Master's thesis

Master's Programme in Computer Science

Benchmarking Cloud-Edge IoT Communication Protocols in a Hybrid Docker-Kubernetes Architecture

Leevi Leinonen

June 18, 2025

Faculty of Science

University of Helsinki

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Leevi Leinonen

Työn nimi — Arbetets titel — Title

Benchmarking Cloud-Edge IoT Communication Protocols in a Hybrid Docker-Kubernetes Architecture

Ohjaajat — Handledare — Supervisors

Prof. Mika Mäntylä

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	June 18, 2025	142 pages

Tiivistelmä — Referat — Abstract

The choice of communication protocol is critical for Internet of Things (IoT) systems, especially in cloud-edge architectures, significantly impacting performance, latency, and resilience. This thesis conducts a systematic empirical performance evaluation of five key IoT protocols: MQTT (QoS 0, 1, 2), ZeroMQ, AMQP (RabbitMQ), CoAP (Non-confirmable messages), and gRPC.

A containerized cloud-edge testbed using Docker and Kubernetes was implemented to simulate realistic deployments. The evaluation measured end-to-end latency and message throughput under increasing message frequencies, varying payload sizes, and diverse network impairments including packet loss and incrementally increasing static latency. Qualitative network traffic analysis was also performed.

Key findings highlight distinct performance trade-offs. ZeroMQ demonstrated superior high-frequency throughput and low latency. AMQP and MQTT QoS 0 also offered strong throughput. Conversely, MQTT QoS 1 and 2, ensuring reliability, incurred significant performance penalties. CoAP (NON) provided the lowest latency for delivered messages in impaired networks but suffered message loss and struggled with larger payloads. gRPC showed good performance at moderate loads. Crucially, no single protocol proved universally optimal; selection must be tailored to specific application needs regarding reliability, latency, data volume, and network quality.

This research contributes a robust testbed design, a comprehensive comparative performance dataset, and data-driven guidance for protocol selection in modern cloud-edge IoT systems.

ACM Computing Classification System (CCS)

Networks → Network performance evaluation → Network measurement

Computer systems organization → Architectures → Distributed architectures

Networks → Network protocols → Application layer protocols

Avainsanat — Nyckelord — Keywords

IoT, Edge Computing, Cloud Computing, Performance Metrics, Kubernetes, Docker

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Software study track

Contents

1 Introduction **1**

2 Background and Related Work **4**

2.1 Introduction 4
2.2 IoT Communication Protocols 5
2.2.1 Message Queuing Telemetry Transport (MQTT) 5
2.2.2 ZeroMQ (ZMQ) 9
2.2.3 Advanced Message Queuing Protocol (AMQP) - RabbitMQ 12
2.2.4 Constrained Application Protocol (CoAP) 14
2.2.5 gRPC (Google Remote Procedure Call) 16
2.3 Overall Comparative Analysis of IoT Protocols 18
2.3.1 Architecture and Communication Paradigms 18
2.3.2 Performance Characteristics 19
2.3.3 Practical Deployment Considerations 21
2.3.4 Guidelines for Protocol Selection 22
2.3.5 Concluding Remarks on Comparative Analysis 23
2.4 Cloud-Edge Computing Architectures 24
2.4.1 Cloud-Centric vs. Edge-Centric IoT Architectures 24
2.4.2 Containerization and Kubernetes in IoT 27
2.5 Performance Metrics and Benchmarking in IoT 29
2.5.1 Latency, Jitter, and Throughput 29
2.5.2 Message Ordering and Integrity 30
2.5.3 Resource Utilization and Network Overhead 31
2.6 Related Work 32
2.6.1 Comparative Studies on IoT Communication Protocols 32
2.6.2 Cloud-Edge Benchmarking in Prior Research 33
2.6.3 Gaps in Existing Research 34
2.7 Summary 35

3 System Design and Methodology **36**
3.1 Introduction 36
3.2 Overall System Architecture 36
3.2.1 Edge Layer: Sensor Data Generation and Protocol Connectors . . . 39
3.2.2 Cloud Layer: Kubernetes-Based Processing Services 39
3.3 Sensor Source Module 41
3.4 Protocol-Specific Connectors 43
3.4.1 Broker-Based Protocols 43
3.4.2 Brokerless Protocols 44
3.4.3 Implementation Considerations 45
3.5 Cloud Backend Implementation on Kubernetes 45
3.5.1 Message Processing Architecture 46
3.5.2 Handling Brokered and Brokerless Protocols 46
3.5.3 Scalability and Reliability in Kubernetes 47
3.6 Experimental Procedure 48
3.6.1 Baseline Performance and Load Characterization 48
3.6.2 Network Impairment Resilience Testing 49
3.6.3 General Experimental Conduct and Data Collection 50
3.7 Performance Metrics 51
3.7.1 Latency, Throughput, and Message Integrity 51
3.7.2 Resource Utilization and Monitoring Framework 52
3.8 Summary 52

4 Implementation Details **54**
4.1 Edge-Side Implementation and Components 54
4.1.1 Sensor Source Implementation 54
4.1.2 Protocol-Specific Cloud Connectors 58
4.1.3 Edge Docker Setup 64
4.2 Cloud-Side Deployment and Infrastructure 67
4.2.1 Kubernetes Deployment Architecture 68
4.2.2 Cloud Message Brokers 68
4.2.3 Cloud Connector Services 72
4.2.4 Deployment Configuration Files 76
4.3 Instrumentation and Logging Setup 78
4.3.1 Prometheus Metric Collection 79

4.3.2 Metrics Collection in Protocol-Specific Readers 80
4.3.3 Implementation of Prometheus Metrics in Cloud Readers 81
4.4 Testing Tools and Environment Setup 83
4.4.1 Simulating Network Conditions 83
4.4.2 Prometheus Query Language (PromQL) 84
4.4.3 Grafana Dashboards 86
4.5 Conclusion 87

5 Performance Evaluation 88
5.1 Experimental Setup and Methodology Recap 88
5.1.1 Testbed Configuration 88
5.1.2 Testing Scenarios 89
5.1.3 Metrics 89
5.2 Network-Level Protocol Analysis with Wireshark 90
5.2.1 MQTT (Message Queuing Telemetry Transport) 90
5.2.2 ZeroMQ (ZMQ) 92
5.2.3 AMQP (Advanced Message Queuing Protocol) via RabbitMQ . . . 92
5.2.4 CoAP (Constrained Application Protocol) 93
5.2.5 gRPC (Google Remote Procedure Call) 93
5.2.6 Comparative Observations from Wireshark Packet Exchanges . . . 94
5.3 Baseline Performance Comparison 96
5.3.1 Latency and Throughput Under Increasing Load 96
5.3.2 Summary of Baseline Observations 100
5.4 Performance Under Network Impairment: 10% Packet Loss 101
5.4.1 Impact on Latency and Message Throughput 102
5.4.2 Summary of Performance with 10% Packet Loss 104
5.5 Impact of Increasing Message Payload Size 104
5.5.1 Observed Latency, Message, and Data Throughput 107
5.5.2 Summary of Performance with Increasing Payload Size 109
5.6 Impact of Increasing Static Network Latency 110
5.6.1 Effect on Latency and Message Throughput 112
5.6.2 Summary of Performance with Increasing Static Network Latency . 114
5.7 Performance Under Combined Network Impairments 114
5.7.1 Observed Behavior under Mixed Impairments 116
5.7.2 Summary of Performance with Combined Impairments 118

5.8 Discussion of Results 118
5.8.1 Dominance of Transport Layer and Reliability Mechanisms 119
5.8.2 Throughput, Scalability, and Bottlenecks 120
5.8.3 Resilience to Network Impairments 120
5.8.4 Protocol-Specific Nuances 121
5.9 Chapter Conclusion 122

6 Analysis and Discussion 124
6.1 Revisiting Research Questions 124
6.1.1 RQ1: End-to-End Latencies under Baseline and High-Load Conditions 124
6.1.2 RQ2: Message Throughput Variation and Scalability Bottlenecks . 125
6.1.3 RQ3: Impact of Network Impairments on Reliability and Latency . 126
6.2 Key Performance Trade-offs and Protocol Suitability 128
6.2.1 Reliability vs. Latency and Throughput 128
6.2.2 Impact of Payload Size 129
6.2.3 Broker-based vs. Brokerless Architectures 129
6.2.4 Suitability for Containerized Cloud-Edge Deployments 130
6.3 Alignment with Existing Research 131

7 Conclusion and Future Work 133
7.1 Summary of Key Findings 133
7.2 Answering Research Questions and Contributions 134
7.3 Limitations of the Study 135
7.4 Future Work 136
7.5 Concluding Remarks 136

Bibliography 139

1 Introduction

The Internet of Things (IoT) is rapidly transforming industries by enabling real-time monitoring, automation, and data-driven decision-making. From smart homes and wearable health trackers to industrial automation and precision agriculture, connected devices are generating unprecedented volumes of data. By 2030, the number of connected IoT devices is projected to surpass 125 billion (Moraes et al., 2019), creating significant challenges for data transmission and processing. The choice of communication protocol is critical in these deployments, directly impacting system performance, reliability, and scalability, particularly in resource-constrained environments (Çorak et al., 2018). Delays in transmitting data from a medical sensor, for instance, could lead to delayed interventions, while packet loss in an industrial control system could disrupt critical operations. Therefore, understanding the performance trade-offs of different IoT communication protocols is paramount.

Traditional cloud-centric architectures, where IoT devices send data directly to remote data centers, often suffer from high latency, network congestion, and inefficient energy use (Chan et al., 2022). Edge computing, which brings computation closer to the data source, offers a compelling alternative, reducing latency and bandwidth consumption (Chan et al., 2022). Hybrid cloud-edge architectures, combining the benefits of both paradigms, are becoming increasingly prevalent. In these architectures, IoT communication protocols are the essential enablers of efficient and reliable data exchange between devices and cloud services. However, the performance of these protocols under realistic cloud-edge conditions, particularly within containerized deployments, remains an open research question.

This thesis addresses this gap by conducting a systematic performance evaluation of five widely used IoT communication protocols: MQTT, ZeroMQ, AMQP (using RabbitMQ), CoAP, and gRPC. These protocols represent a diverse range of architectural approaches, including publish-subscribe (MQTT), message queuing (AMQP, ZeroMQ), and Remote Procedure Call (RPC) (gRPC), as well as different transport mechanisms (TCP, UDP). The evaluation is performed within a realistic cloud-edge testbed, built using Docker and Kubernetes, to reflect modern deployment practices. Crucially, the experimental setup simulates real-world IoT constraints by enforcing outbound-only connections from edge devices, mimicking scenarios where devices are behind firewalls or NAT.

This research is guided by the following key research questions:

1. What are the average and 95th percentile end-to-end latencies of MQTT (with QoS levels 0, 1, and 2), ZeroMQ, AMQP (RabbitMQ), CoAP, and gRPC under baseline and high-load conditions in a containerized cloud-edge IoT environment?
2. How does message throughput vary for each protocol as the number of simulated sensors and message frequency increase, and what are the primary bottlenecks limiting scalability?
3. What is the impact of network impairments (added latency, packet loss, and jitter) on the message delivery reliability and end-to-end latency of each protocol?

These questions are designed to be specific, measurable, and directly address the performance characteristics of the protocols under investigation.

The primary contributions of this research are:

- **Cloud-Edge IoT Testbed:** Development of a standardized, containerized testbed for evaluating IoT communication protocols in a realistic cloud-edge environment, leveraging Docker and Kubernetes.
- **Comprehensive Performance Evaluation:** A quantitative comparison of latency, jitter, throughput, message reliability, and resource utilization across the five protocols under varying network conditions and load scenarios.
- **Protocol Trade-off Analysis:** Empirical insights into the trade-offs between publish-subscribe, message queuing, and RPC-based communication models in the context of cloud-edge IoT deployments.
- **Practical Guidance:** Data-driven recommendations for selecting appropriate IoT communication protocols based on specific application requirements and deployment constraints.

The remainder of this thesis is organized as follows. Chapter 2 provides a comprehensive review of relevant background information and related work, covering IoT communication protocols, cloud-edge computing, and performance benchmarking. Chapter 3 details the system architecture and experimental methodology. Chapter 4 describes the implementation of the testbed, including the edge and cloud components, instrumentation, and

testing tools. Chapter 5 presents the empirical results and analysis of the performance evaluation. Chapter 6 will provide a detailed discussion. Finally, Chapter 7 concludes the thesis, summarizing the key findings and outlining future research directions.

2 Background and Related Work

2.1 Introduction

The rapid expansion of the Internet of Things (IoT) has led to a proliferation of devices and services generating and exchanging massive amounts of data. As these systems scale, ensuring reliable and efficient communication between heterogeneous devices, edge nodes, and cloud infrastructures becomes increasingly critical. A wide range of protocols has emerged to facilitate machine-to-machine (M2M) exchanges—each offering distinct advantages and trade-offs in terms of latency, reliability, resource utilization, and security.

This chapter provides a comprehensive overview of the key technologies and concepts shaping modern IoT deployments. First, we examine major IoT messaging protocols, including MQTT, ZeroMQ, AMQP (RabbitMQ), CoAP, and gRPC, highlighting their architectural differences, communication patterns, and suitability for various application requirements. We pay particular attention to how each protocol handles quality of service (QoS), message routing, and security—factors that directly influence performance in both resource-constrained and large-scale environments.

Next, we explore cloud-edge computing architectures, emphasizing the shift away from strictly centralized cloud models toward hybrid deployments that bring computational resources closer to data sources. We discuss the benefits and challenges of distributing workloads across cloud and edge layers, such as reduced latency, lower bandwidth usage, and enhanced resilience. The role of containerization (Docker) and orchestration (Kubernetes) in these architectures is also detailed, showing how lightweight, portable containers and automated scaling mechanisms can significantly improve fault tolerance and resource allocation in complex IoT systems.

Finally, we outline key performance metrics—including latency, jitter, throughput, and resource utilization—that are crucial for benchmarking IoT protocols and architectures. Existing comparative studies on messaging protocols are reviewed to underscore current knowledge gaps, particularly regarding their behavior in Kubernetes-based cloud-edge environments. By synthesizing these foundational elements, this chapter lays the groundwork for the experimental evaluations that follow, where we will systematically assess how

different IoT protocols perform under real-world conditions in containerized, distributed deployments.

2.2 IoT Communication Protocols

Efficient communication between IoT devices and cloud-based infrastructures is a critical factor in the design of scalable and high-performance IoT systems. Different protocols have been developed to facilitate machine-to-machine (M2M) communication, each with distinct characteristics in terms of messaging architecture, reliability, latency, and resource utilization. This section provides an overview of the main IoT communication protocols examined in this study, including MQTT, ZeroMQ, gRPC, AMQP, and CoAP.

2.2.1 Message Queuing Telemetry Transport (MQTT)

Message Queuing Telemetry Transport (MQTT) is a lightweight machine-to-machine (M2M) communication protocol designed for constrained environments such as the Internet of Things (IoT). Originally developed by IBM in 1999, it has since been standardized under OASIS as MQTT version 5.0 (Standard, 2019). The protocol follows a publish-subscribe model and operates over TCP/IP to provide reliable message delivery.

MQTT Architecture

The MQTT protocol is designed around a publish-subscribe architecture, which enables asynchronous communication between devices. This model eliminates the need for direct connections between message producers (publishers) and consumers (subscribers), thereby reducing network congestion and improving scalability in distributed systems. A typical MQTT communication flow, illustrating these interactions, is depicted in Figure 2.1.

An MQTT system consists of three primary components:

- *Publisher*: A device that sends messages to a topic managed by a broker. Examples in Figure 2.1 include ‘`Sensor 1`‘, ‘`Sensor 2`‘, and ‘`Sensor 3`‘.
- *Broker*: A central entity (the ‘`MQTT Broker`‘ in Figure 2.1) responsible for receiving messages from publishers and forwarding them to subscribers based on their topic subscriptions.

- *Subscriber*: A client that registers interest in specific topics and receives messages when they are published, such as the `‘Dashboard‘` or `‘Mobile App‘` shown in Figure 2.1.

Topics define the logical channels through which messages are routed. Publishers send messages to a specific topic (e.g., `home/temp`, `home/co2` in Figure 2.1), while subscribers express interest in one or more topics to receive relevant messages. MQTT also supports wildcard subscriptions, allowing clients to subscribe to multiple related topics dynamically. For instance, a subscriber registered to `home/#` (as the `‘Mobile App‘` is in Figure 2.1) will receive messages from all subtopics under `home/`, including temperature, humidity, and air quality readings if published to respective subtopics.

The example in Figure 2.1 further clarifies this: multiple sensors publish environmental data to the broker under distinct topics (like `home/temp`, `home/co2`, `home/pressure`). Different subscribers, such as a monitoring dashboard, an industrial control system (like the `‘Ventilation System‘`), and a data logging service (`‘Storage DB‘`), then receive only the messages relevant to their specific topic subscriptions, demonstrating efficient and targeted data distribution.

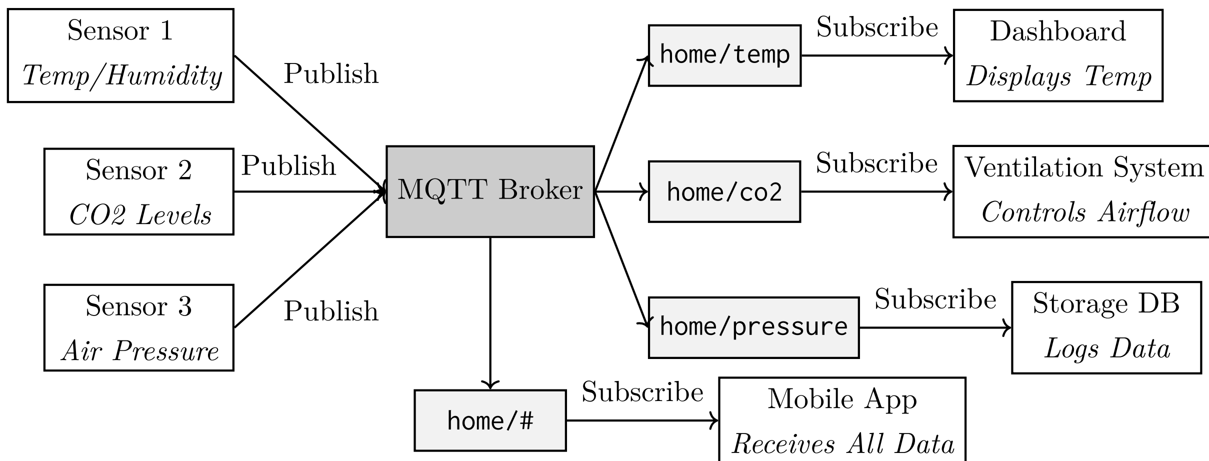

Figure 2.1: MQTT Architecture: Publish-Subscribe Model with Topics

Quality of Service in MQTT

To ensure reliable message delivery, MQTT defines three levels of Quality of Service (QoS), which determine how messages are transmitted between clients and brokers. These QoS levels are summarized in Table 2.1.

Table 2.1: MQTT Quality of Service Levels

QoS Level	Description	Message Duplication
0	At most once (fire and forget)	Possible
1	At least once (acknowledged delivery)	Yes
2	Exactly once (four-step handshake)	No

QoS 0 provides the lowest overhead but does not guarantee message delivery, making it suitable for applications where occasional packet loss is acceptable. QoS 1 ensures that messages arrive at least once, but duplicate messages may occur if acknowledgments are lost. QoS 2 guarantees exactly-once delivery using a four-step exchange, making it the most reliable but also the most resource-intensive option (Standard, 2019).

Several studies have evaluated the trade-offs between different QoS levels. Research has shown that while higher QoS levels improve reliability, they introduce additional latency and resource consumption (Moraes et al., 2019). For example, in scenarios with high network congestion, QoS 2 can lead to increased message delays due to its additional acknowledgment overhead. In contrast, QoS 0 is more efficient in low-latency applications but is unsuitable for mission-critical systems where message loss cannot be tolerated (Bender et al., 2021).

Performance and Scalability

MQTT is widely adopted in IoT deployments due to its efficiency in low-bandwidth environments. However, its reliance on a centralized broker introduces scalability challenges, particularly in large-scale cloud-edge architectures. Studies have found that as the number of connected devices increases, broker performance becomes a bottleneck, impacting overall system responsiveness (Bender et al., 2021).

Performance evaluations focusing on MQTT indicate that it can introduce additional communication delays compared to some protocols that avoid broker mediation (Moraes et al., 2019). While the broker-based model simplifies message routing, it requires sufficient computational resources to handle high message throughput. Broker clustering and load balancing mechanisms have been proposed to mitigate these limitations, but they increase system complexity and operational costs.

In terms of latency, MQTT performs well under normal network conditions but can experience significant degradation in high-traffic scenarios. Studies have shown that MQTT

exhibits higher latency under sustained loads, partly due to its reliance on TCP connection overhead (Bender et al., 2021). Protocols that operate over UDP, such as CoAP, can offer lower latency in lossy networks but at the expense of guaranteed delivery (Moraes et al., 2019). Furthermore, a recent open-source evaluation of MQTT brokers found that implementation language and broker architecture significantly affect resource usage, throughput, and latency, highlighting the importance of deployment-specific testing for large-scale MQTT-based systems (Bender et al., 2021).

Security Considerations

One of the main limitations of MQTT is its lack of built-in security mechanisms. The protocol does not natively support end-to-end encryption, making it vulnerable to man-in-the-middle attacks and unauthorized message interception (Standard, 2019). While TLS/SSL can be used to encrypt MQTT communication, this requires additional configuration and computational resources, which may be impractical for resource-constrained IoT devices.

Authentication in MQTT is typically handled through username-password mechanisms, but these are insufficient for securing large-scale deployments without additional measures such as token-based authentication or certificate-based security models (Bender et al., 2021). Furthermore, MQTT brokers can become single points of failure if not properly secured against denial-of-service (DoS) attacks.

Security enhancements introduced in MQTT 5.0 include support for enhanced authentication methods and fine-grained access control policies. However, these features require application-layer implementations and do not inherently protect against network-level threats. Research suggests that additional security layers, such as hardware-based encryption or decentralized identity management, may be necessary to secure MQTT-based IoT deployments (Moraes et al., 2019).

In summary, MQTT remains a widely used protocol for IoT communication, especially in scenarios that require minimal bandwidth and can tolerate broker-based architectures. The following sections will discuss other protocols, such as ZeroMQ, AMQP, CoAP, and gRPC, before a broader comparison of their respective trade-offs.

2.2.2 ZeroMQ (ZMQ)

ZeroMQ (ZMQ) is a high-performance, asynchronous messaging library that enables direct peer-to-peer communication without requiring a central broker. Unlike MQTT's brokered publish-subscribe model, ZeroMQ is a decentralized messaging framework designed for low-latency, high-throughput applications. Originally developed by iMatix and maintained as an open-source project, ZeroMQ provides a flexible socket-based API that abstracts network communication complexities (Hintjens et al., 2011; Lauener et al., 2017).

Message Queuing Architecture

ZeroMQ operates as a lightweight messaging layer that supports multiple transport mechanisms, including TCP, inter-process communication (IPC), in-process messaging, and multicast. It is not a traditional message broker but instead functions as a distributed messaging library that enables direct communication between endpoints. This brokerless model reduces dependency on a central server and minimizes communication overhead, making ZeroMQ suitable for real-time systems and large-scale distributed architectures (Pamadi et al., 2020).

ZeroMQ supports multiple messaging patterns, allowing developers to implement different communication models based on application requirements. These patterns include:

- *Publish-Subscribe*: Allows multiple subscribers to receive messages from a publisher without requiring a broker.
- *Request-Reply*: Implements a synchronous message exchange model where clients send requests and wait for server responses.
- *Pipeline (Push-Pull)*: Facilitates distributed task execution by pushing messages to worker nodes.
- *Pair Communication*: Establishes a one-to-one bidirectional messaging channel between two endpoints.
- *Router-Dealer*: A flexible, asynchronous pattern where a ROUTER socket manages multiple DEALER sockets, providing dynamic request routing and load balancing (Hintjens et al., 2011; Lauener et al., 2017).

ZeroMQ is fundamentally brokerless, which means it does not require a dedicated server to mediate message exchanges. However, its ROUTER-DEALER pattern can emulate broker-like behavior through a 'soft broker' model. In this model, the ROUTER socket dynamically routes messages from multiple DEALER sockets, effectively handling load balancing and message addressing without a centralized broker. As detailed in (Hintjens et al., 2011; Lauener et al., 2017), this approach offers lower latency and greater scalability, though it shifts some responsibility for ensuring message reliability to the application layer, making it suitable for scalable, fault-tolerant cloud-edge architectures.

Performance and Scalability

Performance evaluations have demonstrated that ZeroMQ excels in high-throughput environments, particularly for applications requiring real-time data exchange. Benchmarking results show that ZeroMQ achieves lower message latency and higher message throughput than many broker-based protocols under heavy load conditions (Pamadi et al., 2020).

However, ZeroMQ lacks built-in quality of service (QoS) mechanisms, meaning that messages can be lost if a subscriber is offline or experiences network failure. The trade-off between reliability and performance must therefore be considered when selecting a protocol for cloud-edge IoT architectures. Nonetheless, ZeroMQ's brokerless architecture often provides efficient horizontal scaling by enabling direct peer-to-peer communication without the overhead of a central broker (Kang and Dubey, 2020).

The ROUTER-DEALER pattern further enhances ZeroMQ's scalability by allowing a single ROUTER node to handle multiple DEALER clients concurrently, implementing dynamic load balancing across worker nodes. This enables distributed message processing without the need for centralized message queues or brokers, reducing bottlenecks in high-load scenarios (Hintjens et al., 2011).

Some studies also emphasize that ZeroMQ's performance can vary significantly depending on the language bindings and socket patterns in use. Thorough testing of each deployment scenario is recommended to ensure optimal throughput and latency (Lauener et al., 2017).

Security Considerations

One of the major challenges of ZeroMQ is its lack of built-in security features. Unlike some brokered protocols that offer native encryption or authentication, ZeroMQ requires additional security layers to provide encryption, authentication, and access control. Implemen-

tations such as CurveZMQ can offer cryptographic security for ZeroMQ communication, but these must be explicitly integrated into applications (Lauener et al., 2017).

The lack of native security support means that ZeroMQ is not inherently suitable for public or untrusted networks without additional measures. While its peer-to-peer design can reduce latency and simplify routing, it places the onus on developers to incorporate adequate encryption (e.g., TLS or CurveZMQ), certificate management, and other protective mechanisms.

Use Cases and Deployment Scenarios

ZeroMQ is particularly well-suited for applications that require low-latency, high-throughput messaging, such as:

- *High-frequency financial trading*, where minimal communication delay is critical for executing trades.
- *Distributed monitoring and logging*, where multiple components generate large volumes of real-time log data.
- *Parallel computing and task distribution*, where computational workloads are assigned to multiple worker nodes.
- *Industrial automation and control systems*, where real-time sensor data must be processed with minimal delay.
- *Cloud-edge IoT deployments*, where the ROUTER-DEALER pattern allows asynchronous processing of sensor data across multiple edge nodes before sending results to cloud platforms.

In cloud-edge IoT systems, ZeroMQ can facilitate direct communication between edge devices and cloud services, thereby reducing latency compared to broker-based protocols. The ROUTER-DEALER pattern further enhances flexibility by enabling dynamic workload distribution across edge and cloud resources, making ZeroMQ a strong candidate for large-scale, decentralized IoT architectures (Hintjens et al., 2011; Kang and Dubey, 2020).

However, the lack of built-in message persistence and security mechanisms means that additional measures must be implemented to ensure reliable and secure data transmission. Future research continues to explore security solutions and adaptive QoS mechanisms to further optimize ZeroMQ for critical IoT applications.

2.2.3 Advanced Message Queuing Protocol (AMQP) - RabbitMQ

The Advanced Message Queuing Protocol (AMQP) is a message-oriented middleware protocol designed to facilitate reliable, flexible, and secure communication between distributed applications. Originally developed by JPMorgan Chase in 2003 and later standardized by the Organization for the Advancement of Structured Information Standards (OASIS), AMQP is widely used in enterprise environments for asynchronous messaging, message queuing, and workload distribution (Prajapati, 2021). RabbitMQ, one of the most commonly used open-source implementations of AMQP, provides a robust queuing mechanism, message persistence, and flexible routing.

AMQP Architecture and Messaging Model

Unlike simple publish-subscribe protocols, AMQP introduces a more structured architecture with exchanges, queues, and routing mechanisms. Producers publish messages to an exchange, which then determines how messages are routed to one or more queues based on predefined rules. Consumers retrieve messages from these queues at their own pace, ensuring reliable delivery even in cases of intermittent connectivity or consumer failures (Uy and Nam, 2019).

A distinguishing feature of AMQP is its support for multiple exchange types, allowing fine-grained control over message routing. The most commonly used are direct, fanout, topic, and headers exchanges, each serving different routing needs. This level of flexibility makes AMQP well-suited for distributed IoT applications that require prioritized message handling and workload balancing (Naik, 2017).

Reliability and Quality of Service

AMQP provides built-in mechanisms to ensure reliable message delivery, distinguishing it from simpler broker-based protocols. Message acknowledgments (ACKs), persistent storage, and transactional messaging allow AMQP to maintain message integrity even in cases of network failures or broker restarts (Luzuriaga et al., 2015). RabbitMQ supports message persistence by storing messages on disk rather than in volatile memory, ensuring durability across broker restarts. Additionally, acknowledgment mechanisms prevent message loss, as consumers must explicitly confirm receipt of messages before they are removed from the queue.

Compared to protocols without explicit ACK-based flows, AMQP inherently offers strong delivery guarantees but at the cost of increased processing overhead (Uy and Nam, 2019). This trade-off makes AMQP particularly effective in use cases where message durability is critical, such as industrial control systems and financial transactions.

Performance and Scalability

AMQP is widely adopted in scenarios requiring high throughput and strong reliability guarantees. Studies have found that while it can introduce higher latency in low-bandwidth environments due to additional processing steps in its queuing model, it provides better throughput and reliability under high-load conditions (Uy and Nam, 2019; Luzuriaga et al., 2015). Since AMQP stores messages until they are successfully consumed, it is well-suited for systems needing guaranteed delivery (Prajapati, 2021). However, its added overhead and more complex transaction model may make it less efficient for extremely resource-constrained devices or networks.

In environments with moderate to high message rates, AMQP's robust queuing and exchange-routing features shine. By decoupling producers and consumers, the protocol supports asynchronous communication patterns and can load-balance tasks across multiple consumers. Empirical evaluations also indicate that AMQP meets enterprise demands for security and message ordering, although it may not be the optimal choice in severely bandwidth-limited or power-constrained settings (Luzuriaga et al., 2015).

Security Considerations

Security is an important aspect of AMQP. It offers built-in support for encryption, authentication, and fine-grained access control policies. RabbitMQ, for instance, provides role-based access control (RBAC), allowing administrators to set permissions for individual users and applications (Naik, 2017). This security model ensures that only authorized entities can access and modify queues, reducing the risk of unauthorized data exposure.

The additional security features, such as TLS/SSL encryption and certificate-based authentication, can introduce computational overhead, which may pose challenges for constrained IoT deployments (Prajapati, 2021). Nevertheless, for enterprise-scale or cloud-based message processing where security is paramount, AMQP's built-in mechanisms facilitate secure data exchange and reduce the complexity of integrating external security layers.

Suitability for Cloud-Edge IoT Architectures

AMQP's reliability, flexible routing, and strong security mechanisms make it well-suited for cloud-based deployments requiring guaranteed message delivery and centralized control. It is frequently used in enterprise environments that demand message queuing and load balancing for large-scale distributed applications (Prajapati, 2021; Luzuriaga et al., 2015).

Edge computing scenarios, which may emphasize minimal overhead and real-time responsiveness, should weigh the benefits of AMQP's strong delivery guarantees against the additional latency introduced by its message queuing approach (Uy and Nam, 2019). Many organizations choose AMQP for components that manage critical or high-value data, leveraging its durability and security features in complex, mixed cloud-edge architectures.

2.2.4 Constrained Application Protocol (CoAP)

The Constrained Application Protocol (CoAP) is a lightweight application-layer protocol designed for constrained environments, including low-power, lossy networks (LLNs) and resource-constrained devices. CoAP was standardized by the Internet Engineering Task Force (IETF) under RFC 7252 (Shelby et al., 2014). The protocol follows a Representational State Transfer (REST) architecture, making it a compact and efficient alternative to HTTP for Internet of Things (IoT) applications.

CoAP Architecture

CoAP operates over the User Datagram Protocol (UDP), offering a lightweight communication model that minimizes protocol overhead compared to TCP-based alternatives (Thangavel et al., 2014). By leveraging UDP, CoAP reduces connection establishment latency and enables efficient communication in energy-constrained networks.

The protocol supports two primary message types: Confirmable (CON) and Non-Confirmable (NON) messages. Confirmable messages require an acknowledgment from the receiver, ensuring reliable delivery in unreliable networks. Non-Confirmable messages, on the other hand, are sent without expecting a response, optimizing performance for applications where occasional packet loss is acceptable (Shelby et al., 2014).

CoAP utilizes a request-response model, similar to HTTP, where clients issue requests (e.g., GET, POST, PUT, DELETE) to servers hosting resources identified by Uniform Resource Identifiers (URIs). Unlike HTTP, CoAP is designed for constrained environ-

ments and supports multicast communication, allowing a single request to be delivered to multiple recipients, reducing network traffic (Betzler et al., 2016).

Performance and Scalability

Studies have evaluated CoAP's performance in terms of latency, bandwidth consumption, and message reliability. Research by (Thangavel et al., 2014) demonstrates that CoAP often outperforms protocols with higher overhead in high-packet-loss environments, as its UDP-based design avoids the overhead of connection-oriented transport. In lower-packet-loss scenarios, connection-oriented protocols can sometimes exhibit lower end-to-end delay.

Experimental evaluations also indicate that CoAP reduces additional traffic overhead, especially when the message size is small and the network exhibits moderate packet loss ($\leq 25\%$) (Thangavel et al., 2014). This can make CoAP a favorable choice for battery-powered IoT devices requiring energy-efficient communication.

Reliability and Message Ordering

To mitigate the lack of built-in TCP reliability mechanisms, CoAP implements a retransmission and deduplication system. Confirmable messages are retransmitted with exponential backoff until an acknowledgment is received or a timeout occurs (Shelby et al., 2014). Additionally, CoAP uses Message IDs and Tokens to ensure each message is processed only once, reducing duplicate responses in lossy networks.

As CoAP does not guarantee strict message sequencing across multiple hops, applications that need strict in-order delivery can implement sequencing at the application layer (Martí et al., 2019).

Security Considerations

Security in CoAP is achieved through Datagram Transport Layer Security (DTLS), which provides encryption and authentication comparable to TLS but adapted for UDP communication (Shelby et al., 2014). DTLS ensures end-to-end confidentiality and integrity but may introduce additional computational overhead, which can be significant for constrained devices.

Compared to protocols that use TLS at the transport layer, CoAP's reliance on DTLS introduces extra handshake overhead, potentially increasing connection setup time (Martí

et al., 2019). However, DTLS is beneficial in lossy networks where TCP-based encryption methods may suffer from frequent retransmissions. Research also suggests that lightweight security mechanisms such as OSCORE (Object Security for Constrained RESTful Environments) may further enhance CoAP's security in IoT deployments (Shelby et al., 2014).

Use Cases and Deployment Scenarios

CoAP is widely used in applications where low power consumption and minimal bandwidth usage are critical. Examples include:

- *Smart energy and building automation*, where devices periodically report sensor data with minimal overhead (Shelby et al., 2014).
- *Environmental monitoring*, where sensors deployed in remote areas must operate efficiently over long periods (Martí et al., 2019).
- *Industrial IoT*, where CoAP's multicast capability reduces network congestion in large-scale deployments (Betzler et al., 2016).

Despite its strengths for lightweight IoT applications, CoAP's reliance on UDP may limit its suitability for deployments needing guaranteed message delivery or persistent connections, where other protocols with broker-based architectures or advanced QoS may be more appropriate (Thangavel et al., 2014).

2.2.5 gRPC (Google Remote Procedure Call)

Google Remote Procedure Call (gRPC) is an open-source communication framework designed to facilitate efficient and scalable remote interactions between applications. Built on top of HTTP/2, gRPC leverages Protocol Buffers (protobuf) for data serialization, allowing for high-performance communication with reduced overhead. While originally developed for microservice architectures, its potential applications in Internet of Things (IoT) environments have been increasingly explored (Kampars et al., 2021).

gRPC Architecture and Communication Model

Unlike traditional message-oriented protocols, gRPC follows an RPC-based model where clients invoke methods on remote servers as if they were local functions. It supports

various communication paradigms, including:

- *Unary RPC*: The client sends a single request and receives a single response.
- *Server Streaming RPC*: The client sends a request and receives multiple responses as a stream.
- *Client Streaming RPC*: The client sends a stream of requests and receives a single response.
- *Bidirectional Streaming RPC*: Both client and server exchange multiple messages asynchronously.

This flexibility enables gRPC to handle a wide range of IoT communication patterns, particularly in scenarios requiring efficient data streaming and low-latency interactions (Bolanowski et al., 2022).

Performance Considerations in IoT

gRPC's primary advantage over REST-based APIs and traditional messaging protocols lies in its efficiency. Studies indicate that gRPC significantly reduces latency and bandwidth consumption compared to RESTful HTTP, particularly when handling large data payloads (Bolanowski et al., 2022). The use of HTTP/2 multiplexing allows multiple requests and responses to be processed concurrently over a single connection, minimizing the overhead associated with establishing new connections.

However, despite these benefits, the suitability of gRPC for IoT applications remains a subject of debate. Unlike protocols such as MQTT, which are optimized for resource-constrained devices and unreliable networks, gRPC operates over TCP and lacks built-in mechanisms for message persistence and Quality of Service (QoS) guarantees. This makes it less suitable for lossy networks and constrained IoT environments (Kampars et al., 2021).

Security Considerations

gRPC incorporates built-in security mechanisms through SSL/TLS encryption, ensuring data integrity and confidentiality during transmission. It also supports token-based

authentication and other security frameworks, making it a secure option for cloud-to-edge communications (Arora et al., 2024). However, in edge-based IoT deployments, the overhead of maintaining secure TCP connections can introduce additional computational demands, which may not be ideal for battery-powered devices.

Use Cases in IoT

Given its characteristics, gRPC is particularly well-suited for scenarios requiring fast, structured data exchange between IoT edge nodes and cloud services. These include:

- *Cloud-based analytics*: IoT sensors transmitting structured data to cloud platforms for real-time processing.
- *Machine learning inference*: Edge devices sending data to cloud-based AI models for inference and response.
- *Industrial automation*: High-frequency control commands between edge controllers and centralized monitoring systems.

In contrast, for heavily constrained environments where ultra-low power usage and robust offline functionality are paramount, protocols such as MQTT or CoAP typically remain more suitable (Kampars et al., 2021).

2.3 Overall Comparative Analysis of IoT Protocols

The protocols discussed in the previous sections—MQTT, ZeroMQ, AMQP, CoAP, and gRPC—each offer distinct advantages and trade-offs for Internet of Things (IoT) deployments. Selecting the optimal protocol or protocol combination for a given scenario depends on multiple factors, including network reliability, power constraints, message delivery guarantees, scalability requirements, and security considerations. This section integrates insights from existing research to provide a broader, more holistic comparative analysis of these protocols.

2.3.1 Architecture and Communication Paradigms

Brokered vs. Brokerless MQTT and AMQP adopt a broker-based approach, centralizing message routing and facilitating publish-subscribe (MQTT) or queue-based (AMQP)

communication. In contrast, ZeroMQ is brokerless, typically operating in a peer-to-peer manner, thus offering lower latency and fewer architectural components but also placing responsibility for message reliability and topic management on the application layer (Lauener et al., 2017; Pamadi et al., 2020; Kang and Dubey, 2020). However, ZeroMQ's ROUTER-DEALER pattern introduces a soft-brokered model where messages are dynamically routed between endpoints, enabling load balancing, request delegation, and message multiplexing without requiring a dedicated broker process (Hintjens et al., 2011). This hybrid approach allows ZeroMQ to scale efficiently while maintaining flexibility in distributed environments. CoAP follows a request-response model that eliminates a standalone broker, although proxy or intermediary components can be introduced if needed (Shelby et al., 2014). By comparison, gRPC uses remote procedure calls, effectively pairing each function invocation on the client side with an implementation on the server side. While not a message-oriented middleware per se, gRPC's RPC-based abstraction enables a variety of streaming modes (unary, client streaming, server streaming, bidirectional streaming), positioning it as a flexible option for cloud-edge communications (Kampars et al., 2021).

Communication Models MQTT specializes in publish-subscribe, making it highly suitable for loosely coupled device networks requiring decoupled one-to-many or many-to-many communications (Standard, 2019; Moraes et al., 2019). AMQP introduces more structured exchanges and queues, thereby allowing a mix of publish-subscribe and point-to-point semantics with robust delivery guarantees and transactional operations (Prajapati, 2021). CoAP, while reminiscent of HTTP, aligns with resource-constrained devices using request-response over UDP (Shelby et al., 2014; Thangavel et al., 2014). ZeroMQ thrives in high-speed, real-time requirements or in specialized patterns (e.g., pipeline, pub-sub, request-reply) without a central broker (Lauener et al., 2017; Pamadi et al., 2020). Meanwhile, gRPC supports multiple RPC modes that handle both microservice calls and streaming in ways reminiscent of advanced publish-subscribe systems but requires a connection-oriented channel (Bolanowski et al., 2022).

2.3.2 Performance Characteristics

Latency and Bandwidth Consumption Latency is often highlighted as a critical factor for IoT devices that rely on quick responses and real-time analytics. Research by (Moraes et al., 2019) and (Bender et al., 2021) notes that MQTT, being broker-based, can experience higher latency under heavy loads compared to ZeroMQ or gRPC, which

use direct connections or HTTP/2 multiplexing, respectively. ZeroMQ's brokerless design can achieve very low latency in real-time applications, but it lacks built-in QoS features, making it less resilient if there is a high risk of packet loss (Kang and Dubey, 2020). CoAP outperforms TCP-based protocols under moderate packet loss conditions due to its lightweight UDP-based design (Thangavel et al., 2014; Betzler et al., 2016), though it may not scale as seamlessly to extremely large deployments if advanced topic-based routing or guaranteed ordering is required.

For workloads involving large data transfers (e.g., file uploads or media streaming), ZeroMQ and gRPC can demonstrate stronger throughput due to reduced overhead and efficient streaming (Pamadi et al., 2020; Bolanowski et al., 2022). MQTT performs adequately for moderate message sizes but may struggle with significant volumes if the broker or network saturates (Standard, 2019). AMQP can handle large queues reliably but may incur higher overhead because of richer metadata and configuration details in each exchange and queue (Prajapati, 2021; Uy and Nam, 2019).

Reliability and QoS Protocols like AMQP and MQTT offer explicit Quality of Service levels, guaranteeing at-least-once or exactly-once delivery (Standard, 2019; Prajapati, 2021). MQTT's QoS 1 and QoS 2 modes ensure message delivery but can add latency and overhead in unstable networks (Moraes et al., 2019). AMQP's transactional semantics excel in enterprise use cases requiring persistent, fault-tolerant message queues (Uy and Nam, 2019). In contrast, ZeroMQ, CoAP, and gRPC either lack built-in message persistence or rely on a simpler acknowledgment model (Lauener et al., 2017; Shelby et al., 2014; Kampars et al., 2021). CoAP's Confirmable messages (CON) and optional reliability features (exponential backoff and deduplication) suffice in many sensor and IoT contexts (Shelby et al., 2014), but for strict ordering or multi-hop reliability, additional logic is often required at the application level (Martí et al., 2019). gRPC does not natively support brokered persistence but does permit advanced streaming with HTTP/2's flow control, making it robust for short disconnections or congested segments, though still not as robust under very high packet-loss conditions as a protocol specifically designed for unreliable links (Kampars et al., 2021).

Scalability and Resource Utilization Protocols that rely on a central broker (AMQP, MQTT) may encounter bottlenecks if the broker is not clustered or load-balanced. Studies have documented that broker scaling mechanisms for MQTT are non-trivial but can handle

a large number of connected devices if well configured (Bender et al., 2021). ZeroMQ's peer-to-peer nature scales linearly by adding more nodes without augmenting a broker node, though each node must handle subscription, routing, and reliability tasks on its own (Kang and Dubey, 2020). CoAP typically shines in small or moderate-scale sensor networks but can be extended to large systems if proxies or additional infrastructures are introduced (Betzler et al., 2016; Böhm and Wirtz, 2022). gRPC scales well in microservice-based systems that require many microservices to communicate, especially when load balancing at the layer of orchestrators or service meshes (Bolanowski et al., 2022).

Security Implications From a security standpoint, MQTT and AMQP incorporate TLS/SSL for encryption and can enforce authentication with username-password or certificates (Standard, 2019; Prajapati, 2021). CoAP uses DTLS to secure communications, though DTLS's additional handshake overhead can hamper performance on tiny devices (Shelby et al., 2014; Martí et al., 2019). ZeroMQ typically necessitates external security layers like CurveZMQ or TLS tunnels to protect data in transit (Lauener et al., 2017), raising complexity in public or untrusted networks. gRPC integrates SSL/TLS by default and supports token-based or other pluggable authentication frameworks (Arora et al., 2024). However, each additional layer of security can increase computational demands, complicating usage in battery-constrained IoT endpoints (Kampars et al., 2021).

2.3.3 Practical Deployment Considerations

Edge vs. Cloud Integration For highly constrained edge nodes with minimal CPU and memory, simpler protocols such as CoAP, MQTT, or ZeroMQ typically excel (Thangavel et al., 2014). CoAP or MQTT can reduce overhead and manage ephemeral connectivity for large sensor fleets (Mishra, 2018; Çorak et al., 2018). If the edge device is more powerful (e.g., an edge server with GPU capabilities), gRPC can be advantageous, particularly for streaming large data sets or interactive control commands requiring low-latency round trips (Böhm and Wirtz, 2022; Pamadi et al., 2020). AMQP may be favored in enterprise-grade deployments, bridging multiple edge clusters with robust routing and transactional semantics, especially in financial or industrial automation domains (Uy and Nam, 2019; Prajapati, 2021).

Developer Experience and Tooling MQTT's popularity is reflected in extensive tooling for IoT device provisioning, online brokers, and open-source client libraries (Standard,

2019). ZeroMQ's minimalistic approach is appreciated by developers needing custom patterns or extreme throughput in real-time systems (Lauener et al., 2017; Kang and Dubey, 2020). CoAP integrates well with existing REST paradigms, thus easing transitions from HTTP to more lightweight IoT protocols (Betzler et al., 2016). AMQP has broad enterprise support and mature infrastructure (e.g., RabbitMQ, ActiveMQ), though it can demand more overhead in specifying routing keys, exchanges, or queue configurations (Prajapati, 2021; Naik, 2017). gRPC, while newer in IoT contexts, boasts robust cross-language support and automatic code generation from protobuf definitions, making it a top contender for microservice or cloud-edge systems that demand fast, structured data exchange (Bolanowski et al., 2022; Kampars et al., 2021).

Evolving Hybrid or Mixed Models Given the varied constraints across IoT networks, many modern deployments combine multiple protocols. For instance, resource-limited sensors might push data to an MQTT broker at the edge, and data might be relayed to the cloud over a gRPC-based microservice architecture (Chan et al., 2022; Böhm and Wirtz, 2022). CoAP endpoints can similarly leverage an HTTP/CoAP cross-proxy, bridging local UDP-based sensor communications with an HTTPS-based cloud interface (Shelby et al., 2014). ZeroMQ can embed seamlessly into specialized sub-networks requiring extreme real-time performance, with optional bridging to other protocols if needed (Pamadi et al., 2020; Lauener et al., 2017).

2.3.4 Guidelines for Protocol Selection

Drawing upon the comparative insights above, the following guidelines can assist architects and engineers in deciding on a protocol or protocol mix:

1. **Network Reliability and Constraints**: If the network is prone to packet loss or device power constraints are stringent, a UDP-based approach (CoAP) or a broker-based pattern with simpler QoS levels (MQTT) might be more appropriate (Thangavel et al., 2014).

2. **Message Size and Throughput Requirements**: For small telemetry messages, MQTT or CoAP are typically sufficient. For large data streams (e.g., video frames, aggregated logs), gRPC or ZeroMQ can provide higher throughput (Bolanowski et al., 2022; Pamadi et al., 2020).

3. **Real-time and Latency-Critical Applications**: ZeroMQ or gRPC are well-suited to scenarios demanding sub-millisecond latencies, particularly if advanced streaming is needed (Kang and Dubey, 2020; Kampars et al., 2021).

4. **Enterprise-Grade Reliability and Routing**: AMQP excels in structured enterprise environments requiring sophisticated routing, message queuing, or transactional guarantees (Uy and Nam, 2019; Prajapati, 2021).

5. **Security and Access Control**: All protocols can integrate TLS/DTLS, but overhead differs. If strong authentication and encryption are needed for cloud-edge interactions, gRPC or AMQP provide robust built-in support (Arora et al., 2024; Prajapati, 2021).

6. **Scalability and Integration**: MQTT and AMQP have proven track records in scaling up to thousands of connections, with support for clustering. ZeroMQ scales horizontally without a central broker but requires careful application logic. gRPC can integrate well with orchestrated microservices (Kubernetes, service meshes) to achieve large-scale deployments (Bolanowski et al., 2022).

No single protocol can satisfy all IoT or edge-cloud scenarios simultaneously. Many large distributed systems deliberately combine multiple protocols, each fulfilling specialized roles in the data flow pipeline (Moraes et al., 2019; Böhm and Wirtz, 2022).

2.3.5 Concluding Remarks on Comparative Analysis

MQTT, ZeroMQ, AMQP, CoAP, and gRPC each address different dimensions of the IoT protocol design space. MQTT's simplicity, low overhead, and popularity make it well-suited for basic telemetry from constrained sensors. CoAP's REST-like abstraction over UDP excels in battery-powered sensor networks with intermittent connectivity, while ZeroMQ offers raw speed and low latency for real-time, brokerless communications. AMQP addresses enterprise-level demands with advanced routing, queueing, and transactional messaging. Finally, gRPC stands out for high-performance microservice and cloud-edge integration through powerful streaming modes and protocol buffer serialization.

Although these protocols are often presented as alternatives, real-world solutions commonly integrate multiple ones, taking advantage of each protocol's strengths. Future trends in IoT and edge computing—such as serverless edge functions, AI-based data processing, and cross-domain data marketplaces—are likely to intensify the need for flexible

protocol stacks that combine efficient transport, security, and application-level features. Studies have increasingly demonstrated the effectiveness of bridging or proxying among them (e.g., CoAP-HTTP proxies, MQTT-gRPC bridging) to unify distributed systems while minimizing overhead (Shelby et al., 2014; Pamadi et al., 2020).

In conclusion, a careful assessment of requirements such as QoS, bandwidth, latency, security, and developer ecosystem is essential when selecting protocols for IoT-based or microservice-based architectures. By examining the synergy between network constraints and application design, engineers can design robust, performant, and secure solutions that leverage the optimal features of MQTT, ZeroMQ, AMQP, CoAP, and gRPC in tandem.

2.4 Cloud-Edge Computing Architectures

The increasing adoption of IoT applications has led to challenges in processing large volumes of data efficiently. Traditional cloud-centric architectures rely on remote data centers for storage and computation, which can introduce significant latency, network congestion, and scalability limitations. To mitigate these issues, edge computing has emerged as a paradigm that brings computational resources closer to the data sources, enabling real-time processing and reducing dependency on centralized cloud services. This section discusses the fundamental differences between cloud and edge architectures, followed by an analysis of the role of containerization technologies such as Docker and Kubernetes in managing IoT workloads.

2.4.1 Cloud-Centric vs. Edge-Centric IoT Architectures

The rapid adoption of Internet of Things (IoT) technologies has led to an unprecedented increase in data generation, necessitating efficient computational architectures to handle large-scale data processing. Traditional cloud-centric architectures leverage centralized data centers (the `‘Cloud Server (Centralized)‘` in Figure 2.2) to process and store IoT-generated data, offering scalable and cost-effective solutions. However, with the increasing number of connected devices, this approach introduces significant limitations, including high latency, bandwidth congestion, and reliability concerns, particularly for time-sensitive applications (Böhm and Wirtz, 2022).

As an alternative, edge computing has emerged as a paradigm that moves computational resources closer to the data source. This is exemplified by the `‘Edge Node (Local`

Processing)` shown in Figure 2.2, which receives raw data from devices like a `Smart Thermostat`, `Camera`, and `Automated Valve`. Edge computing enables real-time processing and can provide control feedback locally (e.g., `Control` to the thermostat, `Feedback` to the valve), thereby reducing reliance on remote cloud infrastructure (Chan et al., 2022). Edge computing complements cloud computing rather than replacing it, often forming a hybrid cloud-edge model that balances the workload. In such a model, the edge node might send processed data to the centralized cloud server for long-term storage or further analytics, as indicated by the `Processed Data` flow in Figure 2.2 (Duan et al., 2020). This figure illustrates the fundamental interaction within a hybrid IoT environment, highlighting the distinct roles and data flows between IoT devices, edge nodes, and cloud servers.

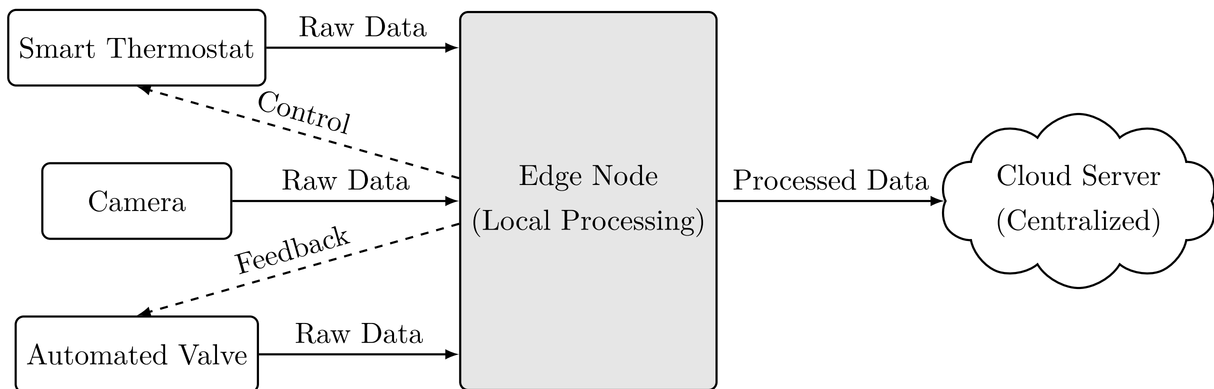

Figure 2.2: Cloud-Edge Computing Architecture in an IoT Environment, illustrating data flow from devices to an edge node for local processing and control, with subsequent transmission of processed data to a centralized cloud server.

Cloud computing remains an essential component of IoT architectures due to its ability to provide robust storage, large-scale data analytics, and seamless device interoperability. In cloud-centric architectures, IoT devices act as data collectors, transmitting raw sensor readings to cloud services for processing and long-term storage. While this model benefits from the extensive computational power of cloud data centers, it also introduces several challenges. The latency associated with multiple network hops is often prohibitive in applications requiring real-time responsiveness, such as industrial automation and smart transportation (Böhm and Wirtz, 2022). Additionally, the continuous transmission of large volumes of IoT data increases bandwidth costs and network congestion, which can degrade system performance. The dependence on centralized cloud services also raises concerns regarding system reliability, as connectivity disruptions may lead to service interruptions

or data loss.

Edge computing addresses these challenges by decentralizing data processing, enabling computations to occur on intermediary nodes such as IoT gateways, micro data centers, or even directly on devices themselves. By reducing the volume of data sent to the cloud, edge computing optimizes bandwidth usage while enhancing system responsiveness. This localized processing capability is particularly valuable in applications requiring low-latency decision-making, such as predictive maintenance in manufacturing and autonomous vehicle navigation (Böhm and Wirtz, 2022). Additionally, edge computing can improve security and privacy by limiting the exposure of sensitive data to external networks, as preliminary processing can be performed at the edge before transmitting only relevant insights to the cloud (Duan et al., 2020).

A key advantage of edge computing is its ability to enhance system resilience. Unlike cloud-centric architectures that rely heavily on continuous internet connectivity, edge computing allows IoT devices to function independently during network disruptions. This is particularly crucial in mission-critical applications, such as healthcare monitoring systems, where uninterrupted operation is essential (Chan et al., 2022). Moreover, by distributing workloads across edge nodes, computational resources are utilized more efficiently, preventing bottlenecks that can occur in centralized cloud environments.

Despite its benefits, edge computing is not without its challenges. Resource constraints on edge devices, such as limited processing power and memory, can restrict the complexity of tasks that can be executed locally. Additionally, managing a large-scale distributed edge infrastructure introduces challenges in terms of orchestration, security, and workload balancing (Böhm and Wirtz, 2022). Hybrid cloud-edge architectures have emerged as a viable solution, leveraging the strengths of both paradigms by dynamically allocating workloads between edge and cloud layers. In such architectures, latency-sensitive tasks are processed at the edge, while the cloud handles long-term data storage and computationally intensive analytics.

Real-world deployments of hybrid cloud-edge architectures have demonstrated their effectiveness in various IoT applications. In smart city monitoring systems, for example, traffic cameras and environmental sensors perform preliminary data filtering at the edge before transmitting relevant information to centralized cloud platforms for city-wide analysis. Industrial IoT deployments utilize edge computing for real-time process control while leveraging cloud services for predictive maintenance and historical trend analysis (Duan et al., 2020). Similarly, in healthcare IoT, wearable devices perform on-device analytics for

real-time patient monitoring while cloud-based platforms aggregate and analyze long-term health data for clinical insights.

Future advancements in cloud-edge computing architectures are expected to focus on improving resource scheduling, security, and fault tolerance mechanisms. Research is ongoing to enhance dynamic workload allocation strategies, ensuring optimal performance across both cloud and edge environments. As IoT ecosystems continue to expand, the seamless integration of cloud and edge computing will play a crucial role in enabling scalable, efficient, and resilient IoT infrastructures.

2.4.2 Containerization and Kubernetes in IoT

The complexity of modern IoT deployments has led to the widespread adoption of containerization as a means to efficiently manage distributed applications. Traditional virtualization approaches, such as virtual machines (VMs), provide strong isolation but suffer from high overhead, long startup times, and inefficient resource utilization. Containerization, by contrast, offers a lightweight alternative by allowing applications to run in isolated environments while sharing the host operating system kernel. This significantly reduces resource consumption and improves scalability, making it particularly suitable for constrained edge devices (Morabito et al., 2017).

Docker has emerged as the dominant containerization technology, enabling the packaging of applications along with their dependencies into portable, self-sufficient units. Unlike traditional VMs, Docker containers eliminate the need for a full guest operating system, resulting in faster deployment times and lower memory usage. These benefits are especially valuable in edge computing environments, where computational resources are limited, and applications must be deployed dynamically in response to changing conditions (Morabito et al., 2017). Comparative evaluations have shown that Docker outperforms VM-based approaches in terms of startup latency and resource consumption, making it more efficient for real-time IoT applications (Böhm and Wirtz, 2022).

Despite its advantages, managing large-scale containerized applications across cloud-edge infrastructures requires robust orchestration mechanisms. Kubernetes has become the de facto standard for automating the deployment, scaling, and management of containerized workloads. Originally designed for cloud environments, Kubernetes has been extended to support edge computing through lightweight distributions such as K3s and MicroK8s, which are optimized for resource-constrained devices (Böhm and Wirtz, 2022). These

variants provide a reduced memory footprint and lower CPU utilization, making them well-suited for IoT deployments at the edge. Comparative studies have evaluated the performance trade-offs of K3s and MicroK8s, showing that they provide a balance between efficiency and functionality, making them ideal for edge applications (Böhm and Wirtz, 2022).

Kubernetes enables several critical capabilities for IoT applications. It provides automated workload scheduling, ensuring that applications are efficiently distributed across available compute nodes. This is particularly beneficial for edge computing, where network topology and resource availability can vary dynamically. Additionally, Kubernetes enhances fault tolerance by automatically restarting failed containers and redistributing workloads in response to node failures. These self-healing capabilities improve the resilience of IoT systems, ensuring continuous operation even in unpredictable network conditions (Böhm and Wirtz, 2022). In addition, studies have demonstrated the effectiveness of Kubernetes' latency-aware orchestration strategies for optimizing resource distribution in edge-based industrial IoT systems (Böhm and Wirtz, 2022).

Resource management is another key advantage of Kubernetes in IoT environments. In traditional cloud-based deployments, resource allocation follows predefined static configurations, which may lead to inefficient utilization. Kubernetes introduces dynamic resource scheduling, allowing compute, memory, and network resources to be allocated based on real-time demand. This adaptive scheduling is essential for edge computing, where workloads fluctuate depending on sensor inputs and application requirements (Böhm and Wirtz, 2022). Furthermore, Kubernetes supports custom metrics-based scaling, enabling workload adjustments based on latency, bandwidth availability, and energy consumption metrics (Böhm and Wirtz, 2022).

While Kubernetes offers numerous benefits, its adoption in IoT environments presents several challenges. Edge devices typically have limited processing power and memory, making it difficult to run full-fledged Kubernetes clusters. Moreover, IoT networks often experience intermittent connectivity, which can disrupt centralized control mechanisms. To address these issues, research has explored lightweight Kubernetes alternatives and decentralized orchestration approaches, such as multi-cluster federation, which distributes workloads across multiple geographically dispersed clusters to improve fault tolerance and reduce latency (Böhm and Wirtz, 2022). Recent studies have further examined methods for extending Kubernetes clusters to low-resource edge devices, enabling efficient resource allocation and reducing dependency on high-performance cloud infrastructure (Böhm and

Wirtz, 2022).

A notable real-world example of Kubernetes in IoT is its integration with cloud-edge monitoring systems. Studies have demonstrated how Kubernetes, combined with monitoring tools such as Prometheus and Grafana, can provide real-time insights into edge device performance and network conditions. These platforms enable adaptive resource allocation and predictive scaling, further enhancing the efficiency of IoT deployments (Chan et al., 2022). Kubernetes-based cloud-edge architectures have also been successfully deployed in large-scale smart city monitoring systems to optimize resource management and service availability (Chan et al., 2022).

The integration of Docker and Kubernetes in IoT enables a flexible and scalable approach to workload management across cloud and edge layers. By leveraging containerization for application portability and Kubernetes for orchestration, modern IoT architectures can achieve high availability, efficient resource utilization, and reduced operational overhead. Future research aims to refine Kubernetes for edge computing by improving latency-aware scheduling, energy-efficient workload placement, and secure multi-cluster coordination. As IoT ecosystems continue to evolve, the seamless integration of containerization and orchestration technologies will play a pivotal role in enabling next-generation distributed computing environments.

2.5 Performance Metrics and Benchmarking in IoT

The evaluation of IoT communication protocols requires a structured approach to benchmarking key performance metrics. Given the constraints imposed by resource-limited edge devices and variable network conditions, factors such as latency, jitter, throughput, message ordering, and resource utilization play a crucial role in determining the efficiency of different protocols. These metrics directly impact the reliability and scalability of IoT systems, particularly in real-time applications such as industrial automation, remote healthcare, and smart infrastructure monitoring (Kang and Dubey, 2020).

2.5.1 Latency, Jitter, and Throughput

Latency is a fundamental performance metric in IoT networks, referring to the time taken for a message to travel from the sender to the receiver. Many IoT applications, such as industrial process control and real-time health monitoring, require minimal latency to

ensure timely decision-making. Several factors influence latency, including network congestion, processing delays, and protocol overhead (Althoubi et al., 2021). In cloud-based IoT architectures, additional latency is introduced by long transmission paths between edge devices and remote data centers.

Closely related to latency is jitter, which quantifies the variation in packet delay over time. High jitter can degrade the performance of real-time IoT applications, particularly those relying on streaming sensor data or time-synchronized event processing. Studies have shown that edge computing can mitigate jitter by reducing reliance on remote cloud servers, leading to more predictable response times (Althoubi et al., 2021).

Throughput represents the amount of data successfully transmitted over a network within a given time frame. Protocol efficiency, message size, and available bandwidth all influence throughput. Comparative studies of IoT messaging protocols indicate that ZeroMQ typically achieves higher throughput than MQTT due to its peer-to-peer design, which avoids the overhead associated with broker-based architectures (Kang and Dubey, 2020). Performance evaluations conducted on MQTT broker implementations demonstrate that message delivery time increases significantly under high-load conditions, highlighting scalability concerns (Mishra, 2018).

2.5.2 Message Ordering and Integrity

Maintaining correct message order is essential in IoT applications that process sequential data, such as predictive maintenance and industrial automation. Sequence numbers embedded in sensor payloads enable receivers to detect missing or out-of-order messages. AMQP provides built-in transactional sequencing, while MQTT ensures order within a single brokered session but does not guarantee order across multiple connections (Kang and Dubey, 2020).

MQTT's reliance on a central broker helps maintain order within a single connection but does not guarantee message sequencing across multiple connections. Conversely, ZeroMQ, which follows a brokerless architecture, enables direct peer-to-peer communication but requires additional mechanisms to preserve ordering in multi-hop deployments. In high-latency networks, protocols with acknowledgment-based retransmission schemes, such as gRPC, introduce additional delays but improve message integrity (Pamadi et al., 2020).

Network-induced packet reordering is another challenge, particularly in IoT deployments using wireless and lossy networks. Analytical models of sensor networks indicate that both

edge latency and core network latency contribute to message disorder, necessitating buffer management strategies to reassemble out-of-order packets (Althoubi et al., 2021). Studies comparing MQTT and CoAP suggest that while broker-based architectures like MQTT maintain message order within a session, they introduce additional transmission delays compared to UDP-based alternatives like CoAP, which prioritize low overhead over strict sequencing guarantees (Silva et al., 2021).

2.5.3 Resource Utilization and Network Overhead

The efficiency of an IoT communication protocol is also determined by its computational footprint and network overhead. Resource constraints are particularly relevant for battery-powered edge devices, where excessive CPU and memory usage can reduce operational lifetime. Studies comparing MQTT, ZeroMQ, and CoAP reveal that MQTT's broker-based model imposes additional CPU overhead as the number of connected clients increases (Morabito et al., 2017).

Network overhead refers to the additional metadata and signaling required for communication. Protocols that introduce high header overhead or require frequent acknowledgments consume more bandwidth, making them less suitable for low-power, constrained networks. Empirical evaluations show that MQTT's reliance on TCP results in higher network overhead compared to UDP-based alternatives like CoAP, which trade delivery guarantees for reduced transmission overhead (Pamadi et al., 2020). However, when Quality of Service (QoS) levels are introduced in MQTT, the protocol incurs greater computational and memory overhead due to the need for message tracking and retransmissions (Mishra, 2018).

Comparative benchmarking studies demonstrate that MQTT, while effective for reliable message delivery, performs poorly in networks with high congestion due to its persistent TCP connections and acknowledgment overhead. In contrast, CoAP, which operates over UDP, exhibits lower latency and reduced energy consumption but sacrifices reliability and message ordering guarantees (Çorak et al., 2018). These trade-offs highlight the importance of selecting protocols based on application-specific constraints, particularly in cloud-edge architectures where resource efficiency and network performance must be balanced.

To optimize IoT protocol performance, recent research has explored adaptive scheduling techniques that dynamically allocate processing resources based on network conditions.

Edge-computing strategies can further reduce resource consumption by offloading computationally intensive tasks from constrained devices to nearby gateways or cloud infrastructure (Böhm and Wirtz, 2022). Future developments in IoT protocol design will likely focus on reducing protocol-induced overhead while maintaining reliable data delivery in heterogeneous network environments.

2.6 Related Work

The growing adoption of IoT applications has led to the development of various communication protocols, each designed to address specific network constraints, reliability requirements, and scalability challenges. Selecting an appropriate messaging protocol is critical for ensuring efficient data exchange, particularly in cloud-edge environments where latency, resource utilization, and fault tolerance directly impact system performance. Several studies have attempted to compare these protocols across different use cases, evaluating their efficiency in constrained IoT environments. This section reviews prior research on IoT communication protocols, their performance in cloud-edge architectures, and existing gaps that motivate this study.

2.6.1 Comparative Studies on IoT Communication Protocols

Numerous studies have examined the performance of IoT communication protocols, focusing on key metrics such as latency, throughput, message reliability, and resource efficiency. The most widely evaluated protocols include MQTT, CoAP, AMQP, gRPC, and ZeroMQ, each of which presents distinct advantages and trade-offs depending on the target application and network environment.

Comparative analyses of MQTT and CoAP indicate that while both protocols are lightweight and efficient for IoT applications, CoAP's UDP-based transport offers lower latency and better resilience in lossy networks, whereas MQTT provides higher reliability through its broker-based architecture and Quality of Service (QoS) mechanisms (Silva et al., 2021; Betzler et al., 2016). Studies evaluating the performance of AMQP, particularly through RabbitMQ implementations, highlight its enterprise-oriented features such as message queuing, routing, and delivery guarantees, making it suitable for cloud-based IoT backends but potentially introducing higher latency compared to MQTT and CoAP (Prajapati, 2021; Uy and Nam, 2019).

ZeroMQ has been assessed for its brokerless design, which enables lower communication latency and higher throughput compared to broker-based protocols like MQTT and AMQP (Pamadi et al., 2020). Additionally, ZeroMQ's socket abstraction provides flexible messaging patterns, including pub-sub, request-reply, and multi-hop message routing via mechanisms such as ROUTER-DEALER, which allow for decentralized coordination without a broker (Hintjens et al., 2011). However, its lack of built-in message persistence and QoS features makes it less suited for environments requiring guaranteed message delivery (Kang and Dubey, 2020). Similarly, gRPC has been evaluated as a high-performance alternative for cloud-native IoT applications, offering efficient structured data exchange via Protocol Buffers and HTTP/2 multiplexing. Studies comparing gRPC with MQTT suggest that while gRPC is optimized for cloud communication and microservices, its reliance on TCP and lack of built-in message persistence limit its applicability in constrained IoT networks (Kampars et al., 2021; Bolanowski et al., 2022).

Despite these comparative analyses, most existing studies focus on evaluating these protocols in isolated environments, without accounting for the complexities of cloud-edge architectures and containerized deployments. As IoT systems increasingly leverage Kubernetes-based cloud-edge infrastructures, a more comprehensive evaluation of how these protocols perform in such environments is needed.

2.6.2 Cloud-Edge Benchmarking in Prior Research

Cloud-edge computing architectures have been extensively analyzed in the context of reducing latency, managing network overhead, and optimizing resource allocation. Research has demonstrated that traditional cloud-only IoT architectures introduce significant transmission delays, as data processing occurs in remote servers, making them unsuitable for real-time applications (Böhm and Wirtz, 2022). Edge computing mitigates these challenges by enabling local data processing, improving response times, and reducing bandwidth consumption.

Prior research on Kubernetes-based edge computing highlights the advantages of containerized microservices in terms of scalability and fault tolerance. Böhm et al. evaluated Kubernetes in cloud-edge deployments and found that while edge computing reduces latency, it also introduces challenges related to workload orchestration and resource scheduling (Böhm and Wirtz, 2022). Another study examined hybrid cloud-edge models, showing that dynamic workload balancing between cloud and edge layers enhances computational

efficiency (Duan et al., 2020).

While there is a growing body of research on cloud-edge architectures, limited studies have analyzed the impact of different IoT communication protocols in Kubernetes-based environments. The majority of existing work focuses on general cloud-edge computing strategies rather than protocol-level performance benchmarking. This thesis aims to fill this gap by systematically evaluating the efficiency of MQTT, ZeroMQ, AMQP, gRPC, and CoAP in a Kubernetes-based cloud-edge infrastructure.

Despite growing research in cloud-edge orchestration, limited studies analyze the impact of messaging protocols in distributed Kubernetes deployments, where factors such as broker placement (e.g., MQTT, AMQP), decentralized message routing (e.g., ZeroMQ, CoAP), and network overhead introduced by connection management (e.g., gRPC) play critical roles in determining overall system efficiency.

2.6.3 Gaps in Existing Research

While prior work has provided valuable insights into IoT communication protocols and cloud-edge computing, certain areas remain underexplored. Most comparative studies evaluate messaging protocols in controlled environments without considering the additional complexities introduced by containerization, network constraints, and distributed workload orchestration.

A key limitation in existing research is the lack of comprehensive benchmarking of IoT protocols within a Kubernetes-based cloud-edge architecture. Although some studies analyze protocol efficiency in traditional cloud or embedded environments, they often do not account for the additional overhead and performance trade-offs introduced by containerized deployments (Mishra, 2018). Given the increasing reliance on container orchestration for scalable IoT systems, a systematic evaluation of these protocols in Kubernetes environments is necessary.

Additionally, while existing studies provide insights into protocol behavior under different network conditions, they rarely examine how factors such as dynamic workload scaling, latency-aware scheduling, and containerized execution affect protocol performance in hybrid cloud-edge infrastructures. This thesis addresses these gaps by benchmarking MQTT, ZeroMQ, AMQP, gRPC, and CoAP in a real-world Kubernetes-based cloud-edge testbed. The evaluation focuses on key performance metrics such as latency, jitter, throughput, message integrity, and resource utilization.

By conducting these experiments in a cloud-edge system, this study aims to provide a more holistic understanding of how these communication protocols perform in modern IoT deployments. The findings will contribute to informed decision-making regarding protocol selection and optimization strategies for scalable, efficient, and reliable IoT architectures.

2.7 Summary

This chapter surveyed the foundational components of modern IoT systems, focusing on messaging protocols, cloud-edge computing paradigms, and key performance metrics. We began by comparing MQTT, ZeroMQ, AMQP (RabbitMQ), CoAP, and gRPC, illustrating how architectural choices—such as brokered versus brokerless communication, TCP versus UDP transport, or RPC-based versus publish-subscribe interactions—affect latency, scalability, and reliability. Although each protocol excels in specific scenarios, it became clear that no single solution can address every IoT use case, highlighting the importance of context-driven protocol selection.

We then examined the cloud-edge computing continuum and its advantages over purely cloud-centric models, particularly for time-sensitive or bandwidth-intensive applications. Here, containerization (Docker) and orchestration frameworks (Kubernetes) emerged as pivotal tools for managing distributed IoT workloads, offering automated deployment, resource scaling, and service continuity in hybrid environments, despite inherent complexities.

To round out the discussion, performance metrics vital for evaluating IoT deployments—most notably latency, jitter, throughput, and resource overhead—were reviewed. Existing studies, while informative, often fail to account for the orchestrated, containerized conditions increasingly prevalent in practice. This gap underscores the need for comprehensive benchmarking that encompasses both protocol characteristics and distributed orchestration overhead.

The next chapter will present the system architecture and experimental methodology designed to investigate these issues. It will detail the planned approach for deploying the aforementioned IoT protocols in a Kubernetes-enabled cloud-edge testbed and measuring their performance under varying conditions.

3 System Design and Methodology

3.1 Introduction

The efficient exchange of data between IoT devices and cloud services is critical for real-world applications, ranging from industrial automation to smart cities. However, different communication protocols offer varying trade-offs in terms of latency, reliability, scalability, and resource consumption. This study aims to systematically evaluate the performance of multiple IoT communication protocols, MQTT, ZeroMQ, AMQP, gRPC, and CoAP, within a cloud-edge deployment.

This chapter presents the system architecture and methodology used to benchmark these protocols in a controlled environment. The architecture consists of an edge layer, responsible for generating and transmitting synthetic sensor data, and a cloud layer, deployed on a Kubernetes cluster, which receives, processes, and monitors protocol-specific performance metrics. A modular framework is employed, where broker-based protocols (MQTT, AMQP) are contrasted against brokerless protocols (ZeroMQ, gRPC, CoAP) to assess their suitability for different IoT use cases.

The experimental methodology ensures comparability between protocols using a consistent sensor data source, standardized message payloads, and identical network conditions. A Prometheus-based monitoring stack collects real-time performance data, allowing a quantitative comparison of protocol efficiency.

The following sections provide a detailed breakdown of the system architecture, including the sensor source module, protocol-specific connectors, cloud backend implementation, and the experimental procedure. These components collectively form the foundation for evaluating communication protocols in a realistic, cloud-native IoT setting.

3.2 Overall System Architecture

The system architecture designed for benchmarking IoT communication protocols in this thesis adopts a distributed cloud-edge model, as illustrated in Figure 3.1. This architecture is intentionally structured to facilitate controlled and reproducible experiments across

various protocols. It comprises two principal layers: the *Edge Layer*, containerized using Docker, which is responsible for synthetic sensor data generation and protocol-specific data transmission; and the *Cloud Layer*, deployed on Kubernetes, which hosts the corresponding protocol endpoints, message brokers where applicable, and the metric collection infrastructure.

Figure 3.1 provides a visual overview of this planned testbed. On the **Edge (Docker)** side, a `‘Sensor Source Python App (Data Generator)‘` produces simulated sensor readings. This data is initially published to a `‘Local MQTT Broker (QoS 0)‘`, acting as an internal distribution hub. A suite of distinct `‘Connectors‘` (CoAP, gRPC, AMQP, ZeroMQ, MQTT QoS 2, MQTT QoS 1, and MQTT QoS 0) subscribe to this local broker. Each connector is then responsible for forwarding the received data to its designated counterpart in the cloud using its specific protocol. The `‘Browser Interface‘` shown on the edge allows for runtime configuration and monitoring of the sensor source.

On the **Cloud (Kubernetes)** side, each protocol has a dedicated processing pipeline. For instance, the `‘CoAP Connector‘` on the edge sends data to a `‘CoAP Server Reader/Metrics‘` in the cloud. Similarly, brokered protocols like AMQP and MQTT have their respective brokers (`‘RabbitMQ Broker‘`, `‘MQTT Broker (QoS x)‘`) and corresponding `‘Reader/Metrics‘` services. ZeroMQ utilizes a `‘ZeroMQ Broker‘` (representing the soft-broker pattern used). All these cloud-based reader services are designed to parse incoming messages and expose performance metrics. These metrics are then scraped by a `‘Prometheus‘` instance, which is part of the `‘Grafana Kube Stack‘`, enabling visualization via `‘Grafana Dashboards‘`.

This layered approach, clearly delineating edge responsibilities from cloud processing, aligns with contemporary cloud-edge computing paradigms where initial data handling might occur closer to the source before transmission to centralized cloud infrastructure for further analysis or storage (Andriulo et al., 2024). The interaction between these modular components is designed to ensure that each communication protocol is evaluated under identical input conditions, which is paramount for a fair comparative analysis. The subsequent subsections will detail the planned functionalities of the Edge and Cloud layers.

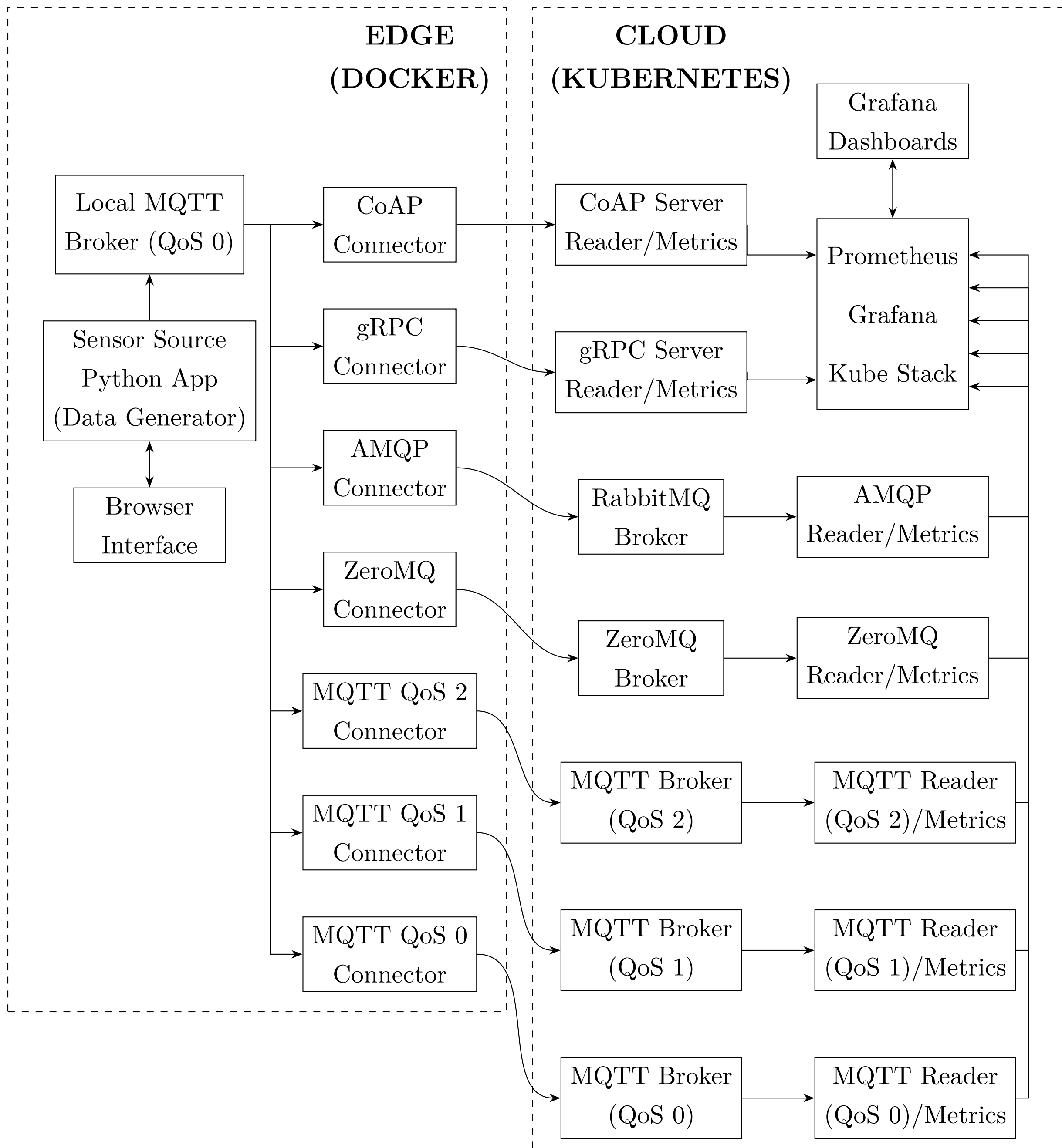

Figure 3.1: Planned Overall System Architecture for IoT Protocol Benchmarking, depicting Edge (Docker) and Cloud (Kubernetes) components and data flow for each evaluated protocol.

3.2.1 Edge Layer: Sensor Data Generation and Protocol Connectors

The edge layer is responsible for generating synthetic IoT sensor data and forwarding it to the cloud using different communication protocols. A Python-based *Sensor Source* application runs inside a Docker Engine on a local edge device, simulating multiple sensors that continuously publish structured data at predefined intervals. Each sensor message contains essential metadata, including a unique identifier, a timestamp, a sequence number, and sensor-specific values. The inclusion of sequence numbers allows for the detection of message loss or reordering, which are critical factors in protocol evaluation.

All sensor messages are initially published to a local MQTT broker running on the same edge device. This broker serves as a central distribution hub, allowing protocol-specific connectors to retrieve data in a consistent manner before forwarding it to the cloud layer. Each protocol under evaluation has a dedicated connector process that reads from the local MQTT broker and transmits messages using its respective communication mechanism. The protocols included in this study—MQTT (with different QoS levels), ZeroMQ, AMQP, gRPC, and CoAP—represent a diverse set of messaging architectures, covering broker-based and brokerless paradigms.

The MQTT connectors forward messages to dedicated cloud-hosted MQTT brokers configured with varying Quality of Service (QoS) levels, ensuring a comparative assessment of reliability mechanisms. ZeroMQ utilizes a dealer-router broker model, providing a brokerless alternative with direct message routing (Silva et al., 2021). The AMQP connector interacts with a RabbitMQ instance, leveraging message queuing and delivery guarantees (Uy and Nam, 2019). For CoAP and gRPC, the connectors establish direct communication with cloud-hosted services, bypassing the need for an intermediary broker. These design choices ensure that each protocol's performance characteristics are measured within a standardized experimental framework.

3.2.2 Cloud Layer: Kubernetes-Based Processing Services

The cloud layer is deployed in a Kubernetes cluster, where protocol-specific services receive, process, and monitor incoming messages. Each protocol operates within an isolated environment to ensure independent benchmarking, with services deployed as Kubernetes pods. These pods are responsible for handling incoming data streams, logging performance

metrics, and exposing monitoring endpoints for further analysis (Čilić et al., 2023).

For protocols that inherently rely on a message broker, such as MQTT and AMQP, messages are received by dedicated cloud-hosted broker instances (Mosquitto and RabbitMQ, respectively). These brokers act as intermediaries between senders and receivers. Additional metric collection services then subscribe to these brokers to extract relevant performance data, including message latency, throughput, and integrity checks. In contrast, protocols like and CoAP process messages directly at their respective cloud service endpoints, as they do not depend on centralized message brokers. ZeroMQ, while fundamentally brokerless, is handled in this testbed via a custom *ROUTER-DEALER* soft broker deployed in the cloud, which facilitates message relay and distribution to a ZMQ consumer service; however, this soft broker primarily serves as a message routing proxy rather than a persistent queuing broker in the traditional sense of MQTT or AMQP. All cloud services, whether consuming from a broker or receiving directly, are designed to expose standardized `/metrics` endpoints, allowing Prometheus to aggregate data in a structured manner (Böhm and Wirtz, 2022).

The Kubernetes cluster facilitates efficient scaling and resource allocation, ensuring that performance benchmarks are not biased by infrastructure limitations. Each protocol-specific service logs critical performance indicators, including:

- *End-to-end latency*, measuring the time taken for a message to travel from the edge to the cloud.
- *Jitter*, evaluating variations in message delay.
- *Throughput*, quantifying the rate at which messages are successfully processed.
- *Message ordering integrity*, verifying whether messages arrive in the correct sequence.
- *Resource utilization*, monitoring CPU and memory consumption of protocol services.

Prometheus serves as the central monitoring framework, continuously collecting performance data across all protocol implementations. The integration of Grafana allows for real-time visualization of benchmarking results, enabling comparative analysis of protocol behavior under varying network conditions.

This architecture ensures a rigorous and controlled evaluation of IoT communication protocols, highlighting the trade-offs between reliability, latency, and resource efficiency. The

subsequent sections delve deeper into the implementation details of each component, providing a comprehensive view of the methodology employed in this study.

3.3 Sensor Source Module

The Sensor Source module is a core component of the benchmarking framework, responsible for generating structured IoT sensor data in a controlled and reproducible manner. This module operates within a Docker container on the edge device and simulates multiple sensors that periodically emit measurements. The generated data is then disseminated to cloud-hosted services via protocol-specific connectors.

Each sensor message follows a standardized JSON format that includes key metadata fields necessary for benchmarking IoT communication protocols. These fields include the sensor identifier, sensor type, timestamp, sequence number, measured value, and an additional padding field for payload size manipulation. The inclusion of structured metadata ensures that each protocol's performance is evaluated consistently and facilitates comparative analysis across different messaging architectures (Milošević et al., 2021). Listing 3.1 presents an example of the structured payload format used in this study.

```
{
 "sensor_id": "sensor-1",
 "sensor_type": "temperature",
 "seq": 32694,
 "timestamp": "2025-03-06T17:23:59.382037+00:00",
 "value": 11.08,
 "padding": " "
}
```

Listing 3.1: Standardized JSON Payload Structure

Each field in the JSON message plays a critical role in benchmarking:

- **sensor_id** uniquely identifies the source of the message, allowing protocol performance to be analyzed per device.
- **sensor_type** specifies the nature of the data, which can include temperature, humidity, or pressure readings.
- **seq** (sequence number) enables detection of message loss, duplication, and out-of-order delivery—essential metrics in assessing communication reliability.

- **`timestamp`** records the exact moment of message creation and is synchronized using Network Time Protocol (NTP) to ensure high accuracy, thereby mitigating clock drift and enabling precise latency measurements.

- **`value`** represents the simulated measurement, generated using randomized distributions that reflect real-world environmental sensor data.

- **`padding`** is an adjustable field that allows for testing protocol efficiency under varying message sizes.

To maintain consistency across all protocol connectors, the Sensor Source module initially publishes generated messages to a local MQTT broker running within the same Docker environment. This intermediary broker ensures that all protocol connectors receive identical input data before forwarding it to their respective cloud endpoints. The use of MQTT as an internal distribution layer eliminates discrepancies that could arise from inconsistent message generation, reinforcing the validity of protocol comparisons (Čilić et al., 2023).

Another key feature of the Sensor Source module is its configurability. The module allows dynamic adjustments to test parameters, including:

- **Number of active sensors**: The number of simulated devices can be modified to test scalability and protocol performance under different loads.

- **Message generation frequency**: The system supports adjustable transmission intervals, ranging from high-frequency updates (e.g., every 10ms) to low-frequency reporting (e.g., once per second), providing insights into how each protocol handles varying message rates.

- **Payload size variation**: The padding field enables precise control over message size, facilitating direct comparisons between protocols that employ different encoding schemes, such as MQTT's text-based format versus gRPC's binary encoded protocol buffers.

Beyond message generation, the Sensor Source module provides a RESTful API for runtime configuration and performance monitoring. This API enables real-time modifications to experimental parameters without restarting the system, ensuring flexibility during benchmarking. Additionally, the module continuously tracks key statistics, such as the total number of messages sent, bytes transmitted, and real-time data rates, allowing for comprehensive performance evaluations under varying conditions.

By combining configurable message generation, precise timestamping, and real-time monitoring, the Sensor Source module provides a robust foundation for evaluating IoT communication protocols. Its ability to simulate realistic IoT workloads, integrate with multiple messaging protocols, and adjust dynamically to experimental conditions ensures that the benchmarking process remains systematic, reproducible, and reflective of real-world deployments.

3.4 Protocol-Specific Connectors

The protocol-specific connectors serve as intermediary modules that facilitate the transmission of sensor data from the edge device to the cloud. Each connector subscribes to the locally hosted MQTT broker, receives messages from the Sensor Source module, and forwards them to the cloud using a specific IoT communication protocol. This approach ensures that each protocol processes identical input data, enabling fair performance comparisons under controlled conditions. Given the diversity of IoT communication architectures, the connectors are categorized into two groups: broker-based and brokerless messaging protocols.

3.4.1 Broker-Based Protocols

Broker-based protocols rely on an intermediary message broker to facilitate communication between the sender and receiver. In this study, MQTT and AMQP are evaluated under this paradigm, where messages are first routed through a cloud-hosted broker before being processed.

MQTT Connectors: The MQTT implementation follows a two-tier broker architecture, where messages are initially published to a local MQTT broker at the edge and then forwarded to a cloud-hosted MQTT broker. To assess the impact of different Quality of Service (QoS) levels, separate MQTT connectors are implemented for QoS 0, QoS 1, and QoS 2. These connectors preserve the original payload structure while modifying the reliability guarantees for message delivery. QoS 0 provides best-effort delivery with no acknowledgment, QoS 1 ensures at-least-once delivery, and QoS 2 guarantees exactly-once delivery through a multi-step acknowledgment process. While QoS 2 enhances reliability, it also increases transmission overhead and processing latency (Ferrari et al., 2019).

AMQP (RabbitMQ) Connector: The AMQP connector transmits messages to a Rab-

bitMQ instance hosted in the cloud. AMQP employs a message queuing model, where messages are stored in queues and processed asynchronously. This approach ensures reliable message delivery, as messages persist even in the event of temporary network disruptions. However, compared to MQTT, AMQP introduces higher protocol overhead due to its transactional message-handling mechanisms (Sebrechts et al., 2021). While this architecture improves reliability, it may also impact throughput and introduce additional processing delays.

3.4.2 Brokerless Protocols

Brokerless protocols eliminate the need for an intermediary broker, enabling direct communication between the edge device and cloud services. This study evaluates ZeroMQ, gRPC, and CoAP as brokerless alternatives.

ZeroMQ Connector: ZeroMQ implements a direct messaging model using a `DEALER-ROUTER` pattern, which allows for efficient message distribution without requiring a centralized broker. Unlike MQTT and AMQP, which impose additional overhead due to broker mediation, ZeroMQ enables low-latency peer-to-peer communication (Milošević et al., 2021). However, ZeroMQ does not provide built-in message persistence or acknowledgment mechanisms, meaning messages can be lost if the network experiences disruptions. This design offers advantages in high-throughput applications but requires additional application-level logic to ensure reliability.

gRPC Connector: The gRPC connector follows a Remote Procedure Call (RPC) model, encapsulating sensor data in structured messages serialized using Protocol Buffers. The connector establishes a persistent HTTP/2-based connection to a cloud-hosted gRPC service, supporting both unary request-response and streaming communication. Compared to MQTT and AMQP, gRPC reduces serialization overhead and improves efficiency in structured data exchanges. However, gRPC lacks built-in message persistence, making it less resilient to temporary network failures (Happ et al., 2017). Additionally, its reliance on TCP can introduce connection establishment delays, impacting real-time message transmission.

CoAP Connector: The CoAP connector employs a lightweight request-response model over UDP, optimizing message transmission for constrained IoT environments. Unlike MQTT and AMQP, which support persistent connections and message queuing, CoAP follows a stateless approach where each message is processed independently. However,

handling large payloads or multiple simultaneous sensor transmissions introduces additional complexity.

To manage large data transfers, CoAP relies on Block-Wise Transfers (RFC 7959), which split messages into smaller blocks that must be reassembled at the receiver. While this avoids IP-layer fragmentation, it increases the need for custom application logic to track and reorder blocks, handle retransmissions, and maintain consistency across concurrent transmissions (Bormann and Shelby, 2016). This added complexity can impact real-time performance, particularly when multiple sensors send fragmented data simultaneously, requiring additional coordination at the application layer.

3.4.3 Implementation Considerations

Each protocol connector is deployed as an independent Docker container, ensuring modularity and isolation. The connectors continuously operate in a loop, receiving messages from the local MQTT broker and forwarding them to their respective cloud endpoints. Since each protocol differs in message reliability, transport efficiency, and serialization overhead, these connectors allow for direct comparisons of how well each approach handles varying network conditions and transmission loads.

By integrating both broker-based and brokerless communication models, this study enables a comprehensive analysis of protocol performance in a cloud-edge IoT environment. The next section details the cloud-based infrastructure responsible for processing and monitoring the incoming messages.

3.5 Cloud Backend Implementation on Kubernetes

The cloud backend is deployed on a Kubernetes cluster, providing a scalable and resilient environment for processing IoT messages. This architecture is designed to receive, process, and log sensor data transmitted from the edge layer while ensuring that each protocol operates within an isolated and controlled environment. Given the diverse characteristics of IoT messaging protocols, the backend follows a modular approach that distinguishes between brokered and brokerless architectures (Sebrechts et al., 2021).

3.5.1 Message Processing Architecture

The cloud backend is responsible for receiving sensor messages, parsing and validating payloads, tracking message order, and exposing structured performance data. Messages are processed either through dedicated brokers (for protocols such as MQTT and AMQP) or via direct service endpoints (for ZeroMQ (soft-broker), CoAP, and gRPC).

All received messages undergo standardized processing, including:

- **Parsing and validation** of incoming data to ensure structural consistency.
- **Tracking of sequence numbers** to detect lost or out-of-order messages.
- **Timestamp synchronization using Network Time Protocol (NTP)** to ensure precise latency measurements (Milošević et al., 2021).
- **Forwarding of performance metrics** to a monitoring system for further analysis.

By implementing NTP-based timestamp adjustments, the backend ensures that latency evaluations accurately reflect transmission delays rather than discrepancies caused by system clock drift (Ferrari et al., 2019). This is critical for maintaining benchmarking accuracy across distributed cloud-edge environments.

3.5.2 Handling Brokered and Brokerless Protocols

The cloud backend distinguishes between two primary categories of IoT communication protocols: brokered and brokerless architectures.

Brokered Protocols (MQTT, AMQP): Brokered protocols rely on message brokers that act as intermediaries between the edge and cloud applications. The backend includes:

- **MQTT brokers**, each configured with a specific quality-of-service (QoS) level to allow direct comparisons of different reliability mechanisms.
- **An AMQP-based RabbitMQ service**, which enables message queuing, delivery guarantees, and routing mechanisms.

These brokers are deployed as stateful services within Kubernetes, ensuring message persistence and reliable delivery. Load balancer services expose the broker endpoints, simplifying network access for edge devices.

Brokerless Protocols (ZeroMQ, CoAP, gRPC): Brokerless protocols eliminate the need for a centralized message broker, allowing direct communication between edge devices and cloud services. These protocols are handled as follows:

- **ZeroMQ** employs a *ROUTER-DEALER* proxy to manage message routing efficiently.
- **CoAP** operates over UDP and follows a request-response model, making it particularly lightweight. However, CoAP's block-wise transfer mechanism introduces complexity when handling large messages or multiple concurrent sensor streams, requiring custom handling to differentiate message fragments correctly (Liri et al., 2018).
- **gRPC** follows an *RPC-based model*, enabling structured data exchange over HTTP/2, which improves efficiency but lacks built-in message persistence (Gheorghe-Pop et al., 2020).

These services are deployed as Kubernetes pods, with load balancer services ensuring efficient connectivity between the edge layer and cloud-based endpoints.

3.5.3 Scalability and Reliability in Kubernetes

To handle varying message loads, the cloud backend leverages Kubernetes' built-in horizontal scaling capabilities. Each protocol-specific service scales dynamically based on traffic patterns, ensuring that high-throughput workloads do not overload individual pods (Andriulo et al., 2024).

Reliability mechanisms integrated within Kubernetes include:

- **Automated pod restarts** in the event of failures.
- **Health monitoring** to detect and mitigate service disruptions.
- **Load balancing** across multiple service replicas to distribute network traffic efficiently.

For brokered protocols such as MQTT and AMQP, Kubernetes manages stateful workloads to maintain persistence across pod restarts. This prevents message loss and ensures continuity in data transmission.

The cloud backend, deployed on a Kubernetes cluster, is responsible for handling IoT communication protocols with a focus on modularity, scalability, and resilience. Brokered protocols (MQTT, AMQP) rely on stateful broker instances, while brokerless protocols (ZeroMQ, CoAP, gRPC) operate as independent service endpoints. By leveraging Kubernetes' orchestration capabilities and built-in networking infrastructure, the system provides a reliable and adaptable environment for benchmarking IoT communication protocols in real-world cloud-edge deployments.

3.6 Experimental Procedure

To ensure a systematic and unbiased evaluation of IoT communication protocols, a series of experiments will be conducted under controlled conditions. These experiments are designed to reflect potential real-world cloud-edge IoT deployment challenges while maintaining comparability across protocols. The methodology will account for establishing baseline performance, assessing behavior under varying load conditions (message frequency and payload size), and evaluating resilience to common network impairments. These tests aim to capture meaningful performance differences relevant to the research questions of this thesis.

3.6.1 Baseline Performance and Load Characterization

The initial phase of experimentation will focus on establishing baseline performance benchmarks and observing how protocols respond to increasing message loads.

- **Increasing Message Frequency Scenario:**
  - **Setup:** A logical sensor source will progressively increase its message generation rate over time, starting from a low frequency and escalating to very high frequencies.
  - **Payload:** Each message will utilize a standardized JSON format with an approximate size of 150 bytes.
  - **Network Conditions:** Nominal network conditions without any artificially induced impairments will be maintained to isolate intrinsic protocol performance.

- **Objective:** This test is intended to identify the maximum message throughput capacity of each protocol, observe how end-to-end latency evolves with increasing load, and pinpoint saturation points or bottlenecks under ideal network conditions.

- **Varying Message Payload Size Scenario:**
  - **Setup:** A constant, moderate message rate (e.g., 5 messages/second from a small set of simulated sensors) will be maintained. The JSON payload size of each message will be systematically increased at intervals, progressing from small (e.g., 150 bytes) to very large (e.g., up to 1 MB).
  - **Network Conditions:** Nominal network conditions without induced impairments.
  - **Objective:** This test aims to understand the efficiency of each protocol in transmitting larger data units, the impact on end-to-end latency, and any limitations encountered with increasing message size, such as fragmentation issues or processing overheads.

3.6.2 Network Impairment Resilience Testing

To analyze the robustness and performance characteristics of each protocol under non-ideal network conditions, several fault injection scenarios will be implemented. These disruptions are designed to simulate common real-world constraints.

- **Packet Loss Simulation:**
  - **Setup:** A persistent packet loss (e.g., 10%) will be introduced on the communication link. The message load will be set at a moderate level (e.g., initially 50 messages/second, potentially increasing to 150 messages/second) that protocols can handle under ideal conditions.
  - **Objective:** This test aims to evaluate protocol-level recovery mechanisms, the impact on message delivery reliability (throughput), and the resulting effect on end-to-end latency when packets are frequently lost and potentially retransmitted.

- **Static Network Latency Injection:**

  - **Setup:** A constant, moderate message load (e.g., 50 messages/second) will be maintained while artificial static network latency between the edge and cloud is incrementally increased at intervals (e.g., from 25 ms up to 1500 ms).
  - **Objective:** This test is designed to assess protocol responsiveness and stability under conditions of high and increasing round-trip times (RTT), particularly for protocols reliant on acknowledgments or multi-stage handshakes.

- **Combined Network Impairments Simulation:**
  - **Setup:** A moderate message load (e.g., 40 messages/second with 500-byte payloads) will be subjected to a simultaneous combination of average static delay (e.g., 50ms), jitter (e.g., ±50ms), and packet loss (e.g., 5%).
  - **Objective:** This test aims to observe protocol behavior in a more complex, realistic adverse network environment, evaluating the combined impact of multiple simultaneous impairments on latency and throughput.

While other impairments like isolated jitter or deliberate message reordering could be considered, the focus will be on these fundamental and highly impactful network conditions to provide clear, comparable results within the scope of this thesis.

3.6.3 General Experimental Conduct and Data Collection

All experiments will be conducted within the consistent Kubernetes cluster environment for the cloud components and the Dockerized edge setup described previously. Performance metrics, primarily end-to-end latency (95th percentile), message throughput, and data throughput, along with indicators of message loss, will be collected using the Prometheus and Grafana monitoring stack. Each test condition will be run for a sufficient duration to allow for system stabilization and observation of clear performance trends. Where feasible and deemed necessary for data clarity, key tests or test phases may be repeated.

The planned experimental conditions are designed to provide a practical yet comprehensive assessment of protocol efficiency, behavior under varying loads, and resilience to common network problems, directly addressing the research questions of this thesis.

3.7 Performance Metrics

To systematically evaluate the performance of each communication protocol, a set of well-defined metrics is collected in the cloud environment. These metrics provide insights into message delivery efficiency, reliability, and system resource utilization. The data collection process is facilitated using the kube-prometheus-stack, a widely adopted cloud-native monitoring solution that integrates Prometheus for metric aggregation and Grafana for visualization (Sebrechts et al., 2021).

All metrics are collected from protocol-specific cloud services running in Kubernetes. Each service exposes a standardized `/metrics` endpoint, allowing Prometheus to scrape and store performance data at fixed intervals. The monitoring system ensures that results are gathered consistently across all protocols, minimizing variability due to external infrastructure factors.

3.7.1 Latency, Throughput, and Message Integrity

End-to-end latency is measured as the time difference between when a message is generated by the *Sensor Source* at the edge and when it is received by the cloud backend. To ensure precision, both the edge and cloud components synchronize their clocks using NTP (Network Time Protocol), mitigating errors introduced by clock drift (Ferrari et al., 2019). Since different protocols employ varying transport mechanisms, latency measurements reveal how transmission delays are influenced by factors such as broker mediation, TCP overhead, and message queuing strategies.

Jitter quantifies the variability in latency over a defined window of messages, capturing fluctuations in network conditions or protocol-induced variations in message delivery times. Low jitter values indicate predictable transmission performance, whereas high jitter may indicate unstable protocol behavior under fluctuating load conditions (Milošević et al., 2021).

Message throughput is recorded as the number of successfully received messages per second. This metric is critical for assessing protocol scalability, particularly under high sensor density or increased transmission frequency. In addition to message count, data throughput is measured in bytes per second to analyze protocol efficiency in handling different payload sizes. Protocols with high header overhead may exhibit lower data throughput despite maintaining stable message rates (Happ et al., 2017).

Message order integrity is evaluated by tracking sequence numbers embedded in each sensor message. If a message arrives with a lower sequence number than expected, it is classified as an *out-of-order* message. Missing sequence numbers indicate *dropped messages*, which contribute to the overall error rate. The error rate represents the proportion of lost messages, providing insights into how well each protocol handles unreliable networks, congestion, or fault injection scenarios (Čilić et al., 2023).

3.7.2 Resource Utilization and Monitoring Framework

To assess the computational overhead of each protocol, CPU and memory consumption are continuously monitored within the Kubernetes cluster. These resource usage metrics help quantify the cost of running each protocol in cloud environments and identify potential inefficiencies in protocol implementation (Gheorghe-Pop et al., 2020). For brokered protocols like MQTT and AMQP, CPU load may be influenced by broker-side processing, whereas brokerless protocols such as gRPC and ZeroMQ rely on direct message handling at the application layer, which can have different performance implications.

All metrics are collected through Prometheus and visualized in Grafana dashboards. The kube-prometheus-stack provides a centralized observability framework that enables fine-grained analysis of protocol performance under varying conditions. Metrics are scraped at a configurable interval to balance data resolution with system overhead, ensuring efficient monitoring without excessive resource consumption (Andriulo et al., 2024).

By combining network performance, message integrity, and computational resource analysis, this framework provides a holistic assessment of IoT communication protocols in a cloud-edge deployment. The results offer valuable insights into trade-offs between reliability, efficiency, and scalability, guiding protocol selection for real-world IoT applications.

3.8 Summary

This chapter outlined the design and methodology for benchmarking IoT communication protocols in a cloud-edge architecture. The system comprises two main layers:

Edge Layer: A Docker-based Sensor Source module generates structured IoT data, publishing messages to a local MQTT broker. Protocol-specific connectors forward this data to the cloud using different messaging paradigms, including brokered (MQTT, AMQP) and brokerless (ZeroMQ, gRPC, CoAP) protocols.

Cloud Layer: A Kubernetes-based backend processes incoming messages and logs structured performance metrics. Brokered protocols (MQTT, AMQP) rely on stateful message brokers, while brokerless protocols (ZeroMQ, gRPC, CoAP) interact with direct cloud-hosted endpoints.

The system is designed for scalability and resilience, leveraging Kubernetes' built-in load balancing, automatic scaling, and self-healing mechanisms to ensure consistent protocol benchmarking.

A structured experimental procedure was defined to evaluate protocols under controlled conditions, including baseline tests, scalability analysis, and fault injection scenarios. Key performance metrics—latency, jitter, throughput, message integrity, and resource utilization—are continuously monitored using Prometheus and Grafana.

This methodology ensures a fair, reproducible, and data-driven comparison of IoT communication protocols, providing insights into their real-world applicability for cloud-edge deployments. The subsequent chapters will present the implementation details of the testbed, followed by the empirical results derived from these experiments.

4 Implementation Details

4.1 Edge-Side Implementation and Components

The edge-side implementation consists of multiple modular components that work together to generate, process, and transmit sensor data to the cloud. These components are designed to be containerized and orchestrated using Docker, enabling deployment on a wide range of edge devices. In this experimental setup, the edge device is not restricted to specialized hardware; instead, it can be any system capable of running Docker Compose configurations, ensuring flexibility and reproducibility across different environments.

The key responsibilities of the edge-side components include:

- **Generating Sensor Data**: The Sensor Source module simulates IoT sensor readings and publishes structured messages to an MQTT broker.
- **Managing Local Communication**: A lightweight MQTT broker runs on the edge device, serving as the central message hub for sensor data before it is relayed to the cloud.
- **Protocol-Specific Cloud Forwarding**: Dedicated connector modules subscribe to the local MQTT broker and transmit sensor messages to cloud endpoints using different communication protocols.
- **Containerized Execution**: Each component is deployed as a Docker container, allowing for simplified configuration, portability, and scalability.

This architecture ensures that the edge device remains independent of specific hardware requirements while maintaining modularity and scalability in data transmission. The subsequent sections provide a detailed breakdown of each component, beginning with the Sensor Source module.

4.1.1 Sensor Source Implementation

The Sensor Source module is a Python-based application designed to simulate IoT sensor data and publish it via MQTT. It integrates a RESTful API using Flask for dynamic

configuration and performance monitoring. The application also synchronizes timestamps using NTP and employs multi-threading to run the sensor publishing loop in parallel with the REST API server.

NTP Time Adjustment

Accurate timestamps are essential for performance benchmarking. The application utilizes an NTP (Network Time Protocol) server to adjust local system time.

```
def get_ntp_offset(ntp_server='pool.ntp.org'):
    try:
        client = ntplib.NTPClient()
        response = client.request(ntp_server, version=3)
        return response.offset
    except Exception:
        return 0
ntp_offset = get_ntp_offset()
```

Listing 4.1: NTP Time Synchronization

The function contacts the NTP server to compute the offset between the local system clock and the reference time. This offset is stored globally and applied to all timestamps.

Global Configuration and Synchronization

A shared configuration dictionary, protected by a threading lock, allows real-time updates of parameters such as publishing frequency, number of sensors, and payload size. Additionally, a statistics tracking system maintains counts of published messages and transmitted data.

MQTT Client Initialization

The application uses the `paho.mqtt.client` library to publish sensor data asynchronously. The MQTT client connects to a broker and starts a network loop to handle communication.

Sensor Data Generation and Publishing

Each sensor message includes:

- A unique `sensor_id`: Identifies the specific sensor generating the message, allowing for per-source analysis if needed.
- The `sensor_type`: Specifies the kind of measurement being reported (e.g., temperature, humidity, pressure).
- A global `sequence` number: A monotonically increasing integer used by the receiving end to detect message loss and out-of-order arrivals, critical for assessing communication reliability.
- A precise `timestamp`: Records the message creation time at the edge, corrected using NTP offset calculations to ensure accuracy for latency measurements across distributed components.
- A simulated `value`: The actual sensor reading, numerically representing the measurement (e.g., degrees Celsius for temperature).
- A `padding` field: A string field whose length can be dynamically adjusted. This field is specifically included to control and vary the overall size of the JSON message payload. By modifying the content of this padding string, experiments can simulate different message sizes without altering the core data structure, enabling the evaluation of protocol performance under varying data transmission volumes.

The sensor value is generated as follows:

```
1 def generate_sensor_value(sensor_type):
2     ranges = {'temperature': (10, 40), 'humidity': (30, 80), 'pressure':
    (980, 1000)}
3     return round(random.uniform(*ranges.get(sensor_type, (0, 100))), 2)
```

Listing 4.2: Sensor Value Simulation

Message Padding and MQTT Publication

To simulate realistic network payload sizes, messages are padded before transmission. If a message is smaller than the configured payload size, it is expanded with a filler field.

```
1 def pad_payload(message_str, target_size):
2     current_size = len(message_str.encode('utf-8'))
3     if current_size >= target_size:
4         return message_str
```

```
    padding_needed = target_size - current_size
    return message_str[:-1] + ', "padding": "' + (' ' * padding_needed) + '
   "}'
```

Listing 4.3: Message Padding

Messages are generated at a configurable frequency and published via MQTT.

- The publisher retrieves the latest configuration parameters.
- It calculates the delay per message based on the total interval and number of sensors.
- For each sensor, a message is generated, padded, and published to the MQTT broker.
- The system updates the message count and data size statistics.
- The publisher thread sleeps for the computed delay before generating the next message.

REST API for Configuration and Monitoring

The application provides a REST API that allows users to:

- Retrieve the current sensor configuration.
- Dynamically update publishing parameters (interval, number of sensors, payload size).
- Monitor real-time statistics such as total messages sent, bandwidth usage, and message rate.

This API ensures external systems can adjust the testbed parameters dynamically without restarting the application.

Summary

The Sensor Source module is a multi-threaded, MQTT-based sensor simulator with:

- NTP-based timestamp synchronization.
- Configurable sensor parameters via a REST API.

- Real-time statistics tracking for performance evaluation.

This modular design ensures accurate and flexible IoT data simulation for cloud-edge benchmarking.

4.1.2 Protocol-Specific Cloud Connectors

Each cloud connector is responsible for relaying sensor data from the local MQTT broker to a cloud-hosted service using a designated protocol. While the underlying transport mechanisms differ, the overall structure of each connector remains largely consistent. This design ensures modularity, ease of maintenance, and the ability to systematically compare different protocols under identical conditions.

General Structure of the Connectors

All protocol-specific connectors share a common architectural framework, consisting of the following components:

- **Local MQTT Subscriber:** Each connector initializes an MQTT client that subscribes to the `sensor/data` topic on the edge device.
- **Message Forwarding Mechanism:** Upon receiving a message from the MQTT broker, the connector translates it into the respective protocol format.
- **Cloud Communication Client:** The message is then transmitted to the cloud-hosted service using the designated protocol.
- **Fault Tolerance and Reconnection Handling:** The connectors implement basic error handling mechanisms, such as automatic reconnection upon failure, ensuring robustness in message transmission.

This modular architecture allows for a direct comparison of different IoT communication protocols while minimizing implementation discrepancies. The primary differences across connectors lie in the transport protocol and the associated reliability mechanisms.

Implementation Variability

The key implementation differences across the connectors are summarized as follows:

- **Transport Mechanism:** Each connector employs a different communication protocol for forwarding messages to the cloud:
  - MQTT connectors publish messages to a cloud-hosted MQTT broker.
  - ZeroMQ connectors send messages via a DEALER socket.
  - AMQP connectors enqueue messages in a RabbitMQ queue.
  - CoAP connectors send messages using a RESTful POST request.
  - gRPC connectors use remote procedure calls (RPC) for structured message transmission.
- **Reliability Guarantees:** Protocols provide different levels of message delivery assurance:
  - MQTT supports three Quality of Service (QoS) levels (0, 1, and 2), offering trade-offs between performance and reliability.
  - AMQP enforces message durability and acknowledgments through its queuing mechanism.
  - ZeroMQ relies on asynchronous message queuing but lacks built-in acknowledgment mechanisms.
  - CoAP offers confirmable (CON) and non-confirmable (NON) message types.
  - gRPC guarantees reliable message delivery with built-in request-response semantics.
- **Message Formatting and Encoding:** While MQTT, AMQP, and gRPC natively support structured messages, ZeroMQ and CoAP require additional encoding and handling before transmission.
- **Synchronous vs. Asynchronous Processing:** Some protocols, such as gRPC and CoAP, operate asynchronously, whereas MQTT, AMQP, and ZeroMQ predominantly use synchronous messaging, though ZeroMQ can also function asynchronously.

By maintaining a consistent connector structure while varying only the transport protocol and reliability mechanisms, this implementation provides a controlled environment for evaluating the performance characteristics of different IoT communication protocols. The subsequent sections detail the specific implementation of each connector.

MQTT Connector

The MQTT-based cloud connector is responsible for forwarding sensor data from the local MQTT broker to a cloud-hosted MQTT broker. This connector establishes two separate MQTT clients:

- A **local MQTT client**, which subscribes to the `sensor/data` topic on the local broker. This subscription always operates at **QoS 0**, as the local broker resides on the same edge device, ensuring immediate message availability without requiring delivery guarantees.
- A **cloud MQTT client**, which republishes the received messages to a cloud-hosted MQTT broker. This connection allows for different levels of Quality of Service (QoS) to be configured, impacting message delivery guarantees.

The forwarding mechanism operates as follows:

1. The local MQTT client subscribes to the `sensor/data` topic using QoS 0.
2. Upon receiving a message, the local MQTT client forwards it to the cloud MQTT broker.
3. The cloud MQTT client republishes the message with a configurable QoS level, ensuring proper delivery semantics based on the desired reliability.

The three MQTT connectors differ only in the QoS level applied at the **cloud MQTT broker**:

- **QoS 0**: At-most-once delivery (best-effort, without acknowledgment).
- **QoS 1**: At-least-once delivery (message is acknowledged by the receiver).
- **QoS 2**: Exactly-once delivery (ensures no duplication or loss).

The QoS setting is applied when publishing to the cloud broker, as illustrated below:

```
def on_message(client, userdata, msg):
    # Forward the received message to the cloud broker with a configurable QoS
    cloud_mqtt_client.publish(mqtt_topic, payload=msg.payload, qos=1)
```

Listing 4.4: Publishing to Cloud MQTT Broker with Configurable QoS

Since the local broker operates within the same device as the sensor source and the connector, using QoS 0 for the local subscription is sufficient.

ZeroMQ Connector

The ZeroMQ (ZMQ) cloud connector establishes a bridge between the local MQTT broker and a ZMQ-based cloud endpoint. This connector utilizes a DEALER socket to enable asynchronous message forwarding.

To relay messages, the ZeroMQ connector performs the following steps:

1. The MQTT client subscribes to the `sensor/data` topic on the local broker.
2. Upon receiving a message, it forwards the payload to the ZMQ broker via a DEALER socket.
3. The ZMQ DEALER socket maintains an active connection to ensure efficient message transmission.
4. The connector handles error cases and ensures continuous message flow.

The ZMQ connection setup is as follows:

```
context = zmq.Context()
zmq_socket = context.socket(zmq.DEALER)
zmq_socket.setsockopt_string(zmq.IDENTITY, "forwarder")
zmq_socket.connect(f"tcp://{ZMQ_BROKER_ADDRESS}:{ZMQ_BROKER_PORT}")
```

Listing 4.5: ZMQ Connection Setup

This implementation ensures efficient message forwarding between the edge and cloud using ZeroMQ.

AMQP Connector

The AMQP connector enables message forwarding from the MQTT broker to a RabbitMQ queue, ensuring message persistence and reliable delivery.

The forwarding process consists of:

1. The MQTT client subscribes to the `sensor/data` topic on the local broker.

2. Received messages are published to a RabbitMQ queue.

3. The queue declaration ensures that messages persist in case of system restarts.

4. The connector ensures continuous operation and error handling.

The RabbitMQ connection setup is as follows:

```
credentials = pika.PlainCredentials(rabbitmq_user, rabbitmq_pass)
connection_params = pika.ConnectionParameters(host=rabbitmq_host, port=
    rabbitmq_port, credentials=credentials)
rabbit_connection = pika.BlockingConnection(connection_params)
channel = rabbit_connection.channel()
channel.queue_declare(queue=rabbitmq_queue, durable=True)
```

Listing 4.6: RabbitMQ Connection Setup

This approach ensures robust message queuing and delivery to the cloud service.

CoAP Connector

The CoAP connector forwards MQTT messages to a CoAP-based cloud endpoint using an asynchronous event loop. It is designed to handle the translation of messages received from the local MQTT broker into CoAP POST requests destined for the cloud service.

Sensor data is relayed to the cloud endpoint through the following steps:

1. The MQTT client subscribes to the `sensor/data` topic on the local MQTT broker.

2. Upon message reception, the payload is placed into an asynchronous processing queue.

3. A dedicated coroutine retrieves messages from this queue.

4. Each message is then transmitted to the CoAP cloud endpoint via a CoAP POST request. For all experiments conducted and reported in this thesis, CoAP Non-confirmable (NON) messages were exclusively utilized for these POST requests. This choice was made to evaluate CoAP's performance in a lightweight, best-effort delivery mode, minimizing protocol overhead for acknowledgment handling. While CoAP also supports Confirmable (CON) messages for reliable delivery, their evaluation was outside the scope of the experimental work presented herein.

The CoAP client context, necessary for creating and sending messages, is initialized asynchronously as shown in Listing 4.7.

```
import aiocoap

async def init_coap_context():
    # Creates a client context for sending CoAP messages
    context = await aiocoap.Context.create_client_context()
    return context
```

Listing 4.7: CoAP Client Context Initialization

This method enables efficient, low-overhead communication for constrained IoT environments.

gRPC Connector

The gRPC connector transmits structured sensor data from MQTT to a gRPC-based cloud service using Protocol Buffers.

Messages are forwarded by:

1. The MQTT client subscribes to the `sensor/data` topic on the local broker.
2. Received messages are parsed into gRPC request objects.
3. The request is sent asynchronously to the gRPC service.
4. The connector handles errors and ensures stable communication.

The following Protocol Buffers definition structures the transmitted data:

```
syntax = "proto3";

package sensordata;

service SensorDataService {
  rpc SendSensorData(SensorDataRequest) returns (SensorDataResponse);
}

message SensorDataRequest {
  string sensor_id = 1;
  string sensor_type = 2;
  int64 seq = 3;
```

```
13   string timestamp = 4;
14   double value = 5;
15   string padding = 6;
16 }
17
18 message SensorDataResponse {
19   string message = 1;
20 }
```

Listing 4.8: gRPC Protocol Buffer Definition

This protocol buffer definition, as detailed in Listing 4.8, structures the transmitted data, ensuring low-overhead message transmission suitable for high-performance cloud processing.

4.1.3 Edge Docker Setup

All edge components are containerized using Docker, ensuring consistency, portability, and ease of deployment. The edge architecture consists of the following core components:

- **Sensor Source**: The central component responsible for generating and publishing simulated sensor data to the local MQTT broker.
- **Local MQTT Broker**: Acts as an intermediary message broker that receives sensor data from the Sensor Source before it is forwarded to the cloud.
- **Protocol-Specific Cloud Connectors**: Independent services that subscribe to the local MQTT broker and relay sensor data to cloud-based services using different communication protocols.

The Sensor Source continuously generates structured sensor readings and transmits them to the Local MQTT Broker. The protocol-specific cloud connectors then retrieve these messages and forward them to the respective cloud services, ensuring a modular and extensible design.

Deployment and Containerization

The containerized applications follow a uniform deployment approach, ensuring that each service operates independently while being part of a shared network. The deployment involves:

- **Docker Compose for Service Orchestration**: Defines how services are interconnected and managed.
- **Minimal Docker Images**: Ensuring small, optimized containers with only required dependencies.
- **Persistent Configuration Management**: Using mounted volumes or environment variables for dynamic configuration.

The following example demonstrates the deployment of the local MQTT broker, which receives data from the Sensor Source:

```
version: '3.8'

networks:
  network:
    external: true

services:
  mosquitto:
    image: eclipse-mosquitto:latest
    container_name: mosquitto_broker
    restart: unless-stopped
    networks:
      - network
    ports:
      - "1883:1883"
    volumes:
      - ./mosquitto.conf:/mosquitto/config/mosquitto.conf
```

Listing 4.9: Docker Compose Configuration for Local MQTT Broker

The corresponding Dockerfile ensures a minimal setup for the MQTT broker:

```
FROM eclipse-mosquitto:latest
COPY mosquitto.conf /mosquitto/config/mosquitto.conf
EXPOSE 1883
```

Listing 4.10: Dockerfile for MQTT Broker

Sensor Source and Cloud Connectors

The Sensor Source is the primary data generator in the system. It simulates real-world IoT sensor readings and publishes them to the local MQTT broker. Cloud connectors then

subscribe to this broker and forward the messages to their respective cloud endpoints.

- The **Sensor Source** runs as a Python-based container, exposing an HTTP API for monitoring and configuration while continuously publishing sensor readings.
- Each **Protocol-Specific Cloud Connector** runs independently, subscribing to the local MQTT broker and transmitting messages to cloud services.
- Images are built using a base Python environment with required dependencies installed via `requirements.txt`.

For example, the Sensor Source is deployed similarly:

```
version: '3.8'

networks:
  network:
    external: true

services:
  sensor_app:
    build: .
    container_name: sensor_source
    restart: unless-stopped
    networks:
      - network
    ports:
      - "3000:3000"
    volumes:
      - .:/app
```

Listing 4.11: Docker Compose Configuration for Sensor Source

The Dockerfile follows a minimal and consistent structure across all edge components:

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY sensor-source.py .
COPY index.html .
EXPOSE 3000
CMD ["python", "sensor-source.py"]
```

Listing 4.12: Dockerfile for Sensor Source

All protocol-specific cloud connectors utilize a similar deployment approach, with only minor modifications in their respective Python scripts and dependencies. Each connector subscribes to the local MQTT broker and forwards sensor data to a cloud-based service, ensuring seamless data transmission across different protocols.

4.2 Cloud-Side Deployment and Infrastructure

The cloud-side deployment is hosted on Azure Kubernetes Service (AKS), a managed Kubernetes environment that provides scalability, reproducibility, and seamless integration with cloud-based monitoring and networking services. This platform is utilized to ensure the system operates within a controlled and high-performance computing environment. By leveraging AKS, the infrastructure benefits from automated resource management, dynamic scaling, and efficient orchestration of protocol-specific services.

To maintain consistency and facilitate controlled experimental conditions, all Kubernetes workloads are deployed within a dedicated namespace. Additionally, a single node pool with substantial computational resources is allocated to ensure consistent performance across all cloud-side components. The node pool specifications are as follows:

- Allocated CPU: 8 cores (allocatable: 7820m)
- Allocated memory: 32GB (allocatable: 27.9GB)
- High IOPS storage and network throughput for efficient data processing
- Operating system: Ubuntu 22.04.5 LTS
- Kubernetes version: v1.30.6, running containerd as the container runtime

All cloud-exposed services utilize the Azure Load Balancer mechanism, which assigns public IP addresses dynamically to enable external communication with edge devices. This approach ensures that all protocol-specific cloud endpoints remain accessible while maintaining high availability and security. Unlike ClusterIP-based services, which are restricted to internal cluster communication, the LoadBalancer configuration allows direct communication between edge-side components and cloud-hosted processing services.

4.2.1 Kubernetes Deployment Architecture

The Kubernetes deployment consists of multiple protocol-specific services and a monitoring infrastructure. All cloud-side components are deployed within the same namespace, ensuring uniform resource allocation and simplified network communication. The primary components of the deployment are outlined as follows:

- **Protocol-Specific Services**: Dedicated cloud service endpoints are provisioned for each communication protocol (MQTT, ZeroMQ, gRPC, AMQP, and CoAP). These services receive data from edge devices, process incoming sensor messages, and expose performance metrics.

- **Message Brokers**: The MQTT and RabbitMQ brokers are deployed as standalone services, enabling message queuing and forwarding. In contrast, ZeroMQ utilizes a lightweight custom soft broker, which functions as a message relay rather than a fully-fledged message queue.

- **Prometheus Monitoring Stack**: A Prometheus instance is deployed to collect performance data, including message latency, throughput, and ordering integrity.

- **Grafana Visualization**: A Grafana dashboard is configured to visualize the collected performance metrics in real time, facilitating comparative evaluation of different communication protocols.

Each of these components is deployed as a Kubernetes workload and is managed within the designated namespace. By running all services under a unified namespace, resource usage remains consistent, and inter-service communication is streamlined. The utilization of Azure Load Balancer ensures that external connections to cloud-hosted services remain stable and responsive, thereby providing a robust framework for evaluating the performance of IoT communication protocols in a cloud-edge environment.

4.2.2 Cloud Message Brokers

To facilitate communication between edge devices and cloud services, multiple message brokers are deployed in the cloud. These brokers receive sensor data from protocol-specific connectors and enable message forwarding to cloud-based consumers. This section details the deployment of MQTT and AMQP brokers and explains the role of the ZeroMQ soft broker.

MQTT Cloud Brokers

Three separate MQTT brokers are deployed, each corresponding to a different Quality of Service (QoS) level. These brokers are implemented using the Eclipse Mosquitto MQTT broker and deployed on Kubernetes as independent services.

The deployment of each MQTT broker consists of:

- A Kubernetes `Deployment` that runs a single Mosquitto broker instance.
- A corresponding `ConfigMap` that defines the broker configuration, including the listener port and authentication settings.
- A Kubernetes `Service` of type `LoadBalancer`, exposing the broker for external connections.

An example configuration for the QoS 0 broker is shown below:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mosquitto-qos0
  namespace: monitoring
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mosquitto-qos0
  template:
    metadata:
      labels:
        app: mosquitto-qos0
    spec:
      containers:
        - name: mosquitto
          image: eclipse-mosquitto:latest
          ports:
            - containerPort: 1883
          volumeMounts:
            - name: config-volume
              mountPath: /mosquitto/config/mosquitto.conf
              subPath: mosquitto.conf
      volumes:
        - name: config-volume
```

```
          configMap:
            name: mosquitto-config-qos0
```

Listing 4.13: Kubernetes Deployment for MQTT QoS 0

Each broker configuration allows anonymous connections and listens on port 1883 for MQTT traffic.

RabbitMQ Broker for AMQP

For AMQP-based communication, a RabbitMQ message broker is deployed using a similar approach. The RabbitMQ deployment consists of:

- A Kubernetes `Deployment` that runs a single RabbitMQ instance with management UI support.
- A `ConfigMap` that defines the RabbitMQ configuration, including TCP listener settings.
- A Kubernetes `Service` of type `LoadBalancer`, exposing the broker on ports 5672 (AMQP) and 15672 (management UI).

An example RabbitMQ deployment configuration is shown below:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rabbitmq
  namespace: monitoring
spec:
  replicas: 1
  selector:
    matchLabels:
      app: rabbitmq
  template:
    metadata:
      labels:
        app: rabbitmq
    spec:
      containers:
        - name: rabbitmq
          image: rabbitmq:3-management
          ports:
```

```
            - containerPort: 5672
            - containerPort: 15672
          volumeMounts:
            - name: config-volume
              mountPath: /etc/rabbitmq/rabbitmq.conf
              subPath: rabbitmq.conf
      volumes:
        - name: config-volume
          configMap:
            name: rabbitmq-config
```

Listing 4.14: Kubernetes Deployment for RabbitMQ

RabbitMQ enables persistent message queuing, ensuring reliability in AMQP-based cloud communication.

ZeroMQ Soft Broker

Unlike MQTT and AMQP, ZeroMQ does not include a native broker. Instead, a custom ZeroMQ-based soft broker is implemented as an intermediary message relay.

The soft broker operates as follows:

- The front-end uses a `ROUTER` socket to receive messages from publishers.
- The back-end uses a `DEALER` socket to forward messages to subscribers.
- A polling mechanism continuously monitors both sockets and transfers messages between them.
- Hexadecimal logging is implemented for debugging and ensuring message integrity.

The ZeroMQ soft broker is implemented as follows:

```
import zmq
import os
import logging
import binascii

def main():
    context = zmq.Context()
    frontend = context.socket(zmq.ROUTER)
    frontend.bind(f"tcp://*:{os.getenv('FRONTEND_PORT', 5555)}")
    backend = context.socket(zmq.DEALER)
```

```
    backend.bind(f"tcp://*:{os.getenv('BACKEND_PORT', 5556)}")

    poller = zmq.Poller()
    poller.register(frontend, zmq.POLLIN)
    poller.register(backend, zmq.POLLIN)

    while True:
        sockets = dict(poller.poll(timeout=1000))
        if frontend in sockets:
            message = frontend.recv_multipart()
            backend.send_multipart(message)
        if backend in sockets:
            message = backend.recv_multipart()
            frontend.send_multipart(message)
```

Listing 4.15: ZeroMQ Soft Broker Implementation

This soft broker acts as a lightweight proxy, enabling message routing between edge devices and cloud consumers without the overhead of a full-fledged message broker.

4.2.3 Cloud Connector Services

To process and analyze sensor data in the cloud, protocol-specific connector services are deployed. These services subscribe to their respective cloud brokers, receive sensor data, and expose standardized performance metrics for analysis.

Standardized Cloud Connector Architecture

Each cloud connector follows a modular architecture, ensuring a uniform approach across different protocols. The key design principles include:

- **Abstract Protocol Handler**: An abstract base class defines the necessary methods for protocol-specific implementations.
- **Standardized Message Processing**: All connectors share a common processing pipeline that extracts key metrics, monitors latency, and tracks message integrity.
- **Unified Prometheus Metrics Endpoint**: Regardless of the protocol used, each cloud connector exposes a standardized set of performance metrics via a Prometheus HTTP endpoint.

Protocol Handler Abstraction

To maintain consistency across different cloud connectors, an abstract base class defines the core functionality required for message handling. Each protocol-specific implementation extends this base class and implements the necessary connection, subscription, and message retrieval logic.

```
import abc

class ProtocolHandler(abc.ABC):
    @abc.abstractmethod
    def connect(self):
        pass

    @abc.abstractmethod
    def subscribe(self):
        pass

    @abc.abstractmethod
    def receive_message(self, callback):
        pass

    @abc.abstractmethod
    def disconnect(self):
        pass
```

Listing 4.16: Abstract Base Class for Cloud Connectors

Each protocol handler (MQTT, ZeroMQ, AMQP, etc.) extends this base class to provide its own communication logic while adhering to a common structure.

Example: MQTT Cloud Connector

The MQTT cloud connector subscribes to an MQTT broker, receives sensor messages, and updates performance metrics in real time.

```
import paho.mqtt.client as paho

class MQTTHandler(ProtocolHandler):
    def __init__(self):
        self.client = paho.Client()
        self.client.on_connect = self._on_connect
        self.client.on_message = None
```

```
8         self.client.on_disconnect = self._on_disconnect
9         self.client.on_subscribe = self._on_subscribe
10
11     def connect(self):
12         self.client.connect(MQTT_BROKER, MQTT_PORT, 60)
13
14     def subscribe(self):
15         self.client.subscribe(MQTT_TOPIC, qos=0)
16
17     def receive_message(self, callback):
18         self.client.on_message = callback
19         self.client.loop_start()
20
21     def disconnect(self):
22         self.client.disconnect()
```

Listing 4.17: MQTT Cloud Connector Implementation

This structure allows for the easy extension of additional protocol handlers while maintaining a common interface. Each cloud connector service can thus be deployed independently, while exposing consistent metrics for monitoring and performance evaluation.

Subsequent sections will provide deeper insights into the Prometheus-based monitoring and logging framework used to analyze cloud-side performance.

CoAP and gRPC Cloud Connectors

Unlike MQTT, ZeroMQ, and AMQP, which utilize a cloud-based broker or message queue, the CoAP and gRPC cloud connectors operate as standalone server instances. These connectors expose endpoints that directly receive sensor data from the edge, process it, and provide performance metrics.

gRPC Cloud Connector

gRPC is a high-performance RPC (Remote Procedure Call) framework that enables efficient communication between edge devices and cloud services. The gRPC cloud connector acts as a server that listens for incoming sensor data, processes it, and forwards it to the analysis pipeline. The implementation follows the standardized `ProtocolHandler` interface, ensuring consistency with other connectors.

A dedicated gRPC servicer class is implemented to handle incoming requests. Each re-

ceived message is parsed, structured into a JSON-like object, and forwarded for processing. The cloud connector does not require an explicit subscription mechanism, as gRPC inherently follows a request-response model.

The gRPC connector initializes a server thread and binds to a predefined port:

```
def connect(self):
    self.server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    servicer = GRPCHandler.SensorDataServiceServicer(lambda: self._callback)
    sensordata_pb2_grpc.add_SensorDataServiceServicer_to_server(servicer,
   self.server)
    self.server.add_insecure_port(f'[::]:{self.port}')
    self.server.start()
```

Listing 4.18: gRPC Server Initialization

Each sensor message received via gRPC is converted into a structured format and processed accordingly. The connector leverages the protocol buffer schema (Listing 4.8) to ensure structured message handling.

CoAP Cloud Connector

CoAP (Constrained Application Protocol) is a lightweight protocol designed for constrained IoT environments. Unlike MQTT, which relies on a publish-subscribe model, CoAP follows a request-response paradigm, making it ideal for direct sensor-to-cloud communication.

The CoAP cloud connector operates as an asynchronous server, listening for incoming POST requests on a dedicated endpoint. When a sensor device transmits data, the CoAP server processes the message and forwards it to the internal pipeline. The implementation relies on the `aiocoap` library for asynchronous event handling.

The server is initialized as follows:

```
def connect(self):
    self.loop = asyncio.new_event_loop()
    def start_server(loop):
        asyncio.set_event_loop(loop)
        self.site = resource.Site()
        self.site.add_resource(('.well-known', 'core'), resource.WKCResource
   (self.site.get_resources_as_linkheader))
        loop.run_until_complete(aiocoap.Context.create_server_context(self.
   site, bind=("0.0.0.0", 5683)))
```

```
8            loop.run_forever()
9        self.thread = threading.Thread(target=start_server, args=(self.loop,),
         daemon=True)
10       self.thread.start()
```

Listing 4.19: CoAP Server Initialization

Each incoming CoAP request is handled asynchronously, extracting relevant sensor data and forwarding it to the processing pipeline. The `receive_message` function dynamically registers a resource endpoint where sensor data is received.

These implementations ensure that CoAP and gRPC messages are processed with the same standardized metrics collection, making them functionally equivalent to their broker-based counterparts.

4.2.4 Deployment Configuration Files

Each cloud-side service is deployed using Kubernetes YAML manifests, ensuring modularity and scalability. The deployment strategy varies depending on whether a service requires external access or operates solely within the cluster.

To facilitate controlled experimentation, the services are deployed within a single Kubernetes namespace. Services that require direct communication with edge devices, such as gRPC, utilize Azure Load Balancer services to expose a publicly accessible endpoint. Conversely, services that only communicate internally, such as MQTT readers, are deployed using ClusterIP services to maintain isolation within the cluster.

Internal Service Deployment: MQTT Reader

The MQTT reader is responsible for processing sensor data received from the MQTT broker. Since the broker itself is already exposed externally via an Azure Load Balancer, the reader service remains internal to the cluster. The service configuration follows a standard approach:

- The deployment specifies a single replica within the monitoring namespace.
- The MQTT broker address is configured to reference the internal service name within the Kubernetes cluster.
- The service is defined with a `ClusterIP` type to prevent external access.

- Prometheus annotations allow automatic metric scraping from the exposed endpoint on port 8000.

A snippet from the deployment manifest is shown below:

```
apiVersion: v1
kind: Service
metadata:
  name: mqtt-reader-svc-qos0
  namespace: monitoring
spec:
  selector:
    app: mqtt-reader-qos0
    protocol: mqtt
  ports:
    - name: metrics
      protocol: TCP
      port: 8000
      targetPort: metrics
  type: ClusterIP
```

Listing 4.20: MQTT Reader Deployment Configuration

Externally Exposed Service: gRPC Reader

Unlike the MQTT reader, the gRPC reader must be accessible from edge devices, as there is no intermediary message broker. To achieve this, the service is exposed via an Azure Load Balancer, allowing external devices to communicate with the gRPC endpoint. The deployment includes:

- A deployment specifying a single replica within the monitoring namespace.
- The gRPC server listens on port 50051, while Prometheus metrics are exposed on port 8000.
- A LoadBalancer-type service exposes the gRPC endpoint to external edge devices.
- A separate ClusterIP service is used for internal Prometheus metric collection.

A snippet from the deployment manifest is shown below:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: grpc-reader-svc-grpc
  namespace: monitoring
spec:
  selector:
    app: grpc-reader
    protocol: grpc
  ports:
    - name: grpc
      protocol: TCP
      port: 50051
      targetPort: grpc
  type: LoadBalancer
```

Listing 4.21: gRPC Reader Service Configuration

The use of Azure Load Balancers ensures that edge devices can reliably communicate with cloud-hosted gRPC services while maintaining isolation for internal monitoring and processing components. By structuring the deployment in this manner, the system provides a robust, modular architecture suited for benchmarking cloud-edge communication protocols.

4.3 Instrumentation and Logging Setup

To systematically measure and log key performance metrics, the system is instrumented using Prometheus for metric collection and Grafana for visualization. Each cloud service exposes a standardized `/metrics` endpoint, which Prometheus scrapes periodically to collect performance data. The integration of Prometheus within Kubernetes is managed via the kube-prometheus-stack, which deploys Prometheus, Alertmanager, and Grafana as part of a unified monitoring solution.

An essential aspect of ensuring precise latency measurements is the synchronization of time across all components. The cloud protocol readers synchronize their system time using NTP (Network Time Protocol) in the same way as the Sensor Source at the edge. This synchronization allows for accurate measurement of message delay by adjusting timestamps against a common reference.

4.3.1 Prometheus Metric Collection

Each cloud protocol-specific reader is responsible for exposing a Prometheus-compatible `/metrics` endpoint. These endpoints allow the collection of real-time performance data, enabling quantitative analysis of different communication protocols. The following key metrics are monitored:

- **Latency**: The time elapsed from message generation at the edge to reception at the cloud endpoint. This is recorded using a `Gauge` metric to track the most recent latency and a `Histogram` to provide a distribution of observed latencies.
- **Throughput**: The number of successfully processed messages per second. This is measured using a `Gauge` that tracks messages per second and byte-based throughput (kilobytes and megabytes per second).
- **Error Rate**: The number of lost, malformed, or out-of-sequence messages. These are counted separately using `Counter` metrics for dropped messages and out-of-order arrivals.
- **Resource Utilization**: CPU and memory consumption of cloud services to assess computational efficiency.

To enable automatic metric collection, each reader is configured with a ServiceMonitor resource, which allows Prometheus to dynamically discover and scrape service endpoints within the Kubernetes cluster. A representative ServiceMonitor configuration is shown below:

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: mqtt-reader-monitor-qos0
  namespace: monitoring
  labels:
    release: my-prom-stack
spec:
  selector:
    matchLabels:
      app: mqtt-reader-qos0
  endpoints:
    - port: metrics
```

```
14       interval: 15s
15   namespaceSelector:
16     matchNames:
17       - monitoring
```

Listing 4.22: ServiceMonitor Configuration for MQTT Reader

This configuration ensures that Prometheus periodically scrapes metrics from the MQTT reader every 15 seconds. Similar ServiceMonitor definitions are applied to all protocol-specific cloud readers to enable consistent monitoring.

4.3.2 Metrics Collection in Protocol-Specific Readers

Each cloud-based protocol reader collects and exposes Prometheus-compatible metrics using the `prometheus_client` Python library. These metrics allow for comprehensive analysis of protocol performance. The following key metrics are collected:

- **Messages Received** (`messages_received_total`): A `Counter` metric that increments for every successfully received message.
- **Processing Time** (`message_processing_seconds`): A `Summary` metric tracking the time taken to process a single message.
- **Latency** (`message_latency_seconds`): A `Gauge` metric representing the delay between message generation and reception.
- **Latency Distribution** (`message_latency_seconds_histogram`): A `Histogram` that captures variations in message delays using predefined latency buckets.
- **Out-of-Order Messages** (`messages_out_of_order_total`): A `Counter` tracking instances where messages arrive with sequence numbers out of order.
- **Dropped Messages** (`messages_dropped_total`): A `Counter` tracking messages that were skipped due to gaps in the sequence numbering.
- **Throughput**:
  - `current_throughput_messages_per_second`: Number of messages processed per second.
  - `current_throughput_bytes_per_second`: Byte throughput.

  - `current_throughput_kilobytes_per_second`: Throughput measured in kilobytes per second.
  - `current_throughput_megabytes_per_second`: Throughput measured in megabytes per second.

- **Jitter** (`message_jitter_seconds`): A `Gauge` metric that calculates the standard deviation of latency over a rolling window.
- **Total Bytes Received** (`total_bytes_received`): A `Counter` metric tracking cumulative bytes received.
- **Unique Sensors** (`unique_sensors_current`): A `Gauge` that estimates the number of unique sensor sources observed in a given time window.

A sample of the actual metrics exposed at the `/metrics` endpoint of a cloud reader is shown below:

```
# HELP message_latency_seconds Latency of the *most recent* message
  processed
# TYPE message_latency_seconds gauge
message_latency_seconds 0.028383
# HELP message_latency_seconds_histogram Histogram of message latencies
# TYPE message_latency_seconds_histogram histogram
message_latency_seconds_histogram_bucket{le="0.01"} 7807.0
message_latency_seconds_histogram_bucket{le="0.1"} 259812.0
message_latency_seconds_histogram_bucket{le="0.5"} 272564.0
message_latency_seconds_histogram_bucket{le="1.0"} 272655.0
message_latency_seconds_histogram_count 278532.0
message_latency_seconds_histogram_sum 28780.81618199989
# HELP messages_out_of_order_total Total number of out-of-order messages
  received
# TYPE messages_out_of_order_total counter
messages_out_of_order_total 4.0
```

Listing 4.23: Example Output from /metrics Endpoint

4.3.3 Implementation of Prometheus Metrics in Cloud Readers

Prometheus instrumentation within the cloud protocol readers is implemented through a standardized `/metrics` endpoint that is periodically scraped to facilitate real-time monitoring. This mechanism provides insights into system performance and reliability. While

the full source code comprises an extensive suite of metrics, the following excerpt presents a subset of key metrics employed for latency monitoring and sequence integrity verification.

```
from prometheus_client import start_http_server, Gauge, Counter, Histogram

# Define selected Prometheus metrics
latency = Gauge('message_latency_seconds', 'Latency of the most recent
    message processed')
latency_histogram = Histogram('message_latency_seconds_histogram', '
    Histogram of message latencies', buckets=(.01, .025, .05, .075, .1,
    .125, .15, .175, .2, .25, .3, .35, .4, .5, .75, 1, 2.5, 5, 10, float('
    inf')))
out_of_order_messages = Counter('messages_out_of_order_total', 'Total number
     of out-of-order messages received')

# Start Prometheus HTTP server (port defined externally)
start_http_server(PROMETHEUS_PORT)
```

Listing 4.24: Selected Prometheus Metrics from the Source Code

Within the data processing pipeline, each received message undergoes timestamping, and its latency is computed through adjustment with an NTP offset to maintain precise synchronization. The following excerpt illustrates this latency computation:

```
import datetime
from datetime import timedelta

# Adjust cloud's current time using NTP offset
now = datetime.datetime.now(datetime.timezone.utc) + timedelta(seconds=
    ntp_offset)
time_diff = (now - timestamp).total_seconds()

latency.set(time_diff)
latency_histogram.observe(time_diff)
```

Listing 4.25: Latency Calculation in Cloud Readers

Furthermore, the system ensures the preservation of message ordering by verifying sequence numbers to detect any out-of-order packets. This validation process is essential in evaluating the reliability of each protocol. The following logic performs out-of-order detection and updates Prometheus counters accordingly:

```
# Within the DataProcessor class:
if self.is_first_message:
    self.last_seq_number = seq_num - 1
```

```
        self.is_first_message = False
else:
    expected_seq_num = self.last_seq_number + 1
    if seq_num < expected_seq_num:
        out_of_order_messages.inc()
    elif seq_num > expected_seq_num:
        # Handle dropped messages accordingly (code omitted for brevity)
        pass
self.last_seq_number = seq_num
```

Listing 4.26: Out-of-Order Message Tracking

This excerpt demonstrates the core implementation responsible for monitoring latency and message ordering across cloud protocol readers. These metrics are fundamental for evaluating the efficiency and reliability of different IoT communication protocols within the system.

4.4 Testing Tools and Environment Setup

To ensure a comprehensive evaluation of the system, a variety of testing tools are utilized to simulate real-world network conditions, analyze collected performance data, and visualize results. The key tools employed in this testing framework include:

- **NetEm and Clumsy**: Used for network impairment simulations such as adding artificial latency, inducing packet loss, and introducing jitter.
- **Prometheus Query Language (PromQL)**: Used to query collected metric data to derive statistical insights regarding latency, throughput, and error rates.
- **Grafana Dashboards**: Configured to provide a comparative visualization of protocol performance metrics, allowing direct analysis of trends and anomalies.

4.4.1 Simulating Network Conditions

A critical component of testing involves introducing network impairments that IoT edge-to-cloud communication systems commonly experience in real-world deployments. To simulate these conditions, different tools are used depending on the platform running the edge stack.

Linux-based Simulation with NetEm On Linux-based edge environments, NetEm (Network Emulator) is used to introduce controlled delays, packet drops, and bandwidth restrictions. NetEm is configured using the `tc` (traffic control) command-line utility. The following commands illustrate common configurations:

```
# Add 100ms delay with ±10ms variation (jitter)
sudo tc qdisc add dev eth0 root netem delay 100ms 10ms

# Introduce 2% packet loss
sudo tc qdisc change dev eth0 root netem loss 2%

# Remove all network impairments
sudo tc qdisc del dev eth0 root netem
```

Listing 4.27: Simulating 100ms Latency and 2% Packet Loss with NetEm

These configurations allow precise control over network conditions, enabling validation of how each protocol responds to different levels of network degradation.

Windows-based Simulation with Clumsy For Windows-based environments, Clumsy is used as an alternative to NetEm. Clumsy provides a graphical interface to introduce network impairments, including:

- **Lag**: Adds artificial latency to outgoing packets.
- **Drop**: Randomly discards a percentage of packets.
- **Throttle**: Limits network bandwidth.

Clumsy provides an intuitive way to test the robustness of protocols under adverse conditions without requiring command-line configuration.

4.4.2 Prometheus Query Language (PromQL)

Prometheus provides a powerful and flexible query language, PromQL, essential for processing and analyzing the collected time-series metrics. This language allows for the aggregation, transformation, and calculation of meaningful statistics from the raw telemetry data. The following examples illustrate typical PromQL queries employed in this research to derive key performance indicators such as throughput, error rates, and latency percentiles.

Analyzing Throughput Trends To evaluate system throughput over time, the following PromQL query calculates the average throughput (bytes per second) over a rolling 30-second window:

```
avg_over_time(current_throughput_bytes_per_second[30s])
```

Listing 4.28: Query for Average Throughput

Example output:

```
| Time                |Throughput (Bytes/sec)|
|---------------------|----------------------|
| 10:00:00            | 12,500               |
| 10:00:30            | 14,300               |
| 10:01:00            | 11,900               |
```

This output is visualized as a line chart in Grafana, showing fluctuations in message throughput.

Detecting Message Drops To monitor protocol reliability, the following query computes the rate of dropped messages per second over a 30-second window:

```
rate(messages_dropped_total[30s])
```

Listing 4.29: Query for Dropped Messages

Example output:

```
| Time                | Dropped Messages/sec |
|---------------------|----------------------|
| 10:00:00            | 0.2                  |
| 10:00:30            | 0.5                  |
| 10:01:00            | 0.0                  |
```

This query helps identify whether message loss increases under network stress, providing insights into protocol resilience.

Measuring Latency Percentiles Latency distribution is a key performance metric. The following histogram quantile query extracts the 95th percentile latency over a 30-second interval:

```
histogram_quantile(0.95, rate(message_latency_seconds_histogram_bucket[30s])
   )
```

Listing 4.30: Query for 95th Percentile Latency

Example output:

```
| Time                 | 95th Percentile Latency  |
|----------------------|--------------------------|
| 10:00:00             | 0.175                    |
| 10:00:30             | 0.210                    |
| 10:01:00             | 0.160                    |
```

This data is often plotted as a time-series graph in Grafana, allowing for easy visualization of how latency evolves over time.

4.4.3 Grafana Dashboards

Grafana is used to visualize the collected Prometheus metrics, enabling comparative analysis of protocol performance. The dashboards are configured to display:

- Time-series plots of throughput (messages per second, bytes per second).
- Latency histograms illustrating distribution percentiles.
- Packet loss statistics, including out-of-order and dropped messages.

Example Visualization: A time-series plot of the 95th percentile latency over time.

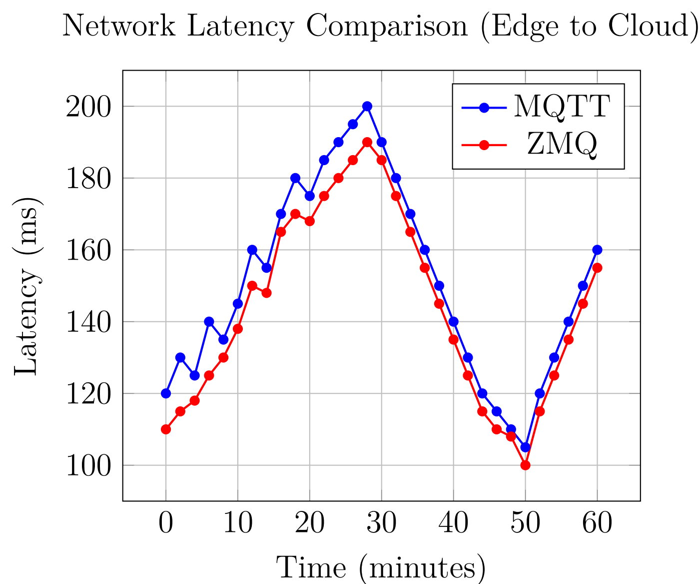

Figure 4.1: Grafana visualization of the 95th percentile message latency.

Exporting Data for Further Analysis Grafana allows exporting collected data in multiple formats, including:

- CSV: For detailed offline processing in tools like Python or Excel.
- JSON: For automated parsing and post-processing.
- Image Snapshots: For report documentation.

By integrating NetEm/Clumsy for network simulation, PromQL for statistical analysis, and Grafana for visualization, the system provides a rigorous and reproducible methodology for benchmarking protocol performance under controlled conditions.

4.5 Conclusion

This chapter detailed the practical implementation of the experimental testbed. Key components, including the edge-side Sensor Source and protocol connectors, their Docker-based deployment, the cloud-side Kubernetes infrastructure with its message brokers and services, and the Prometheus-based monitoring stack, were described. This implemented system forms the foundation for the systematic protocol benchmarking presented in the subsequent chapter. The following chapter, Chapter 5, will now present the empirical results and analysis derived from this testbed.

5 Performance Evaluation

This chapter presents the results of the empirical performance evaluation of the five IoT communication protocols: MQTT (with Quality of Service levels 0, 1, and 2), ZeroMQ, AMQP (RabbitMQ), CoAP, and gRPC. These protocols were implemented and deployed within the cloud-edge testbed detailed in Chapter 4, adhering to the system design and methodology outlined in Chapter 3. The primary objective of this evaluation is to quantitatively assess and compare the performance characteristics of these protocols under a range of simulated IoT conditions. This analysis aims to furnish insights into their respective suitability for diverse IoT application scenarios, focusing on key metrics such as end-to-end latency, message throughput, and message loss, as detailed in Section 5.1.3.

5.1 Experimental Setup and Methodology Recap

Before delving into the performance results, this section provides a concise recapitulation of the experimental environment and the methodologies employed. This summary serves to contextualize the data presented and ensure clarity regarding the conditions under which the evaluations were conducted.

5.1.1 Testbed Configuration

The experimental testbed comprises a distributed architecture with an edge component responsible for synthetic sensor data generation and a cloud component, hosted on Azure Kubernetes Service (AKS), tasked with data reception and processing. The edge layer leverages a Python-based Sensor Source application (Section 3.3) to emulate IoT device behavior, publishing structured messages to a local MQTT broker. Subsequently, protocol-specific connectors relay this data to the cloud infrastructure using their designated communication protocols. On the cloud side, Kubernetes orchestrates the protocol-specific services and associated message brokers (Section 3.5). A comprehensive monitoring suite, integrating Prometheus for metrics aggregation and Grafana for visualization, underpins the data collection process.

5.1.2 Testing Scenarios

The performance evaluation encompasses a series of experiments conducted under meticulously controlled conditions, as elaborated in Section 3.6. These scenarios are broadly classified as follows:

- **Baseline Performance Assessment:** Initial measurements conducted under nominal operating conditions, characterized by an increasing message rate from a single sensor without artificially induced network impairments. This establishes a foundational performance benchmark.
- **Network Impairment Resilience Tests:** Evaluation of protocol robustness against common network adversities, including tests with persistent packet loss, incrementally increasing static network latency, and combined impairments (latency, jitter, and packet loss).
- **Impact of Payload Size:** Analysis of protocol performance with a constant message rate but progressively increasing message payload sizes.

While the term *Scalability Analysis* was used in Chapter 3 to broadly cover varying loads, the specific tests presented herein focus on increasing message frequency from a single source (Baseline), increasing payload size, and performance under various impairments, rather than explicitly scaling the number of concurrent sensor devices in separate dedicated tests within this chapter's results.

5.1.3 Metrics

The core metrics for this performance evaluation, detailed extensively in Section 3.7 of Chapter 3, primarily focus in this chapter on:

- End-to-end Latency (typically 95th percentile)
- Message Throughput (messages per second) and Data Throughput (cumulative bytes)
- Message Loss Rate (implicitly observed via received message counts against offered load)

While jitter and explicit out-of-order message counts were instrumented for collection, the presented results concentrate on the above metrics as they most clearly differentiated protocol behaviors in these specific test scenarios. Out-of-order message delivery was not observed as a significant issue in these experiments; TCP-based protocols inherently ensure in-order delivery within a single stream, and for CoAP over UDP, the primary observation related to message integrity was packet loss rather than reordering.

5.2 Network-Level Protocol Analysis with Wireshark

To complement the quantitative performance metrics derived from the testbed, this section presents a qualitative analysis of the network traffic generated by each IoT communication protocol. Utilizing Wireshark for packet capture and inspection allows for a granular examination of the underlying communication patterns, protocol-specific overheads, handshake procedures, and message encapsulation structures. This analysis focuses on the transmission of a standardized JSON payload, as exemplified below, from the edge connector to the cloud service, ensuring a consistent basis for comparison across protocols:


```
{"sensor_id": "sensor-1", "sensor_type": "temperature", "seq": 14,
 "timestamp": "2025-05-09T16:56:10.750110+00:00", "value": 25.87, "padding": " "}
```


The network captures were performed on the edge device, specifically monitoring the outbound traffic originating from the protocol-specific connector modules destined for their respective cloud-based endpoints. For these baseline captures, transport-level security (e.g., TLS/DTLS) was not enabled to allow for direct inspection of application protocol headers and payloads, unless inherent to the protocol's typical secure deployment. The listings below present illustrative snippets of these captures, focusing on the key packet exchanges for a single message transmission. The source IP 'Edge' and destination IP 'Cloud' are used as placeholders, and 'TS Delta' indicates the time difference from the previous displayed packet in the selected snippet.

5.2.1 MQTT (Message Queuing Telemetry Transport)

MQTT operates over TCP/IP. The following subsections analyze Wireshark capture snippets for MQTT PUBLISH operations at Quality of Service (QoS) levels 0, 1, and 2. It is

assumed that the TCP connection and MQTT CONNECT/CONNACK handshake have already occurred prior to these captured message exchanges.

MQTT QoS 0 (At-most-once delivery)

The packet exchange for MQTT QoS 0, as depicted in Listing 5.1, is minimal. The Edge sends a single MQTT 'Publish Message' packet. The Cloud responds with a TCP 'ACK', acknowledging the underlying transport segment but providing no MQTT-specific acknowledgment.

```
TS Delta Source Destination Proto Len Info
0.184100 Edge   Cloud       MQTT  202 Publish Message [sensor/data]
0.074691 Cloud  Edge        TCP   60  [ACK]
0.998817 Edge   Cloud       MQTT  202 Publish Message [sensor/data]
0.032453 Cloud  Edge        TCP   60  [ACK]
```

Listing 5.1: Illustrative Wireshark Packet Exchange for MQTT (QoS 0)

This *fire and forget* mechanism results in the lowest protocol overhead for MQTT message transmission.

MQTT QoS 1 (At-least-once delivery)

For MQTT QoS 1, a two-way handshake at the MQTT protocol level ensures message delivery, as shown in Listing 5.2. The Edge sends a 'Publish Message' with a unique Message ID. The Cloud then returns an MQTT 'Publish Ack' (PUBACK) for that ID.

```
TS Delta Source Destination Proto Len Info
2.610612 Edge   Cloud       MQTT  204 Publish Message (id=3) [sensor/data]
0.032366 Cloud  Edge        MQTT  60  Publish Ack (id=3)
0.044360 Edge   Cloud       TCP   54  [ACK] (for PUBACK)
0.999196 Edge   Cloud       MQTT  204 Publish Message (id=4) [sensor/data]
0.032397 Cloud  Edge        MQTT  60  Publish Ack (id=4)
```

Listing 5.2: Illustrative Wireshark Packet Exchange for MQTT (QoS 1)

This exchange involves more packets per logical message compared to QoS 0, reflecting the cost of guaranteed at-least-once delivery.

MQTT QoS 2 (Exactly-once delivery)

MQTT QoS 2 employs a four-part handshake for the highest level of delivery assurance, detailed in Listing 5.3.

```
TS Delta Source Destination Proto Len Info
1.905713 Edge   Cloud       MQTT  204 Publish Message (id=2) [sensor/data]
0.032410 Cloud  Edge        MQTT  60  Publish Received (id=2)
0.000703 Edge   Cloud       MQTT  58  Publish Release (id=2)
0.032413 Cloud  Edge        MQTT  60  Publish Complete (id=2)
0.048482 Edge   Cloud       TCP   54  [ACK] (for PUBCOMP)
```

Listing 5.3: Illustrative Wireshark Packet Exchange for MQTT (QoS 2)

The sequence is: PUBLISH (Edge), PUBREC (Cloud), PUBREL (Edge), and PUBCOMP (Cloud). This robust mechanism incurs the highest network overhead among MQTT QoS levels due to the multiple packet exchanges per message.

5.2.2 ZeroMQ (ZMQ)

ZeroMQ, using a DEALER-ROUTER pattern over TCP in this configuration, encapsulates its messages using ZMTP. Listing 5.4 illustrates the data transmission.

```
TS Delta Source Destination Proto Len Info
1.343255 Edge   Cloud       TCP   188 [PSH, ACK] (ZMTP data)
0.040583 Cloud  Edge        TCP   60  [ACK]
0.961659 Edge   Cloud       TCP   188 [PSH, ACK] (ZMTP data)
0.031979 Cloud  Edge        TCP   60  [ACK]
```

Listing 5.4: Illustrative Wireshark Packet Exchange for ZeroMQ

The Edge sends ZMTP data within TCP segments (often with PSH flag set), and the Cloud acknowledges these at the TCP layer. Similar to MQTT QoS 0, this presents a lean exchange once the TCP connection is established, with ZMTP handling the message framing.

5.2.3 AMQP (Advanced Message Queuing Protocol) via RabbitMQ

AMQP 0-9-1, as used by RabbitMQ, is a TCP-based protocol. After connection and channel setup, publishing a message involves distinct AMQP frames, as shown in Listing 5.5.

```
TS Delta Source Destination Proto Len Info
5.074494 Edge   Cloud       AMQP  105 Basic.Publish, Content-Header
0.000325 Edge   Cloud       AMQP  194 Content-Body
0.032128 Cloud  Edge        TCP   60  [ACK] (for Content-Header)
0.000000 Cloud  Edge        TCP   60  [ACK] (for Content-Body)
```

Listing 5.5: Illustrative Wireshark Packet Exchange for AMQP

The Edge sends a 'Basic.Publish' frame (including the content header) followed by a 'Content-Body' frame, often in quick succession. The Cloud acknowledges these at the TCP layer. This two-packet transmission from the sender for a single logical message is characteristic of AMQP's structured approach.

5.2.4 CoAP (Constrained Application Protocol)

CoAP typically operates over UDP. Listing 5.6 illustrates a message exchange using Non-confirmable (NON) messages for the primary data transmission from the Edge.

```
TS Delta Source Destination Proto Len Info
1293.557 Edge   Cloud       CoAP  191 NON, MID:35494, POST, TKN:b085, /sensor
0.040179 Cloud  Edge        CoAP  60  NON, MID:13348, 2.01 Created, TKN:b085
1.000968 Edge   Cloud       CoAP  191 NON, MID:35495, POST, TKN:b086, /sensor
0.039458 Cloud  Edge        CoAP  60  NON, MID:13349, 2.01 Created, TKN:b086
```

Listing 5.6: Illustrative Wireshark Packet Exchange for CoAP (Non-confirmable Request with Non-confirmable Response)

In this scenario, the Edge sends a CoAP 'NON' (Non-confirmable) POST request containing the sensor data. Since the request is NON, the Edge does not expect a CoAP-level acknowledgment for its delivery. However, the Cloud server may still choose to send a response, which, as shown in Listing 5.6, is also a 'NON' message indicating '2.01 Created'. This response informs the Edge of successful processing but is itself sent unreliably (as a NON message) and is not acknowledged by the Edge. This pattern represents a *fire-and-forget* request with an optional, unacknowledged *fire-and-forget* response, minimizing protocol overhead for applications where such best-effort communication is acceptable.

5.2.5 gRPC (Google Remote Procedure Call)

gRPC utilizes HTTP/2 over TCP. Listing 5.7 shows packet exchanges within an established HTTP/2 session for a unary RPC, including initial HTTP/2 SETTINGS and sub-

sequent data exchanges.

```
TS Delta Source Destination Proto     Len Info
0.066126 Cloud  Edge        HTTP2     63  SETTINGS[0]
0.000153 Edge   Cloud       GRPCHTTP2 410 HEADERS[1]: POST ..., DATA[1] (GRPC
    )(PROTOBUF)...
0.032024 Cloud  Edge        TCP       60  [ACK]
0.000151 Cloud  Edge        HTTP2     71  PING[0]
0.000461 Edge   Cloud       HTTP2     71  PING[0] (Ack)
0.000322 Cloud  Edge        GRPCHTTP2 211 HEADERS[1]: 200 OK, DATA[1] (GRPC)(
    PROTOBUF)...
0.000912 Edge   Cloud       HTTP2     71  PING[0]
... (Further PINGs and data exchanges for subsequent messages) ...
0.999467 Edge   Cloud       GRPCHTTP2 180 HEADERS[3]: POST ..., DATA[3] (GRPC
    )(PROTOBUF)...
0.034642 Cloud  Edge        GRPCHTTP2 122 HEADERS[3]: 200 OK, DATA[3] (GRPC)(
    PROTOBUF)...
```

Listing 5.7: Illustrative Wireshark Packet Exchange for gRPC

The initial part of Listing 5.7 shows HTTP/2 session setup frames like 'SETTINGS'. For a gRPC call, the Edge sends HTTP/2 'HEADERS' and 'DATA' frames for the request (payload is Protocol Buffer encoded). The Cloud responds with its own HTTP/2 'HEADERS' and 'DATA' frames for the response. Interspersed are HTTP/2 'PING' frames for session management, contributing to the overall traffic.

5.2.6 Comparative Observations from Wireshark Packet Exchanges

The analysis of packet exchanges, exemplified in Listings 5.1 through 5.7, reveals distinct network-level behaviors:

- **Connection Establishment and Session Management:** All TCP-based protocols (MQTT, ZeroMQ, AMQP, gRPC) necessitate an initial TCP handshake. AMQP and gRPC (via HTTP/2) generally involve more intricate protocol-level session setup, including frames like HTTP/2 'SETTINGS' for gRPC (Listing 5.7), compared to the simpler MQTT connection or ZeroMQ socket initialization. CoAP, leveraging UDP, avoids connection setup overhead. gRPC (HTTP/2) further demonstrates session-level keep-alives through PING frames, as seen in Listing 5.7.
- **Packets per Logical Message (Data Phase):** The number of application-level packets exchanged for transmitting a single sensor reading varies significantly:

  - **Minimal Exchange (1 app-level packet from sender):** MQTT QoS 0 (Listing 5.1), ZeroMQ (Listing 5.4), and CoAP with NON requests (Listing 5.6, considering only the sender's packet) are very lean. The CoAP NON scenario may include an unacknowledged NON response from the server, making it two packets total but still *fire-and-forget* for both.
  - **Two-Way Exchange (2 app-level packets total for reliable delivery or request-response):** MQTT QoS 1 (Listing 5.2) uses a PUBLISH and a PUBACK. gRPC for a unary call (Listing 5.7) involves a logical request and a logical response. AMQP (Listing 5.5) requires two packets from the sender for one logical message, which are then acknowledged at the TCP layer.
  - **Four-Way Exchange (4 app-level packets total):** MQTT QoS 2 (Listing 5.3) has the most extensive exchange (PUBLISH, PUBREC, PUBREL, PUBCOMP) for its exactly-once semantics.

- **Header Overhead:** CoAP is designed for minimal headers. MQTT headers are also relatively compact. AMQP employs more descriptive and structured framing. gRPC, built on HTTP/2, benefits from binary framing and header compression (HPACK), though the HTTP/2 infrastructure itself is more complex than protocols like CoAP or MQTT. ZMTP adds its own framing over TCP for ZeroMQ.

- **Payload Encapsulation:** The listings illustrate that protocols like MQTT, AMQP (by default for text), CoAP (if content-type is JSON, as in Listing 5.6), and ZeroMQ (if directly sending JSON) transmit the payload in a human-readable text format. In contrast, gRPC mandates Protocol Buffers, resulting in a binary payload observed within the HTTP/2 DATA frames (Listing 5.7), which is generally more efficient for serialization/deserialization and bandwidth.

- **Acknowledgment Mechanisms:** Reliability is achieved through diverse means. TCP ACKs provide transport-level reliability for all TCP-based protocols. CoAP NON messages (Listing 5.6) offer no CoAP-level acknowledgment for the request, though a NON response may be sent. For reliable CoAP, a CON/ACK sequence (not shown with the updated snippet but previously discussed) would be used. MQTT QoS 1 and 2 implement explicit MQTT-level acknowledgments (Listings 5.2 and 5.3). AMQP supports publisher confirmations (though not explicitly seen in the provided snippet). gRPC relies on the reliable, ordered stream delivery of HTTP/2 over TCP, with the RPC response acting as an application-level acknowledgment.

- **Transport Protocol Implications:** TCP offers reliability, ordering, and congestion control at the cost of connection setup. UDP, used by CoAP, is connectionless and lightweight, enabling potentially lower latency but requiring application-level mechanisms for reliability if needed (e.g., CoAP CON/ACK, distinct from the NON example in Listing 5.6).

These qualitative observations regarding packet counts and protocol interactions per message provide a foundational context for interpreting the quantitative performance metrics, such as latency and throughput, presented in the subsequent sections of this chapter.

5.3 Baseline Performance Comparison

This section presents the results of the baseline performance tests. In this scenario, the system was subjected to an increasing message load originating from a single simulated sensor. The message generation rate was progressively increased over time, allowing for an observation of how each protocol handles escalating throughput demands and the corresponding impact on end-to-end latency. The payload for each message was approximately 150 bytes, structured as a JSON object containing sensor metadata and a randomly generated temperature value, similar to the example provided in Section 5.2. All tests were conducted under nominal network conditions within the Azure Kubernetes Service environment, without any artificially induced impairments.

The performance is primarily evaluated using two key metrics observed over the duration of the test (approximately 26 minutes):

1. **End-to-end Latency:** The 95th percentile of the time taken for a message to travel from the edge sensor source to the cloud processing service.
2. **Message Throughput:** The number of unique messages successfully received and processed per second by the cloud service for each protocol.

5.3.1 Latency and Throughput Under Increasing Load

Figure 5.1 illustrates the 95th percentile end-to-end latency for each of the seven protocol configurations as the message sending rate increases over the test period. Concurrently, Figure 5.2 displays the corresponding message throughput (messages received per second)

for each protocol. These two figures should be interpreted together, as the ability of a protocol to maintain low latency is often coupled with its capacity to sustain high throughput. It is also critical to consider that some protocols might achieve a certain received throughput by dropping messages, while others might queue extensively, leading to high latencies but no overt message loss up to a certain point.

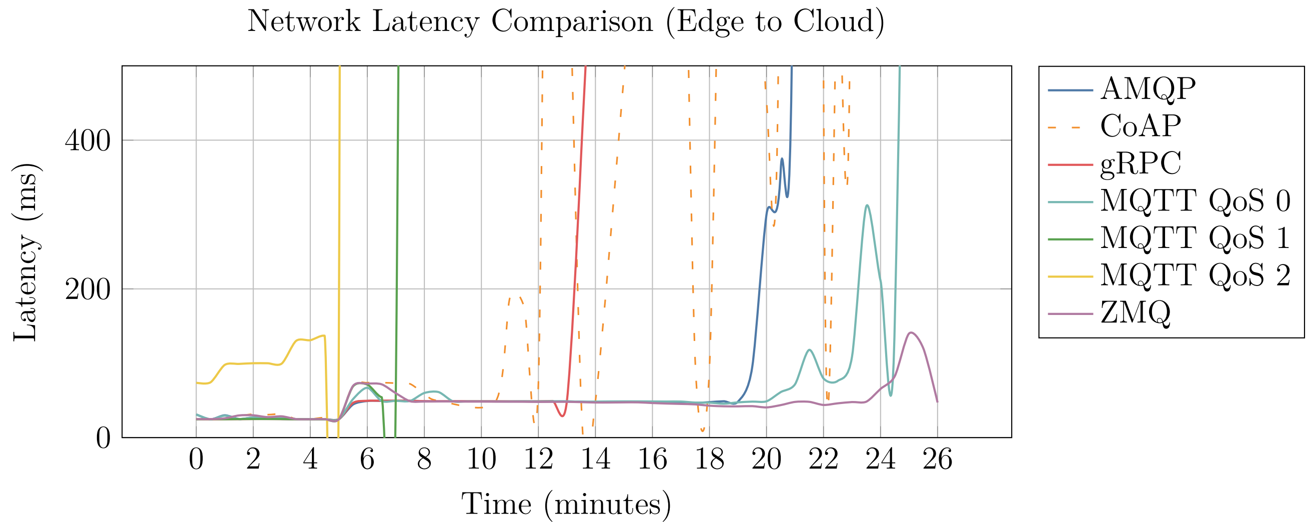

Figure 5.1: Network Latency Comparison (95th Percentile, Edge to Cloud) under Increasing Message Load

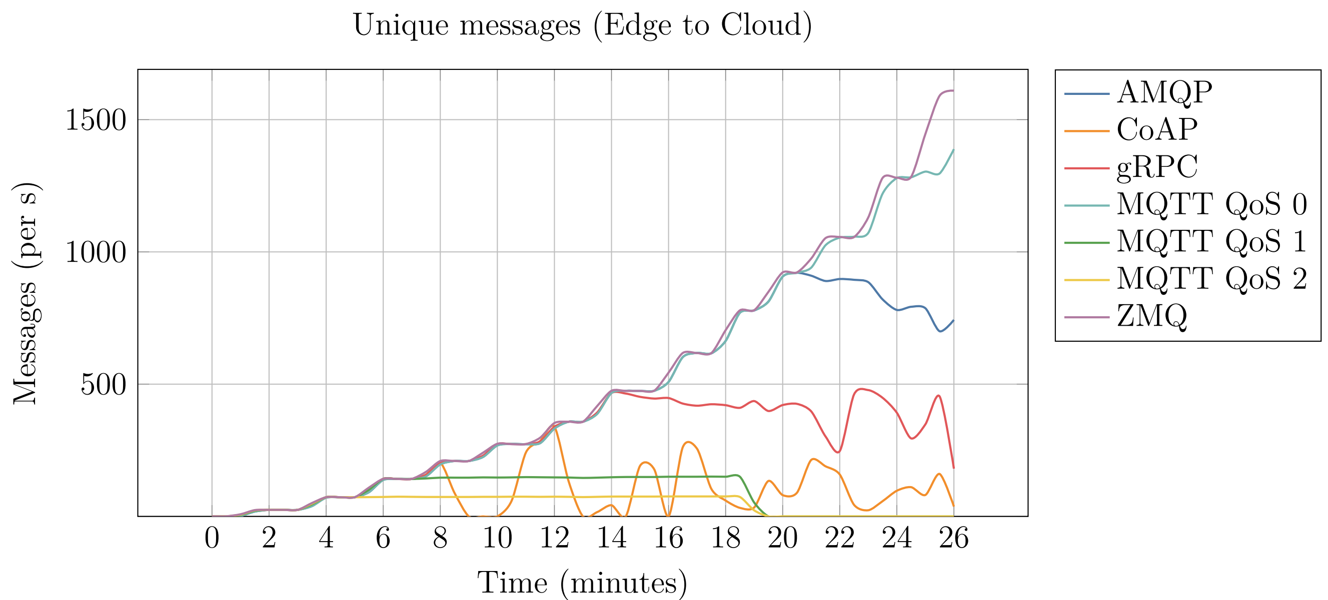

Figure 5.2: Unique Messages Received (per second, Edge to Cloud) under Increasing Message Load

Observing Figure 5.1 and Figure 5.2, several distinct performance characteristics emerge for the protocols under evaluation as the offered load escalates:

ZeroMQ (ZMQ): ZeroMQ demonstrates remarkable resilience and performance throughout the test. As seen in Figure 5.2, its received message rate closely tracks the increasing sending rate, successfully processing over 3000 messages per second towards the end of the test. Correspondingly, Figure 5.1 shows that ZeroMQ maintains a consistently low 95th percentile latency, generally remaining well below 50 ms and exhibiting only a modest increase to around 150 ms even at the highest throughput levels. This performance underscores ZeroMQ's suitability for high-throughput, low-latency scenarios, attributable to its lightweight, brokerless architecture and efficient ZMTP messaging.

MQTT (Message Queuing Telemetry Transport): The performance of MQTT varies significantly based on the Quality of Service (QoS) level employed:

- **MQTT QoS 0 (At-most-once):** This configuration initially performs well, maintaining low latency similar to ZeroMQ. However, as the message rate approaches approximately 2500 messages per second (around the 24-minute mark in Figure 5.2), its throughput begins to plateau and then decline. Concurrently, its latency (Figure 5.1) starts to increase significantly, eventually exceeding 300 ms. This suggests that while efficient at lower loads due to its *fire and forget* nature, the Mosquitto broker or the TCP connections begin to experience contention or processing delays at very high message influx rates, potentially leading to message drops which contribute to the declining received throughput.

- **MQTT QoS 1 (At-least-once) and QoS 2 (Exactly-once):** These reliable MQTT configurations prioritize message delivery over maintaining low latency or high throughput. As shown in Figure 5.2, their throughput capabilities are significantly lower; QoS 1 peaks erratically around 250-300 messages per second, while QoS 2 struggles to sustain rates above 150-200 messages per second. Their throughput lines appear relatively *stable* (though low) before the test for these QoS levels was terminated around the 19-minute mark. This apparent stability in received messages masks the severe degradation in latency. Figure 5.1 reveals that MQTT QoS 2 is the first to exhibit extreme latency, consistently exceeding 100 ms very early (around 2 minutes) and climbing rapidly. MQTT QoS 1 follows a similar pattern, with latency rising sharply around the 4-minute mark. For both QoS 1 and 2, latencies escalate to many seconds (potentially up to 100 seconds or more if allowed to continue indefinitely) as the system attempts to guarantee message delivery by

queuing or retransmitting, without dropping packets at the MQTT protocol layer. The test runs for QoS 1 and 2 were halted prematurely (around 19 minutes) because their latencies had become unacceptably high, and further continuation would not yield new insights into their behavior under these overload conditions.

CoAP (Constrained Application Protocol): CoAP, operating over UDP and using Non-confirmable (NON) messages for this baseline test, displays a highly variable latency profile as seen in Figure 5.1. There are periods of low latency, but also frequent, very large spikes where latency exceeds 1 second. Its throughput (Figure 5.2) is also erratic, peaking around 500 messages per second but often dropping significantly, indicative of substantial packet loss. This behavior is characteristic of UDP-based protocols without robust, built-in congestion control or guaranteed delivery at the transport layer when faced with high sending rates. The NON messages minimize overhead but also mean that packet loss is not recovered, directly impacting the received throughput and contributing to the observed latency spikes (which might represent the delay of sporadically successful messages amidst many lost ones).

AMQP (Advanced Message Queuing Protocol): AMQP, implemented via RabbitMQ, demonstrates robust performance, scaling well to higher throughputs than gRPC in this test. Figure 5.2 shows AMQP's throughput increasing steadily and sustaining rates up to approximately 1700-1800 messages per second (around the 18-20 minute mark) before it starts to show signs of instability and decline. Its 95th percentile latency (Figure 5.1) remains relatively low and stable during this ramp-up phase. Beyond this saturation point, as the offered load continues to increase, the latency begins to rise more noticeably and exhibits larger fluctuations. The structured messaging and queueing capabilities of AMQP allow it to handle a significant volume of messages efficiently, but like other TCP-based brokered systems, it eventually encounters a bottleneck at very high, unbatched message frequencies.

gRPC (Google Remote Procedure Call): gRPC, leveraging HTTP/2 over TCP, initially maintains low latency. As shown in Figure 5.2, its throughput scales to approximately 900-1000 messages per second (around 12-14 minutes). Beyond this point, its received throughput starts to fluctuate and then declines significantly, particularly after the 22-minute mark, suggesting message loss or processing failure at higher loads. The latency (Figure 5.1) remains relatively stable and low until this throughput inflection point, after which it begins to rise sharply, exhibiting large spikes and eventually exceeding 1 second. This indicates that while gRPC is efficient for moderate loads, in this testbed con-

figuration, it encounters a performance ceiling earlier than AMQP, possibly related to the overhead of HTTP/2 stream management or application-level request/response processing under extreme duress, leading to message drops when overwhelmed.

5.3.2 Summary of Baseline Observations

The baseline tests (Section 5.3), characterized by an increasing message rate from a single sensor, reveal clear performance differentiations:

- **ZeroMQ** stands out for its ability to sustain the highest throughput (over 3000 messages/sec) while maintaining the lowest and most stable 95th percentile latency throughout the entire test duration.
- **AMQP** demonstrates strong performance, achieving high throughput (around 1700-1800 messages/sec) with stable latency before reaching its saturation point.
- **MQTT QoS 0** performs well up to a high throughput (around 2500 messages/sec) before latency increases and received throughput drops, indicating it is a strong contender for high-volume, best-effort delivery where some message loss at peak load is tolerable.
- **gRPC** shows good performance up to a moderate-to-high throughput range (around 900-1000 messages/sec) before its latency characteristics degrade and received throughput declines, indicating message loss under heavy load.
- **MQTT QoS 1 and QoS 2** exhibit significantly lower throughput capacities (around 150-300 messages/sec). Their primary characteristic under overload is extremely high latency as they attempt to ensure message delivery without dropping packets at the MQTT layer, rather than a drop in received messages (until the test was halted for them). This highlights the severe performance cost of their reliability mechanisms for each message at high frequencies.
- **CoAP (NON messages over UDP)** exhibits the most erratic latency and the lowest consistent throughput, with clear indications of significant packet loss at higher sending rates.

These baseline results establish a clear hierarchy in terms of raw throughput capacity, latency stability, and behavior under increasing message load. ZeroMQ leads in raw performance, followed by AMQP and MQTT QoS 0 in terms of sustainable throughput before

significant degradation. gRPC offers good performance but saturates earlier than AMQP. The reliable MQTT QoS levels trade performance for guaranteed delivery, resulting in extreme latencies, while CoAP (NON) struggles with reliability and consistency at high loads.

5.4 Performance Under Network Impairment: 10% Packet Loss

To assess the resilience and behavior of the communication protocols under adverse network conditions, a test was conducted with a persistent, artificially induced packet loss of 10% on the communication link between the Edge and the Cloud. This scenario is representative of challenging real-world deployments, such as those involving wireless communication or congested network segments. During this test, the message load was also varied: for the first 10 minutes, data was generated at a rate equivalent to 50 messages per second; subsequently, the rate was increased to 150 messages per second for the remainder of the test (approximately 8 more minutes). These message rates were chosen as they were comfortably handled by most protocols in the baseline tests (Section 5.3), allowing the impact of packet loss to be the primary differentiating factor.

The key metrics observed are again the 95th percentile end-to-end latency (Figure 5.3) and the unique messages received per second (Figure 5.4).

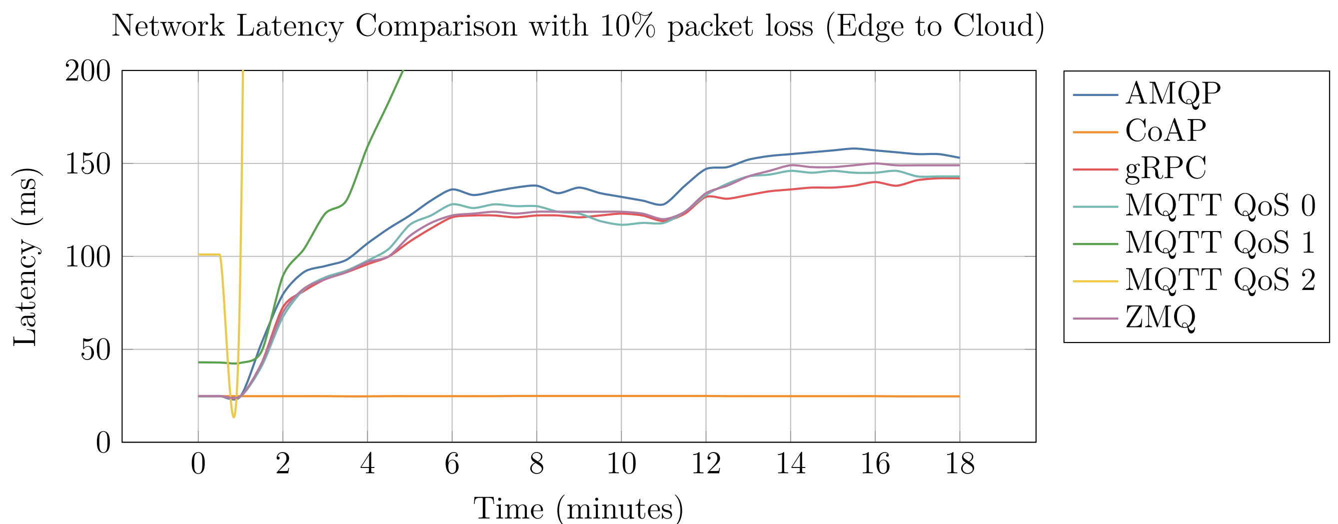

Figure 5.3: Network Latency Comparison (95th Percentile, Edge to Cloud) with 10% Packet Loss

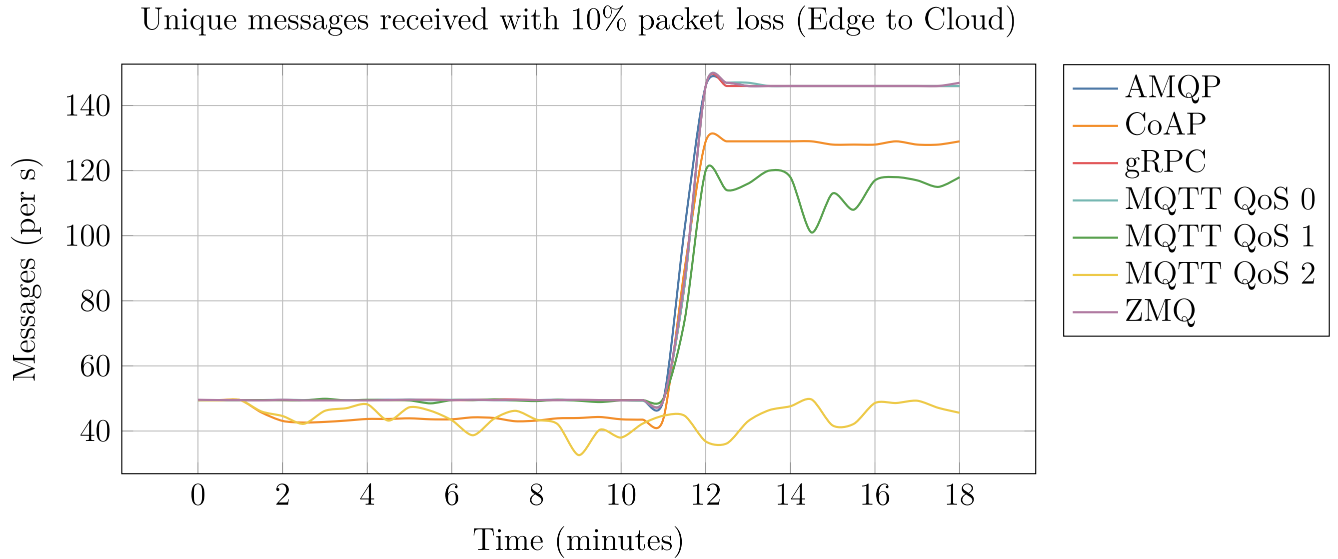

Figure 5.4: Unique Messages Received (per second, Edge to Cloud) with 10% Packet Loss

5.4.1 Impact on Latency and Message Throughput

The introduction of a 10% packet loss significantly alters the performance landscape, revealing how different protocol mechanisms cope with unreliable transport:

TCP-Based Protocols (MQTT, ZeroMQ, AMQP, gRPC): A common characteristic observed across all TCP-based protocols (MQTT - all QoS levels, ZeroMQ, AMQP, and gRPC) is their ability to maintain message delivery integrity despite the 10% packet loss. As seen in Figure 5.4, after an initial stabilization period and once the message rate increases at the 10-minute mark, these protocols successfully deliver close to the offered load (initially 50 messages/sec, then 150 messages/sec). There are no significant persistent message drops attributable to the packet loss itself; the received throughput matches the sending rate once the system adapts.

However, this reliability comes at the cost of increased latency, as clearly depicted in Figure 5.3. The TCP transport layer's inherent mechanisms for detecting lost packets (e.g., via duplicate ACKs or timeouts) and retransmitting them are responsible for this behavior. When a packet containing application data is lost, TCP will retransmit it, ensuring the application eventually receives the data. This retransmission process inherently adds delay. The 10% packet loss means that, on average, 1 in 10 packets needs retransmission, significantly impacting the round-trip time and, consequently, the end-to-end latency.

Among the TCP-based protocols:

- **MQTT QoS 0, MQTT QoS 1, AMQP, gRPC, and ZeroMQ** all exhibit a noticeable increase in their 95th percentile latency compared to the baseline no-loss scenario. Initially, with 50 messages/sec, their latencies hover around 25-50 ms, but quickly rise to the 100-150 ms range once packet loss starts to take effect and retransmissions occur. When the load increases to 150 messages/sec around the 10-12 minute mark, there's a further, albeit smaller, sustained increase in latency for these protocols, generally stabilizing between 120 ms and 160 ms. The tight clustering of their latency profiles suggests that the TCP retransmission mechanism becomes a dominant factor in determining latency under these conditions, somewhat overshadowing the finer differences in their application-level protocols seen in the baseline tests.

- **MQTT QoS 2** displays the most significant latency impact. While it also maintains message delivery (Figure 5.4), its 95th percentile latency (Figure 5.3) escalates much more rapidly and to a higher degree than the other TCP-based protocols. It quickly surpasses 100 ms within the first minute of packet loss and continues to climb, reaching and exceeding 200 ms. This is because packet loss can disrupt its more complex four-way handshake (PUBLISH, PUBREC, PUBREL, PUBCOMP). If any of these MQTT control packets are lost, the entire sequence for that message is stalled until TCP retransmits the lost MQTT packet and the handshake can resume. This compounding effect of potential losses at multiple stages of its delivery protocol makes MQTT QoS 2 particularly sensitive to packet loss in terms of latency.

CoAP (Constrained Application Protocol): CoAP, using Non-confirmable (NON) messages over UDP, behaves distinctly. As shown in Figure 5.4, its received message throughput is consistently lower than the offered load. When sending 50 messages/sec, it receives approximately 40-45 messages/sec. When the rate increases to 150 messages/sec, it receives around 125-130 messages/sec. This shortfall of roughly 10-15% directly reflects the 10% packet loss rate introduced at the network layer, as CoAP NON messages have no built-in retransmission mechanism. Lost UDP packets containing CoAP NON messages are lost permanently.

The most striking feature of CoAP in this scenario is its latency profile (Figure 5.3). Despite the 10% packet loss, the 95th percentile latency for the messages that do arrive remains exceptionally low and stable, consistently around 25 ms throughout the entire

test, irrespective of the change in sending rate. This occurs because UDP is connectionless; successfully transmitted packets are not delayed by retransmission queues or complex state management that TCP employs. If a CoAP NON message makes it through the network without being dropped, its transit time is primarily dictated by the raw network path delay. This makes CoAP (NON) a unique proposition: if an application can tolerate a certain level of message loss inherent to the network conditions and prioritizes minimal latency for successfully delivered messages, CoAP (NON) offers a compelling advantage.

5.4.2 Summary of Performance with 10% Packet Loss

The introduction of a 10% packet loss (Section 5.4) highlights critical operational differences:

- **TCP-based protocols (MQTT, AMQP, gRPC, ZeroMQ)** successfully overcome the packet loss to deliver nearly all messages, demonstrating the effectiveness of TCP's reliability mechanisms. However, this comes at the cost of a significant increase in end-to-end latency (generally stabilizing between 120-160 ms for most, higher for MQTT QoS 2) due to packet retransmissions.
- **MQTT QoS 2** is particularly affected by packet loss in terms of latency due to the potential for disruption at multiple stages of its four-way handshake for each message.
- **CoAP (NON messages over UDP)** experiences message loss roughly proportional to the network packet loss rate. However, for the messages that are successfully delivered, it maintains exceptionally low and stable latency (around 25 ms).
- This test underscores a fundamental trade-off: TCP-based protocols prioritize reliable delivery over latency in lossy conditions, while CoAP (NON over UDP) prioritizes low latency for successful deliveries at the expense of message reliability. The choice between them depends critically on whether the application can tolerate data loss or requires guaranteed delivery even if it means higher and more variable delays.

5.5 Impact of Increasing Message Payload Size

This section evaluates the performance of the communication protocols when subjected to a constant message rate but with progressively increasing message payload sizes. In this

scenario, five simulated sensors collectively generated a total of 5 messages per second. The size of the JSON payload for each message was systematically increased at distinct intervals over the test duration (approximately 24 minutes). The payload sizes tested were: 150 bytes, 1 kB, 2 kB, 4 kB, 8 kB, 16 kB, 32 kB, 64 kB, 128 kB, 256 kB, 512 kB, and finally 1024 kB (1 MB). This test aims to identify how protocol overheads, serialization/deserialization efficiencies, and transport layer characteristics affect performance as the data volume per message grows. All tests were conducted under nominal network conditions.

Performance is analyzed using the following metrics:

1. **End-to-end Latency:** The 95th percentile of message latency (Figures 5.5 and 5.6).

2. **Message Throughput:** The number of unique messages successfully received per second (Figure 5.7).

3. **Data Throughput:** The total volume of data (bytes) successfully received over time (Figure 5.8).

Figure 5.6 provides a focused view of the latency for protocols that maintained lower latencies, while Figure 5.5 shows the full scale, including protocols that experienced extreme latency increases.

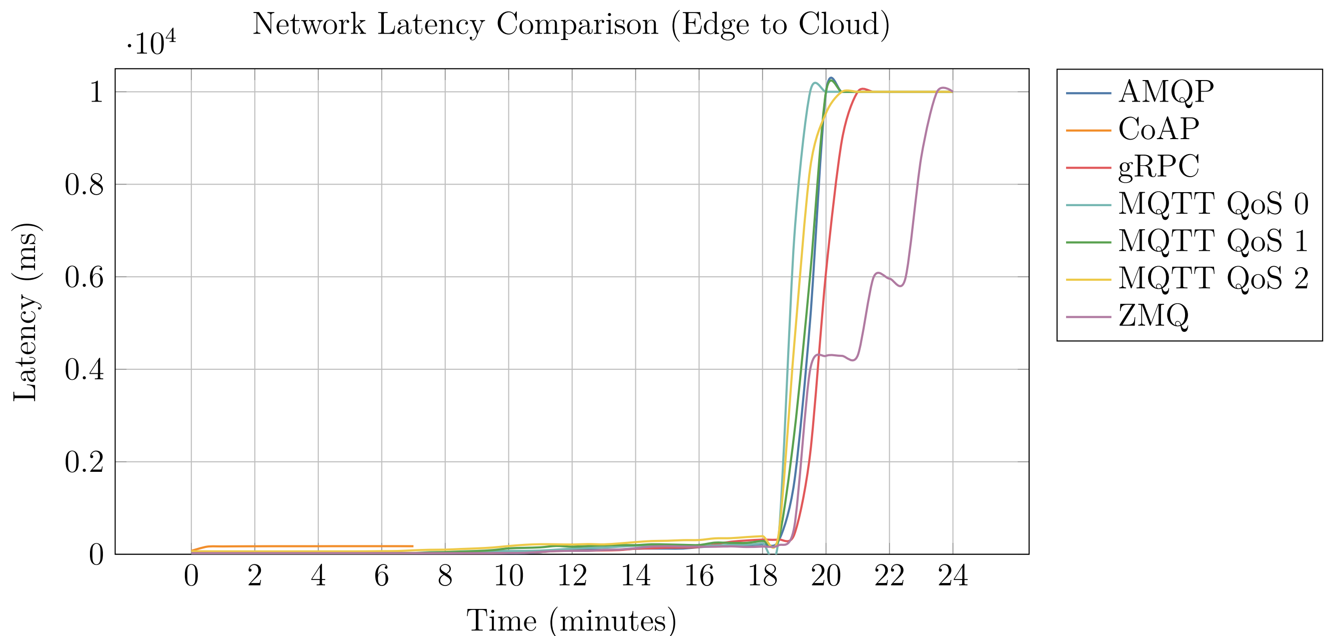

Figure 5.5: Network Latency Comparison (95th Percentile, Edge to Cloud) with Increasing Payload Size

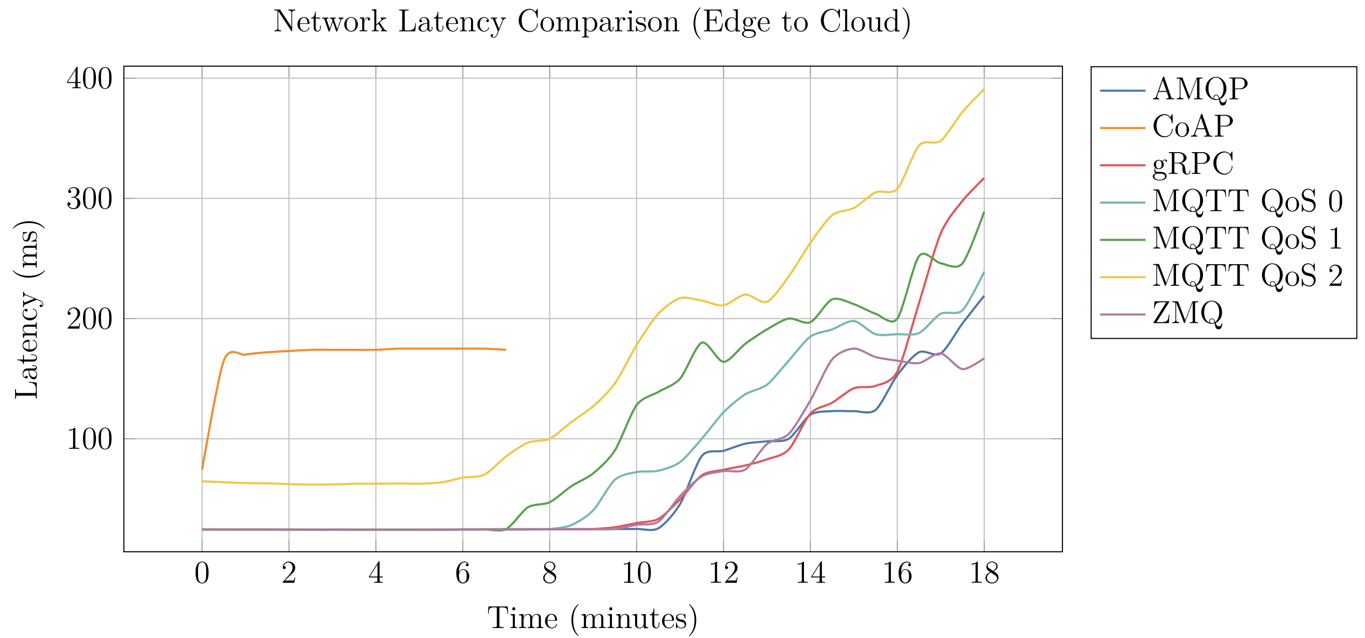

Figure 5.6: Network Latency Comparison (95th Percentile, Edge to Cloud) - Focused View for Lower Latency Protocols with Increasing Payload Size

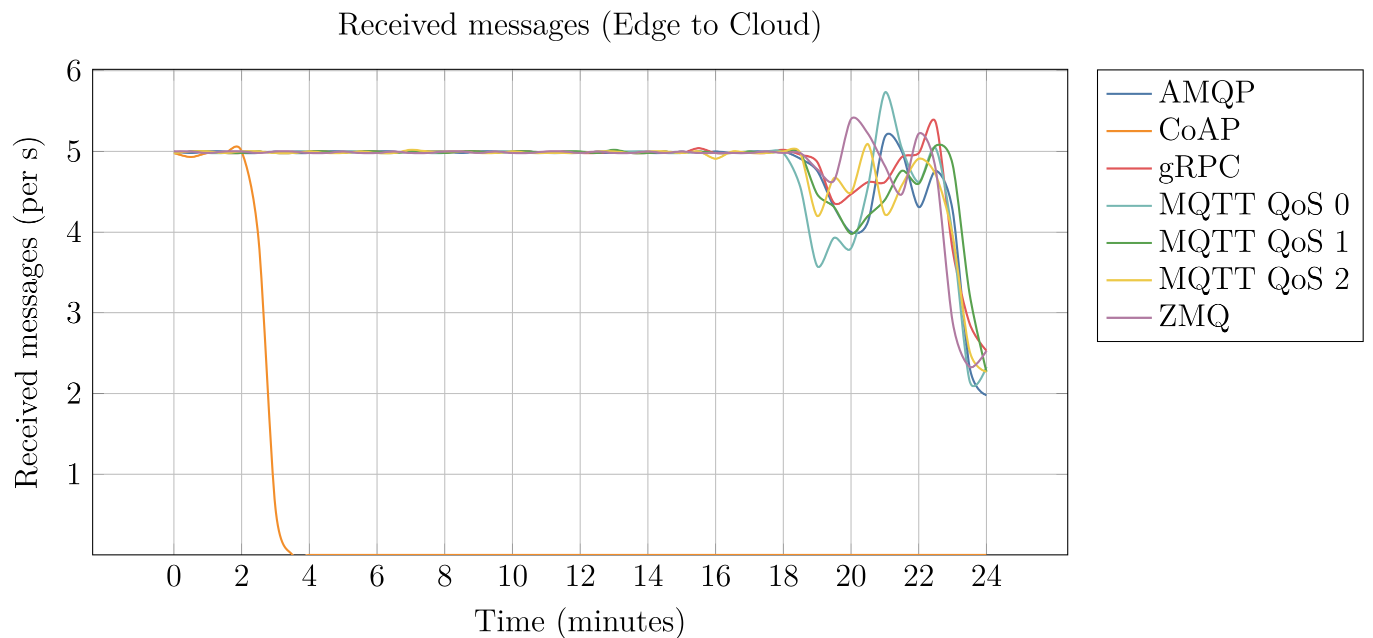

Figure 5.7: Unique Messages Received (per second, Edge to Cloud) with Increasing Payload Size

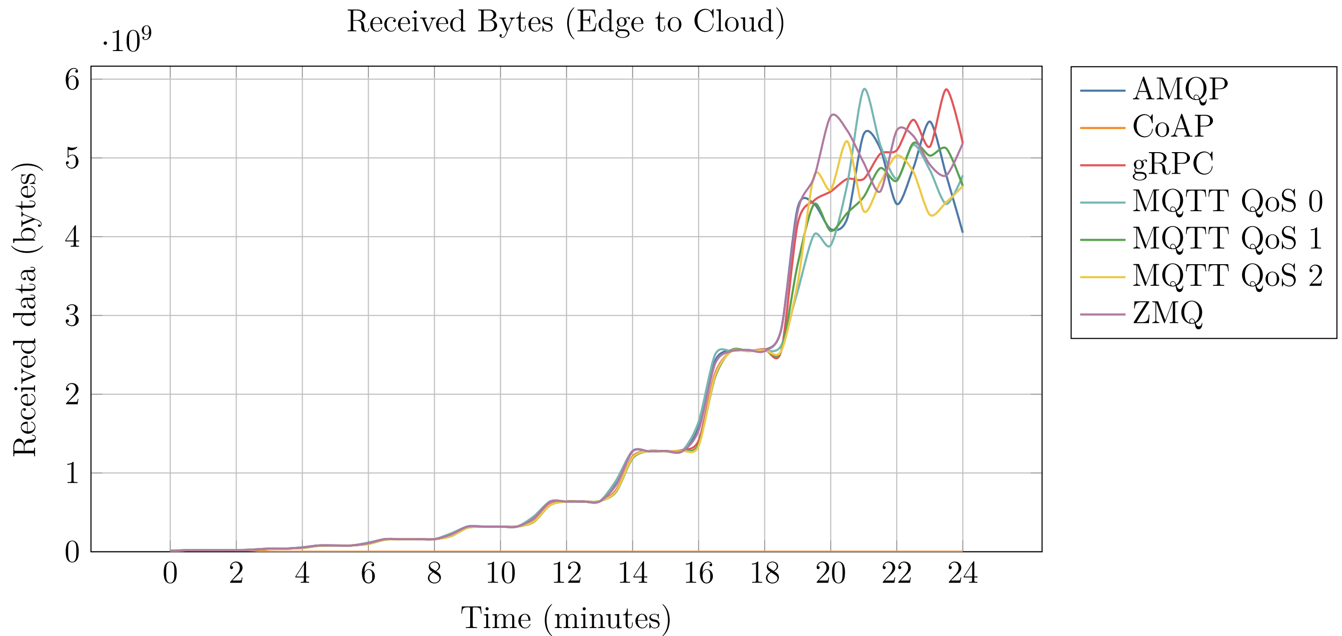

Figure 5.8: Cumulative Bytes Received (Edge to Cloud) with Increasing Payload Size

5.5.1 Observed Latency, Message, and Data Throughput

The results from this payload size escalation test reveal significant differences in how protocols manage larger data transfers:

Message Throughput (Messages per Second): As observed in Figure 5.7, most protocols (AMQP, gRPC, MQTT QoS 0, QoS 1, QoS 2, and ZMQ) successfully maintain the target message rate of 5 messages per second for a significant portion of the test, specifically up to the point where payload sizes become very large (around 256 kB to 512 kB, corresponding to approximately 18-20 minutes into the test). Beyond this point, some protocols begin to struggle to maintain the 5 messages/sec rate, indicating that processing or transmitting these very large individual messages introduces bottlenecks that slow down the overall message handling frequency.

CoAP (Constrained Application Protocol): CoAP exhibits a starkly different behavior. As seen in Figure 5.7, CoAP's message throughput drops drastically to near zero very early in the test, around the 2-3 minute mark. This corresponds to the point where the payload size increases beyond a small initial value (likely beyond what a single UDP datagram can comfortably carry without IP fragmentation or requiring CoAP block-wise transfer). The current implementation does not utilize CoAP's block-wise transfer mech-

anism (RFC 7959), which is essential for reliably transmitting messages larger than the path MTU over UDP. Without block-wise transfer, large CoAP messages sent over UDP are highly susceptible to being dropped by the network or failing at the receiver, leading to the observed collapse in message throughput. Consequently, CoAP is largely absent from meaningful comparison in the latency and byte throughput figures for larger payloads.

Data Throughput (Bytes Received): Figure 5.8 shows the cumulative bytes received over time. For the protocols that successfully transmit messages, the curves show stepwise increases in the slope each time the payload size is increased, reflecting the higher data volume per message. Protocols like ZMQ, AMQP, gRPC, and MQTT QoS 0 track closely together for a large portion of the test, indicating efficient data transfer. MQTT QoS 1 and QoS 2 also transfer the data but, as will be seen with latency, at a higher cost. The flattening of the curves towards the end for some protocols (e.g., around 20-22 minutes) corresponds to the drop in message throughput seen in Figure 5.7 when handling extremely large payloads (512kB, 1MB).

End-to-End Latency: The impact of increasing payload size on latency is varied and highly informative, as shown in Figure 5.5 (full scale) and Figure 5.6 (focused view):

- **CoAP:** As expected from its message throughput failure, CoAP's latency (Figure 5.6) becomes largely irrelevant after the initial small payloads. The brief period where it does transmit (payloads around 150 bytes) shows a relatively higher initial latency (around 60-70 ms) compared to other protocols, which then spikes as message delivery fails with larger payloads.

- **ZeroMQ (ZMQ):** ZMQ consistently exhibits the lowest latency across almost all payload sizes, as seen in Figure 5.6. Even as payloads increase up to 128 kB (around 16-18 minutes), its 95th percentile latency remains impressively low, generally below 200 ms. This underscores its efficiency in handling bulk data transfer with minimal protocol overhead. A slight upward trend is visible, which is expected as more data per message requires more transmission and processing time.

- **MQTT QoS 0, AMQP, and gRPC:** These three protocols show very similar latency behavior for smaller to medium payloads (up to around 32-64 kB, roughly 12-16 minutes). Their latencies (Figure 5.6) gradually increase with payload size, starting from very low values (sub-20ms) and rising to around 100-150 ms. As payload sizes grow larger (128 kB and beyond), their latencies begin to diverge more and increase more steeply. gRPC's latency tends to climb more sharply than AMQP

and MQTT QoS 0 at the largest payload sizes (Figure 5.5). This suggests that while efficient with Protocol Buffers, the overhead of HTTP/2 or the application-level processing for very large gRPC messages becomes significant.

- **MQTT QoS 1 and QoS 2:** These reliable MQTT variants start with higher baseline latencies compared to QoS 0, ZMQ, AMQP, and gRPC, even for the smallest 150-byte payloads (around 60-70 ms for QoS 2, slightly less for QoS 1, as seen in Figure 5.6). As the payload size increases, their latencies escalate much more dramatically than the other TCP-based protocols. Figure 5.5 shows that by the time payloads reach 8-16 kB (around 8-10 minutes), MQTT QoS 2 latency is already climbing past 200 ms, and both QoS 1 and QoS 2 exhibit very high latencies (approaching or exceeding 400 ms and then rapidly increasing to many seconds) for payloads of 64 kB and larger. The per-message acknowledgment handshakes (two-way for QoS 1, four-way for QoS 2) become increasingly burdensome as the amount of data being acknowledged in each step grows, leading to significant queuing and processing delays. For the largest payloads (512kB, 1MB), their latencies become extreme, going off the scale of Figure 5.5 (indicated by the lines hitting the top and representing values in the tens of seconds).

5.5.2 Summary of Performance with Increasing Payload Size

The evaluation with increasing message payload sizes (Section 5.5) reveals the following key insights:

- **CoAP's Limitation:** CoAP, without implemented block-wise transfer, is unsuitable for transmitting messages larger than a few hundred bytes to 1 kB, as its message throughput collapses almost immediately.

- **ZeroMQ's Efficiency:** ZMQ consistently provides the lowest latency across a wide range of payload sizes, making it highly effective for applications requiring timely delivery of both small and large messages.

- **MQTT QoS 0, AMQP, gRPC Grouping:** These protocols perform comparably for small to medium payloads, with latency gradually increasing. For very large payloads, their differentiation becomes more apparent, with ZMQ generally leading, followed by AMQP and MQTT QoS 0, while gRPC's latency climbs more steeply at the extreme end.

- **Reliable MQTT Overhead:** MQTT QoS 1 and especially QoS 2 incur a substantial latency penalty as payload sizes increase. The overhead of their per-message reliability handshakes becomes a dominant factor, leading to extremely high latencies for large messages.

- **Throughput Maintenance vs. Latency Trade-off:** Most TCP-based protocols maintain the target 5 messages/second rate until very large payloads cause processing or network saturation, at which point message throughput can decline. The primary differentiator for sustained message delivery is how latency is affected by the increasing data volume per message.

This test highlights that while many protocols can handle a fixed rate of small messages, their ability to efficiently transport larger individual messages varies significantly, with simpler, lower-overhead protocols like ZeroMQ showing advantages, and protocols with extensive per-message handshakes (like MQTT QoS 2) struggling significantly with latency.

5.6 Impact of Increasing Static Network Latency

This section investigates the performance of the communication protocols when subjected to a progressively increasing static network latency applied to the link between the Edge and Cloud components. For this test, a constant message load of 50 messages per second was generated (from 50 simulated sensors, each sending 1 message per second). The artificial static latency was incremented every minute, following this progression in milliseconds: 25, 50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200, 1300, 1400, and finally 1500 ms. This methodical increase allows for observation of how additional round-trip time (RTT) affects the end-to-end latency and message delivery throughput of each protocol.

The primary metrics for this analysis are the 95th percentile end-to-end latency (Figure 5.9) and the unique messages received per second (Figure 5.10).

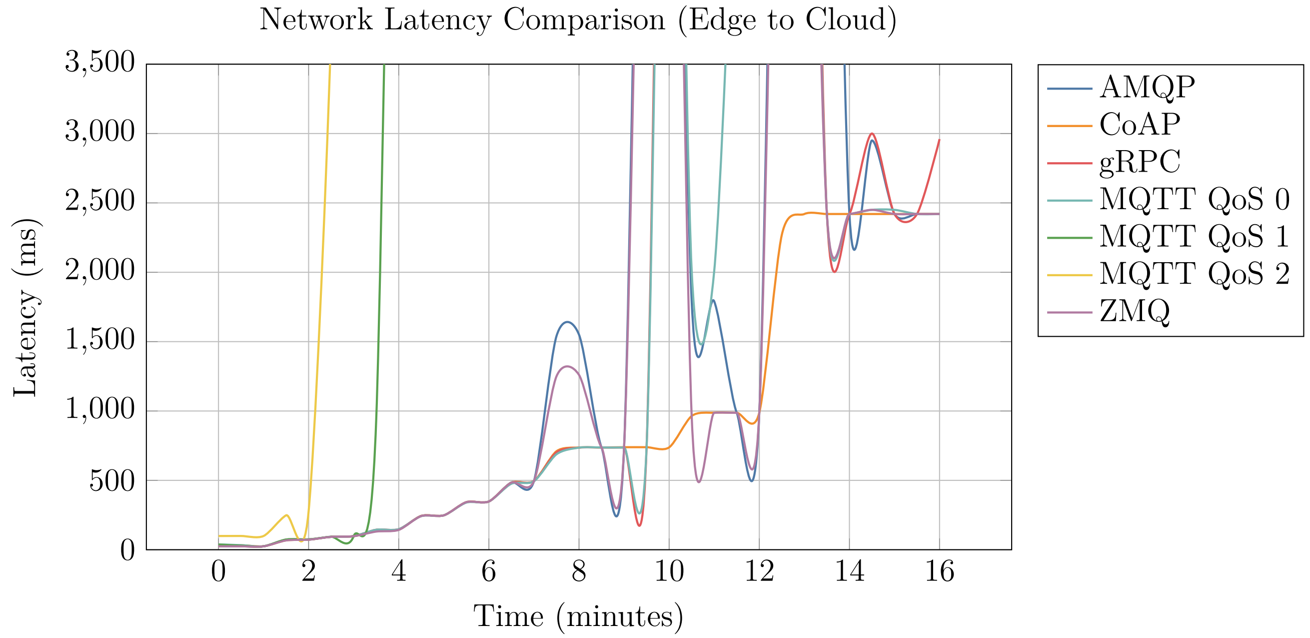

Figure 5.9: Network Latency Comparison (95th Percentile, Edge to Cloud) with Incrementally Increasing Static Network Latency

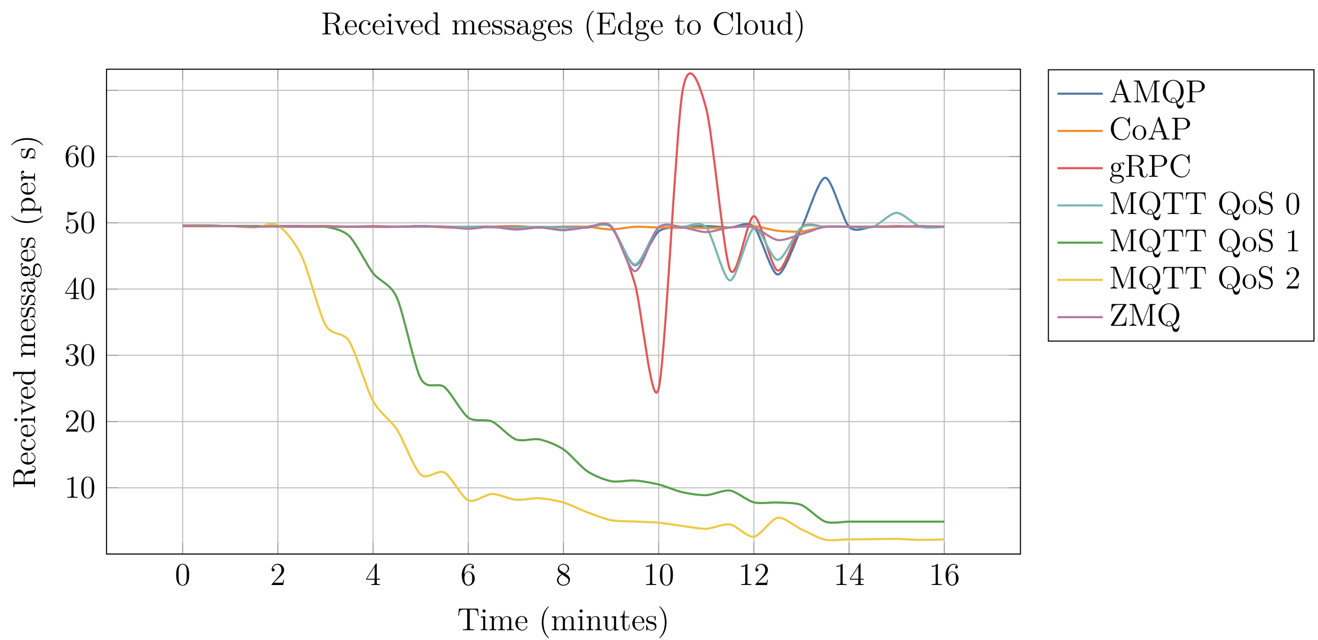

Figure 5.10: Unique Messages Received (per second, Edge to Cloud) with Incrementally Increasing Static Network Latency

5.6.1 Effect on Latency and Message Throughput

The introduction of increasing static network latency reveals distinct responses from the protocols, particularly differentiating those reliant on connection-oriented, acknowledged transport from those using connectionless transport:

End-to-End Latency (Figure 5.9): As anticipated, all protocols exhibit an increase in their measured 95th percentile end-to-end latency as the underlying static network latency is increased. However, the magnitude and rate of this increase vary significantly:

- **CoAP (Constrained Application Protocol):** CoAP (using NON messages over UDP) demonstrates the most direct and proportional response to the added static latency. Its measured end-to-end latency increases almost linearly with the added network latency, maintaining a relatively small delta above the artificial latency value. For example, with 500ms added static latency, its measured latency is slightly above 500ms. This behavior is expected because, for successfully delivered NON messages, the primary component of end-to-end latency becomes the one-way network transit time. The optional NON response from the server would add another one-way transit time if measured as a round trip, but for one-way data push, CoAP (NON) shows minimal additional protocol-induced delay beyond the network itself.
- **MQTT QoS 0, ZeroMQ, AMQP, and gRPC:** These TCP-based protocols also show a clear increase in latency that correlates with the added static network latency. However, their measured latencies are generally higher than CoAP's at any given static latency level. This is due to the overheads of TCP connection management, potential TCP-level acknowledgments contributing to measured RTTs if the application waits for certain confirmations, and the protocol-specific processing at each end. As the static latency increases to very high values (e.g., 1000 ms and above, around 10 minutes onwards), the end-to-end latencies for these protocols climb significantly, often exceeding the added static latency by a considerable margin (e.g., reaching 2000-3000 ms when 1000-1500 ms static latency is applied). This indicates that high RTTs can start to impede TCP windowing mechanisms, flow control, or cause internal queuing within the application or broker if processing cannot keep pace due to delayed acknowledgments or responses. gRPC, AMQP, ZMQ, and MQTT QoS 0 exhibit somewhat similar trends in latency increase, though with some variations in their final values.

- **MQTT QoS 1 and QoS 2:** These reliable MQTT configurations are the most severely affected by increasing static network latency. Their end-to-end latencies escalate very rapidly and dramatically. MQTT QoS 2's latency begins to climb steeply almost immediately as static latency is introduced, quickly reaching multi-second values. MQTT QoS 1 follows a similar, though slightly less severe, trajectory. This extreme sensitivity is due to their multi-round-trip acknowledgment handshakes per message (two-way for QoS 1, four-way for QoS 2). Each leg of these handshakes is now subject to the increased static network latency. For instance, with QoS 2, a 500ms static one-way latency could theoretically add 2 seconds (4 * 500ms) to the protocol-level exchange time for a single message, plus TCP overheads. This explains why their latencies reach extremely high values (off the scale of Figure 5.9, up to 10000 ms) very early in the test.

Message Throughput (Figure 5.10): The impact on message throughput also varies:

- **CoAP, MQTT QoS 0, ZeroMQ, AMQP, and gRPC:** For a significant portion of the test, these protocols largely maintain the target throughput of 50 messages per second, even as static latency increases up to around 700-800 ms (approximately 7-8 minutes into the test). This indicates that, while latency is increasing, the protocols and their underlying TCP connections (for non-CoAP) can still handle the message rate. However, as static latency becomes excessively high (1000 ms and beyond), some of these protocols, notably gRPC, begin to show instability and dips in their received message throughput. This suggests that extreme RTTs can start to affect the ability to reliably sustain the message flow, perhaps due to internal timeouts, buffer overflows as messages queue due to slow acknowledgments, or TCP congestion window issues.

- **MQTT QoS 1 and QoS 2:** These protocols exhibit a rapid decline in their ability to sustain the 50 messages/second throughput. As seen in Figure 5.10, MQTT QoS 2's throughput starts to drop almost immediately when static latency is introduced, falling significantly within the first few minutes. MQTT QoS 1 maintains the throughput for a bit longer but also begins to degrade noticeably as static latency climbs past 100-200 ms. The long delays in completing their per-message acknowledgment handshakes directly limit how many new messages can be processed or sent per unit of time. Effectively, the system becomes "backlogged" waiting for confirmations, and the message processing rate cannot keep up with the sending rate.

5.6.2 Summary of Performance with Increasing Static Network Latency

The introduction of progressively increasing static network latency (Section 5.6) reveals several key behaviors:

- **CoAP (NON over UDP)** demonstrates the most resilient latency profile in terms of proportionality, with its end-to-end latency closely tracking the added static network latency plus a small processing overhead. It maintains its message throughput fairly well, as its *fire-and-forget* nature is less impacted by RTT for individual message sends.
- **TCP-based protocols with simpler exchanges (MQTT QoS 0, ZMQ, AMQP, gRPC)** show increasing end-to-end latency that is greater than just the added static latency, reflecting TCP and application-level overheads. They generally maintain throughput until very high static latencies cause instability or processing bottlenecks.
- **MQTT QoS 1 and QoS 2** are extremely sensitive to increases in static network latency. Their multi-round-trip reliability mechanisms cause their end-to-end latencies to escalate dramatically and their sustainable message throughput to degrade rapidly. This highlights their unsuitability for applications requiring timely data delivery over high-latency links if per-message guarantees are strictly enforced at these QoS levels.
- The results underscore that protocols with more "chatter" or more round-trips per logical operation (like MQTT QoS 1 and especially QoS 2) suffer disproportionately more from increased network RTT. Simpler, one-way or lean two-way exchanges are less impacted in terms of their ability to maintain throughput, though all experience increased absolute latency.

5.7 Performance Under Combined Network Impairments

This section evaluates the protocol performance under a more complex scenario where multiple network impairments are simultaneously introduced. This test aims to simulate a

challenging but realistic network environment characterized by inherent delays, variability in those delays (jitter), and a degree of unreliability (packet loss). For this evaluation, a constant message load of 40 messages per second was generated, with each message having a payload size of 500 bytes (resulting in a total data rate of approximately 20 kB/s). The following network conditions were applied to the link between the Edge and Cloud components using the ‘tc‘ (traffic control) utility with NetEm:

- Average static delay: 50 ms
- Jitter (variation around the average delay): ±50 ms (uniformly distributed, resulting in actual delays between 0 ms and 100 ms for each packet, averaging 50 ms)
- Packet loss: 5%

This test ran for approximately 8 minutes. The primary metrics observed are the 95th percentile end-to-end latency (Figure 5.11) and the unique messages received per second (Figure 5.12).

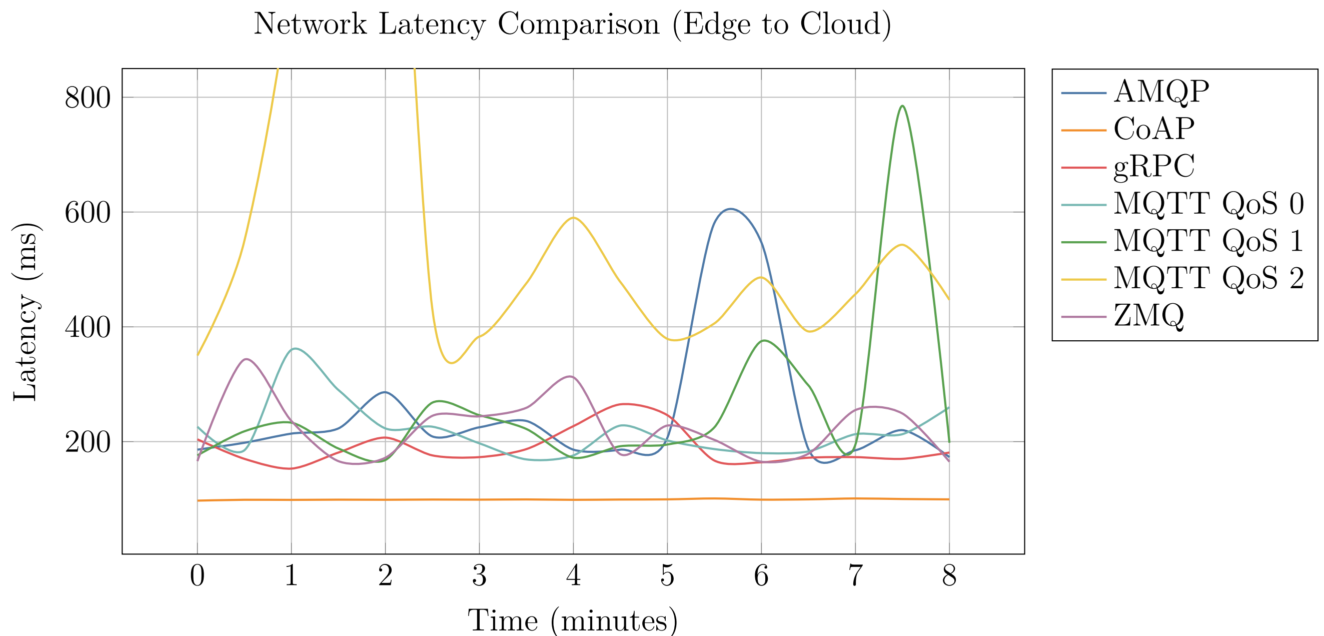

Figure 5.11: Network Latency Comparison (95th Percentile, Edge to Cloud) under Combined Latency (50ms±50ms) and 5% Packet Loss

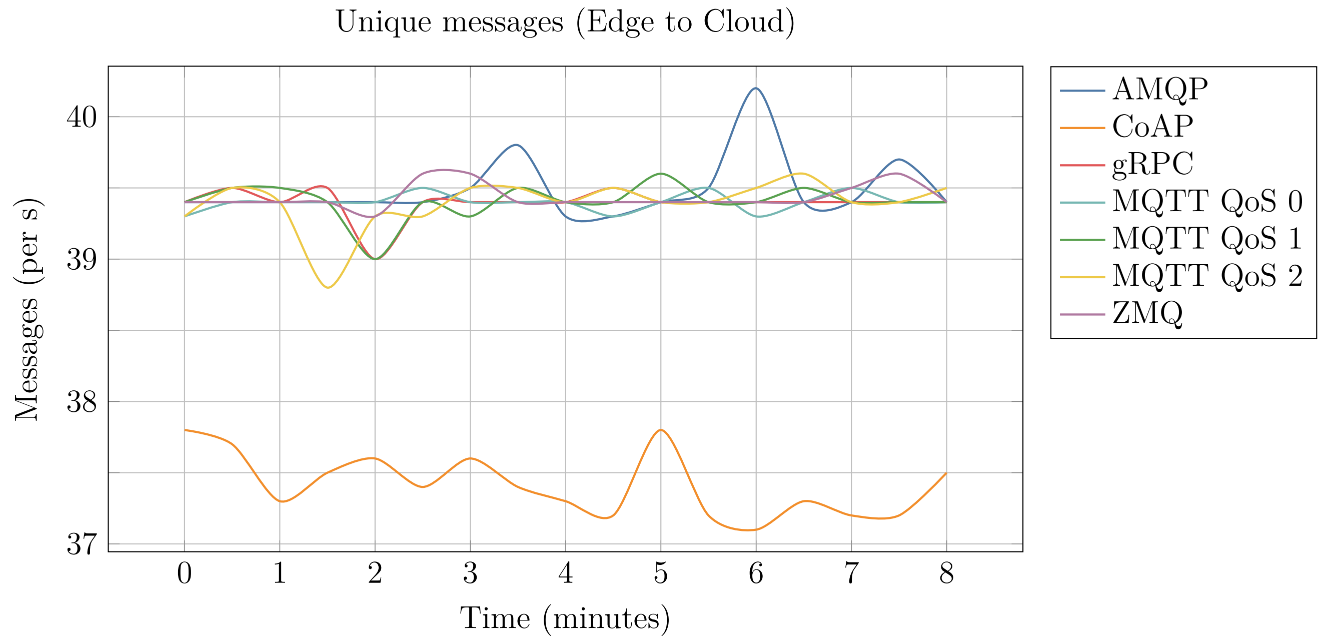

Figure 5.12: Unique Messages Received (per second, Edge to Cloud) under Combined Latency (50ms±50ms) and 5% Packet Loss

5.7.1 Observed Behavior under Mixed Impairments

The combined effect of these impairments provides further insight into the robustness and behavior of each protocol:

Message Throughput (Figure 5.12): The ability to maintain the target throughput of 40 messages per second varies significantly:

- **TCP-Based Protocols (MQTT - all QoS levels, ZeroMQ, AMQP, gRPC):** These protocols demonstrate resilience in terms of message delivery. Despite the 5% packet loss, they all manage to deliver close to the 40 messages/second offered load, as shown by their lines hovering near the 39.5-40 messages/sec mark. There are minor fluctuations, likely due to the interplay of jitter and packet retransmissions, but no sustained, significant drop in message delivery that would indicate an inability to cope with the 5% loss at this message rate. This again highlights TCP's capability to recover lost packets.

- **CoAP (Constrained Application Protocol):** CoAP (using NON messages over UDP) exhibits a clear and consistent reduction in throughput. It processes approximately 37-37.8 messages per second, which represents a loss of about 2.2-3 messages

per second from the offered 40 messages/sec. This loss rate (around 5.5-7.5%) is consistent with, and slightly higher than, the artificially introduced 5% network packet loss. Since CoAP NON messages do not have an inherent retransmission mechanism, packets lost at the network layer are permanently lost to the application.

End-to-End Latency (Figure 5.11): The latency profiles reflect the combined impact of the base 50ms delay, the ±50ms jitter, and the delays introduced by recovering from 5% packet loss:

- **CoAP:** CoAP maintains the lowest and most stable 95th percentile latency, consistently around 100 ms. This value is slightly above the maximum instantaneous delay introduced by the 50ms base + 50ms jitter (0-100ms range). For the messages that successfully traverse the network (around 93-94.5% of them), their transit is quick, unburdened by TCP retransmission delays or complex protocol handshakes.
- **MQTT QoS 0, ZeroMQ, AMQP, gRPC:** These TCP-based protocols exhibit higher and more variable latencies compared to CoAP. Their 95th percentile latencies generally fluctuate between 150 ms and 350 ms, with occasional spikes. This range reflects the base delay, jitter, and, critically, the additional time taken for TCP to detect packet loss (via timeouts or duplicate ACKs) and retransmit the missing data. The 5% loss rate means retransmissions are frequent enough to keep the latency elevated and variable. Among this group, ZMQ and MQTT QoS 0 often show slightly lower average latencies than AMQP and gRPC, but all are clearly impacted by the need to recover lost packets.
- **MQTT QoS 1:** MQTT QoS 1 displays a latency profile that is generally higher and more volatile than the previous group, often ranging between 200 ms and 400 ms, with larger peaks. The combination of TCP retransmissions for lost segments (which could carry either the PUBLISH or the PUBACK) and the MQTT-level acknowledgment adds to the overall delay and variability.
- **MQTT QoS 2:** MQTT QoS 2 is the most severely affected in terms of latency. Its 95th percentile latency is consistently the highest, typically fluctuating between 300 ms and well over 800 ms, with significant peaks. The multi-stage handshake of QoS 2 (PUBLISH, PUBREC, PUBREL, PUBCOMP) means that a 5% chance of packet loss at any of these four stages (or their underlying TCP segments) can stall the entire message exchange, requiring TCP retransmissions. This multiplicative effect

of potential loss points across its extended handshake amplifies the impact of packet loss and jitter on its end-to-end latency.

5.7.2 Summary of Performance with Combined Impairments

This test with combined latency, jitter, and 5% packet loss (Section 5.7) further clarifies the operational trade-offs:

- **Reliability vs. Loss Tolerance:** TCP-based protocols successfully deliver nearly all messages by retransmitting lost packets, ensuring data integrity at the cost of increased and more variable latency. CoAP (NON) sacrifices this guaranteed delivery, experiencing message loss proportional to network conditions.
- **Latency under Duress:** CoAP (NON) offers the lowest latency for delivered messages due to its lightweight UDP nature and avoidance of retransmission delays. However, this comes with the acceptance of data loss.
- **Impact of Protocol Complexity:** Protocols with more round-trips or more complex state management per message (notably MQTT QoS 2, and to a lesser extent MQTT QoS 1) experience a greater cumulative impact on latency when faced with combined impairments. Each lost packet in their extended handshakes causes significant delays. Simpler TCP-based exchanges (MQTT QoS 0, ZMQ) are still delayed by TCP retransmissions but have fewer application-level stages where loss can compound delays.
- The choice of protocol in such challenging network environments becomes a critical decision. If consistent, low latency for (potentially fewer) messages is paramount and some data loss is acceptable, CoAP (NON) is a strong candidate. If reliable delivery of all messages is required, a TCP-based protocol is necessary, but the application must be prepared to handle higher and more variable latencies. Among TCP options, those with simpler acknowledgment schemes (or no application-level acknowledgment beyond TCP like ZMQ or MQTT QoS 0) tend to offer better latency than those with more involved per-message handshakes when packet loss is a factor.

5.8 Discussion of Results

The empirical evaluations presented in this chapter, encompassing baseline performance under increasing message frequency (Section 5.3), varying message payload sizes (Section

5.5), and diverse network impairments (Sections 5.4, 5.6, and 5.7), provide a comprehensive understanding of the operational characteristics and trade-offs inherent in the selected IoT communication protocols. Several overarching themes and specific protocol behaviors emerge from the collective results.

5.8.1 Dominance of Transport Layer and Reliability Mechanisms

A primary observation across multiple tests is the profound impact of the underlying transport protocol (TCP vs. UDP) and the application-level reliability mechanisms (e.g., MQTT QoS levels) on performance.

TCP's Double-Edged Sword: TCP-based protocols (MQTT, ZeroMQ, AMQP, gRPC) consistently demonstrated reliable message delivery in the face of packet loss, as detailed in Section 5.4. This inherent reliability, achieved through acknowledgments and retransmissions, is a significant advantage for applications where data integrity is paramount. However, this came at the cost of increased latency, especially when packet loss was present or when static network latency was high (Section 5.6). The TCP retransmission delays became a dominant factor in these scenarios, often overshadowing finer application-protocol differences in the less reliable configurations (e.g., MQTT QoS 0, ZMQ).

UDP's Low-Latency, Best-Effort Nature: CoAP, when using Non-confirmable (NON) messages over UDP, consistently exhibited the lowest latency for successfully delivered messages, particularly under impaired network conditions (Sections 5.4 and 5.6). This is attributable to UDP's connectionless, *fire-and-forget* nature, which avoids the overhead of connection setup, acknowledgments, and retransmission queues. However, this came with the direct consequence of message loss proportional to the network's unreliability, as noted in Section 5.4 and 5.7. The tests also highlighted CoAP's limitations with larger payloads without block-wise transfer (Section 5.5), where its throughput collapsed.

Application-Level Reliability Costs: The experiments starkly illustrated the performance cost of application-level reliability. MQTT QoS 1 and, more dramatically, MQTT QoS 2, with their multi-round-trip handshakes per message (Section 5.2), consistently showed the highest latencies and lowest sustainable throughputs. This was evident under increasing message frequency (Section 5.3), increasing payload size (Section 5.5), high static network latency (Section 5.6), and combined impairments (Section 5.7). While they aim to guarantee delivery, this guarantee imposes significant overhead that makes them less suitable for high-volume or very time-sensitive applications over challenging networks

if individual message acknowledgment is strictly enforced as per the protocol.

5.8.2 Throughput, Scalability, and Bottlenecks

High-Throughput Contenders: ZeroMQ consistently demonstrated the highest raw throughput capacity in the baseline high-frequency test (Section 5.3), successfully handling over 3000 messages/second while maintaining low latency. Its lightweight, brokerless architecture (with a soft-broker model in this testbed) proved highly efficient. AMQP and MQTT QoS 0 also showed strong throughput capabilities, scaling well to 1700-2500 messages/second before encountering saturation in the same test.

Saturation Points: All protocols eventually hit a saturation point where increasing the offered load led to sharply increased latency, reduced message throughput, or both (Section 5.3). For TCP-based protocols, this often manifested as TCP backpressure, queuing, or internal processing bottlenecks within the broker or application services. For CoAP (NON), saturation was primarily observed as increased packet loss rather than just high latency for delivered messages.

Impact of Message Size: The test with increasing payload sizes (Section 5.5) revealed that while most protocols could handle a fixed rate of small messages, their efficiency with larger messages varied. ZeroMQ maintained its latency advantage. MQTT QoS 0, AMQP, and gRPC performed well up to a point, while MQTT QoS 1 and QoS 2 struggled significantly with latency as payload sizes increased due to the larger amount of data involved in their acknowledgment handshakes.

5.8.3 Resilience to Network Impairments

Packet Loss: The 10% packet loss test (Section 5.4) was particularly revealing. TCP-based protocols recovered messages but incurred latency penalties. CoAP (NON) lost messages but maintained low latency for those delivered. This highlights a critical design choice for IoT applications: prioritize delivery certainty or low latency for available data.

Static Latency: High static network latency (Section 5.6) disproportionately affected protocols with more round-trips per message (MQTT QoS 1 & 2). Simpler exchanges or one-way data flows (CoAP NON, MQTT QoS 0) were less impacted in terms of their ability to maintain message throughput, though all experienced increased absolute latency.

Combined Impairments: The test with combined latency, jitter, and packet loss (Sec-

tion 5.7) showed that MQTT QoS 2 was the most severely impacted in terms of latency, while CoAP (NON) provided the lowest latency for delivered messages but suffered message loss. Other TCP-based protocols fell in between, managing to deliver messages reliably but with noticeable latency increases and variability.

5.8.4 Protocol-Specific Nuances

- **ZeroMQ:** Consistently a top performer for low-latency, high-throughput scenarios across various tests. Its brokerless nature (or efficient soft-broker) and lean ZMTP make it highly efficient, but it places more responsibility on the application for aspects like message persistence or complex routing if needed beyond its inherent patterns.
- **MQTT:** The QoS mechanism provides clear trade-offs. QoS 0 is highly scalable for best-effort delivery up to a high threshold. QoS 1 and 2 offer reliability guarantees but at a steep performance cost, especially under adverse conditions or high load, making them suitable for lower-volume critical messages rather than high-frequency telemetry.
- **AMQP:** Demonstrated robust performance, particularly in handling moderate to high message rates (Section 5.3) and larger payloads (Section 5.5) with reasonable latency before saturation. Its feature-rich nature makes it suitable for enterprise-grade messaging where these features are valued.
- **CoAP:** Shines in low-latency, best-effort delivery over UDP, especially if message loss is tolerable (Sections 5.4, 5.6, 5.7). Its major limitation in this study was handling large payloads without block-wise transfer (Section 5.5). For constrained devices and networks where low overhead is key, and data can be resent or is idempotent, it remains a strong candidate.
- **gRPC:** Offers efficient serialization with Protocol Buffers and benefits from HTTP/2. It performed well under moderate loads and with smaller payloads. However, its performance degraded more noticeably than AMQP under very high message rates (Section 5.3) and its latency increased more than some others with extremely large payloads (Section 5.5). Its reliance on TCP means it shares the latency implications of that transport in impaired networks.

The Wireshark analysis (Section 5.2) provided a qualitative underpinning for these quantitative results, illustrating the differences in packet exchanges and protocol *chattines* that contribute to the observed performance characteristics.

5.9 Chapter Conclusion

This chapter presented an extensive empirical performance evaluation of MQTT (QoS 0, 1, 2), ZeroMQ, AMQP, CoAP (NON), and gRPC within a containerized cloud-edge IoT testbed. The experiments systematically investigated their behavior under varying message loads (frequency and payload size) and diverse network impairment conditions (packet loss, static latency, and combined impairments).

The key findings can be summarized as follows:

1. **No Single Best Protocol:** The results unequivocally demonstrate that no single protocol excels across all metrics and scenarios. The optimal choice is highly dependent on specific application requirements regarding reliability, latency, throughput, message size, and network conditions.

2. **Latency vs. Reliability Trade-off:** A fundamental trade-off exists between ensuring message delivery reliability and achieving low end-to-end latency. Protocols with strong, per-message reliability guarantees (e.g., MQTT QoS 1 and QoS 2) invariably incurred higher latencies, especially under stress or impairments. Conversely, protocols prioritizing speed and low overhead with best-effort delivery (e.g., CoAP NON, MQTT QoS 0, ZeroMQ relying on TCP's base reliability) offered lower latency but with potential for message loss (CoAP) or eventual saturation leading to drops (MQTT QoS 0, gRPC).

3. **Throughput Capabilities:** ZeroMQ consistently demonstrated the highest throughput capacity in high-frequency scenarios (Section 5.3), followed by AMQP and MQTT QoS 0. gRPC also showed good throughput but saturated earlier than AMQP in the same test. The reliable MQTT QoS levels had significantly lower throughput ceilings.

4. **Impact of Network Impairments:**

    - **Packet Loss (Section 5.4):** TCP-based protocols effectively recovered from packet loss, maintaining message delivery at the cost of increased latency. CoAP

(NON) suffered message loss proportional to the network loss but maintained low latency for delivered messages.

- **Static Latency (Section 5.6):** Increased network RTT disproportionately affected protocols with multiple round-trips per message (MQTT QoS 1 & 2), severely degrading their latency and throughput.

5. **Payload Size Considerations (Section 5.5):** For transmitting larger payloads at a fixed message rate, ZeroMQ maintained the best latency profile. MQTT QoS 1 and QoS 2 struggled significantly with latency as payload sizes grew. CoAP (without block-wise transfer) was found unsuitable for payloads beyond a few kilobytes.

6. **Architectural Implications:** Brokerless or efficiently brokered protocols (like ZeroMQ) generally offered lower latency and higher raw throughput compared to more feature-rich brokered systems (like AMQP or MQTT with a central broker) when pushed to their limits, though the latter provide valuable abstractions like persistent queues and flexible routing.

These findings provide empirical data to guide architects and developers in selecting appropriate IoT communication protocols. The choice involves carefully weighing the application's tolerance for data loss against its sensitivity to delay, the expected message volume and size, and the anticipated reliability of the underlying network infrastructure. Future work could extend this analysis to include even larger-scale deployments with a higher number of concurrent sensors, more complex network topologies, the impact of security and encryption overheads for each protocol, and a deeper investigation into broker clustering and scaling strategies for MQTT and AMQP.

6 Analysis and Discussion

The empirical performance evaluation presented in Chapter 5 yielded a rich dataset on the behavior of MQTT (QoS 0, 1, 2), ZeroMQ, AMQP, CoAP (NON), and gRPC under diverse operational conditions. This chapter provides a comprehensive analysis and discussion of these findings, interpreting the quantitative results in the context of the protocols' architectural characteristics, the research questions outlined in Chapter 1, and the existing body of knowledge detailed in Chapter 2. The discussion will focus on the emergent trade-offs, performance envelopes, and practical implications for selecting and deploying these protocols in cloud-edge IoT environments, particularly within containerized orchestrations like Kubernetes.

6.1 Revisiting Research Questions

The study was guided by three key research questions (Chapter 1). This section addresses each question based on the experimental evidence.

6.1.1 RQ1: End-to-End Latencies under Baseline and High-Load Conditions

Research Question 1: What are the average and 95th percentile end-to-end latencies of MQTT (with QoS levels 0, 1, and 2), ZeroMQ, AMQP (RabbitMQ), CoAP, and gRPC under baseline and high-load conditions in a containerized cloud-edge IoT environment?

The baseline performance tests involving increasing message frequency (Section 5.3, Figure 5.1) provided clear insights into latency characteristics. Under lower load conditions (initial phases of the baseline test), ZeroMQ, MQTT QoS 0, AMQP, and gRPC exhibited the lowest 95th percentile latencies, often sub-50ms. CoAP (NON), while also capable of low latency, showed more initial variability. MQTT QoS 1 and particularly QoS 2 started with inherently higher latencies due to their acknowledgment handshakes.

As the message load (frequency) increased:

- **ZeroMQ** maintained its latency advantage most effectively, showing only a modest

increase even at throughputs exceeding 3000 messages/second. This aligns with its lightweight, brokerless design (or efficient soft-broker in this testbed) discussed in Section 5.2.2 and 5.8.4.

- **MQTT QoS 0**, **AMQP**, and **gRPC** demonstrated good latency performance up to their respective saturation points (approx. 2500, 1800, and 1000 messages/second, respectively). Beyond these thresholds, their latencies increased sharply. This behavior reflects the eventual overwhelming of their TCP connections, broker processing (for MQTT and AMQP), or application-level request handling (gRPC).
- **MQTT QoS 1 and QoS 2** showed rapid and extreme latency degradation as message frequency increased. Their per-message reliability handshakes (two-way and four-way, respectively, see Section 5.2.1) created significant bottlenecks, leading to latencies escalating into multiple seconds. This confirms the trade-off discussed in (Moraes et al., 2019) where higher QoS incurs latency penalties.
- **CoAP (NON)** exhibited erratic latency under high message frequency in the baseline test, coupled with significant message loss. While individual successfully delivered packets might be fast, the overall 95th percentile was skewed by delays and an inconsistent delivery rate.

The containerized cloud-edge environment, managed by Kubernetes, provided a consistent platform, but the inherent protocol behaviors were the primary differentiators for latency under load.

6.1.2 RQ2: Message Throughput Variation and Scalability Bottlenecks

Research Question 2: How does message throughput vary for each protocol as the number of simulated sensors and message frequency increase, and what are the primary bottlenecks limiting scalability?

While this study primarily focused on increasing message frequency from a single source cluster (simulating increasing data rate rather than an increasing number of distinct, low-rate sensors in dedicated tests), the baseline test with escalating message frequency (Section 5.3, Figure 5.2) provided significant insights into throughput limits.

- **Highest Throughput:** ZeroMQ achieved the highest sustained throughput, followed by AMQP and MQTT QoS 0. This aligns with expectations for protocols with lower per-message overhead (ZeroMQ, MQTT QoS 0) or efficient brokering and flow control (AMQP).
- **Bottlenecks for TCP-based protocols:** For MQTT (all QoS levels), AMQP, and gRPC, the eventual bottleneck appeared to be a combination of broker processing capacity (for MQTT and AMQP using Mosquitto and RabbitMQ respectively), TCP connection management under high load, and application-level service processing limitations within the Kubernetes pods. As message rates peaked, received throughput either plateaued and then declined (MQTT QoS 0, AMQP, gRPC), suggesting the system was dropping messages or unable to process them quickly enough, or (for MQTT QoS 1 and 2) maintained a lower, stable received rate while latency skyrocketed, indicating severe backpressure and queuing.
- **Bottlenecks for CoAP (NON):** CoAP's throughput was limited by its UDP nature and lack of transport-level reliability. At high frequencies, packet loss became the dominant bottleneck, as seen by its erratic and lower overall throughput. Its inability to handle larger payloads (Section 5.5) without block-wise transfer was another critical bottleneck for data volume.
- **MQTT QoS Limitations:** The per-message acknowledgment handshakes of MQTT QoS 1 and QoS 2 were clearly the primary bottlenecks limiting their scalability in terms of message frequency. They could not sustain high throughput due to the time taken for these reliable exchanges.

The primary bottlenecks limiting scalability in these tests were thus related to the processing overhead of reliability mechanisms, broker capacity under high load, TCP/IP stack performance, and the inherent limitations of UDP for reliable high-volume transport without robust application-level controls.

6.1.3 RQ3: Impact of Network Impairments on Reliability and Latency

Research Question 3: What is the impact of network impairments (added latency, packet loss, and jitter) on the message delivery reliability and end-to-end latency of each protocol?

The network impairment tests (Sections 5.4, 5.6, and 5.7) provided clear answers:

- **Impact of Packet Loss (Section 5.4):**
  - **TCP-based protocols (MQTT, ZMQ, AMQP, gRPC)** maintained message delivery reliability due to TCP's retransmission mechanisms. However, this resulted in significantly increased end-to-end latency. MQTT QoS 2 was most affected latency-wise due to its multi-stage handshake.
  - **CoAP (NON)** experienced message loss roughly proportional to the network packet loss but maintained very low latency for successfully delivered messages. This starkly contrasts with TCP-based approaches, as also noted by studies like (Silva et al., 2021) and (Betzler et al., 2016) which highlight CoAP's resilience in lossy networks for certain use cases.
- **Impact of Added Static Latency (Section 5.6):**
  - All protocols saw increased end-to-end latency.
  - **CoAP (NON)** latency increased almost linearly with the added network latency.
  - **MQTT QoS 1 and QoS 2** were extremely sensitive, with their latencies and throughput degrading severely due to the multiplicative effect of RTT on their multi-round-trip acknowledgments. This aligns with the general understanding that chatty protocols suffer more on high-latency links.
  - Other TCP-based protocols **(MQTT QoS 0, ZMQ, AMQP, gRPC)** showed increased latency greater than just the added static RTT, reflecting TCP and application overheads, and eventually showed throughput degradation at very high static latencies.
- **Impact of Combined Impairments (Latency, Jitter, Packet Loss - Section 5.7):**
  - Trends were generally an amalgamation of individual impairment effects. TCP-based protocols maintained reliability but with higher, more variable latency. CoAP (NON) lost messages but kept latency low for successful ones. MQTT QoS 2 was most severely impacted in latency.
  - Jitter (50ms ±50ms) contributed to the variability in latency for all protocols, but its impact was more pronounced when combined with packet loss, as it could affect TCP's RTT estimation and retransmission timers.

These tests confirm that network conditions fundamentally shape protocol performance, and the choice of protocol must consider the expected operating environment. The findings echo sentiments from (Çorak et al., 2018) regarding the importance of protocol selection in resource-constrained and unreliable environments.

6.2 Key Performance Trade-offs and Protocol Suitability

The experimental results highlight several critical trade-offs inherent in IoT communication protocols, influencing their suitability for different application requirements within a cloud-edge architecture.

6.2.1 Reliability vs. Latency and Throughput

This is arguably the most dominant trade-off observed.

- **High Reliability Focus (MQTT QoS 1 & 2):** These configurations provide strong guarantees of message delivery (at-least-once and exactly-once). However, this comes at a significant cost to maximum throughput and average/p95 latency, especially under high load or impaired network conditions (high static RTT, packet loss). The per-message acknowledgment overhead becomes prohibitive for high-frequency data streams. They are best suited for low-volume, critical messages where loss is unacceptable and higher latency can be tolerated.
- **Best-Effort with Low Overhead (MQTT QoS 0, ZeroMQ, CoAP NON):** These options prioritize speed and low resource consumption.
  - **MQTT QoS 0 and ZeroMQ** (over TCP) offer a good balance, providing TCP's inherent transport reliability against typical network issues (like transient loss) but without application-level per-message acknowledgments, leading to high throughput and low latency up to their saturation points. They are suitable for high-volume telemetry where occasional loss due to extreme broker/network overload might be acceptable if not using further application-level checks.

  - **CoAP (NON)** over UDP offers the lowest latency potential for delivered messages, especially in impaired networks, but accepts message loss. This is ideal for idempotent sensor data or applications where the latest value is more important than every single value, and where network conditions might be poor but low delay for received data is critical.

- **Balanced Approaches (AMQP, gRPC):** These protocols offer robust, reliable messaging (AMQP with its queuing and acknowledgments, gRPC over TCP with HTTP/2's reliability) and performed well in many scenarios. AMQP excelled in sustained high throughput before saturation. gRPC offers efficient serialization and RPC semantics. Their performance under impairments was generally bounded by TCP's behavior. They represent good general-purpose choices for many IoT applications that require reliable messaging with moderate to high throughput and can tolerate TCP-induced latency variations.

6.2.2 Impact of Payload Size

As demonstrated in Section 5.5, not all protocols handle large messages with equal grace, even at low message rates.

- **Efficient with Large Payloads:** ZeroMQ showed excellent latency characteristics even with large payloads. AMQP and MQTT QoS 0 also managed large payloads relatively well up to a point.

- **Struggling with Large Payloads:** MQTT QoS 1 and QoS 2 saw their latencies explode due to the large amount of data involved in their multi-stage acknowledgments. CoAP (without block-wise transfer) was entirely unsuitable for payloads beyond a few kilobytes. gRPC's latency also increased notably with very large payloads.

This implies that for applications transmitting large data objects (e.g., images, firmware updates, aggregated data packets), protocols with lean handling of bulk data are preferred.

6.2.3 Broker-based vs. Brokerless Architectures

While not a direct A/B comparison of the same protocol in brokered vs. brokerless mode (except for ZeroMQ's soft-broker vs. direct potential), inferences can be drawn:

- **Brokerless Efficiency (ZeroMQ):** ZeroMQ's performance suggests that direct or efficiently intermediated (soft-broker) communication can yield very high throughput and low latency by minimizing hops and centralized processing overhead.

- **Brokered Abstraction (MQTT, AMQP):** Brokers like Mosquitto (for MQTT) and RabbitMQ (for AMQP) provide valuable services like message persistence, fan-out, topic-based routing, and decoupling of producers and consumers. However, the broker itself can become a bottleneck under extreme load, as seen with MQTT QoS 0 saturation, or contribute to overall latency. The choice and configuration of the broker are critical.

- **RPC Model (gRPC):** gRPC's client-server RPC model is conceptually similar to direct communication but structured around service definitions. Its performance is tied to HTTP/2 and TCP characteristics.

The choice often depends on the need for centralized message management and decoupling versus raw point-to-point or pattern-based performance.

6.2.4 Suitability for Containerized Cloud-Edge Deployments

All tested protocols were successfully deployed in a containerized cloud-edge environment using Docker and Kubernetes. The performance observed is thus representative of such modern deployment paradigms.

- The resource footprint of the protocol services and brokers within Kubernetes pods becomes a factor for cost and density, although this was not a primary metric quantitatively analyzed in the results chapter.

- Kubernetes' networking and service discovery mechanisms generally abstracted the underlying infrastructure well. However, factors like LoadBalancer performance, inter-pod communication latency, and node resource contention could indirectly influence results at scale, though efforts were made to isolate tests.

- The outbound-only connection constraint from the edge, mimicking firewalled environments, was handled by all protocols, as clients initiated connections to cloud services/brokers.

The study confirms the viability of these protocols in such architectures, with performance largely dictated by their intrinsic characteristics rather than specific limitations imposed by containerization itself at this scale.

6.3 Alignment with Existing Research

The findings of this thesis both align with and extend existing research in IoT communication protocols. Many studies, such as those by (Moraes et al., 2019; Çorak et al., 2018; Silva et al., 2021), have compared subsets of these protocols, often highlighting similar trade-offs between MQTT's QoS levels, or MQTT vs. CoAP. For instance, the observation that higher MQTT QoS levels increase latency and reduce throughput is well-established. Similarly, CoAP's suitability for lossy environments (when loss is tolerable) due to its UDP base is a common theme.

This research contributes in several specific ways:

1. **Comprehensive Five-Protocol Comparison:** It evaluates a broader set of five widely used protocols (MQTT, ZeroMQ, AMQP, CoAP, gRPC) under a unified testbed and methodology, including different architectural paradigms (pub-sub, message queuing, RPC, brokerless).

2. **Cloud-Edge Containerized Context:** The evaluation is specifically situated within a realistic, containerized cloud-edge testbed built with Docker and Kubernetes, reflecting modern deployment practices. Many existing studies use simpler setups or simulation environments. This provides insights relevant to deploying these protocols in orchestrated microservice environments at the edge and in the cloud.

3. **Diverse Impairment Scenarios:** The systematic testing under various load conditions (message frequency, payload size) and a range of network impairments (packet loss, static latency, combined effects) offers a multi-faceted view of protocol resilience and behavior.

4. **Focus on Outbound-Only Edge Connections:** The experimental design explicitly simulated edge devices behind firewalls/NAT, a common real-world constraint, ensuring all tested protocols could operate under this condition.

While prior work like (Bender et al., 2021) has focused on MQTT broker performance, or (Pamadi et al., 2020) on ZeroMQ's strengths, this thesis provides a comparative benchmark across a wider spectrum, within a contemporary deployment architecture. The observed high performance of ZeroMQ aligns with findings emphasizing its low-latency capabilities. The challenges faced by CoAP with large messages without block-wise transfer also reinforce known limitations if not implemented correctly for such use cases. The performance ceilings observed for gRPC and AMQP under extreme load offer practical data points for system dimensioning.

The results, therefore, offer valuable, empirically-backed guidance for practitioners making protocol choices for IoT systems that are increasingly being built on cloud-native principles.

7 Conclusion and Future Work

This thesis embarked on a systematic performance evaluation of five prominent Internet of Things (IoT) communication protocols—MQTT (QoS 0, 1, 2), ZeroMQ, AMQP, CoAP (NON), and gRPC—within a realistic, containerized cloud-edge testbed. The primary motivation was to address the critical challenge of selecting appropriate communication protocols for modern IoT deployments, which directly impacts system performance, reliability, and scalability, particularly in resource-constrained or challenging network environments.

7.1 Summary of Key Findings

The research was guided by specific questions concerning latency, throughput, and resilience to network impairments. The experimental results, detailed in Chapter 5 and discussed in Chapter 6, yielded several key conclusions:

1. **No Universal Protocol:** The core finding is that no single protocol universally outperforms others across all tested scenarios and metrics. Protocol selection must be a nuanced decision, carefully aligned with specific application requirements and anticipated operational conditions.

2. **Performance Profiles Under Load:** Under high message frequency, ZeroMQ demonstrated superior raw throughput and low latency. AMQP and MQTT QoS 0 also exhibited strong throughput capabilities up to their respective saturation points. In contrast, MQTT QoS 1 and QoS 2, while offering stronger reliability guarantees, showed significantly lower throughput ceilings and rapidly escalating latencies due to their per-message acknowledgment overhead. CoAP (NON) displayed erratic throughput and latency under high frequency due to its UDP nature and message loss. gRPC performed well at moderate loads but saturated earlier than AMQP.

3. **Resilience to Network Impairments:** TCP-based protocols (MQTT, ZeroMQ, AMQP, gRPC) effectively handled packet loss by retransmitting data, thereby ensuring message reliability at the cost of increased latency. CoAP (NON), operating over UDP, suffered message loss proportional to network conditions but maintained exceptionally low latency for successfully delivered messages. Protocols with more

complex, multi-stage handshakes (notably MQTT QoS 2) were most severely impacted by increased static network latency and combined impairments.

4. **Impact of Message Payload Size:** The ability to efficiently handle large payloads varied. ZeroMQ maintained low latency across a wide range of sizes. MQTT QoS 1 and QoS 2 struggled significantly with latency as payloads grew. CoAP (NON), without block-wise transfer implementation in this study, was unsuitable for payloads beyond a few kilobytes.

5. **Architectural Considerations:** The study implicitly highlighted differences between brokerless (or efficiently soft-brokered like ZeroMQ) and traditional brokered architectures (MQTT, AMQP). While brokers offer valuable decoupling and message management features, they can also introduce bottlenecks or additional latency compared to more direct communication patterns under certain high-load conditions.

6. **Cloud-Edge Containerized Environment Suitability:** All evaluated protocols were successfully deployed and benchmarked within a modern Docker and Kubernetes-based cloud-edge architecture, confirming their applicability in such environments. The observed performance was primarily driven by intrinsic protocol characteristics.

The qualitative Wireshark analysis (Section 5.2) further corroborated these findings by illustrating the underlying packet exchange differences that contribute to the observed performance trade-offs.

7.2 Answering Research Questions and Contributions

This thesis successfully addressed its stated research questions (Section 6.1) by providing empirical data on latency, throughput, and impairment resilience for the selected protocols in a defined cloud-edge context. The primary contributions of this research include:

- The development and documentation of a standardized, containerized cloud-edge testbed for IoT communication protocol evaluation, leveraging Docker and Kubernetes (Chapters 3 and 4).
- A comprehensive, quantitative performance comparison of five key IoT protocols, including multiple MQTT QoS levels, across a range of realistic load and network impairment scenarios.

- Empirical insights into the performance trade-offs associated with different protocol architectures (publish-subscribe, message queuing, RPC) and transport mechanisms (TCP, UDP) within a modern deployment paradigm.

- Data-driven guidance to aid practitioners and researchers in selecting appropriate IoT communication protocols based on specific application requirements and deployment constraints, particularly for cloud-edge systems.

7.3 Limitations of the Study

While this study provides valuable insights, certain limitations should be acknowledged:

- **Single Broker Implementation/Configuration:** For MQTT and AMQP, specific broker implementations (Mosquitto and RabbitMQ) with particular configurations were used. Performance can vary with different brokers or more optimized configurations (e.g., clustering, persistent storage tuning).

- **CoAP Block-Wise Transfer:** The CoAP tests used NON messages and did not implement block-wise transfer, limiting its applicability for larger payloads. A study including CON messages and block-wise transfer would provide a more complete picture of CoAP's capabilities.

- **Security Overheads:** The tests were conducted without transport-level security (TLS/DTLS) enabled on the data path to focus on core protocol performance. The overhead of encryption and secure handshakes would introduce additional latency and processing load, potentially altering relative performance.

- **Network Topology and Scale:** The testbed used a relatively simple point-to-point logical topology between the edge and a single cloud instance. Performance in more complex, multi-hop, or geographically distributed networks, or at a significantly larger scale of devices, might differ.

- **Specific Workload Type:** The workload was primarily synthetic telemetry data. Different data patterns or request-response interactions might favor different protocols.

7.4 Future Work

Building upon the findings and limitations of this research, several avenues for future work emerge:

- **Security Performance Analysis:** Conduct a comparative evaluation that explicitly includes the overhead of TLS/DTLS for TCP-based protocols and CoAP, respectively, and potentially application-level security mechanisms.
- **Scalability with Device Count:** Extend the tests to evaluate performance with a significantly larger number of concurrent simulated edge devices to explore broker scalability limits and the impact of connection management at scale.
- **Broker Optimization and Clustering:** Investigate the performance of MQTT and AMQP with different broker implementations, clustered configurations, and optimized persistence settings.
- **CoAP Enhancements:** Evaluate CoAP with Confirmable (CON) messages and a robust implementation of block-wise transfer to assess its performance with larger payloads and for reliable delivery scenarios.
- **Heterogeneous Network Conditions:** Explore performance over real-world wireless links (e.g., LoRaWAN, NB-IoT, 5G slices) which exhibit more complex and dynamic impairment profiles than the simulated conditions.
- **Resource Utilization on Edge Devices:** Quantify the CPU, memory, and energy consumption of protocol client libraries on resource-constrained edge hardware, which was not the focus of this cloud-centric resource analysis.
- **Hybrid Protocol Approaches:** Investigate hybrid architectures where different protocols are used for different segments of the data path (e.g., CoAP from device to edge gateway, then AMQP or gRPC from gateway to cloud).

7.5 Concluding Remarks

The selection of an appropriate communication protocol is a foundational decision in the design of efficient, reliable, and scalable IoT systems. This thesis has provided a

rigorous, empirical comparison of five leading protocols in a contemporary cloud-edge context. The results underscore the critical importance of understanding the inherent trade-offs of each protocol and aligning that understanding with the specific demands of the intended application and the realities of its operational environment. By offering data-driven insights into these trade-offs, this work aims to empower engineers and researchers to make more informed decisions, ultimately contributing to the development of more robust and performant IoT solutions.

Bibliography

Althoubi, A., Alshahrani, R., and Peyravi, H. (2021). “Delay analysis in IoT sensor networks”. In: *Sensors* 21.11, p. 3876.

Andriulo, F. C., Fiore, M., Mongiello, M., Traversa, E., and Zizzo, V. (2024). “Edge computing and cloud computing for internet of things: A review”. In: *Informatics*. Vol. 11. 4. MDPI, p. 71.

Arora, S., Bhardwaj, A., Kukkar, A., Kaur, S., et al. (2024). “A Comparative Analysis of Communication Efficiency: REST vs. gRPC in Microservice-Based Ecosystems”. In: *2024 International Conference on Emerging Innovations and Advanced Computing (INNOCOMP)*. IEEE, pp. 621–626.

Bender, M., Kirdan, E., Pahl, M.-O., and Carle, G. (2021). “Open-source mqtt evaluation”. In: *2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC)*. IEEE, pp. 1–4.

Betzler, A., Gomez, C., Demirkol, I., and Paradells, J. (2016). “CoAP congestion control for the internet of things”. In: *IEEE Communications Magazine* 54.7, pp. 154–160.

Böhm, S. and Wirtz, G. (2022). “Cloud-edge orchestration for smart cities: A review of kubernetes-based orchestration architectures”. In: *EAI Endorsed Trans. Smart Cities* 6.18, e2.

Bolanowski, M., Żak, K., Paszkiewicz, A., Ganzha, M., Paprzycki, M., Sowiński, P., Lacalle, I., and Palau, C. E. (2022). “Eficiency of REST and gRPC realizing communication tasks in microservice-based ecosystems”. In: *New trends in intelligent software methodologies, tools and techniques*. IOS Press, pp. 97–108.

Bormann, C. and Shelby, Z. (2016). *Block-wise transfers in the constrained application protocol (CoAP)*. Tech. rep.

Chan, Y.-W., Fathoni, H., Yen, H.-Y., and Yang, C.-T. (2022). “Implementation of a cluster-based heterogeneous edge computing system for resource monitoring and performance evaluation”. In: *Ieee Access* 10, pp. 38458–38471.

Čilić, I., Krivić, P., Podnar Žarko, I., and Kušek, M. (2023). “Performance evaluation of container orchestration tools in edge computing environments”. In: *Sensors* 23.8, p. 4008.

Çorak, B. H., Okay, F. Y., Güzel, M., Murt, Ş., and Ozdemir, S. (2018). "Comparative analysis of IoT communication protocols". In: *2018 International symposium on networks, computers and communications (ISNCC)*. IEEE, pp. 1–6.

Duan, Q., Wang, S., and Ansari, N. (2020). "Convergence of networking and cloud/edge computing: Status, challenges, and opportunities". In: *IEEE Network* 34.6, pp. 148–155.

Ferrari, P., Rinaldi, S., Sisinni, E., Colombo, F., Ghelfi, F., Maffei, D., and Malara, M. (2019). "Performance evaluation of full-cloud and edge-cloud architectures for Industrial IoT anomaly detection based on deep learning". In: *2019 II Workshop on Metrology for Industry 4.0 and IoT (MetroInd4. 0&IoT)*. IEEE, pp. 420–425.

Gheorghe-Pop, I.-D., Kaiser, A., Rennoch, A., and Hackel, S. (2020). "A performance benchmarking methodology for MQTT broker implementations". In: *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, pp. 506–513.

Happ, D., Karowski, N., Menzel, T., Handziski, V., and Wolisz, A. (2017). "Meeting IoT platform requirements with open pub/sub solutions". In: *Annals of Telecommunications* 72, pp. 41–52.

Hintjens, P. et al. (2011). "Ømq-the guide". In: *Online: http://zguide.zeromq.org* 23.

Kampars, J., Tropins, D., and Matisons, R. (2021). "A review of application layer communication protocols for the iot edge cloud continuum". In: *2021 62nd International Scientific Conference on Information Technology and Management Science of Riga Technical University (ITMS)*. IEEE, pp. 1–6.

Kang, Z. and Dubey, A. (2020). "Evaluating DDS, MQTT, and ZeroMQ Under Different IoT Traffic Conditions". In: *M4IoT'20: Proceedings of the International Workshop on Middleware and Applications for the Internet of Things*, pp. 7–12.

Lauener, J., Sliwinski, W., and CERN, G. (2017). "How to design & implement a modern communication middleware based on ZeroMQ". In: *Proc of ICALEPCS*. Vol. 17, pp. 45–51.

Liri, E., Singh, P. K., Rabiah, A. B., Kar, K., Makhijani, K., and Ramakrishnan, K. (2018). "Robustness of iot application protocols to network impairments". In: *2018 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. IEEE, pp. 97–103.

Luzuriaga, J. E., Perez, M., Boronat, P., Cano, J. C., Calafate, C., and Manzoni, P. (2015). "A comparative evaluation of AMQP and MQTT protocols over unstable and mobile networks". In: *2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC)*. IEEE, pp. 931–936.

Martí, M., Garcia-Rubio, C., and Campo, C. (2019). "Performance Evaluation of CoAP and MQTT_SN in an IoT Environment." In: *UCAmI*, p. 49.

Milošević, M., Mladenovic, V., and Pešović, U. (2021). "Evaluation of HTTP/3 protocol for internet of things and fog computing scenarios". In.

Mishra, B. (2018). "Performance evaluation of MQTT broker servers". In: *International Conference on Computational Science and Its Applications*. Springer, pp. 599–609.

Morabito, R., Farris, I., Iera, A., and Taleb, T. (2017). "Evaluating performance of containerized IoT services for clustered devices at the network edge". In: *IEEE Internet of Things Journal* 4.4, pp. 1019–1030.

Moraes, T., Nogueira, B., Lira, V., and Tavares, E. (2019). "Performance comparison of IoT communication protocols". In: *2019 IEEE International Conference on Systems, Man and Cybernetics (SMC)*. IEEE, pp. 3249–3254.

Naik, N. (2017). "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP". In: *2017 IEEE international systems engineering symposium (ISSE)*. IEEE, pp. 1–7.

Pamadi, V. N., Chaurasia, D. A. K., and Singh, D. T. (2020). "Comparative Analysis OF GRPC VS. ZeroMQ for Fast Communication". In: *International Journal of Emerging Technologies and Innovative Research (www. jetir. org)* 7.2, pp. 937–951.

Prajapati, A. (2021). "AMQP and beyond". In: *2021 International Conference on Smart Applications, Communications and Networking (SmartNets)*. IEEE, pp. 1–6.

Sebrechts, M., Borny, S., Wauters, T., Volckaert, B., and De Turck, F. (2021). "Service relationship orchestration: Lessons learned from running large scale smart city platforms on kubernetes". In: *IEEE Access* 9, pp. 133387–133401.

Shelby, Z., Hartke, K., and Bormann, C. (2014). *The constrained application protocol (CoAP)*. Tech. rep.

Silva, D., Carvalho, L. I., Soares, J., and Sofia, R. C. (2021). "A performance analysis of internet of things networking protocols: Evaluating MQTT, CoAP, OPC UA". In: *Applied Sciences* 11.11, p. 4879.

Standard, O. (2019). "MQTT Version 5.0". In: *Retrieved June* 22.2020, p. 1435.

Thangavel, D., Ma, X., Valera, A., Tan, H.-X., and Tan, C. K.-Y. (2014). "Performance evaluation of MQTT and CoAP via a common middleware". In: *2014 IEEE ninth international conference on intelligent sensors, sensor networks and information processing (ISSNIP)*. IEEE, pp. 1–6.

Uy, N. Q. and Nam, V. H. (2019). “A comparison of AMQP and MQTT protocols for Internet of Things”. In: *2019 6th NAFOSTED Conference on Information and Computer Science (NICS)*. IEEE, pp. 292–297.