certificate lifecycle management. More generally, TLS addresses only one aspect of system security, secure deployments additionally require robust software update mechanisms, authorization, sound operational practices, etc.

Finally, it should be noted that all measurements include broker processing and therefore reflect end-to-end MQTT behavior. The observed QUIC behavior reflects the evaluated implementation and configuration, and other QUIC stacks may exhibit different characteristics. Furthermore, the experiments were conducted in a limited set of network environments and do not capture all deployment scenarios. Energy consumption and CPU utilization were not evaluated, although both remain important considerations for resource-constrained devices.

Threats towards validity

This study has several general limitations that should be considered when interpreting the results. The experiments

[3]RPi5 is marketed as being approximately 2-3× faster than its predecessor [23].

were conducted for a finite number of runs and durations, while repetition was used to improve robustness, longer-running experiments could further reduce variance, particularly for tail-latency behavior under high load. The observed performance reflects the specific software implementations, hardware platforms, and OS configurations evaluated in this study. Lightweight MVP publisher and subscriber applications and identical automation workflows were used to minimize application-level bias. Custom Bash scripts were developed to automate experiment execution, including controlled start/stop and logging of processes across all nodes. This approach reduced human-induced variability and improved repeatability. However, protocol implementations still differ in internal control flow and buffering behavior. In addition, application-level timestamping introduces unavoidable software overhead. The edge-based MQTT broker was deployed inside Docker containers, introducing an additional networking layer. While this overhead was small at low message rates (below 1 ms), it may become more pronounced under higher load and affect absolute latency measurements. Network conditions were not fully controlled. The Sensor-to-Edge path relied on Wi-Fi, and the Cloud-to-Edge path traversed the public internet, meaning that background interference, contention, and cross-traffic could influence latency. Although relative trends were consistent, stricter network isolation could reduce external variability.

VI. Conclusion and Future Work

This study evaluated the performance cost of secure communication for MQTT using TCP and QUIC across different platforms, security configurations, and traffic patterns. The results show that the primary cost of security arises during connection establishment, where transport choice, authentication algorithm, key type and size, and endpoint computing capability significantly influence latency. In contrast, once connections are established, encrypted data transmission incurs negligible overhead across all evaluated traffic patterns. QUIC consistently reduced per-connection establishment by up to $3.2\times$ reduced latency compared to TCP and exhibited a linear increased tail latency under concurrent connection bursts. Elliptic-curve-based authentication schemes and more capable hardware substantially reduced latency across configurations. Transport protocol and security configuration should be selected based on expected connection dynamics, timing requirements, and endpoint capabilities. We believe the findings are particularly useful in event-driven smart-city applications where maintaining a continuous connection is challenging. Future work could encompass evaluating CPU and energy usage, certificate management, mTLS mechanisms, or QUIC head-of-line blocking benefits in smart-city IoT environments.

Acknowledgment

This work is supported by Vinnova through the project "iSecure" (2023-01899), by the Key Digital Technologies Joint Undertaking through the project "MATISSE" (101140216), by Energimyndigheten through the project "i-EVsFLEX", and by the Clean Energy Transition Partnership through the project "FLEXI" (101069750).

References

[1] T. W. Nowak, M. Sepczuk, Z. Kotulski, W. Niewolski, R. Artych, K. Bocianiak, T. Osko, and J.-P. Wary, "Verticals in 5G MEC-Use Cases and Security Challenges," *IEEE Access*, vol. 9, pp. 87 251–87 298, Jun. 2021.

[2] A. Aly, F. Khomh, and Y.-G. Guéhéneuc, "Enforcing Security in Internet of Things Frameworks: A Systematic Literature Review," *Internet Things*, vol. 6, p. 100050, Jun. 2019.

[3] S. Leclerc, A. Bucaioni, and M. Ashjaei, "Characterizing Time-Critical Internet of Things," *Internet Things*, vol. 34, p. 101721, Nov. 2025.

[4] G. Pettorru and M. Martalà, "A Persistent and Secure Publish-Subscriber Architecture for Low-Latency IoT Communications," *IEEE Trans. Netw. Serv. Manag.*, vol. 23, pp. 729–739, Nov. 2026.

[5] A. Langley et al., "The QUIC Transport Protocol: Design and Internet-Scale Deployment," in *Proc. of the Conf. of the ACM Special Interest Group on Data Commun.*, 2017, pp. 183—196.

[6] L. Almeida, R. P. Pinto, and B. M. C. Silva, "Implementation and Performance Analysis of QUIC Protocol in IoT Monitoring Networks," in *Proc. IEEE Int. Mediterranean Conf. Commun. Netw.*, 2025, pp. 1–6.

[7] D. Saif and A. Matrawy, "An experimental investigation of tuning QUIC-based publish–subscribe architectures in IoT," *IEEE Internet Things J.*, vol. 11, no. 3, pp. 4924–4933, Feb. 2023.

[8] S. Leclerc, E. Hansen, A. Bucaioni, and M. Ashjaei, "Reducing IoT Data at the Edge: A Comparative Evaluation," in *Proc. IEEE Int. Conf. Ind. Technol. (ICIT)*, 2026, pp. 1–6.

[9] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," Internet Requests for Comments, RFC Editor, RFC 8446, Aug. 2018. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8446.html

[10] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," Internet Requests for Comments, RFC Editor, RFC 9000, May. 2021. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9000.html

[11] I. Butun, P. Österberg, and H. Song, "Security of the Internet of Things: Vulnerabilities, Attacks, and Countermeasures," *IEEE Commun. Surv. and Tut.*, vol. 22, no. 1, pp. 616–644, Nov. 2020.

[12] C. C. Byers, "Architectural Imperatives for Fog Computing: Use Cases, Requirements, and Architectural Techniques for Fog-Enabled IoT Networks," *IEEE Commun. Mag.*, vol. 55, no. 8, pp. 14–20, Aug. 2017.

[13] T. K. Boppana and P. Bagade, "Security Risks in MQTT-based Industrial IoT Applications," in *Proc. IEEE Int. Conf. Omni-layer Intell. Syst.*, 2022, pp. 1–5.

[14] M. Has, M. Kušek, and I. P. Žarko, "Evaluating Secure Variants of the MQTT Protocol on Resource-Constrained Devices for Precision Agriculture," in *Proc. IEEE Int. Conf. Telecommun. (ConTEL)*, 2023, pp. 1–7.

[15] Oracle, "Oracle Cloud Infrastructure," [Online]. Available: https://www.oracle.com/cloud, 2026, accessed: Apr. 27, 2026.

[16] EMQ Technologies, "NanoSDK," [Online]. Available: https://github.com/emqx/NanoSDK, 2024, accessed: Apr. 27, 2026.

[17] nanomsg contributors, "NNG: Nanomsg Next Generation," [Online]. Available: https://github.com/nanomsg/nng, 2025, accessed: Apr. 27, 2026.

[18] Microsoft, "MsQuic," [Online]. Available: https://github.com/microsoft/msquic, 2026, accessed: Apr. 27, 2026.

[19] F. Weisbecker, "CPU Isolation: Introduction, Part 1," [Online]. Available: https://www.suse.com/c/cpu-isolation-introduction-part-1, 2022, accessed: Apr. 27, 2026.

[20] EMQ Technologies, "EMQX Documentation," [Online]. Available: https://docs.emqx.com/en/emqx/latest, 2026, accessed: Apr. 27, 2026.

[21] Docker Inc., "Install Docker Engine on Debian," [Online]. Available: https://docs.docker.com/engine/install/debian, 2026, accessed: Apr. 27, 2026.

[22] Eclipse Foundation, "Mosquitto MQTT Broker," [Online]. Available: https://github.com/eclipse-mosquitto/mosquitto, 2026, accessed: Apr. 27, 2026.

[23] Raspberry Pi Ltd, "Raspberry Pi 5," *raspberrypi.com*, 2023. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-5/. (accessed 2026-04-27).

[24] Arm Ltd., "Arm Cortex-A76 Cryptographic Extension," *developer.arm.com*, 2023. [Online]. Available: https://developer.arm.com/documentation/100801/0401/Functional-description/About-the-Cryptographic-Extension. (accessed 2026-04-27).