POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Enabling Location-aware Operation in Decentralized IoT Communications

Tesi di Laurea Magistrale in
Computer Science and Engineering -
Ingegneria Informatica

Author: **Matteo Visotto**

Student ID: 976477
Advisor: Prof. Luca Mottola
Academic Year: 2022-23

Abstract

At the heart of an IoT distributed infrastructure lie communication protocols. Literature presents various communication models and message patterns tailored to different use cases. Among these, the Publish/Subscribe (Pub/Sub) message model stands out as widely utilized, with several publicly available protocols. Zenoh, a recently emerged protocol, adopts a Pub/Sub topic-based paradigm unifying also the Request/Response model. It employs a fully decentralized architecture, ensuring fault tolerance and eliminating the necessity for a central broker. It also supports geo-distributed storage and query capabilities, harmonizing data in motion, in use, and at rest, but it currently lacks location-awareness capabilities. Zenoh is not able to perform subscriptions based on both the data and the location where these data originate.

To address the challenge of associating sensed data with their respective sensing locations for both stationary and moving sources, ensuring independence between the two aspects, we design and implement a location-aware variant of Zenoh. This involves introducing a unique key within the topic structure for conveying geographic encoded data. We introduce a location matching component decoupling the matching process and managing location data externally. This lets us just inject location-based matching results in Zenoh's core and we do not need to make changes in the protocol logic nor in the routing process, guaranteeing performance and functionality of the original version. This solution allows us to pilot message routing managing complex geographical shapes and coordinates, defining location matching rules independently of the rest of the topic. Our approach enables the implementation of three encoding techniques, each of them with a distinct tradeoff between performance and expressiveness in describing geographical data. Our location-aware version of Zenoh demonstrates to create a logical partition of the network according to geographical zones, filtering unnecessary messages, enhancing performance, and reducing latency by more than 50%. In addition, we provide an encoding technique that performs 40% better than the others, making it suitable for congested, slow, and resource-constrained networks.

Keywords: Zenoh, IoT, Protocol, Topic, Publish/Subscribe, Location, Location-awareness

Abstract in lingua italiana

Al centro di un'infrastruttura IoT si trovano i protocolli di comunicazione. Tra i vari proposti in letteratura, il modello Publish/Subscribe (Pub/Sub) è il più diffuso, con diverse implementazioni documentate. Zenoh, un protocollo recentemente emerso, adotta il paradigma Pub/Sub implementando anche il modello Request/Response, supportando architetture decentralizzate, assicurando la tolleranza agli errori e eliminando la necessità di un broker o server. Inoltre, offre il supporto a storage distribuiti e la capacità di eseguire query, integrando la produzione, la memorizzazione, l'utilizzo e la richiesta dei dati, ma attualmente non implementa meccanismi location-aware, ovvero non è in grado di effettuare sottoscrizioni basate sia sui dati che sulla posizione di origine degli stessi.
Per affrontare il problema di associare i dati rilevati da sensori alle rispettive posizioni di rilevamento sia per le fonti stazionarie che in movimento, garantendo l'indipendenza tra i due aspetti, abbiamo progettato e implementato una variante location-aware di Zenoh. Abbiamo introdotto una chiave all'interno del topic contenente i dati geografici codificati. Abbiamo introdotto un componente di match per i dati geografici che disaccoppia il processo di matching gestendoli esternamente. Ciò ci consente di iniettare solo i risultati del match geografico nel core di Zenoh senza apportare modifiche alla logica del protocollo né al processo di routing, garantendo le prestazioni e le funzionalità della versione originale. Questa soluzione ci permette di manipolare il routing dei messaggi gestendo coordinate geografiche e forme geografiche complesse, definendo regole di matching della posizione indipendentemente dal resto del topic. Il nostro approccio ci ha consentito l'implementazione di tre tecniche di codifica, ciascuna un compromesso tra prestazioni ed espressività nella descrizione dei dati geografici.
La soluzione è efficace nel suddividere logicamente la rete in base alle zone geografiche, filtrando i messaggi, migliorando le prestazioni e riducendo la latenza di oltre il 50%. Inoltre, forniamo una tecnica di codifica che, rispetto alle altre, dimostra prestazioni superiori del 40%, rendendola adatta per reti congestionate, lente e con risorse limitate.

Parole chiave: Zenoh, IoT, Protocollo, Topic, Publish/Subscribe, Posizione, Location-awareness

Acknowledgements

I want to thank my advisor, Prof. Luca Mottola, for his advice and the support he gave me throughout my research. Working with him was inspiring, and I am grateful for the opportunity to work under his supervision.

A special thanks to my family, especially my parents, for supporting and motivating me during my studies, for believing in me even when I didn't believe in myself. Thanks for making this possible. I also want to thank my grandparents, who have been my biggest supporters.

Special thanks also to Sveva, the person who made my life happier, who supported me and above all who always put up with the bad moments, encouraging me to continue without too many worries (in her own way).

Then, I want to thank all the NESLab people: Andrea M., Rei, Andrea P., Veronica and Francesco. I want also thank Mattia not only as a member of the NESLab but also for being a friend during all these years.

Finally, I want to thank Daniele for making the house a livable and happier place and becoming a friend.

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Acknowledgements **v**

Contents **vii**

1 Introduction **1**
1.1 Problem and Contribution 2
1.2 Thesis Structure 5

2 State of the Art **9**
2.1 Publish/Subscribe 9
2.1.1 Subscription Models 9
2.1.2 Network Topology 11
2.1.3 Protocols 12
2.2 Request/Response 16
2.2.1 Protocols 16
2.3 Hybrid Approaches 19
2.3.1 Zenoh 19
2.4 Comparison 23
2.5 Location Awareness 25

3 Problem Statement And Design Space **27**
3.1 Motivation 27
3.2 Problem Statement 29
3.3 Solution Space 29
3.3.1 Location as a Topic 30
3.3.2 Location in the Payload 31

3.3.3 Routing 32
3.3.4 Encoding Spatial Information 32
3.4 Solution 33

4 Design **37**
4.1 Overview 37
4.2 Internals Design 39
4.2.1 Zenoh Location-aware API 39
4.2.2 REST API 45

5 Embedding Location **47**
5.1 Base64 47
5.2 MGRS 48
5.3 Bloom Filters 51

6 Implementation Highlights **59**
6.1 Zenoh Internals 59
6.2 Location Key 60
6.2.1 Base64 Key 61
6.2.2 MGRS Key 62
6.2.3 Bloom Filter Key 64
6.3 API Wrapper 66
6.4 REST Interface 70

7 Experimental Evaluation **71**
7.1 Experimental Setup 71
7.1.1 Metrics 72
7.1.2 Baselines 73
7.2 Results Overview 74
7.3 Zoning Network 76
7.3.1 Network Layout 76
7.3.2 Results 78
7.4 Near-field Location-aware Subscribers 80
7.4.1 Network Layout 80
7.4.2 Results 82
7.5 Far-field Location-aware Subscribers 85
7.5.1 Network Layout 85
7.5.2 Results 86

7.6 Computing Capacity 89
7.7 Transport Protocol 93
7.8 Bloom Filter Configuration 97
7.8.1 Setup 97
7.8.2 Results 99

8 Conclusion **101**

Bibliography **105**

List of Figures **109**

List of Tables **111**

1 | Introduction

The proliferation of Internet of Things (IoT) systems revolutionized the way we perceive and interact with our surroundings. At the heart of this transformation lies data [10]. IoT systems quietly gather, transmit, and analyze an astonishing volume of data [17]. Given the constraints on computational capabilities in IoT devices, their primary functions involve sensing and transmitting data. This data is subsequently processed in more powerful nodes. Consequently, a critical aspect lies in the method of data transmission, especially the communication protocols employed.

The Publish/Subscribe (Pub/Sub) message pattern is a widely adopted paradigm to build flexible and scalable communication systems in a distributed environment [2]. This model is suitable for data dissemination and event-driven communication. It offers a decoupled and asynchronous communication pattern, where publishers and subscribers operate without being aware of each other. Due to the flexibility this model provides, various publicly or commercially available systems implement the Pub/Sub paradigm. Each implementation differs in elements such as supported network topology, and subscription model. Among available protocols, some examples are MQTT, MQTT-S, and AMQP, which are commonly employed in IoT for sensor data updates, alert notification, dynamic configuration, and applications where sensors produce data without the need to receive commands frequently.
On the other hand, the Request/Response (Req/Resp) model is also employed in IoT. This model is useful when direct communication between two nodes is necessary, enabling control and coordination of individual devices, as well as specific data retrieval. This model also provides synchronous communication, ensuring that requests are accompanied by timely responses. So, Req/Resp can be applied in scenarios where devices need to listen for incoming requests, such as data retrieval from a sensor and triggering a specific action on a device, like switching on a light bulb. Among different Req/Resp protocols some examples of protocols applied to IoT are HTTP and CoAP [15][4].

An emerging protocol called Zenoh unifies Pub/Sub and Req/Resp models. Zenoh employs a Pub/Sub-based paradigm and integrates geo-distributed storage, query, and com-

putation capabilities with the goal of harmonizing data in motion, in use, and at rest. Its fully decentralized architecture ensures fault tolerance, eliminating the necessity for a central broker or server working both using multiple routers or peer-to-peer. All these protocols offer models and capabilities to meet IoT system requirements but Zenoh seems to be a promising protocol unifying Pub/Sub and Req/Resp models in one unique protocol.

1.1. Problem and Contribution

With the proliferation of IoT systems in our lives and the growth of large-scale systems, the amount of data we produce increases notably. This huge quantity of data must be transmitted and stored. In addition, particularly in large-scale systems, the importance of the data may depend not only on the data but also on the location where it is generated. Let us consider a scenario where multiple cities deploy air quality sensors across various areas. The relevance of both live and stored data is intrinsically tied to the sensor's position within the data value. Hence, it becomes crucial to directly associate data with its location. On the other hand, individuals interested in the data should have the capability to receive real-time information or query stored data based on both the data type and the relevant location. While the nature of the data, such as air quality, remains the same across multiple sensing locations, the area of interest may vary. For instance, one might desire to consult data from the entire city or from a specific zone. For this reason, there is a need to obtain sensor readings using both the data type and the location as two orthogonal and independent parameters.
Among the protocols under consideration, none provide this capability; in other words, they are not *location-aware*. Being location-aware implies that the distributed system is aware, or can become aware upon request, of the position of a device or, in our context, the location to which a message corresponds. Our main goal is to associate a message with both its generated location and the entities that have an interest in it. Practically, our goal is to *link sensed data to its respective sensing location for both stationary and moving sources, ensuring that these two facets remain independent of each other*.

To incorporate location-aware features, we opt not to build a protocol from scratch, an operation that may result in something useless or not properly testable. We opt for Zenoh since it looks like the most promising protocol thanks to the support of a distributed architecture. It also accommodates both the Pub/Sub model for managing real-time data from sensors and the Req/Resp model, which is suitable for implementing distributed storage, historical data retrieval, and on-demand data access. Zenoh is a topic-based protocol, wherein a message is described using a topic. This allows us to articulate

our data using multiple levels for enhanced granularity. Zenoh, in addition, offers the possibility to introduce custom behaviors.

To design our location-aware version of Zenoh we take into account three different aspects:

- Compatibility with the protocol's native version: Ensuring seamless operation for current applications employing Zenoh, even in the modified version. Also, ensuring the ability to concurrently utilize both the native and location-aware versions of the protocol within the same deployed network.
- User-friendliness: Prioritizing ease of use for developers, minimizing the intricacies associated with location management.
- Mobility support: Acknowledging the dynamic nature of IoT devices, particularly those changing locations, such as sensors on vehicles or wearable devices on individuals in motion.

The topic-based nature of Zenoh, suggests us to use the topic for spatial information, keeping the payload for the data. However, specifying explicitly the location of a message using multiple topic levels seemed immediately like a suboptimal solution, lacking freedom both in describing an extended space and in the fact that geographic location would be more tied to the type of data rather than the data itself.
Instead, we chose to dedicate a single topic level to geographic information, placing it within a key that Zenoh can then identify and manage separately. In Figure 1.1 we report the high level architecture we design. This approach allows us to add no more than 6 lines of code to the core of the protocol, by essentially defining only the key prefix, and relocating its entire management to a new component, completely detaching from the regular topic management process. Only at the end of the match the result, whether positive or negative, is communicated to Zenoh's core for comparison with the information from the normal topic. Furthermore, this solution does not require changes to Zenoh's

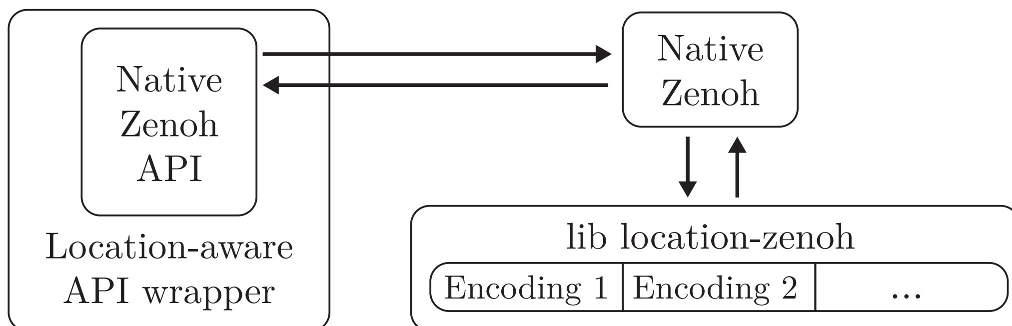


Figure 1.1: High level architecture.

routing component, preserving its functionality and efficiency. In addition, by decoupling the matching process, we ensure complete compatibility with all the devices using the native version of the protocol, leaving unchanged also the topic matching interface. This solution enables us to alter the message routing process from publishers to subscribers, going beyond simple string equality in topics, allowing new and more complex criteria just injecting information within a unique key.

To concretely implement this extension, we worked on two fronts: the APIs intended for developers and the location match module on the router side. Regarding the APIs, we opt to use a wrapper to keep the native APIs unchanged, employing the same function signatures with the addition of necessary parameters. In the wrapper we construct, developers define the topic as in the native version without worrying about including geographic data. The introduction of new functions, in addition to existing ones, allows reading the position, defining an extended space, and specifying the validity of the current position. Transparently to the developer, the wrapper takes care of generating the key containing geographic data, appending it to the topic, and performing the desired operation, whether it is a publish or a subscribe. This approach allows us to treat positional data encoders as plugins within the location-aware system, providing the flexibility to define various encoding techniques with distinct properties and levels of expressiveness.

In our extension, we offer three distinct encoding techniques, each varying in its level of expressiveness and required processing overhead:

- **Base64:** The default method involves the Base64-encoded version of a JSON object. While this results in a longer key, it enables the detailed description of complex areas.
- **Military Grid Reference System (MGRS):** The MGRS is a concise, simple, and lightweight encoding method. It facilitates the description of squares within a grid system, ranging in precision from 6-degree longitudinal bands to $1m^2$. It results in a shorter and more manageable key.
- **Bloom Filters:** A Bloom Filter is a binary data structure that allows querying the membership of an element in a set. As it relies on a bit array, the matching process is lightweight due to bitwise operation. As an encoding technique, it enables the description of relatively complex areas but introduces a probability of false positives.

Subsequently, we enter an evaluation phase to assess the efficiency of our solution. Latency and throughput are the metrics we consider for the evaluation since they measure the performance of the system respectively in terms of time required for a message to reach the destination and the number of messages the system can handle in a second: two

important aspects in a distributed system.
We conduct identical tests using the three location-aware encoding techniques we provide and compare the results with baselines in which messages are routed considering only the normal topic. In baselines' subscribers, we discard messages which do not meet location criteria upon receipt.
We employ three network topologies. Initially, we establish a network with two logically separate but interconnected geographical zones to assess the efficiency of the location-aware mechanism. In this topology, all three location-aware techniques outperform the baselines in both latency and throughput. This demonstrates that in a logically split network, the protocol performs significantly better by preventing out-of-range messages from being routed where unnecessary, thus avoiding resource waste. Using location-aware encoding techniques results in a mean latency reduction of more than 50%.

In the other two topologies, we eliminate the distinct zones, creating a mixed network to examine the performance of the encoding techniques and understand the limitations. In both topologies, we conduct tests under congested network conditions by reducing link capacity and generating a high volume of traffic. We do that to stress the system, to highlight the limits and the differences among techniques. MGRS demonstrates superior performance attributed to its concise representation and straightforward matching process, outperforming Base64 and Bloom Filters by around 40% in latency.

Despite being beyond the intended scope of the location-aware implementation, we also conducted tests on location-aware techniques to address all subscribers in a small network where there is no distinction in location. As expected, the introduced overhead of performing a double match for both topic and location leads to a latency increase, reaching up to 70% compared to the baselines.

In summary, location-aware techniques prove highly effective in scenarios with well-defined geographical zones, decreasing latency of more than 50%. However, in mixed, non-logically split networks with a small number of devices, their effectiveness requires a comprehensive study that considers network structure, performance, and the desired objectives. Regardless of the scenario, MGRS exhibits better performance compared to other encoding techniques, and even in a suboptimal scenario for a location-aware application, it proves to be quite efficient.

1.2. Thesis Structure

This thesis is structured in eight chapters. To begin, we present an overview of current communication protocols applicable to the IoT domain. Subsequently, we delineate the

problem and propose some solutions, offering a rationale for their viability or drawbacks, concluding the section by revealing our chosen solution, elucidating the primary reasons behind this decision, and giving an illustrative example. Following this, we outline the high-level design of our solution, and we discuss some key decisions we make during the implementation phase. Then, we present the setup, metrics, and outcomes of our conducted experimental evaluation.

In the following, we provide a summary of the chapters of our thesis:

- In **Chapter 2** we analyze and compare various communication models and their associated protocols to determine which ones best fulfill the requirements of IoT systems. After evaluating two communication models and six different protocols, Zenoh emerges as the optimal choice for our objectives. Its support for multiple network architectures and both communication models makes it well-suited for a broad range of applications.
- In **Chapter 3** we delve into the issue of linking sensed data to its respective location for both stationary and moving sources, keeping the two aspects separate from each other. We present various potential solutions and explore the information encoding in a single topic level, which represents our solution due to its integration and extendibility.
- In **Chapter 4** we present the high-level design of our solution, along with examples of the final APIs. Crafted for developer-friendly usability, our wrapper APIs encapsulate the native ones, preserving identical function signatures while seamlessly incorporating new functions and parameters for handling location information.
- In **Chapter 5** we introduce the encoding methods for location data. For each method, we outline how we handle the information and execute the matching of the location key data. We also explain our application of Bloom Filter as an encoding technique, outlining the creation of the filter on the client side and the execution of element queries on the router side.
- In **Chapter 6** we delve into the most crucial decisions made in the implementation of Zenoh's location-aware feature, emphasizing the handling of diverse location keys derived from the encoding techniques. We delve into the matching process employed by routers to separate location data from the topic and explore how the final matching results are consolidated.
- In **Chapter 7** we detail the setup of the evaluation, and the rationale behind the

chosen network topologies. We discuss the results obtained for each topology and provide a comparison of the encoding techniques individually. Additionally, we compare the results obtained when reducing computational capacity and changing the transport protocol. Finally, we dedicate a section to discussing the false positive probability of Bloom Filters, explaining the rationale behind the value used during the experiments. The results demonstrate our implementation actually performs a logical split of the network into geographic zones, enhancing system performance through message filtering in routing. Additionally, they indicate that adopting location-awareness in a small, non-location-based network may not be advisable. However, in a slow, resource-constrained, and congested network, employing a location encoding technique can still be beneficial.

- In **Chapter 8** we elaborate on conclusions we derived from our thesis and some possible future work.

2 | State of the Art

The purpose of this chapter is to provide the reader with some basic concepts about technologies and protocols that serve as a background for future reasoning in this thesis. We describe in detail the publish and subscribe paradigm, existing protocols, and their application in different contexts, comparing them with Req/Resp ones. Subsequently, we focus on a new emerging protocol called Zenoh by ZettaScale [26]. Finally, we focus on the location-awareness concept in routing and its application in Publish/Subscribe systems.

2.1. Publish/Subscribe

The Pub/Sub message pattern is a widely adopted paradigm to build flexible and scalable communication systems in a distributed environment [2]. In a Pub/Sub system, generally represented as in Figure 2.1, participants communicate with each other by exchanging messages, or *events*. A participant can be a sender, which publishes a message without the necessity to know who is the receiver thanks to an abstract group it publishes to. On the opposite, a subscriber represents an entity that expresses an interest in one or more categories of events without the knowledge of which publishers, if any, there are.

2.1.1. Subscription Models

Given the widespread usage of the Pub/Sub message pattern, there are various subscription models in literature to best fit possible use cases. There are four distinct types of subscription models based on the level of expressiveness they offer:

- **Channel based:** Publishers publish messages into channels using a channel ID. When a publish occurs, the message is placed in a FIFO queue at the broker side that represents the channel. Subsequently, the message is broadcast to all the subscribers of the queue [18].
- **Topic based:** It represents an improvement of the channel based model introducing a logical channel based on the topic name. It connects the publisher with all

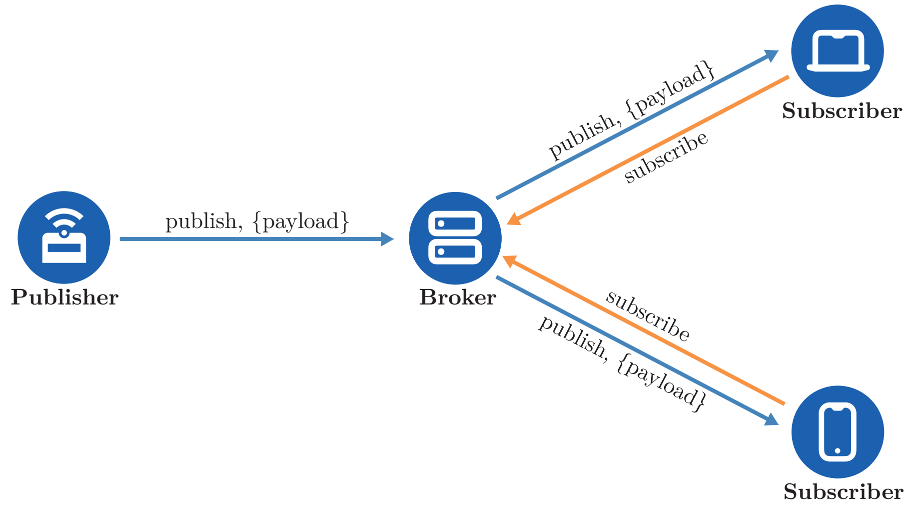


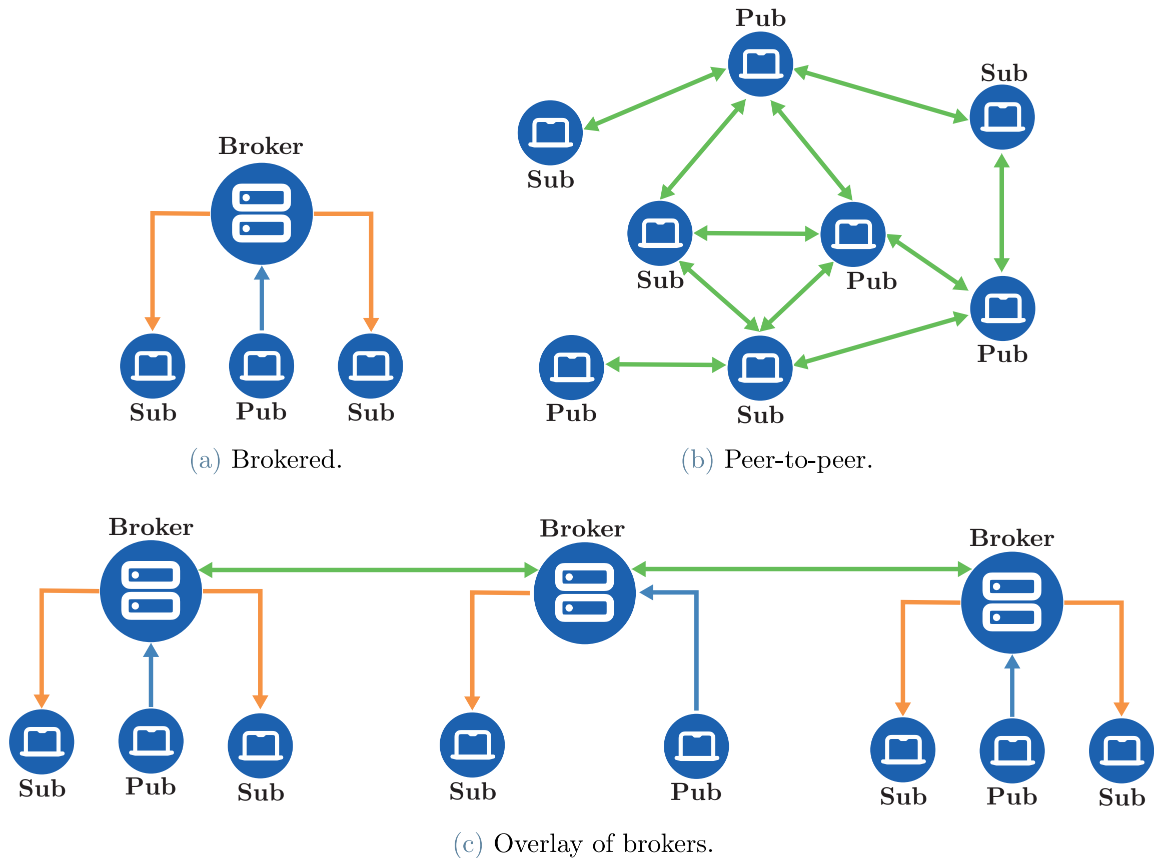
Figure 2.1: Pub/Sub message pattern.

the subscribers that are interested in a particular topic leaving the message content hidden to the broker. In a topic-based model a hierarchical approach is used to provide event classification and the definition of subtopics, for example, `A/B` represents an event belonging to a topic `A` and subtopic `B`. In the same way `A` can have multiple subtopics: `B` and `C`, obtaining a tree structure where `A` is the root [18][2].
In the literature, *subject-based* models represent a synonym of topic-based ones [2].

- **Content based:** This approach offers a different perspective, aiming to enhance the expressiveness of subscriptions. In the content-based model, subscribers and brokers have knowledge of the message content, enabling the application of more robust filters [18]. These filters act as "conditions" on the content, allowing developers to create complex predicates based on message fields [2]. However, this model requires a tradeoff between filter performance, in terms of delay and resource consumption, and the level of expressiveness offered by the filters themselves.

- **Type based:** Eugester, in his paper, presents the type-based approach [7], which relies on data structures. In other words, a subscriber only receives a published message if the structure of the payload aligns with the structure of the payload provided at subscription time [18]. He creates this model to bring Pub/Sub much closer to code structure and expressiveness and to model messages with the language itself. On the other hand, this model creates difficulties in portability since it is not enough to translate the API into a different language but also the message model.

In practice, we focus on topic-based and content-based subscription models, as other models can be easily mapped or represented by these two.



(a) Brokered.

(b) Peer-to-peer.

(c) Overlay of brokers.

Figure 2.2: Possible network topology in Pub/Sub pattern.

2.1.2. Network Topology

Until now, we have discussed publishers and subscribers without addressing the management of messages. In a Pub/Sub system, there are three primary alternatives for organizing the system [2]:

- **Centralized broker:** In this architecture (Figure: 2.2a) a unique logical server entity, called broker, is responsible for receiving, filtering, and dispatching messages from publishers to subscribers [2] [19]. Since the logical central node is unique it represents both a bottleneck and a single point of failure. In spite of that, this architecture is adopted in most of the commonly used protocol deployments.
- **Overlay of brokers:** In an overlay of brokers (Figure: 2.2c) we have multiple brokers interconnected in a network. A client connects to one of the brokers and Pub/Sub functionality is realized by means of a distributed algorithm into the broker network [2]. The architecture is designed to be transparent to the clients interacting with the connected broker. However, as the message distribution is handled by the

brokers, this system scales better than the centralized broker architecture.

- **Peer-to-peer:** In a full peer-to-peer architecture (Figure: 2.2b) there is no need for a middleware, messages are directly routed between peers, which acts both as a client and a router at the same time [2] [12]. This architecture is suitable for event dissemination scenarios involving a limited number of participants and a small geographical scale due to difficulties in network management since nodes can join and leave at any time and each node needs to be aware of the network state.

2.1.3. Protocols

Various publicly or commercially available systems implement the Pub/Sub paradigm. Each implementation differs in elements such as supported network topology, subscription model, Quality of Service (QoS), and application field they best fit in.
In the following section, we introduce some common implementations, highlighting their relevant properties and main application fields.

MQTT (Message Queuing Telemetry Transport Protocol)

MQTT is one of the oldest machine-to-machine Pub/Sub protocols released by IBM in 1999. Table 2.1 summarizes the key properties.

	Property
Year	1999
Architecture	Centralized Broker
Subscription model	Topic-based
Header size	2 Byte
Message size	Up to 256 MB
Quality of Service	QoS0, QoS1, QoS2
Transport protocol	TCP
Security	SSL/TLS
Encoding format	Binary
Methods	Connect, Disconnect, Publish, Subscribe, Unsubscribe, Close

Table 2.1: MQTT properties.

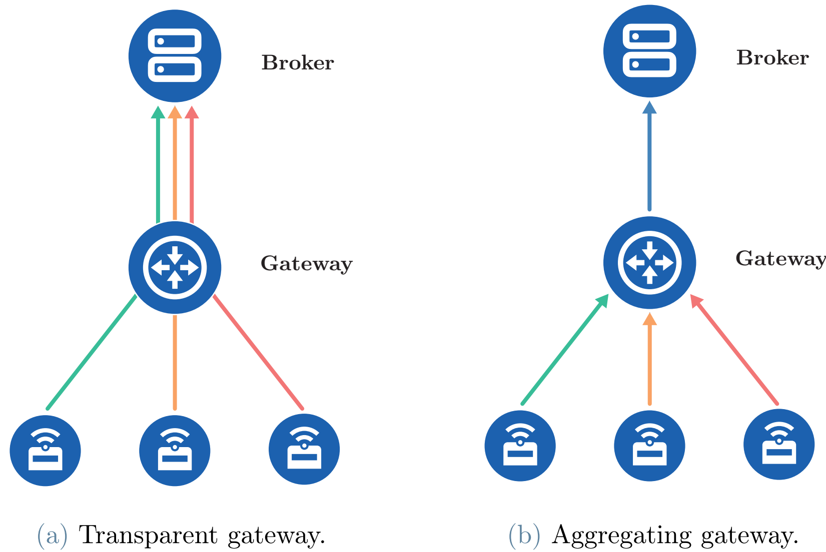
MQTT is a lightweight protocol focused on wireless devices, designed to fit constrained and unreliable networks characterized by high delay and low bandwidth, and based on TCP communication and TLS/SSL security.
Being a binary protocol, MQTT does not impose a fixed payload data type but it requires a maximum payload size of 256MB plus a fixed 2-byte header. MQTT relies on a centralized broker architecture providing a topic-based subscription model [15][13][19]. In the MQTT architecture, clients establish connections with a central broker, and both publishers and subscribers indicate their interest in a specific topic.
Topics in MQTT utilize two wildcard levels, enabling the matching of different data ranges. [19][9]. The single level wildcard (+) replaces one topic level, for example, *building/+/temperature* matches topic like *building/21/temperature* or *building/20/temperature* but it does not match a topic like *building/deib/21/temperature*. On the opposite, the multi level wildcard (#) is able to match multiple topic levels but only at the end of the topic. The topic *building/#/temperature* is not valid since the wildcard is placed in the middle; *building/deib/#* is valid and it matches all the topics that have as root levels *building/deib*, for example, *building/deib/21* as well as *building/deib/20/temperature* and so on.
Furthermore, MQTT supports three different QoS levels between a publisher/subscriber and the broker to ensure different levels of assurance of data distribution [19]:

- QoS0 (At most once): a message is delivered once or not at all. MQTT does not provide any guarantee of reception for the messages sent.
- QoS1 (At least once): a message can be sent at least once, and it can also be sent multiple times by setting the duplicate flag.
- QoS2 (Exactly once): a message is sent exactly once by utilizing a 4-way handshake method. This ensures that a sent message is received without duplication or loss.

Finally, MQTT provides *retained messages*, deliverable also to new subscribers that match the message topic, and *wills* that are messages on a defined topic, the broker sends in case of an unanticipated detach.

MQTT-S (MQTT for Sensors)

MQTT-S is a modified version of the standard MQTT protocol to meet requirements of low-end, battery-operated sensor/actuators devices over bandwidth-constraint wireless sensor networks (WSNs) [9].
According to Hunkeler et al. MQTT-S is designed to be as close as possible to MQTT



(a) Transparent gateway. (b) Aggregating gateway.

Figure 2.3: Gateway types in MQTT-S.

and to support all the features of MQTT to have a smooth integration [9].
The complexity is shifted to the broker/gateway side, allowing the devices to remain simple and lightweight. The protocol utilizes UDP, which does not require a connection-oriented or in-order delivery mechanism. Additionally, packet length is reduced to a maximum of 128 bytes, with a payload limit of 64 bytes. These design choices make MQTT-S suitable for various network types, including IEEE 802.15.4, the same transmission layer ZigBee, for example, uses too. However, two services must be provided:

- Point-to-point data transfer (unicast service) that enables message transport between two points based on the network address;
- One-hop broadcast data transfer that lets a sent message be received by all the nodes in the transmission range: this is typical of all wireless networks.

In MQTT-S, a *gateway* appears between sensor/actuators and the broker as seen in Figure: 2.3. This gateway is responsible for translating messages between MQTT-S and MQTT, as the sensors connect to the gateway, which in turn connects to the broker.
Two types of gateways can be deployed based on how they exchange messages with the broker. The first type is the **transparent gateway**, seen in Figure: 2.3a, which maintains a separate MQTT connection for each MQTT-S device. While this implementation is simpler, it becomes challenging to scale as the number of connections between the gateway and the broker increases with the number of devices in the sensor network.
The second type is the **aggregating gateway**, seen in Figure: 2.3b, which maintains a single connection with the broker and selectively transports messages from the WSN to

the broker. This approach enables for more efficient utilization of resources and better scalability, as the gateway decides which messages to forward to the broker. The aggregating gateway does not act as a broker, it just aggregates multiple messages to reduce the number of active connections between the sensors and the broker while the broker remains the central element for message dispatching.

AMQP (Advanced Message Queuing Protocol)

AMQP is a lightweight machine-to-machine message-oriented protocol designed to provide real-time, low-cost, and reliable group communication. It supports different types of communication like Pub/Sub, Req/Resp, direct messaging and so on [16] but we focus on Pub/Sub version. Table 2.2 summarizes the key properties.

Based on the client/broker model it includes different elements in its architecture: the *exchange* receives messages from a producer (publisher) and routes them to a specific queue; the *routing key* is the virtual address for the target queue and it can be a topic, a header or computed as a predicate of the payload; the *queue* represents the element in which subscribers subscribe, it can be private or shared, durable or transient, permanent or temporary and, combining different properties, we can use it to build different interaction patterns like store-and-forward, Pub/Sub, temporary reply queue [25][13][16][15].
AMQP is a powerful protocol to interconnect applications in an easy way ensuring performance since it can process large volumes of data in a relatively short time compared

	Property
Year	2003
Architecture	Brokered
Subscription model	Multiple
Header size	8 Byte
Message size	Negotiable
Quality of Service	Settle Format, Unsettle Format
Transport protocol	TCP, SCTP
Security	SSL/TLS, IPSec, SALS
Encoding format	Binary
Methods	Consume, Deliver, Publish, Get, Select, Ack, Delete, Nack, Recover, Reject, Open, Close

Table 2.2: AMQP properties.

with other protocols like HTTP. It also provides two QoS levels: the *unsettle format* is unreliable and it is similar to MQTT QoS0 while the *settle format* is the reliable one close to MQTT QoS1 definition.

2.2. Request/Response

The Req/Resp (or Query/Reply) message pattern is one of the basic mechanisms of the Internet. Generally, a Req/Resp interaction is initiated by the client (requester) that contacts a server (replier). The server is responsible for receiving the request, processing the desired information, and returning the response to the client.
This pattern is synchronous, meaning the client waits for the server's response before proceeding. To prevent endless waiting in the event of network failure, the requester sets a timeout. If the timeout expires, the expected response was not received within the specified time, considering, as a consequence, the request unsuccessful.
The Req/Resp pattern provides a mechanism to perform also asynchronous operations, which, for sure, can not stay inside the interval of the timeout. The future-based approach uses an object representing the result of an asynchronous operation. When making a request the function creates and returns a future object, representing the ongoing request. The client can then decide whether to wait for the response or proceed with other tasks. Similarly, the promise-based approach lets a request to crate and return a promise object that represents the response that eventually arises.

2.2.1. Protocols

The Req/Resp paradigm is widely recognized through its primary implementation, HTTP. However, in this section, we explore not only HTTP but also other protocols that follow this pattern.

HTTP

HTTP is the web messaging protocol over TCP (and SSL/TLS) and it represents the standard in the world wide web since 1997. It is a Req/Resp stateless protocol that uses Universal Resource Identifiers (URIs) instead of topics to identify data, the server sends data using URI while clients receive data in a particular URI [15]. Table 2.3 reports a summary of key properties of HTTP/1.1, HTTP/2.0, and HTTP/3.

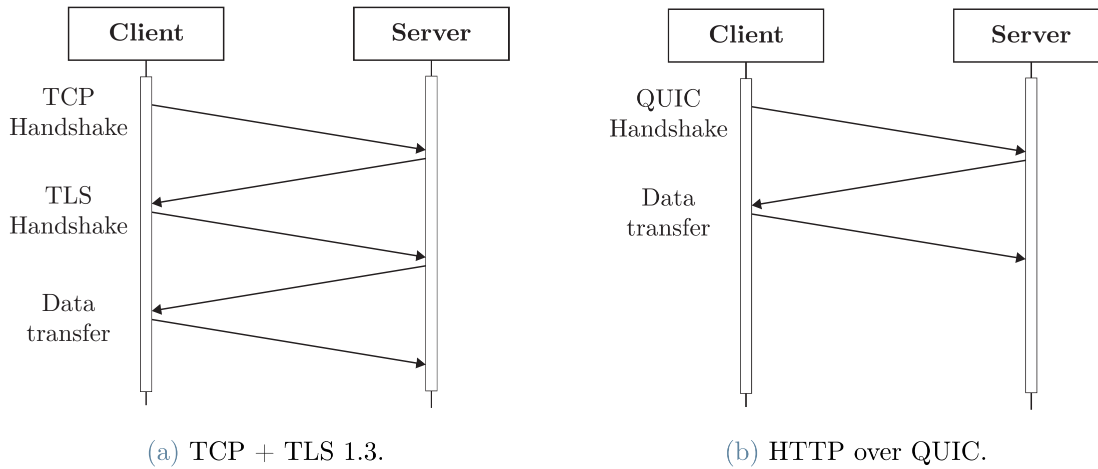
HTTP is a text-oriented protocol, which does not define a predefined header and payload size; instead, these dimensions vary based on the server technology being used. However, due to the widespread acceptance of HTTP, various functionalities such as pipelining,

	HTTP (1.1 and 2.0)	HTTP/3
Year	1997 and later	2022 (proposed)
Architecture	Client/Server	Client/Server
Header size	Undefined	Undefined
Message size	Undefined	Undefined
Quality of Service	Limited (via TCP)	Limited (via TCP)
Transport protocol	TCP	UDP (QUIC), TCP
Security	SSL/TLS	SSL/LTS, QUIC
Encoding format	Text	Text
Methods	Protocol Standard	Protocol Standard

Table 2.3: HTTP properties.

persistent connections, and chunked transfer encoding are typically available in all the commercially available implementations.
HTTP/1.1 represents the default version today even if version 2.0 exists as a standard too. While **HTTP/2.0 is** widely supported by major web browsers and servers, it is not enabled by default and requires specific configuration.
Establishing an HTTP connection between the client and server, especially when using SSL/TLS, involves a three-way handshake that introduces delays that can be significant in certain situations. Version 2.0 partially addresses this issue maintaining the original handshake pattern. In fact, HTTP/2.0 uses a binary format instead of the original plain-text of the previous versions in addition to the introduction of TLS False Start, which allows a client to send application data to the server while they still completing the last handshake for SSL/TLS connection establishment [14].

The next generation of HTTP, known as **HTTP/3**, was officially proposed as a standard in June 2022 (RFC 9114) [3]. The key difference between the previous version is the usage of QUIC (Quick UDP Internet Connection) protocol, originally developed by Google.
In HTTP/3.0, QUIC replaces the three-way handshake used for connection establishment, resulting in reduced delays. This is achieved by combining a fast UDP-based transport with TLS as an integral part of the connection [14][6].
In Figure 2.4 we present the connection establishment difference between HTTP utilizing the latest version of TLS and HTTP over QUIC.
Furthermore, in TLS 1.2 and earlier versions, the handshake process involves one more



(a) TCP + TLS 1.3.

(b) HTTP over QUIC.

Figure 2.4: Connection establishment differences in HTTP.

round trip between the client and server before the secure connection can be established. This additional handshake introduces extra latency, particularly in situations where network conditions are less than optimal or when there are delays in transmitting the handshake messages.

CoAP

Constrained Application Protocol (CoAP) is a lightweight machine-to-machine protocol released in 2010 by the IETF CoRE group. It is a Req/Resp protocol but it also provides a Resource/Observe pattern similar to Pub/Sub [15]. Table 2.4 summarizes the most relevant properties of CoAP.

CoAP is often described as a slim version of HTTP since it is too heavy for constrained devices, or a fresh approach to it, designed specifically for limited resource consumption. The core concept of CoAP involves the use of UDP instead of TCP, along with a simple message layer for retransmitting lost packets. In terms of QoS, CoAP supports only two types of messages: confirmable (requiring acknowledgment) and non-confirmable.

CoAP has a compact 4-byte header and a binary payload, it provides similar functionality to common HTTP methods such as GET, PUT, POST, and DELETE. Resource identification is achieved through the use of URIs, while response codes are encoded in a single byte [15][4].

	Property
Year	2010
Architecture	Client/Server
Header size	4 byte
Message size	Small and undefined
Quality of Service	Confirmable and non-confirmable
Transport protocol	UDP, SCTP
Security	DTLS, IPSec
Encoding format	Binary
Methods	GET, POST, PUT, DELETE

Table 2.4: CoAP properties.

2.3. Hybrid Approaches

Standardized and commonly available protocols can have more than one model available. For example AMQP, we describe as Pub/Sub, provides also Req/Resp capabilities as well as CoAP, which is mainly a Req/Resp protocol, can work also as Resource/Observe.
An emerging protocol, Zenoh, unifies Pub/Sub, Req/Resp, and REST models.

2.3.1. Zenoh

Eclipse Zenoh is a communication protocol developed by ZettaScale and included in the Eclipse Foundation projects [27][26]. It focuses on enabling seamless communication between cloud and edge computing environments, integrating nodes across different network layers under a unified protocol.
Zenoh utilizes a Pub/Sub-based paradigm and incorporates geo-distributed storage, query, and computation capabilities, aiming to unify data in motion, in use, and at rest. Its fully decentralized architecture ensures fault tolerance, eliminating the need for a central broker or server. Nodes can be configured in various ways, including a fully distributed system with peer-to-peer communication.
Zenoh supports multiple communication protocols, including TCP, UDP, TLS/mTLS, QUIC, WebSocket, UNIX-Socket, and even non-IP protocols like Bluetooth and serial connections [12]. Furthermore, Zenoh facilitates the coexistence of multiple network topologies, as depicted in Figure: 2.5, enabling device interconnectivity even over the public internet, typically facilitated by Zenoh routers [23]. Zenoh provides the functionalities of

both producing and receiving data and routing. By combining these functionalities, it is possible to create three different device configurations:

- **Router:** it is a fundamental component of the Zenoh stack, playing a crucial role in managing data flow across different endpoints within the system. Employing a Pub/Sub model, the router efficiently distributes data, ensuring that it reaches the intended destinations. As other protocols, one of its notable features is the ability to automatically detect and adapt to changes in the network topology. This capability ensures that data is consistently delivered to the correct destinations, even in the presence of dynamic network changes. By dynamically adjusting its routing mechanisms, the Zenoh router maintains reliable and efficient data distribution throughout the system.

- **Peer:** it represents any device or system that is running the Zenoh protocol and capable of sending and receiving data within a distributed system. A Zenoh peer can be a physical device, such as a sensor, actuator, or gateway, or it can be a virtual device running on a cloud-based infrastructure. Regardless of its form factor, a Zenoh peer is able to communicate with other peers in the system using the Zenoh protocol, allowing data to be exchanged in a highly scalable and efficient manner since a peer includes the routing capability too. Each Zenoh peer is responsible for managing its own data sources and subscriptions, and can dynamically discover and connect with other peers as needed.

- **Client:** A Zenoh client refers to a software component or application that employs the Zenoh protocol to establish communication with other components or devices within a distributed system. However, unlike routers and peers, a Zenoh client is typically an end device and does not possess the capability to act as a router for other devices. It typically uses the Zenoh API to publish or subscribe to data sources, and may also use other features of the Zenoh stack, such as query and store, to retrieve and manipulate data.

As shown in Figure 2.5, the Zenoh infrastructure accommodates the coexistence of multiple device configurations, enabling the creation of various network topologies based on the configuration each device has. [23]. Some of the configurations that can be implemented include:

- **Mesh:** Mesh topology refers to a network configuration where devices are interconnected in a decentralized manner. So, it allows data transmission directly between any two devices within the network, creating multiple communication paths and increasing redundancy. This redundancy enhances network reliability, as it allows

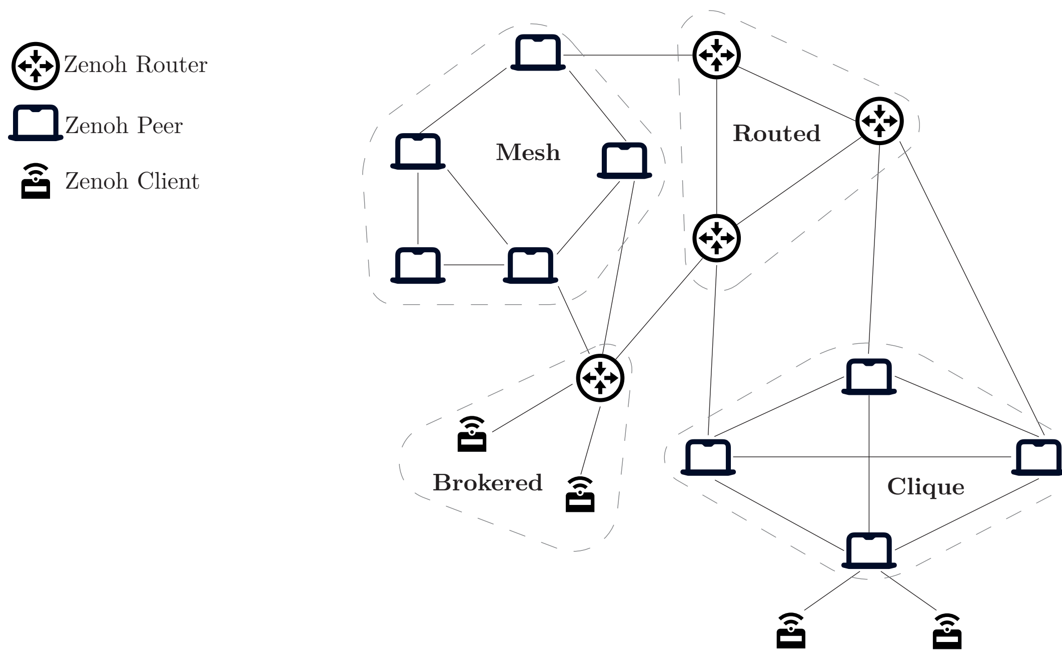


Figure 2.5: Zenoh network topologies.

for alternative routes in case of a connection failure or network congestion. Mesh topologies are common in large-scale networks.

- **Clique:** A clique topology is a fully connected network where each node has a direct link to every other node. This topology promotes robust and efficient communication between devices as there are no intermediaries or bottlenecks in the network. Information can be quickly disseminated to all devices within the clique, making it suitable for scenarios that require high levels of interconnectivity, such as small-scale peer-to-peer networks or collaborative computing environments. However, as the number of devices increases, the complexity and cost of maintaining direct connections between all devices also grows, which limits its scalability.
- **Routed:** A routed topology refers to a network configuration where data flows through a central router or a set of interconnected routers. Devices within the network are connected to the router(s) and communicate with each other through these routing nodes. This topology is suitable for large networks where scalability and centralized control are essential.
- **Brokered:** The brokered topology facilitates communication between connected devices through a central entity called broker, which acts as an intermediary for exchanging messages and coordinating communication. The broker is responsible for receiving messages from devices and delivering them to the appropriate recipients based on predefined rules or subscriptions. This topology promotes decoupling

between devices, as they do not directly communicate with each other but rely on the broker for message routing. In Zenoh, a peer or a router, that is, a node with Zenoh routing capabilities represents the broker.

Finally, within a Zenoh network, both peers and clients have the capability to autonomously discover and connect to a router or another peer through UDP multicast, provided that the network supports this feature. This automatic discovery process is commonly known as *scouting* [23]. However, in scenarios where UDP multicast is not supported or available in the network infrastructure, manual configuration of the links between peers and routers becomes necessary.

Zenoh offers a diverse range of communication APIs that facilitate various forms of data exchange and interaction within a distributed system. Some of these APIs include:

- **Publish-Subscribe API:** This API provides a flexible and efficient way to distribute data between multiple publishers and subscribers within a system. Publishers publish data on a specific topic, and subscribers receive data on that topic. The API also supports wildcards, enabling subscribers to receive data on multiple topics using a single subscription. Wildcards are similar to MQTT ones, in fact, *** matches a single level exactly like *+* while **** has the same meaning of *#*. However, in Zenoh, wildcards can be used at any level, in fact, there are no restrictions, as happens instead in MQTT. Finally, Zenoh introduces a new wildcard *$**, which lets to match a part of the topic level, for example, *building/de$*/temperature* matches *building/deib/temperature* as well as *building/deposit/temperature* but not *building/dica/temperature*.
- **Query API:** This API provides a way to query data stored within a Zenoh network. It allows to build queries by using a simple syntax that supports filtering, aggregation, and other data manipulation operations.
- **Store API:** This API provides a key-value storage to store and retrieve data within a Zenoh network. The storage is distributed and fault-tolerant, providing high availability and scalability.
- **Peer-to-Peer API:** This API provides a way to directly connect two Zenoh peers and exchange data between them. This can be useful for low-latency, high-bandwidth communication between devices in a distributed system.
- **REST API:** This API provides a simple RESTful interface for interacting with a Zenoh network. It allows to publish and subscribe to data, query the network, and

interacting with the key-value store. This also includes an admin space letting a network admin to inspect and analyze connected clients and devices' roles.

2.4. Comparison

We explored two communication paradigms and their widely used protocols. By means of a qualitative analysis, in this section, we delve into their application within Internet of Things (IoT) systems. By referring to an IoT system, we take into account an architecture that considers not only sensors and their communication protocols, but also the edge and cloud infrastructures required for data processing, storage, and retrieval. In this scenario, it is important to select protocols that best fit the required layer in terms of performance, scalability, and, most importantly, resource consumption.
The two communication models, Pub/Sub and Req/Resp, have distinct characteristics and are suited for different scenarios. The Pub/Sub model is suitable for data dissemination and event-driven communication. It offers a decoupled and asynchronous communication pattern, where publishers and subscribers operate without being aware of each other. This model excels in scalability because publishers and subscribers are decoupled, making it suitable for applications that rely on real-time event notification and reaction, such as environmental monitoring or intrusion detection. Additionally, Pub/Sub helps reduce network traffic as subscribers only receive data they are interested in, minimizing unnecessary data transmission.
On the other hand, the Req/Resp model is employed when direct communication between two nodes is necessary, enabling control and coordination of individual devices, as well as specific data retrieval. This model also provides synchronous communication, ensuring that requests are accompanied by timely responses. It is beneficial in scenarios where precise data from particular sensors or devices is required.
In summary, the Pub/Sub model is advantageous for data dissemination, event-driven scenarios, and scalability. On the other hand, the Req/Resp model is suitable for direct communication, control and coordination, specific data retrieval, and synchronous communication. The choice between the two models depends on the specific requirements and objectives of the application.

There are numerous studies in the literature focusing on the performance of protocols, particularly in relation to their implementation in sensor nodes. Sensor nodes are highly resource-constrained devices, characterized by their small size, reliance on battery power, limited computational capacity, and the need to optimize power, network, and space resources.

We consider a range of parameters that play a crucial role in evaluating various protocols. These parameters encompass aspects such as permitted message size, message overhead, resource requirements, latency, bandwidth, interoperability, standardization, reliability, and so on ...
In Table 2.5, we show an overview of each protocol along these properties. By examining these metrics, we gain a deeper understanding of the strengths and weaknesses of each protocol under consideration [15][13][12].

	MQTT	**AMQP**	**HTTP**	**CoAP**	**Zenoh**
Type	Pub/Sub	Pub/Sub	Req/Resp	Req/Resp	Both
Max message size	Mid-Low	Mid-High	High	Low	High
Message overhead	Mid-Low	Mid-High	High	Low	Mid-Low [21]
Resource requirements	Mid-Low	Mid-High	High	Low	Variable
Latency	Mid-Low	Mid-High	High	Low	Low
Bandwidth	Mid-Low	Mid-High	High	Low	Variable
Interoperability	Low	Mid-Low	High	Mid-High	High
Standardization	Low	Mid-Low	High	Mid-High	-
Reliability	High	Mid-High	Low	Mid-Low	High [22]

Table 2.5: Protocol comparison.

In Table 2.5, we employ qualitative labels to indicate each property of the mentioned protocols since numbers coming from studies, in this case, do not meet our goal given that test sets are different in terms of data type, message size, and architecture used. The analysis specifically focuses on comparing the mentioned protocols. The labels "Low" and "High" represent the two extremes, while "Mid-Low" and "Mid-High" indicate a middle position that leans more toward the lower or higher end, respectively. This approach aims to emphasize the strengths and weaknesses of each protocol in IoT system applications.
It is also important to clarify the concept of "Standardization". In this context, it does not imply that a specific protocol is officially recognized as a standard. Rather, it refers to the consistency and uniformity of functionality and guarantees across different implementations of the protocol itself.
We can observe that Zenoh strikes a tradeoff across all the properties. It offers notable advantages, such as high reliability and low latency in scenarios where payload size requires HTTP. Similarly, in situations where MQTT is applicable, Zenoh excels in providing enhanced interoperability. Finally, due to its versatility in adapting to diverse scenar-

ios, Zenoh emerges as a compelling alternative to other protocols, facilitating seamless integration of sensors, cloud environments, and various interconnected systems.

2.5. Location Awareness

Over the last few decades, researchers direct their attention toward location-sensing technologies and applications that are aware of positioning. The primary challenge they address revolves around accurately determining physical locations. Over the years, numerous solutions emerged.
At the forefront of these solutions is the Global Positioning System (GPS). This system achieves global ubiquity and is exceptionally well-suited for outdoor applications, that rely on satellite signals. In indoor environments, where signals might be unreliable or absent, GPS does not represent the optimal solution. Researchers figured out that indoor applications require different approaches and radio systems such as Wi-Fi, Bluetooth, and/or RFID represent possible solutions [8] with an accuracy of 1 to 5 meters. Ultrawide-Band (UWB) technology, instead, emerges as an indoor localization system in the last few years, it exists since 1976 but it was used mainly in military applications. UWB provides accuracy within the centimeter range, high-speed rate, and low cost [24] making it perfect for indoor localization.
Regardless of the solution used to obtain the position of an entity, we can summarize location data as an object whose attributes can be coordinates (x, y, z) or semantic representations of a place, such as (building, floor, room) or (state, region, city).

Orthogonally to the acquisition of position through physical systems, researchers also consider possibilities of localization at the level of communication protocols as well as at the software level, mainly related to the concept of *mobility*. The literature divides mobility into two categories [1]:

- **Physical mobility:** it refers to a device that establishes and terminates connections as it moves across networks. For example, when a device connects to various public Wi-Fi networks, or when we continue to browse on LTE/5G from the university network and then connect to the home network.

- **Logical mobility:** it denotes a device that possesses awareness of its changing location without moving physically, for example, a device that changes the access point under the same network. For instance, when we change different classrooms on campus without disconnecting from Wi-Fi, even though we remain connected to the same network, we change the access point used.

Mobility, whether physical or logical, deals with two opposite concepts: *location transparency* and *location awareness*. In our study, we focus on location awareness since we want both devices and the network to be aware of the location of a device.
Within IoT systems, Pub/Sub protocols align most effectively with a majority of use cases. The literature is relatively sparse concerning studies on the application of Pub/Sub protocols in a location-aware context. Among the existing studies, the emphasis lies on content-based architectures [1][5]. The primary reason behind adopting a content-based architecture is the strength of filters that can be employed during subscription setup. This empowers developers to implement location-based subscriptions by leveraging the positional information contained within published messages. For instance, a newsfeed operates as a content-based system, where a subscriber can choose to follow news topics of interest, such as "technology" news published in the state of "Italy" or in the state the user is in.
However, we deal with more precise positioning, such as that provided by GPS or a specific indoor location. Content-based Pub/Sub solutions in the literature, similar to the one proposed by Cugola et al. [5], take into consideration a client's physical location within the network. They rely on the brokers to which clients are connected, approximating the device's location based on the position of the first element of the communication infrastructure. These systems also necessitate adjustments in the routing system to function effectively. Nevertheless, these solutions do not account for scenarios involving "virtual" positional usage. For instance, they do not address situations where a subscriber deployed in the cloud lacks a physical presence at a specific location but remains associated with it.

Zenoh, being a topic-based Pub/Sub protocol and not content-based, does not feature filters for implementing location awareness applications. Nevertheless, as we discuss in Section 2.4, it is the protocol that offers the best trade-off between its properties and it is most suitable for large-scale IoT applications. For this reason, we consider the possibility of extending Zenoh to fit location-aware applications.

3 | Problem Statement And Design Space

In this chapter, we formulate the research question for this work. We delve into the underlying motivation driving this research, explore the existing solutions documented in literature that are relevant to the chosen scenario and technology, and ultimately, present the solution we opted for, along with a thorough rationale for its selection.
In Section 3.1 we present the motivation that led to this thesis work, in Section 3.2 we state the problem, in Section 3.3 we discuss three possible solutions we considered, and in Section 3.4 we present the high level solution we choose for this work.

3.1. Motivation

In today's interconnected world, the proliferation of Internet of Things (IoT) systems revolutionized the way we perceive and interact with our surroundings. At the heart of this transformation lies the significance of data [10]. In every facet of our daily lives, IoT systems quietly gather, transmit, and analyze an astonishing volume of data. This data encapsulates insights that hold the potential to enhance our lives in unprecedented ways [17]. From optimizing energy consumption in our homes to predicting traffic patterns for smoother commutes, data, and its transmission covers a central role. In a large-scale system such as a city or an entire country the relevance of a piece of data, many of the times, may depend not only on the data but also on the position it is generated.
Let us take into consideration, for example, the city of Milan and the amount of energy consumed per hour. In such a scenario, as illustrated in Figure 3.1, our focus could be on obtaining real-time energy consumption data from a specific area. This data would enable us to make precise energy distribution adjustments customized to local needs without taking into account data coming from neighboring areas. Simultaneously, data from multiple areas provide a comprehensive view of the situation, particularly in critical scenarios, or facilitate statistical analysis.

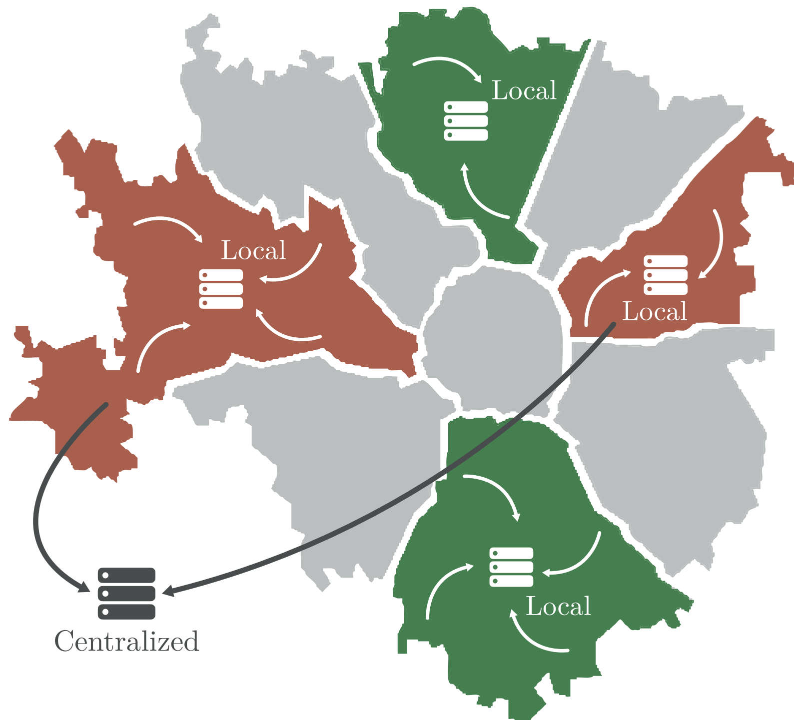


Figure 3.1: Example of data interest in Milano areas.

In a large-scale system, ensuring that a piece of data published in a given location is received only by subscribers in the interested location area represents an open problem. Introducing the location-aware functionality in the Pub/Sub topic-based Zenoh protocol represents the challenge.

In conjunction with data and locations, a large scale IoT system must ensures three core capabilities:

- **Live data:** It refers to real-time information that is constantly updated and reflects the most current state of a system or phenomenon. Live data is dynamic in nature, reflecting the current state or condition of a system, process, or phenomenon, offering immediate insights without any significant delay.
- **Mobility:** It refers to the movement and changes in the attributes of entities that generate, transmit, or interact with data within a given environment. In this case the location changes of an entity that is moving across areas must be reflected into the system.
- **Historical persistency:** It represent the dual w.r.t. live data. It refers to the potential of preserving data and its associated attributes over time, enabling their retrieval for future utilization, particularly for statistical analysis.

3.2. Problem Statement

In large-scale IoT systems, the most commonly employed protocols are those based on the Pub/Sub model with topic-based subscription model [18]. Unfortunately, among these protocols, there is no simple way to effortlessly link location information to a particular data point [20]. The options for a subscriber to effortlessly subscribe to a topic that corresponds to a data point within a designated area of interest are notably absent.

Let us consider, for example, a warning system for drivers that notifies them of road accidents that occur within a 2 km radius from their location. A potential system would publish information related to such an event in a topic like road/notification/accidents, which would also correspond to the subscriber's topic. In this scenario, it is not possible to know a user's location beforehand, neither define "zones" to which the user belongs due to the mobility of the subscriber. The entities within the system evolve over time, vehicles move along the road network, and a potential accident can occur in various locations. If we consider applying the system to a region or even an entire state, it is easy to understand how it is impossible to define geographic parameters beforehand for a driver who is interested only in what happens in a 2km range from himself.
This scenario is an example of why it is important to define a system that is able to adapt to changes over time in addition to being able to manage location information with different granularity while ensuring sufficient performance with regard to its users.

3.3. Solution Space

Among the protocols discussed in Chapter 2, Zenoh appears to have a better trade-off among its properties. Not only Zenoh is a Pub/Sub topic-based protocol, it is also designed to easily integrate devices from sensor to the cloud, making it perfect for our target application.
Furthermore, Zenoh offers good performance also in large-scale systems. In practice, messages are solely routed when a corresponding receiver for a given topic exists. In its routed configuration, Zenoh intelligently selects the optimal path, mitigating flooding events and therefore enhancing overall efficiency.
To establish a connection between data and position within the Zenoh framework, we outline four potential designs:

1. the utilization of multiple topic levels,

2. the usage of the message payload directly,

3. a refinement of routing processes, and
4. the encoding of spatial information to extend the Zenoh key expression language.

3.3.1. Location as a Topic

When considering a topic-based protocol approach, the initial solution that naturally emerges involves incorporating location information within the topic itself. This approach enables the allocation of one or more topic levels to convey location information. While this empowers publishers to accurately indicate their geographical position, subscribers would be required to employ wildcards and undertake filtering processes on the receiver side to make use of this information.
Consider, for instance, the topic <latitude>/<longitude>/temperature/external, which represents the temperature reading of an external sensor located at (<latitude>,<longitude>) point. Within the context of the Pub/Sub paradigm, a subscriber can subscribe to this same topic and receiving published data whenever a new new message is published.
Conversely, a subscriber with an interest in obtaining sensor values within a specific range from its own location, for example, to calculate the mean external temperature, should subscribe to */*/temperature/external since it does not exist a way to describe a space using a point. By doing so, it receives messages from all external temperature sensors and it must discard those values coming from out-of-range locations.

In Figure 3.2 we show a way in which it is possible to make message discarding transparent to the developers. We can insert an intermediary layer between the application and the Zenoh Subscriber API. This layer's purpose is to filter the messages received by

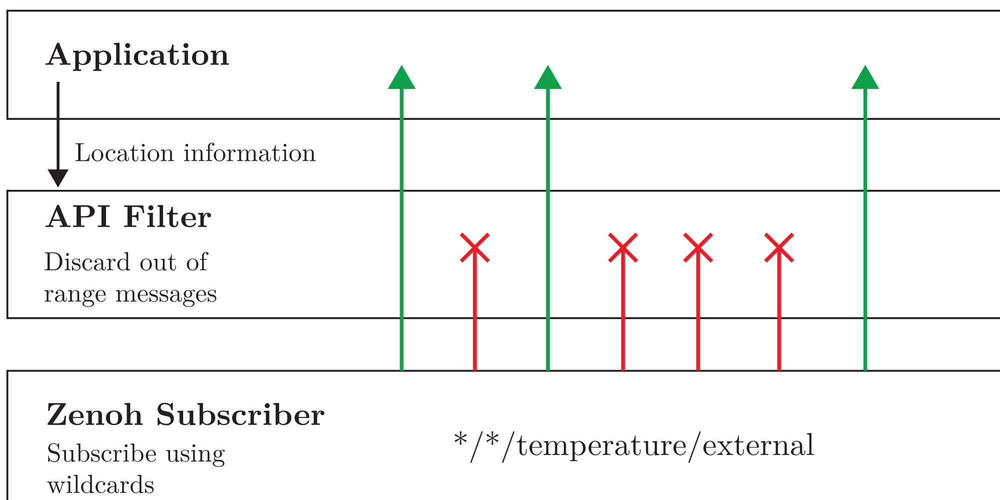


Figure 3.2: Location as a topic.

the subscriber and forward to the application layer only those that match the specified location criteria provided by the application layer.
Although this method does provide a partial solution to the transparency concern for developers, it still falls short of fully addressing the issue of unnecessary message deliveries. This limitation arises because it only allows for subscribing to either a single sensor or all sensors, with no intermediate option available.

3.3.2. Location in the Payload

Another solution that naturally emerges involves incorporating location information within the payload. This solution reduces the topic complexity compared to the one proposed above, but it increases the payload size.

In the same scenario proposed above, the topic temperature/external represents any external temperature sensor reading while the message payload not only contains the temperature reading but also the latitude and the longitude of the sensor location, resulting in a payload like:

```
{
    "latitude": 45.10,
    "longitude": 9.25,
    "temp_c": 12.25
}
```

This time, subscribers do not need a wildcard since the topic matches any location. Nonetheless, a filtering process remains essential for obtaining the desired readings. In

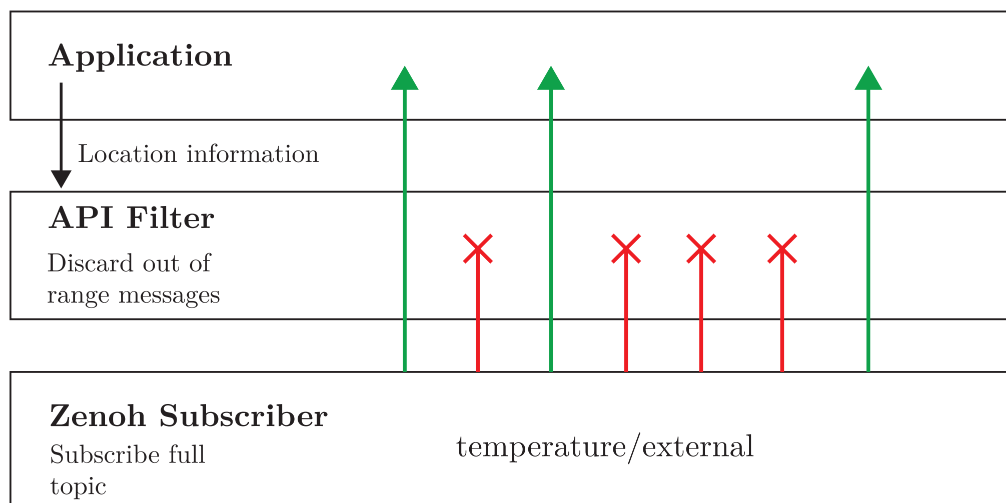


Figure 3.3: Location in the payload.

this case, rather than filtering based on the information within the topic, we rely on the data contained in the payload.

As illustrated in Figure 3.3, it remains feasible to insert an equivalent layer from the previous solution between the application and the Zenoh Subscriber API, maintaining transparency for the developer. While this approach does offer a partial remedy for developer transparency concerns, it does not fully resolve the problem of superfluous message deliveries, as we continue to discard messages at the receiver's side.

3.3.3. Routing

An alternative approach involves making adjustments to Zenoh's routing components.
This would involve incorporating location information into the message by introducing a new header, thereby making location information an inherent attribute of the message rather than including it in the payload or topic. At this stage, Zenoh routers need a logic adjustment to interpret the new header and utilize its data for identifying subscribers of a publisher. This identification should not solely rely on the topic; it should also involve a matching process that incorporates both the location header and the topic.
Zenoh maintains a distributed routing tree that represents the topic hierarchy, which accelerates the routing process when a subscriber is already identified. To implement this solution, we must also make adjustments to the data structure. This involves incorporating location information by introducing distributed tables that establish connections between subscribers based on both topic and location. This is necessary because directly adding location as an attribute to the tree node, which represents a topic level, is not feasible, given that a topic can be associated with multiple locations, and conversely, a location can be linked to multiple topics.
Nevertheless, it is important to emphasize that altering the routing mechanisms and the algorithms employed by Zenoh to maintain distributed information among its nodes carries the potential risk of compromising the performance assurances provided by the current state-of-the-art version. Although this solution would address both transparency and unnecessary message transmission issues, it is quite clear that it is not optimal.

3.3.4. Encoding Spatial Information

Zenoh provides three wildcards: *, **, and $*. According to Zenoh's documentation, the $ symbol serves as a special character that cannot be used as a regular letter within a topic level; instead, it is reserved for representing special expressions. This $ symbol enables the extension of Zenoh's key expression language with custom behaviors at the

topic level where it is utilized. For instance, when using the $* wildcard in a topic, like city_$*/temperature/external, Zenoh recognizes the wildcard and matches all expressions that begin with "city_".
With the $ symbol, Zenoh allows for the incorporation of new specialized keys into its key expression language. Specifically, we can create a new key that includes the necessary location information to introduce location-aware capabilities to Zenoh.

> With the $ symbol, we have the flexibility to define customized matching behaviors for the new key, going beyond simple string equality. This enables us to handle encoded spatial information and perform matches based on device location and areas of interest. We gain the ability to govern how Zenoh routes messages from publishers to subscribers by matching a unique topic in two different and independent ways.

In this solution, we can place the necessary information within the topic using a single level. By introducing a new matching function, we achieve our goal without the need to modify any internal libraries or data structures.

3.4. Solution

After a comprehensive exam of the three alternatives, we believe that introducing a new key expression represents the most optimal choice. This decision is underpinned by three primary reasons.
Firstly, by opting for the creation of a customized key expression, we are able to preserve the integrity of Zenoh's routing mechanisms as well as the performance of the original version. This approach avoids the need for modifications to the core infrastructure, keeping full compatibility with the original version of Zenoh protocol.
Secondly, it allows us to implement a highly customizable matching function. This encompasses not only location-based criteria but also takes into account the diverse attributes of devices, such as their roles as publishers, subscribers, or queryable entities. This level of matching granularity opens the door to a myriad of possibilities for refining the routing process to suit specific use cases, such as custom domain definition for spatial data and/or custom behavior for intersecting domains, such as a Zenoh queryable device belonging to different partially overlapping areas. Furthermore, the use of a custom key expression allows us to control message routing without the need to make changes to Zenoh's core routing library. Instead, we can simply choose target subscribers based on the encoded information within the topic. Thirdly, adopting the encoding approach, joined with a strategic API modification, guarantees a transparent experience for developers. By im-

plementing this matching mechanism, developers are alleviated from the responsibility of explicitly managing location data and matches based on complex criteria.

In practice, the concept involves crafting a unique key denoted by the symbol $ along with a distinctive prefix. To illustrate, consider the prefix $zla_ as our designated marker. Subsequently, the prefix is augmented with a character that identifies the encoding method, followed by a string containing the requisite location information. We introduce also the encoding methods char since we can implement multiple matching functions, targeting multiple encoding methods with different purposes.
To ensure a seamless experience for developers, we strategically position this newly formulated key at the outset of the topic. This incorporation is orchestrated through modifications to the Zenoh API. For instance, when a developer defines a topic like temperature/external, the actual outcome is $zla_x_xxxxxx/temperature/external, where $zla_x_xxxxxx represents the new key expression used to transport location data and pilot message routing. This transformation is concealed from the developer's view.
These special keys are readily recognized, parsed, and submitted to a matching process that differs from mere string equivalence. Rather, the matching process uses the encapsulated information within the key.

In Figure 3.4, we present an example illustrating the high-level behavior of a Zenoh network instance featuring location awareness. Within this network, we define two distinct zones, with each zone accommodating two publishers. These publishers are assigned specific topics, namely, int/temp/c and ext/temp/c, to define temperature values in °C, respectively, for indoor and outdoor conditions. In Zone 01, there is a subscriber who is subscribed to the int/temp/c topic and has a corresponding storage entity for that topic. While there may be other entities subscribing to the same topic, they do not receive messages from Zone 01 as they belong to different zones. In Figure 3.4, we also observe that two storage entities are deployed in a cloud environment, and they are not inherently associated with a specific physical zone. However, they can be configured to belong to one or more zones. Specifically, one storage entity subscribes to ext/temp/c and receives messages from both Zone 01 and Zone 02, while another entity subscribes to int/temp/c but is limited to Zone 02. With this setup, routers direct messages outside of zones only towards links that lead to the target entities matching both the topic and location, as we can see from the arrows depicted in the figure.
In Figure 3.4, we present device topics from the developer's perspective. Conversely, Figure 3.5 displays the same image, revealing the actual hidden topic crucial for zone division. To enhance clarity, we employ the identifiers *zone01* and *zone02* for the two zones, avoiding the use of visually complex encoded information.

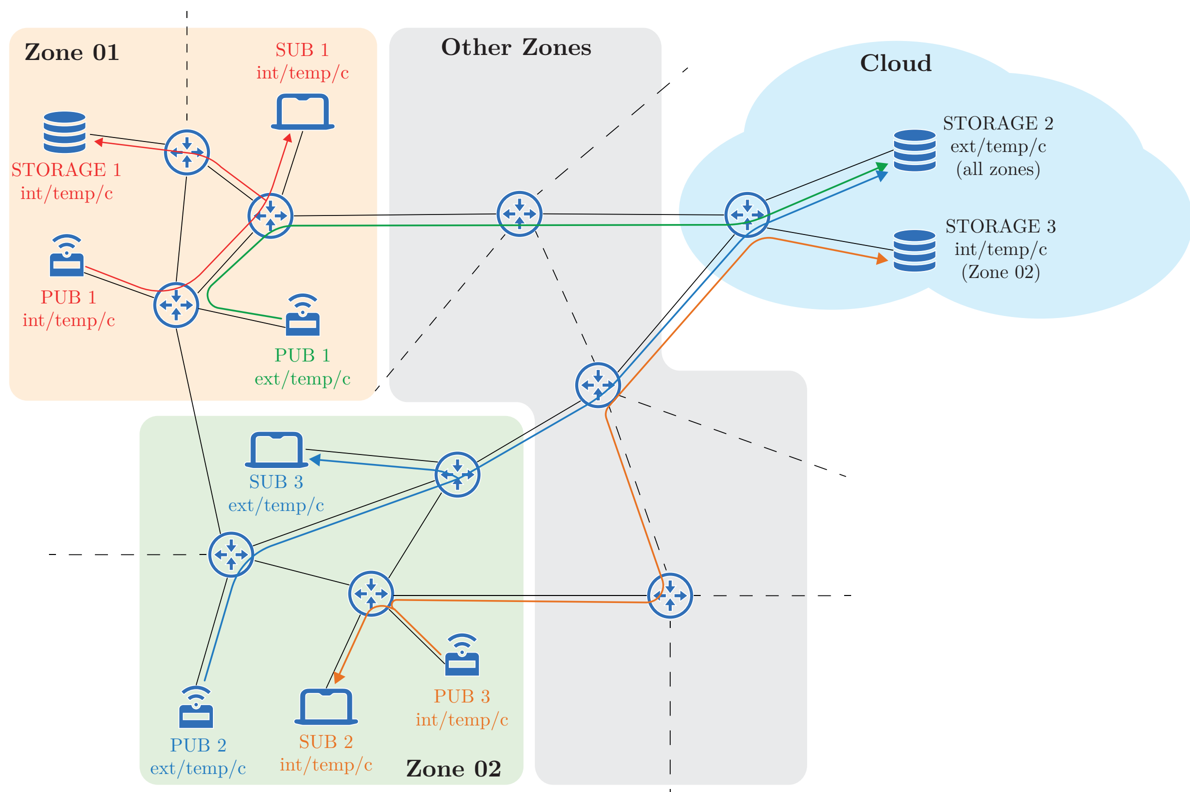


Figure 3.4: Zenoh location awareness example.

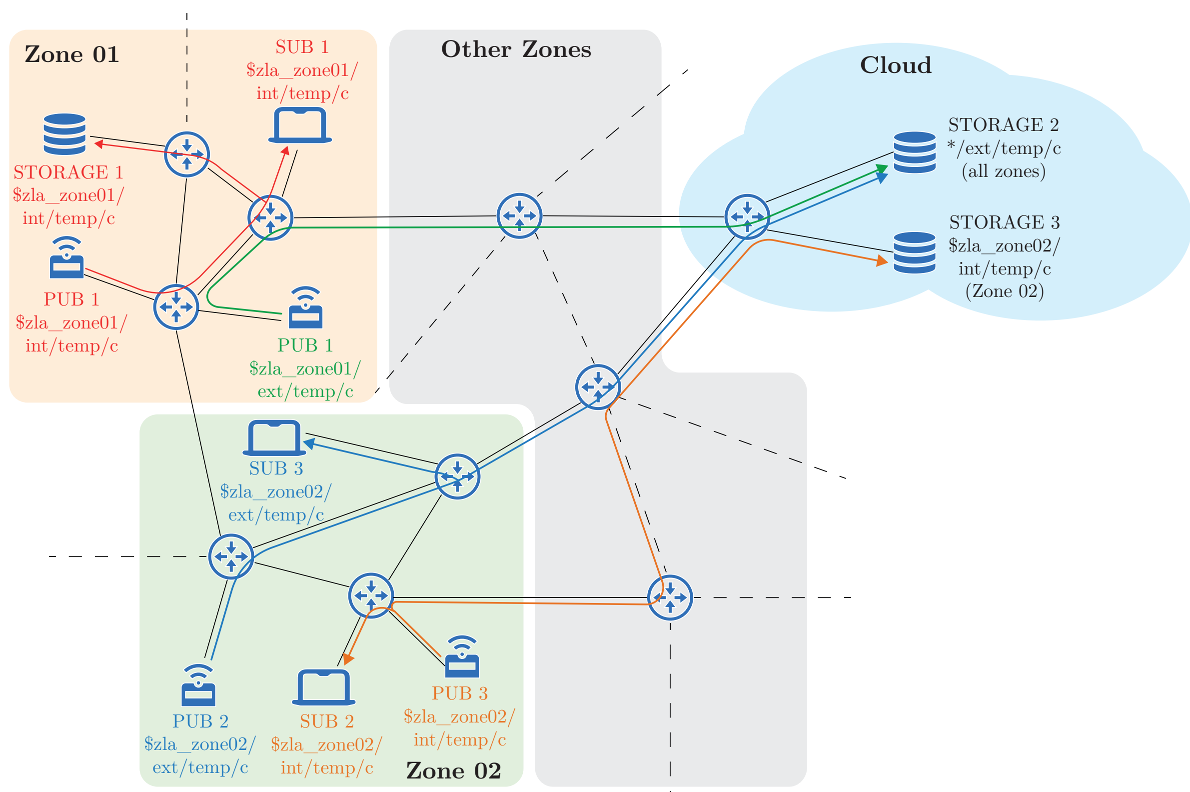


Figure 3.5: Zenoh location awareness example with explicit key.

4 | Design

In this chapter we focus on the regular version of Zenoh. We center our attention on the specific modules that require modifications to enable the implementation of a location-aware version. Furthermore, we also direct our attention towards the creation of modules that serve as support libraries and/or introduce new API functionality.
In Section 4.1 we introduce some concepts linked to our solution while in Section 4.2 we go deeper into our new Zenoh APIs and changes done in internals modules.

4.1. Overview

Before we delve into the modifications to be applied to Zenoh, it is appropriate to introduce some features that entities within the network can have. Specifically, we focus on the properties of a publisher and a subscriber in a location-aware context, how they can be regarded in space, and how they obtain geographic coordinates to determine their position. Primarily, Zenoh must work in both the location-aware version and the regular version. Consequently, in our design, we employ the term *whole network* to indicate an entity that sends/receives messages solely based on the topic, without any spatial information. Conversely, we employ the term *area* to denote an entity that exchanges messages while considering both the topic and location information.
In terms of location, we recognize two types of positional information:

- **Physical location:** It represents a device situated within a specific area, and its location data can be sourced from GPS or other positioning systems, with its value subject to change over time.
- **Virtual location:** It represents an entity or device that does not occupy a specific physical area, as seen in a cloud deployment of a data recorder in Figure 3.4 of the previous chapter. In the case of a virtually located entity, its location is determined by software configuration, meaning the entity's location is hardcoded and may not correspond to the real one.

In these scenarios, a publisher, as summarized in Table 4.1, always behaves in the same

	Physical	Virtual
Whole Network	A publisher, corresponding to a point in space, always includes its location when generating a message. It is the positional value that varies and determines the subscribers to which the message is directed to.	
Area		

Table 4.1: Publisher characteristics.

	Physical	Virtual
Whole Network	No changes w.r.t. regular version.	
Area	Subscribers communicate their real position at subscription time, and update it if necessary over time. They define the coverage area in terms of shape and value at configuration time.	Subscribers define both position, and coverage area definition at configuration time. There is no changes over time.

Table 4.2: Subscriber characteristics.

way, regardless of whether its location is physical or virtual. Furthermore, in spatial terms, we can define a published message as corresponding to a point in space, even though the same point may belong to different area definitions.

On the other hand, a subscriber can receive messages generated from different points. In fact, as summarized in Table 4.2, we must consider not only the area it belongs to, but also how we wish to define the area data should be received from. For this purpose, a subscriber located at a physical location can define its position, for example, via GPS, and be configured to cover a surface, such as a generic flat or circular surface. It can alternatively be set up in an ego-centric manner, delineating the space around itself and maintaining a consistent area definition relative to the subscriber's position, even when in motion.

In the case of a virtual location, the concept remains similar; the position is defined at the configuration level, such as the listening range. The main difference concerns the static nature of the position over time, since a non-real device cannot possess the ability of physical movement.

4.2. Internals Design

We explore the components and libraries that require modification for the implementation of location awareness in Zenoh. Our primary emphasis is on changes to component structures and user-accessible APIs, while the finer implementation details are reserved for Chapter 6.
Before introducing the new location-aware API, it is crucial to clarify the concept of a "topic" in Zenoh. Zenoh employs a key expression, abbreviated as "key_expr," to represent a topic and a "selector" to identify the topic in query operations.

4.2.1. Zenoh Location-aware API

The Zenoh APIs serve as the user interfaces, primarily tailored for developers. To introduce additional features, it is necessary to expand the existing interfaces, enabling the management of location-related information. Zenoh APIs are available in Rust, C, and Python, as well as through a REST interface that can be seamlessly integrated with either a router or a peer. In this section, we use Python to explain the API because it is more readily comprehensible but the result is the same for the other languages, while we discuss the REST API in Section 4.2.2.

In Section 4.1, we distinguish between physical and virtual locations depending on the type of device running the software. In practice, both types require obtaining coordinates regarding their position, whether through a GPS reading or a static value. Therefore, the new APIs necessitate a corresponding mechanism to facilitate this.
The optimal approach is to offer a function for registering a callback, granting developers the flexibility to implement their own custom function. The `declare_position_handler` function requires two parameters: a lifetime value that signifies the duration during which Zenoh considers the last location value as valid, and a handler, which serves as the callback function Zenoh invokes whenever it needs a new location value. Zenoh invokes the handler when there are no location values available or whenever it requires location data but the lifetime of the previous one is expired.

```python
def declare_position_handler(lifetime, handler)
```

Moreover, lifetimes may evolve over time. Therefore, we introduce an additional API that allows for adjusting its value without the need to modify the position handler:

```python
def set_position_lifetime(lifetime)
```

With the exception of the two new APIs, there is no need to introduce additional ones; it

is sufficient to modify the existing ones. Zenoh provides several APIs to developers, with the most important ones relating to the publisher, subscriber, get, and queryable devices all contained in Zenoh `Session` class.
Regarding a publisher, we add only a *precision* parameter. This parameter instructs Zenoh on whether to treat the publisher as a point (the default setting) or to apply some approximation that better fits a different type of encoding method. In the publishing API, there is no need for additional changes, as the position handler already provides the necessary location data.

```
def put(key: IntoKeyExpr,
        value: IntoValue,
        precision: Precision = Precision.POINT,
        encoding=None,
        priority: Priority = None,
        congestion_control: CongestionControl = None,
        sample_kind: SampleKind = None) -> Any

def declare_publisher(key: IntoKeyExpr,
        precision: Precision = Precision.POINT,
        priority: Priority = None,
        congestion_control: CongestionControl = None) -> Publisher
```

The `put` API transmits a message onto the Zenoh network, while the `declare_publisher` API registers a publisher device within Zenoh's network for subsequent publishing activities.

Conversely, subscribe, query, and get operations demand not just the position but also a target area, which can be defined through various methods. To accommodate this, we are incorporating three optional parameters into all four APIs: `location_shape`, an enumerated value for specifying the shape to utilize, `shape_data`, which carries the data required for defining the area, and `precision` parameter, which informs Zenoh whether to interpret the shape definition as punctual or approximate it. It's important to note that all the parameters are optional, and the default values disable the location-aware functionality.

```
def declare_subscriber(key: IntoKeyExpr,
        handler: IntoHandler[Sample, Any, Any],
        location_shape: Shape = Shape.NONE,
        shape_data: Dict = None,
```

```
            precision: Precision = Precision.POINT,
            reliability: Reliability = None) -> Subscriber

    def declare_pull_subscriber(key: IntoKeyExpr,
            handler: IntoHandler[Sample, Any, Any],
            location_shape: Shape = Shape.NONE,
            shape_data: Dict = None,
            precision: Precision = Precision.POINT,
            reliability: Reliability = None) -> PullSubscriber

    def get(selector: IntoSelector,
            handler: IntoHandler[Reply, Any, Receiver],
            location_shape: Shape = Shape.NONE,
            shape_data: Dict = None,
            precision: Precision = Precision.POINT,
            consolidation: QueryConsolidation = None,
            target: QueryTarget = None,
            value: IntoValue = None) -> Receiver

    def declare_queryable(key: IntoKeyExpr,
            handler: IntoHandler[Query, Any, Any],
            location_shape: Shape = Shape.NONE,
            shape_data: Dict = None,
            precision: Precision = Precision.POINT,
            complete: bool = None) -> Queryable
```

Zenoh also provides other APIs that we do not mention since they do not impact location awareness functions and, therefore, remain unchanged.

Example

Let us recreate the example illustrated in Figure 4.1, where we establish two subscribers and a publisher. The first subscriber is set to subscribe to a rectangular area indicated in blue, while the second subscriber subscribes to a circular area centered on the subscriber's position, marked in green. The two areas overlap, and the publisher generates messages within the intersection of the two. In this setup, both subscribers receive the published message.

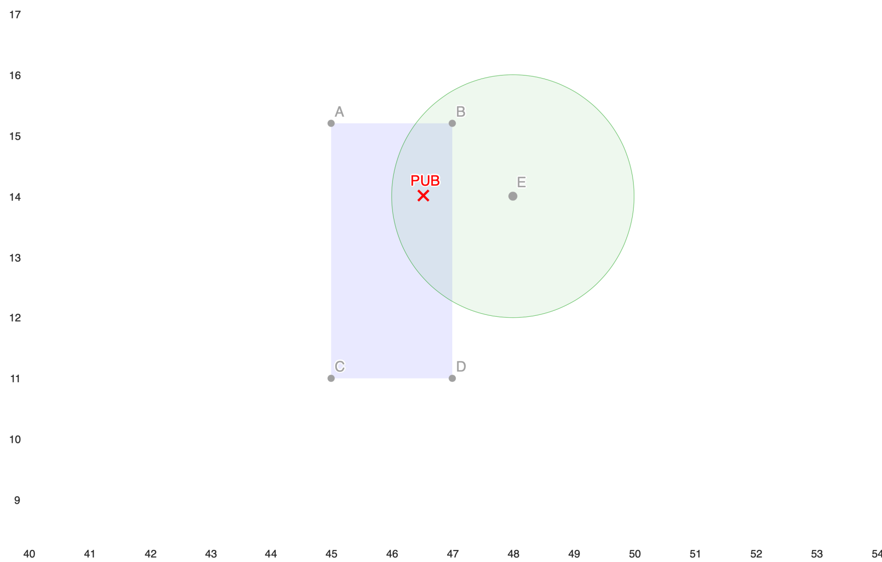


Figure 4.1: Zenoh location-aware intersection example.

We report the code snippets of the publisher and the two subscriber using the new location-aware version of Zenoh python API.

Publisher Code

```
import numpy as np
import zenoh
from zenoh import Reliability, Sample
import location_zenoh
from location_zenoh import Coordinates, Shape, Precision

def position_callback():
    return Coordinates(46.52275, 14.01002)

zenoh.init_logger()
#Open new Location-aware Zenoh session
session = location_zenoh.open(conf)
#Declare the handler with infinity position lifetime
session.declare_position_handler(numpy.infty, position_callback)
#Declare a publisher to 'temperature/external' topic
pub = session.declare_publisher("temperature/external")
while True: #Publish a value
    pub.put(json.dump({"temp_c": 12.5}))
```

Consider, for instance, line 14 where we introduce the `declare_position_handler` API. Within this handler, the position lifetime is set to `numpy.infty`, indicating that the specified position remains valid indefinitely. Subsequently, we specify the callback function's name from which the location data is retrieved. We declare the actual callback function on lines $7-8$, merely returning a Coordinate object representing a point in space. Lastly, on line 16, we declare the publisher. It is noteworthy that the provided topic lacks any information about the location, as the responsibility for handling all location data lies with the API rather than the developer.

In the next code snippet, we declare a subscriber using a rectangular area. On lines $17-20$, we show the new API signature. The topic lacks any location information since the position callback is registered some line above, as for the publisher. However, the `declare_subscriber` API includes the area description. We can observe the utilization of `GEOGRAPHIC_RANGE` to specify the rectangular area, defining the limits in terms of minimum and maximum latitude and longitude within the `shape_data` parameter.

Subscriber Code - Rectangular Area

```
1  #Same imports as publisher
2
3  def position_callback():
4      return Coordinates(45.774260, 12.608570)
5
6  #Print the message payload once received
7  def listener(sample: Sample):
8      print(sample.payload.decode("utf-8"))
9
10 zenoh.init_logger()
11 #Open new Location-aware Zenoh session
12 session = location_zenoh.open()
13 #Declare the handler with infinity position lifetime
14 session.declare_position_handler(numpy.infty, position_callback)
15 #Declare a subscriber for 'temperature/external' topic with a square area of
16 #interest defined in shape_data
17 subscriber = session.declare_subscriber("temperature/external",
18             listener, location_shape= Shape.GEOGRAPHIC_RANGE,
19             shape_data={ "min_lat" : 45.0, "min_long" : 11.0,
20             "max_lat": 47.0, "max_long": 15.2 })
```

Furthermore, we present an alternative area description in the subsequent code snippet, employing an ego-centric definition. On lines 17−19, we declare a subscriber tuned to a circular area using a `CIRCLE` shape. Notably, the circle radius is specified via the `shape_data` parameter, with the center of the circumference determined by the subscriber's own position. The subscriber position is obtained through the standard `position_callback` function. Diverging from the earlier snippets, this subscriber accommodates mobility. On line 14, we set the position lifetime to 180 seconds, indicating that the position is valid for only that duration once retrieved. Upon the expiration of the position lifetime, the API automatically requests a *position update* and adjusts the subscription to align with the new data if necessary.

Subscriber Code - Circular Area

```
1  #Same imports as publisher
2
3  def position_callback():
4      return Coordinates(48.0, 14.0)
5
6  #Print the message payload once received
7  def listener(sample: Sample):
8      print(sample.payload.decode("utf-8"))
9
10 zenoh.init_logger()
11 #Open new Location-aware Zenoh session
12 session = location_zenoh.open()
13 #Declare the handler with 3 minutes position lifetime
14 session.declare_position_handler(180, position_callback)
15 #Declare a subscriber for 'temperature/external' topic with a circular
16 # area of interest centered in subscriber location and having r=2
17 subscriber = session.declare_subscriber("temperature/external",
18             listener, location_shape= Shape.CIRCLE,
19             shape_data={ "r" : 2.0 })
```

For instance, modifying the publisher coordinates to (45.2, 13.0), the publisher remains within the rectangular area but no longer to the circular region. Consequently, only the rectangular area subscriber will receive the published messages. We have omitted the entire code, as the only alteration is in the 8th line of the publisher code, changing it from `return Coordinates(46.52275, 14.01002)` to `return Coordinates(45.2, 13.0)`.

4.2.2. REST API

The Zenoh REST API offers a convenient and web-friendly interface for interacting with Zenoh's data distribution and storage capabilities. It always uses the same url schema `https://host:rest_port/key_expr` and it offers 3 methods:

- **GET:** It binds to the `get` API and the key_expr represents the selector for the operation. If configured with Long-Lived SSE[1] it acts as `declare_subscriber`.
- **PUT:** It binds to the `put` operation, it also requires a body that is the message to publish.
- **DELETE:** It binds the `delete` function, which cancels a put operation.

To align the REST APIs with the same parameters used in Python and RUST, we choose to employ the query string, a standard HTTP method for including additional parameters in a URL.
Therefore, we introduce the query parameters `lat` and `long`. However, the `GET` method necessitates additional information for area definition, which leads us to introduce the `shape` parameter, responsible for shaping the area, along with a set of parameters tailored to the chosen shape, and the `precision` parameter to tell the REST engine if whether to approximate the location point or not.
As example, to invoke the `GET` method with a circular area of a 10-meter radius centered on the caller's position, we formulate the following request:

```
https://host:rest_port/key_expr?lat=10.0&long=20.0&shape=circle&r=10
```

[1]https://zenoh.io/docs/apis/rest/#long-lived-sse-get

5 | Embedding Location

In this chapter, we direct our attention towards the techniques we employ for encoding spatial information within the location key. In Section 5.1 we provide an overview of the basic method, involving the use of a base64-encoded string representing an object. Moving on to Section 5.2, we explore the Military Grid Reference System (MGRS), delving into its features and explaining its role as an encoding system. Lastly, in Section 5.3 we address Bloom Filters, highlighting their characteristics and discussing their utility in describing spatial information.

5.1. Base64

Base64 encoding is a technique that converts binary data into ASCII text format, making it suitable for various applications, including the representation of complex text as a concise string for easy transfer.
We leverage JSON as a standard format to organize information, ensuring compatibility across different programming languages and libraries. In practice, the data the developer configures through the Zenoh APIs, such as the position and shape of the area of interest, are inserted into a JSON object. For example, we describe a subscriber's circular area centered in $(15.5, 20.1)$ having a $10.3m$ radius with the object:

```
{
    "r": 10.3,
    "c": {
        "lat": 15.5,
        "long": 20.1
    }
}
```

and a publisher point simply as:

```
{
    "lat": 16.0,
```

```
        "long": 21,5
    }
```

Once we have the object describing our information of interest, we serialize it. JSON serialization is the operation that transforms the native data structure into a JSON-formatted string. With the JSON string, we can later deserialize it into the original data structure. For example, the circular area we describe above, once serialized, becomes:

{\"r\":10.3,\"c\":{\"lat\":15.5,\"long\":20.1}}

As a final step, we employ Base64 encoding. This final process converts the JSON string into a different, more compact string that represents the same object. It utilizes only characters that can be accommodated within a topic, as certain characters in the original JSON string might not be accepted. The Base64 encoded version of the example results in the string:

e1wiclwiOjEwLjMsXCJjXCI6e1wibGF0XCI6MTUuNSxcImxvbmdcIjoyMC4xfX0=

This representation enables us to encapsulate any desired shape in a single string. On the other hand, routers must know which object is represented to decode it correctly. In Chapter 6, we explain how our solution tells routers the correct object to decode both the Base64 string and the JSON string, which is represented in the topic key and not inside the Base64 encoding.

5.2. MGRS

We propose another encoding method for the location-aware version of Zenoh. The Military Grid Reference System, shortly MGRS, is a geocoordinate system used to specify locations on the Earth's surface. It is commonly employed in military applications, but it is also used in various civilian contexts. MGRS is based on the Universal Transverse Mercator (UTM) coordinate system [11].
Let us see how MGRS system work with the example[1] in Figure 5.1:

- **Grid zones:** The world is divided into a series of 6-degree longitudinal bands, each labeled with a letter from "C" to "X," excluding the letters "I" and "O". These bands are called grid zones.
- **100000-Meter Squares:** Each grid zone is further divided into 100000-meter squares. These squares are identified by a combination of numbers and letters, providing a unique reference within the grid zone.

[1] https://www.maptools.com/tutorials/mgrs/quick_guide

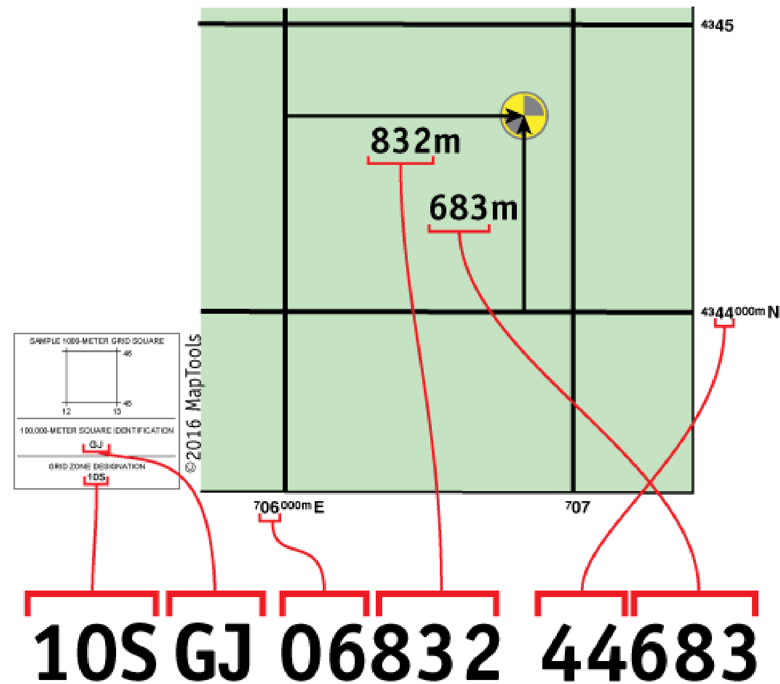


Figure 5.1: MGRS Example.

- **Easting and northing:** The location is specified using easting (horizontal) and northing (vertical) values within the 100000-meter square. Easting values are measured from the left edge, and northing values are measured from the bottom edge.

A complete MGRS coordinate includes the grid zone designation, the 100000-meter square identifier, and the easting and northing values. For example, a full MGRS coordinate might look like "10SGJ0683244683" where "10S" is the grid zone, "GJ" is the 100000-meter square, and "0683244683" represents the easting and northing values. This last value is even because the initial half, "06832," signifies the eastern value, while the latter half, "44683," denotes the northern value.

Furthermore, with MGRS, defining broader or more restricted areas is straightforward. By taking the MGRS Coordinate, it is sufficient to remove or add pairs of values to expand or reduce the area dimension. Let us consider an example:

- "10S GJ 06832 44683": $1m^2$ area
- "10S GJ 0683 4468": $10m^2$ area
- "10S GJ 068 446": $100m^2$ area
- "10S GJ 06 44": $1km^2$ area
- "10S GJ 0 4": $10km^2$ area

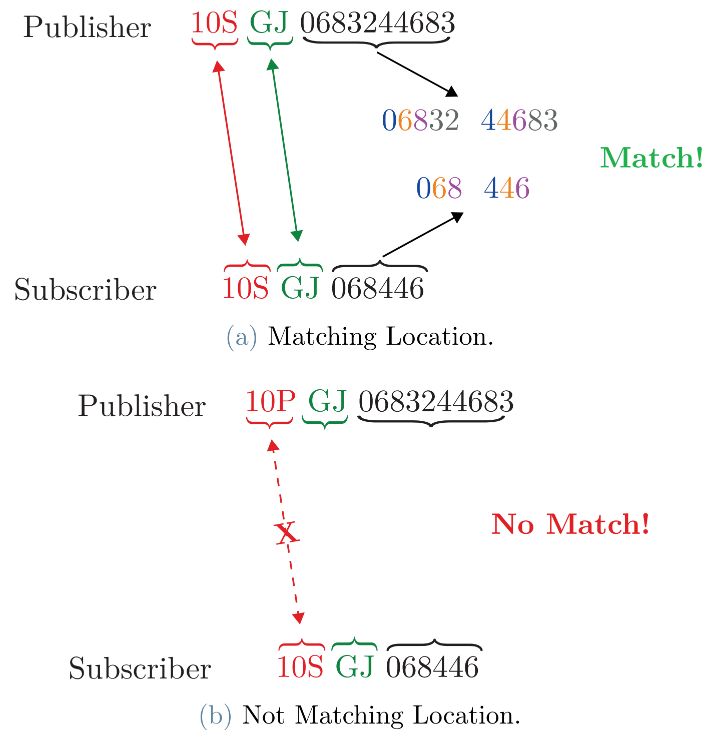
- "10S GJ": $100km^2$ area
- "10S": Grid Zone Junction (GZJ), $6° \times 8°$

Practically, we can use the MGRS coordinate system for managing location information in the location-aware version of Zenoh. While we sacrifice precision and flexibility in describing areas compared to Base64 encoding, we gain the advantage of a shorter string, limited to a maximum of 15 characters. Additionally, the MGRS system offers a simpler way to facilitate matching operations.

In implementing MGRS in the location-aware version of Zenoh, we opt to encode a publisher consistently with the maximum precision it provides, as we always define a publisher as a point. Conversely, we allow the developer to select the preferred precision for subscribers, enabling them to describe either a precise location or a broader area. In Figure 5.2 we show how a match between a publisher with a subscriber is computed, representing both a successful and unsuccessful match case. The steps are:

1. Verify the length of both the publisher and subscriber MGRS strings to determine if they are even or odd. If one string has an even length and the other has an odd length, it indicates a mismatch, and we conclude the process.
2. Compare the first subset of characters:
   - If the length is even, we check the first 4 characters
   - If the length is odd, we check the first 5 characters. For example in Figure 5.2a we can see how the two groups `10S` and `GJ` belong to both publisher and subscriber coordinates.

   If at least one character is different as shown in Figure 5.2b, the publisher does not match the subscriber and we finish. In the figure, we can see that the two grid zones `10S` and `10P` are different.
3. If the previous point does not exit, as the example in Figure 5.2a, we take the second subset representing easting and northing value. We also compute the shortest representation between the subscriber and publisher coordinates.
4. We split the two representations in half to get the easting and northing values separately and then we compare each character until:
   - All the values of the shortest string match the other, therefore, the two locations match.
   - One value is different, therefore, the two locations do not match.



(a) Matching Location.

(b) Not Matching Location.

Figure 5.2: MGRS matching example.

From Figure 5.2a, it is also evident that the last two digits of the easting and northing values in the publisher's coordinates are ignored since the subscriber has already found a match.

5.3. Bloom Filters

A Bloom filter is a space-efficient probabilistic data structure used to test whether an element is a member of a set. It works by using hash functions to map elements into a fixed-size array of bits. Due to its probabilistic nature, there is a chance of encountering false positives but it never yields false negatives. This property makes it suitable for use in a Pub/Sub system. In fact, subscribers can receive at most messages that do not belong to them, but they will never miss messages that belong to them, thus not impacting the system's functionality.
To work with Bloom filters we need to define some parameters:

- Expected number of elements of the domain n: the domain represents all the potential elements the set can include.
- Length of the bit-array m: the overall size of the fixed-size array of bits.
- Number of hash functions k: to enhance precision, we require the application of

more than one hash function to our domain elements.

- Probability of false positive p: it represents the probability that an element will be incorrectly identified as a member of a set even if it is not.

Since each parameter depends on the others, it is not possible to arbitrarily set them all. For example, as the number of elements in the domain increases and the desired probability of false positives decreases, the length of the bit array will also increase. In Equation 5.1, we see how to calculate the optimal length of the bit-array m given the number of elements in the domain n and the desired probability of false positives p.

$$m = -\frac{n \cdot ln(p)}{(ln(2))^2} \tag{5.1}$$

Also the ideal number of hash function k depends on the other parameters. In Equation 5.2 we report the formula to compute k given the length of the bit-array m and the number of elements in the domain n.

$$k = \left\lceil \frac{m}{n} \cdot ln(2) \right\rceil \tag{5.2}$$

Once the number of elements in the domain (n) and the desired probability of false positives (p) are configured, the remaining parameters are computed, allowing us to construct the actual filter.

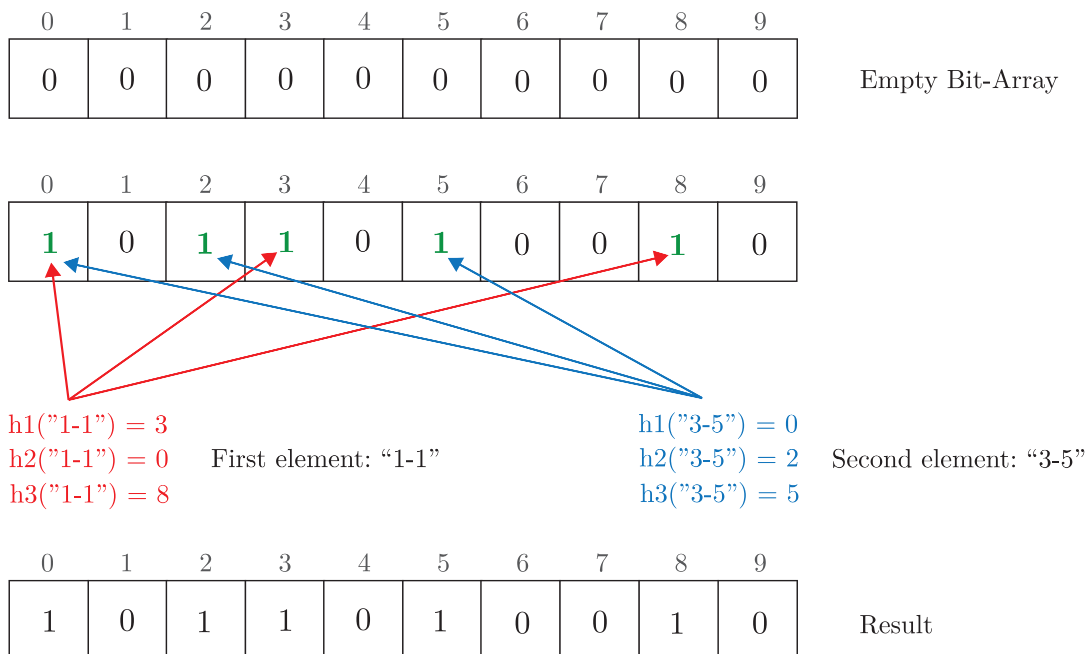


Figure 5.3: Insertion on bit-array.

In Figure 5.3, we provide a visual representation of a Bloom Filter with a 10-bit length array and 3 hash functions. Initially, the bit-array is initialized with all values set to 0. Subsequently, for every element within the domain that we intend to insert into the filter, we calculate the corresponding hash functions. The outcome of each hash function determines the index in the bit-array to which the element is mapped, and the corresponding bit is set to 1. If a bit at a computed index is already set to 1, it remains unchanged. This process is repeated for each hash function and element of the domain to add to the set.

Once we have the resulting bit-array with both 0s and 1s, representing the Bloom Filter, we can see how the querying process works.

We show, using the same example, the process in Figure 5.4. To verify whether an element belongs to the set described by the filter, it is enough to apply the same hash function to the element. If the bits at the positions indicated by the resulting indexes from the hash function are set to 1, the element belongs to the set; otherwise, it does not. During element queries, false positives may arise. Consider, for instance, the element `"1-4"` as depicted in Figure 5.4. The outcomes of the hash functions yield three indexes corresponding to all 1 bits. In this scenario, there may not be an element directly mapped to those indexes. However, the membership determination relies on the combination of various bits set by different elements.

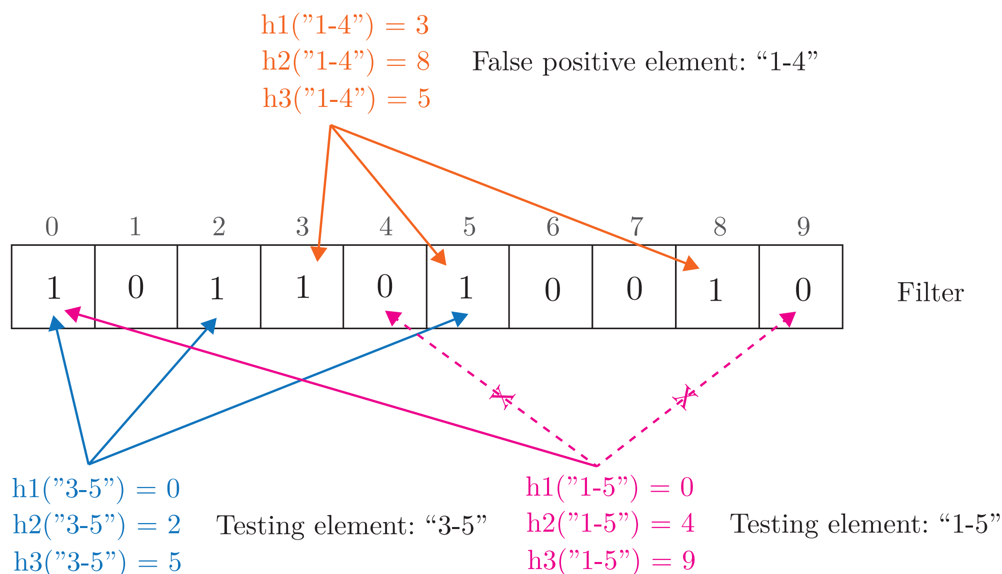


Figure 5.4: Query on Bloom filter.

5	0-5	1-5	2-5	3-5	4-5	5-5	6-5	7-5	8-5	9-5
4	0-4	1-4	2-4	3-4	4-4	5-4	6-4	7-4	8-4	9-4
3	0-3	1-3	2-3	3-3	4-3	5-3	6-3	7-3	8-3	9-3
2	0-2	1-2	2-2	3-2	4-2	5-2	6-2	7-2	8-2	9-2
1	0-1	1-1	2-1	3-1	4-1	5-1	6-1	7-1	8-1	9-1
0	0-0	1-0	2-0	3-0	4-0	5-0	6-0	7-0	8-0	9-0
	0	1	2	3	4	5	6	7	8	9

Figure 5.5: Bloom filter domain definition.

To employ Bloom filters as a technique for encoding spatial data, we require an alternative method for conducting queries. The querying process occurs on Zenoh routers based on the encoded information in the message topic while data definition, and so the encoding process, is up to the clients.

Firstly, it is essential to establish the domain, and we choose to utilize an arbitrary grid defined by the developer. In Figure 5.5, we provide an example of a domain grid with dimensions 10×6, resulting in a total of 60 domain elements. Subsequently, we construct the element identifier as the combination of the row and column indices. These unique identifiers are the elements that we insert into the filter.

Once we define the domain across all the clients, we can map our publishers and subscribers within the domain. We map the publisher to the grid unit in which the publishing point belongs. A publisher always corresponds to a single domain element since, as previously mentioned, we treat it as a point in space. Instead, we map subscribers as *the minimum subset of the domain elements that entirely cover the area the subscriber defines.*

In Figure 5.6 we see a visual example. The publisher, which is the red cross, corresponds to the $\{7-4\}$ element in the Bloom Filter domain. The subscriber, which is more extensive, covering a larger area than a single unit, is mapped to the subset $\{4-2, 5-2, 4-3, 5-3, 4-4, 5-4\}$, which is the minimum number of domain units to

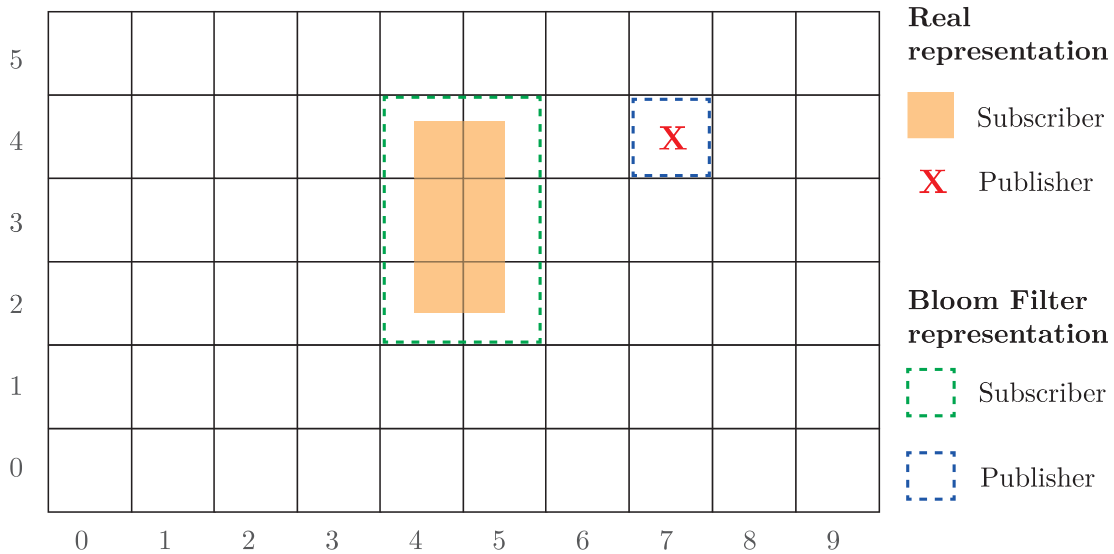


Figure 5.6: Example of Bloom filter domain representation.

completely contain the real subscriber definition.

Now, we can construct the filter. We opted to construct it with a length equal to the nearest integer multiple of 32 greater than the calculated ideal number of bits using Equation 5.1. We make this decision to achieve a representation that is easy to manage once encoded. The developer chooses the false probability value p, and we compute the number of hash functions k using Equation 5.2. The Bloom Filter must include the domain elements that characterize the subscriber, as we need to test the membership of published messages to a subscriber, not the other way around. The subscriber's location key contains the filter, the publisher key should contain the element to test and both these information within the hash functions definition should be distributed to routers for membership testing.

Our goal is to minimize routing overhead, and including all this information within the topic location key leads to a huge message. Therefore, we choose to conduct membership testing in a slightly different manner. Referring to Figure 5.7, we match a publisher with subscribers on the router side by executing the *bitwise* operation, a lightweight operation involving a bit-a-bit AND. In simpler terms, we verify the match between the 1 bits of the subscriber and those of the publisher. To accomplish this, we generate a filter-like representation for the publisher, which serves as the element for membership testing. The publisher's filter is identical to that of a subscriber, with the only distinction being the

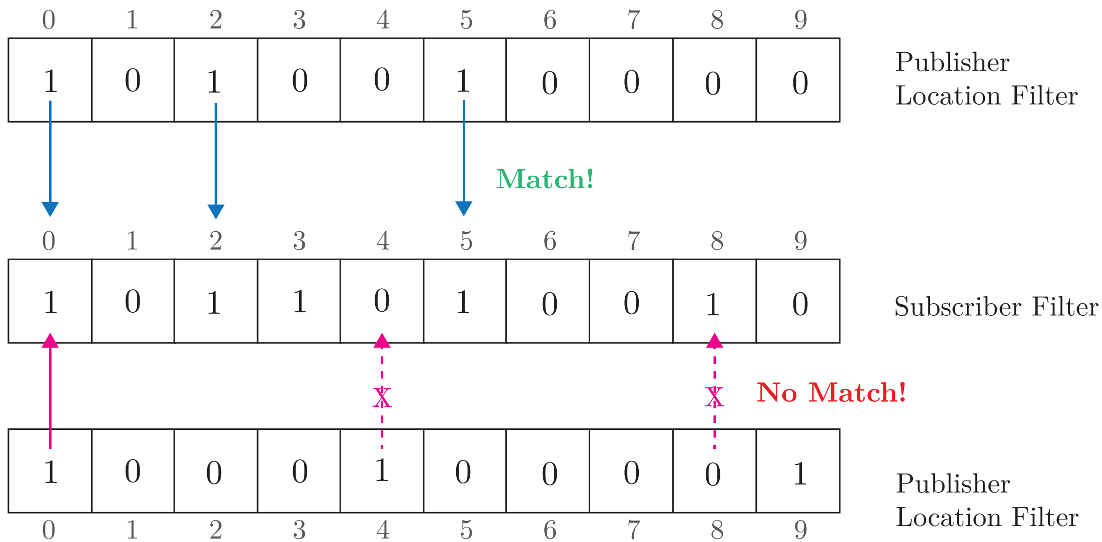


Figure 5.7: Query on Bloom filter in Zenoh router.

position of the bit set to 1, which exclusively represents the unique domain element mapping the publisher.

Once we possess the subscriber filter and the filter-like representation of the publisher, we can execute the match using the bitwise operator. The forthcoming example, illustrating how we carry out the match, utilizes an 8-bit representation instead of 32: consider the integer 90 with a binary representation of 01011010, representing our subscriber filter. To represent a publisher, we can use the number 18, with a binary representation of 00010010. The bitwise operation & works as follows:

```
01011010 & 00010010 = 00010010
```

in decimal representation

```
90 & 18 = 18
```

Finally, as transmitting either the binary or integer representation may lead to a long encoding, we choose to utilize the hexadecimal representation, which is shorter than the decimal one but remains easy to manage. In practice, to integrate the Bloom Filter into the location key, we follow these steps on the client side:

1. Obtain the binary representation of the filter.

2. Divide the filter into sub-arrays of 32 bits in length.
3. Represent each sub-array in base 16 for a more concise representation.
4. Combine the base 16 values into a single string, constituting our encoded location within the location key.

On the router side, we simply perform the bitwise operation on each hexadecimal value between the subscriber filter and the publisher filter. A match is achieved if and only if

```
<subscriber hex> & <publisher hex> = <publisher hex>
```

for each hexadecimal pair of values.

6 | Implementation Highlights

In this chapter, we focus on the implementation of the location-aware version of Zenoh. We deal with the important choices we make during the implementation starting from the new key expression, the wrapper we build around the original API, to the new module in the core of Zenoh. In Section 6.1 we provide an overview of the internal structure of Zenoh and the new library we introduce. In Section 6.2 we introduce the new wildcard and the relative matching in the router component, and in Section 6.3 we present the new API for Zenoh with location-awareness along with the management of mobility cases.

6.1. Zenoh Internals

Zenoh Internals represent the core of Zenoh, which is the set of libraries and modules necessary for the operation of the protocol itself. In Figure 6.1, we represent the main classes that compose it, highlighting the modules we edit and add to implement the location-aware functionality. As our solution relies on expanding the primary expression language by introducing the location key, adjustments need to be made in the `keyexpr` module. Within this module, we pinpoint the stage where Zenoh identifies the $ symbol and extends the recognition process to accommodate not only the `$*` wildcard but also the newly introduced `$zla_` prefix.
To enhance code decoupling, rather than incorporating new matching functions directly into the key expression module, we opt to create a separate library named `location-zenoh`. This library exposes a boolean function responsible for handling the matching of the location key and representing the entry point. Within the `location-zenoh` library, we incorporate the functionality discussed in Chapter 5 to manage encoding techniques. Furthermore, the library provides various recognizers to determine the encoding technique a message uses and it includes the logic to parse the entire key based on the recognized technique, as we explain in the subsequent section.
A component of Zenoh internals is the `rest` plugin library, which provides the encoder and decoder for interfacing with the Zenoh network via a JSON REST interface. We modify the plugin to incorporate location-aware parameters, offering identical functionalities to

those of the API, as detailed in Section 4.2.1, and extend these features to the REST interface. The REST plugin imports and utilizes the `location-zenoh` library.

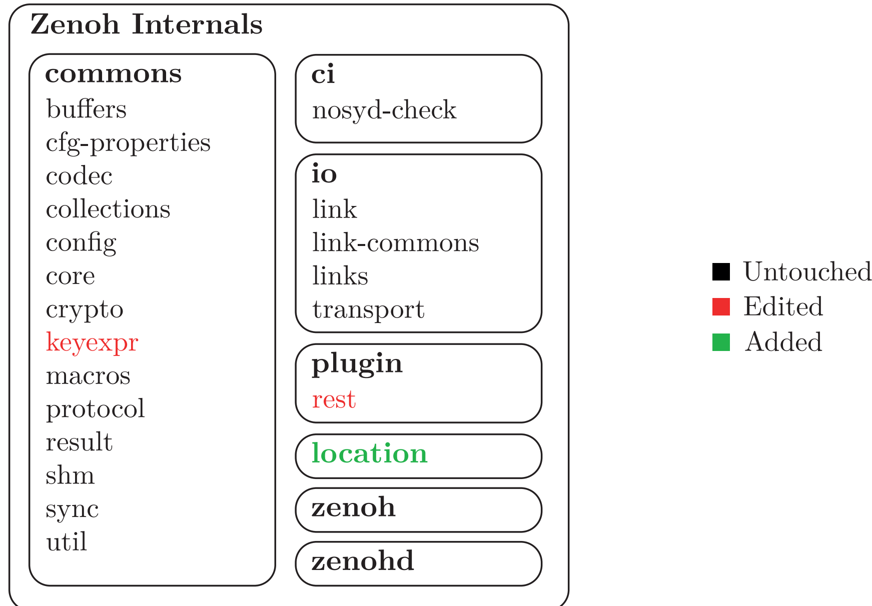


Figure 6.1: Zenoh internals structure.

6.2. Location Key

In previous chapters, we introduced the new wildcard to extend the Zenoh key expression language, forming the location key. We also discussed encoding methods that rely on the location key. Let us now delve into the specifics of this new wildcard, examining its appearance and the information it contains.
The specific structure of the key depends on different aspects:

- **Device type:** The key undergoes changes based on whether the device functions as a publisher, a subscriber, a queryable entity, or a pull subscriber.
- **Encoding method:** A crucial aspect of the key is the flag indicating the encoding method the developer chooses for the respective implementation.
- **Shape of the area:** In Base64 encoding, the shape also plays a role. This is because it informs routers on how to decode the JSON object.

Before delving into the explanation of all possible types of key structures, we present in Figure 6.2 the general format of the location key. An initial fixed prefix `$zla_` informs Zenoh that the wildcard is the location key and should be handled differently w.r.t. a normal topic level. Following the prefix is a flags area, comprising certain characters

Figure 6.2: General structure of the Location Key.

separated by the underscore symbol, which describes the device type, encoding method, and/or shape when necessary. Lastly, the last segment of the key carries the actual encoding of the location information, as discussed in Chapter 5.

6.2.1. Base64 Key

The Base64 encoding method is a precise approach in location definition. Consequently, the location key must convey information about both the device type and the shape encoded in the key. We use the string `YYYYYY` as a placeholder to indicate the location data encoding.
For a publisher, representing a point, the generated key takes the form:

```
$zla_p_YYYYYY
```

Here, the letter `p` in the flags area denotes the device type as a publisher.
The location key associated with other device types, which describe an extended area, is somewhat more intricate. Let us introduce a variable `x` to represent the shape. This variable can take the value 1 to describe a rectangular range using the minimum and maximum latitude and longitude or the value 2 to depict a circular area centered on the device position. In this case, for each device type, we define the following keys based on the value of `x`:

- Subscriber: `$zla_s_x_YYYYYY`
- Pull Subscriber: `$zla_ps_x_YYYYYY`
- Queryable: `$zla_q_x_YYYYYY`
- Get Operation: `$zla_g_x_YYYYYY`

On the router side, the matching process is relatively straightforward. Upon recognizing the `$zla_` prefix in the topic, the key is directed to the location-aware matching function. This function serves as the entry point for the `location-zenoh` library, which we integrate into Zenoh Internals, while the other levels of the topic continue to be matched through Zenoh's normal flow. The location-aware matching function operates as a boolean function, offering only two possible outcomes:

- Match: `true`
- Mismatch: `false`

The function initiates by removing the `$zla_` prefix from the key. After identifying one of the potential values in the device flag, it then splits the remaining part into segments using the underscore as a separator. Following this, it examines the first flag to identify the device submitting the request.
Given that potential matches can occur between a publisher and a subscriber/pull subscriber or between a get operation and a queryable, we categorize these devices into two groups. Subsequently, the function forwards the shape flag and the encoded data to the decoder module, which is responsible for returning the JSON object. It accomplishes this by first decoding the base64 string and then converting the stringified JSON into the corresponding object based on the shape flag. In the case of a publisher, only the base64 string is required since the resulting object is always a point.
Ultimately, the actual match is conducted. In the case of a publisher with a subscriber, it verifies whether the coordinates of the points belong to the subscriber's area, including the perimeter. The match between a get operation and a queryable checks if at least one point exists between the two defined areas. At conclusion, the location-aware function conveys the result back to the Zenoh flow, which compares the location key match result with the one the other topic levels provide. The match between two devices using the Base64 encoding technique is performed in $O(n + m)$ where n is the complexity of the base64 decoder and m is the one of the JSON parser.

6.2.2. MGRS Key

The MGRS encoding method does not differentiate between publishers and subscribers. An MGRS coordinate consistently follows the same format. Matching occurs when a wider area intersects another one, it relies on the concept of containment as explained in Section 5.2.
The MGRS location key comprises the prefix, the flag defining the encoding, and the coordinates, resulting in the format:

```
$zla_m_YYYYYY
```

where, as usual, `YYYYYY` is the coordinate placeholder.

On the router side, the matching process operates similarly to the Base64 method. Initially, Zenoh identifies the location key through the `$zla_` prefix. It then directs the location key to the `location-zenoh` library using the location-aware matching function,

which removes the prefix. The function subsequently recognizes the letter `m`, which doesn't signify any device type but indicates MGRS encoding. The function splits the key using the underscore as a separator and forwards the MGRS coordinates to the respective module. The MGRS matching function is not so straightforward. In fact, it is necessary to proceed through levels within the coordinates, starting from the Grid Zones up to the easting and northing values, if present. Let us examine the function we offer in Rust, as Zenoh Internals are written in the Rust language.

```
1  pub fn mgrs_match(d1: &[u8], d2: &[u8]) -> bool {
2      if d1[0] != d2[0] || d1[1] != d2[1] { return false; }
3      let mut next_i = 2;
4      if d1.len()%2!=0 && d2.len()%2!=0 {
5          if d1[2] != d2[2] { return false; }
6          next_i = 3;
7      }
8      if d1.len() == next_i || d2.len() == next_i { return true; }
9      if d1[next_i] != d2[next_i] || d1[next_i+1] != d2[next_i+1] {
10         return false;
11     }
12     next_i = next_i + 2;
13     if d1.len() == next_i || d2.len() == next_i { return true; }
14     let d1_then: &[u8] = &d1[next_i..];
15     let d2_then: &[u8] = &d2[next_i..];
16     let len1 = d1_then.len();
17     let len2 = d2_then.len();
18     let mut for_len = len1/2;
19     if len1 >= len2 {
20         for_len = len2/2;
21     }
22     for i in 0..=for_len-1 {
23         if d1_then[i] != d2_then[i] ||
24             d1_then[i+(len1/2)] != d2_then[i+(len2/2)] {
25             return false;
26         }
27     }
28     return true;
29 }
```

The function receives two MGRS coordinates, `d1` and `d2`, as input parameters, regardless of the device type. The crucial step in the function involves verifying the alignment of various components:

- Line `2`: the function checks the first two characters that define the grid zone. Since an MGRS coordinate must contain at least the grid zone, its existence is guaranteed. In cases where the grid zone is three characters long, making the full MGRS length odd, the function performs a check on line `4`. If true, it verifies that the third character pairs align, advancing the cursor (`next_i`) one step ahead. If one of the coordinates only contains the grid zone, they match, and the function returns true; otherwise, it proceeds.
- Line `9`: the function checks the two symbols identifying the 1000000-meter square precision. Similar to the previous check, if these symbols differ, the coordinates do not match. If they match, and the 100000-meter square precision is the maximum precision for either coordinate, the location matches, and the function returns true. Otherwise, the function continues.
- After removing the already verified parts, the function determines the minimum length between the two coordinates, as this is needed to cycle over the remaining values. On lines `22-27`, both the easting and northing values are checked. If, after completing all the values of the shortest coordinate, no mismatches occur, the function concludes that there is a location match.

In conclusion, the function communicates the result back to the Zenoh flow, which compares the location key match result with those provided by other topic levels, completing the process. The MGRS matching algorithm examines a maximum of 15 characters, having a complexity of $O(k)$ with k as a constant. Essentially, we can execute the match in constant time, denoted as $O(1)$.

6.2.3. Bloom Filter Key

Utilizing the Bloom Filter as an encoding method for transferring spatial information within the Zenoh topic eliminates the need to differentiate between device types. Instead, it necessitates distinguishing between the filter and the element to determine whether the element belongs to the set. Referring to the encoding process outlined in Section 5.3, which employs a filter-like representation, it becomes crucial to identify the filter and the element. We achieve this by utilizing the flag area of the location key, where the filter is marked with an `f` and the element with an `e`. The location keys take the following form:

```
$zla_b_f_YYYYYY
$zla_b_e_YYYYYY
```

As usual, the placeholder `YYYYYY` represents the encoded location data.

On the router side, the matching process mirrors the other methods. Initially, Zenoh identifies the location key by recognizing the `$zla_` prefix. Subsequently, it forwards the location key to the `location-zenoh` library, utilizing the location-aware matching function, which eliminates the prefix. The function then detects the presence of the letter `b`, signifying Bloom Filter encoding. Following this, the function divides the remaining part using underscore characters, forwarding both the encoded location and the `f`/`e` flag to the matching module.
The Bloom Filter encoding consists of a sequence of hexadecimal values separated by dots, with each representing 32 bits of the filter. It appears as follows:

```
1f34.75a3b.0.34.f123a
```

The Bloom filter matching function, for both the filter and the element, divides the encoding using dots, resulting in two arrays with the values' representation. Since the two arrays have the same length, the function operates as follows for each index `i` of the array:

1. Retrieve the element at position `i` from both the filter and the element array.
2. Convert both hexadecimal strings into a `u32`, representing an unsigned 32-bit integer value.
3. Apply the bitwise operation and compare the result with the element value. A value matches if and only if:

   ```
   <filter> & <element> = <element>
   ```

4. If the result is a match, the function proceeds to the next index; otherwise, it returns false and exits.

A match occurs only if all parts of the element match the corresponding parts in the filter. In conclusion, the function communicates the result back to the Zenoh flow, which compares the location key match result with those provided by other topic levels, completing the process. A Bloom filter match is performed in $O(k)$, where k represents the number of 32-bit array representations derived from the length m of the Bloom filter bit-array.

6.3. API Wrapper

To offer Zenoh's location-aware API to developers, we choose not to develop a totally new API library or modify the existing one. Instead, we opt to create a wrapper around the current Zenoh API. This decision is made to ensure complete compatibility with the existing Zenoh version and to provide developers the same syntax as the current API.

Currently, Zenoh offers its API for different programming languages such as Python, RUST, C, and so on. In our implementation, we choose Python as the main language due to its greater readability. Additionally, we modify the APIs in the RUST language as a demonstration that even though the languages are different, the logic with which the wrapper is created remains unchanged.
To initialize the Zenoh API, the process involves opening a new Zenoh Session through the `open` static method within the `zenoh` module. Typically, this method returns an instance of the `Session` class. In constructing our wrapper, we initially create the `LocationSession` class. This class, in its initializer, invokes the constructor of Zenoh's `Session` class. The outcome of the Session's initialization is stored in a private variable named `_zsession`, which proves essential for communicating with Zenoh. We introduce a new `open` method within the `location-zenoh` module. Unlike the standard `zenoh` module, this modified `open` method returns an instance of the `LocationSession` instead of the regular `Session`. This design allows developers to choose between utilizing the `location-zenoh` module for opening a location-aware version of Zenoh session or the `zenoh` module for the standard one.
The `LocationSession` class incorporates the new API along with modified signatures of the existing methods, as detailed in Section 4.2.1. Before describing the modified methods we introduce some detail about the new API we introduce in the location-aware version of Zenoh.
The `declare_position_handler(lifetime, handler)` function saves the lifetime and the handler callback respectively inside the private variables `_position_lifetime` and `_position_callback`. While the `_position_callback` is useful to read location data from the developer side, the `_position_lifetime` parameter plays a central role in position updates. Every time an API requires position data, a private function `_request_position()` is called.

```
1 def _request_position(self):
2     if self._position_callback is not None:
3         if self._last_p_update is None or
4         (time() - self._last_p_update) > self._position_lifetime:
```

```
5                 position = self._position_callback()
6                 self._position = position.to_dict()
7                 self._last_p_update = time()
8         else:
9             self._position = None
```

The function initially verifies the existence of the _position_callback; if it exists, it then checks for the presence of location data and ensures the validity of the lifetime. If the lifetime has expired, the function necessitates a new position, which is subsequently stored in the private _position variable. Additionally, it updates the timestamp of the last update. If the position is valid, the function simply exits.
We consistently adjust the signatures for all Zenoh API functions, internally we construct them to operate in the same manner. Let us take, for example, the put function:

```
1  def put(self, key: IntoKeyExpr, value: IntoValue,
2              precision: Precision = Precision.POINT,
3              encoding=None, priority: Priority = None,
4              congestion_control: CongestionControl = None,
5              sample_kind: SampleKind = None) -> Any:
6
7          self._request_position()
8          location_object = {"type": ZDevice.PUBLISHER,
9                             "data": self._position,
10                            "precision": precision}
11         #Other data in location_object if necessary
12         key = compute_new_key(location_object, key)
13
14         return self._zsession.put(key, value, encoding, priority,
15                                   congestion_control, sample_kind)
```

The function begins by calling the _request_position() function to update the position data within the class, if necessary. Subsequently, it creates a location object that includes the device type using the ZDevice enum. The location object is then augmented with the position data and the chosen encoding precision, representing the selected encoding technique among POINT (default), MGRS, or BLOOM_FILTER. Additional parameters may also be included in the location-object, such as the shape describing the area in the case of a subscriber.
Following this, the compute_new_key function takes on the responsibility of appending the location key to the topic based on the information within the location_object.

This function is a part of the location-zenoh library, encompassing all the methods for encoding and decoding location data, as well as constructing the location key, as elaborated in Section 6.2. Finally, the result of the original Zenoh Session API function is returned, utilizing the newly generated topic with the location key instead of the simple topic.
This approach ensures full compatibility with the original Zenoh, with the wrapper being specifically tasked with managing location-aware functionality, particularly the inclusion of the location key.

While adhering to this pattern in all APIs that necessitate the location key, the subscriber introduces an additional requirement. Specifically, in the scenario of device mobility, where a device changes its position over time, it is necessary to consistently update the location key with fresh position data when the lifetime expires. Unlike publishers, which request the position each time a new message is published, or subscribers with a fixed area definition that does not utilize their position for the definition, subscribers using an ego-centric subscription, like the circular area centered on the subscriber's position, need to update their subscription over time. To address this situation, we introduce a MobilitySubscriber class instantiated only when the subscriber definition specifies a non-infinite position lifetime.
The MobilitySubscriber is a class containing the parameters that define the subscriber, including encoding techniques, topic, shape, area definition, and more, along with a variable for Zenoh's Session subscriber class. This novel subscriber type initiates a thread responsible for invoking the _request_position() function when the lifetime expires. If the new position differs from the previous one, it unsubscribes the old subscriber and establishes a new subscription with the updated position data. Given that the actual subscriber is an internal component of the MobilitySubscriber class, this entire process remains transparent to the developer, relieving them from concerns about mobility management.
As an illustration, the declare_subscriber function, prior to subscribing using Zenoh's Session function, examines the lifetime. If the lifetime is non-infinite, it initiates the thread and declares the MobilitySubscriber.

```
1    # declare subscriber function as normal
2   if self._position_lifetime != np.infty and
3       self._position_lifetime is not None:
4           daemon = Thread(target=self._session_background_handler, args=(),
5                                           daemon=True,
6                                           name='background_position_update')
```

```
7         daemon.start()
8         mob_subscriber = MobilitySubscriber(self._zsession, subscriber,
9                                             original_key, <all params>,
10                                            sub_type=ZDevice.SUBSCRIBER)
11        self._mobility_subscriber = mob_subscriber
12  # return the mobility subscriber or normal subscriber device
```

The responsibility of updating the position in the background when the lifetime expires lies with the `_session_background_handler` function within the `LocationSession` class:

```
1  def _session_background_handler(self):
2      while True:
3          sleep(self._position_lifetime)
4          current_p = self._position
5          self._request_position()
6          if current_p['mgrs'] is not None:
7              if current_p['mgrs'] != self._position['mgrs']:
8                 self._mobility_subscriber.update_position(self._position)
9          else:
10             if current_p['lat'] != self._position['lat'] or
11                current_p['long'] != self._position['long']:
12                self._mobility_subscriber.update_position(self._position)
13
```

In the background thread, this function sleeps for the entire lifetime, and upon waking up, it requests a new position. Subsequently, it updates the mobility subscriber based on the coordinate type if the new position differs. Finally the `update_position` function of the `MobilitySubscriber` class uses the Zenoh Session, saved as `_inner` variable, to first undeclare the subscriber, then the function computes the new location key, and finally re-declares the publisher with the updated data:

```
1  def update_position(self, position):
2      self._inner.undeclare()
3      key = compute_new_key(<location_object>, self._key)
4      self._inner = self._session.declare_subscriber(key,
5                                                     self._cb,
6                                                     reliability=self._reliability)
```

We address the mobility subscription problem through the utilization of the background thread and the `MobilitySubscriber` class, ensuring a transparent management process.

6.4. REST Interface

The REST interface does not conform to the Zenoh API standards. Instead, it functions as a web server capable of interpreting the URL as the topic and utilizing HTTP methods to execute operations within the Zenoh network, as detailed in Section 4.2.2. In Figure 6.3, we present a diagram illustrating the functioning of the REST interface and its interaction with Zenoh.
We modify the web server to incorporate location-aware functionality. To enhance user-friendliness, we opt to utilize the query string to configure all the required parameters. For example, to publish a message in Zenoh using location-aware it is sufficient to make a POST request to a URL like:

```
https://host:rest_port/key_expr?lat=10.0&long=20.0&shape=circle&r=10
```

Within this URL, the `key_expr` functions as the topic without the inclusion of the location key. The existence of query parameters like `lat`, `shape`, and others, signals to the web server to process this request in location-aware mode. Subsequently, the web server maps these requests into the API detailed in the preceding section, and a potential output generated from the API, is formatted in JSON and returned in the HTTP response.

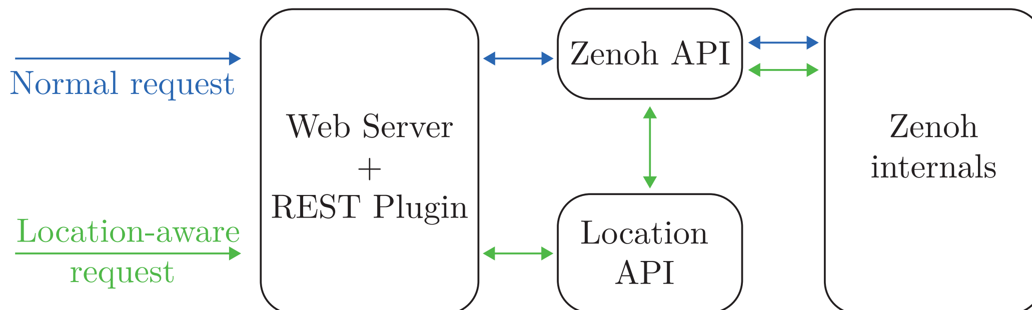


Figure 6.3: Zenoh REST Flow.

7 | Experimental Evaluation

In this chapter, we describe the evaluation of our implementation of the location-aware version of Zenoh. In Section 7.1 we explain the setup we use for the experiment, highlighting the evaluation metric we use, and baselines to confront the results with. In Section 7.2 we provide a summary of the evaluated encoding techniques as a guideline to choose the best technique based on the system characteristic. From Section 7.3 to Section 7.5 we present three network topologies with the goal of each test. In Section 7.6 we deal with the system behavior considering a limited computational capacity on router nodes. In Section 7.7 we discuss about the differences in performances using UDP as transport protocol instead of TCP. Finally, in Section 7.8 we explain the thought behind the configuration of the false positive percentage on Bloom Filter providing an example of threshold percentage.

7.1. Experimental Setup

Evaluating our distributed technique requires the emulation of multiple nodes, in which we ensure consistent conditions across each node throughout the experiments.
We create such environment using a ProxMox cluster running ProxMox Virtual Environment version 7.4-3. ProxMox cluster consists of multiple physical nodes, each equipped with an Intel(R) Xeon(TM) E3-1270 v5 CPU clocked at 3.60GHz, 64 GiB of RAM, and a 1Gbps ethernet connection. On these physical nodes, we create a total of 17 virtual

Virtual machine configuration

OS	Ubuntu Server 22.04.1
vCPU	1 socket, 4 cores
vRAM	4 GiB
HDD	120 GiB

Table 7.1: ProxMox node configuration.

machines (VMs) configured according to the specifications outlined in Table 7.1.

In addition to configuring the virtual machines (VMs), we install MahiMahi into each client VM. MahiMahi is a comprehensive toolset that provides a range of network emulation tools, in particular the **LinkShell**. The LinkShell simulates both dynamic links, such as those in cellular networks, and links with constant speeds. It categorizes packets into uplink and downlink queues based on their intended direction, releasing them according to the respective input packet-delivery trace.
This configuration ensures consistent replication of network conditions across different nodes, maintaining uniformity throughout various experiments. Moreover, it facilitates the emulation of wireless connections for both publishers and subscribers, closer to real-world use cases.

On the router nodes, we opt for **Wondershaper** installation. This tool allows us to regulate the bandwidth of the system network adapter, deliberately slowing down the network speed across the entire system. This capability enables us to emulate slow connections within our VMs, contributing to a more comprehensive testing environment.

The experimental evaluation comprises both our location-aware implementation of Zenoh and its original counterpart. As we construct it using the "Zenoh-0.7.2-rc" version, we employ the same version as a baseline for our evaluation.
The baselines comprise the experimental outcomes derived from the original Zenoh version within our testing environment. This approach allows us to acquire values that are directly comparable to those obtained using Zenoh Location-aware, as they originate from a controlled setting with identical node counts, configurations, and network conditions.

7.1.1. Metrics

In our experimental evaluation, we measure multiple metrics.
Latency refers to the delay or time lag between a published message and the instant a subscriber receives it, typically measured in milliseconds (ms). It is a critical factor in determining the speed and responsiveness of the distributed system. Evaluating latency in a distributed system often requires conducting a ping-pong test to prevent clock synchronization discrepancies among nodes. This test involves measuring the Round Trip Time (RTT), which represents the duration it takes for a message to travel to its destination and return. In this context, latency is calculated as half of the RTT, i.e., $RTT/2$.
In our testing setup, where both publisher and subscriber nodes are virtualized on the

same physical machine, the virtualization environment ensures clock synchronization. Consequently, there is no need to measure Round Trip Time; instead, we directly obtain latency as the ΔT between the publishing instant T_p and the one the message reaches destination T_s. The latency L is

$$L(ms) = \Delta T = T_s - T_p \tag{7.1}$$

Throughput refers to the rate at which messages reach a subscriber. It is measured in messages per second (msg/s). In our experiments we count the number of messages N the subscriber receives and then we compute the time interval ΔT as the difference between the timestamps of the last message (T_N) and the first one (T_0). Finally we compute the throughput R as:

$$R(msg/s) = \frac{N}{\Delta T} = \frac{N}{T_N - T_0} \tag{7.2}$$

7.1.2. Baselines

To evaluate our Location-aware version of Zenoh we need baselines to compare our results to. Baselines definition mainly depends on the differences and the aspects we want to underline, which are: the overhead incurred when parsing and computing the location key match, and the differences in latency and throughput with the receiver selection in the routing process rather than message filtering upon receipt.
From these two main points, we define the following experiments to obtain baselines with the original Zenoh:

- **B - Topic:** We initiate the message publication process by incorporating location details, specifically latitude, and longitude, into the first two levels of the topic structure, resulting in the format: <latitude>/<longitude>/evaluation/test. Subsequently, all subscribers subscribe to the topic pattern */*/evaluation/test, thereby receiving all messages that are published. Within each subscriber, we establish a range for acceptable latitude and longitude values. We then extract coordinates from the received topic and decide whether to accept or disregard incoming messages based on a comparison between the publisher's coordinates and the defined acceptance range. We finally compute metrics only when a message is accepted, emulating the rejection of messages at the subscriber side before reaching the application layer.
- **B - Payload:** We initiate the message publishing process with the publisher including location information within the message payload. This leads to the creation

of the topic structure as evaluation/test, which is two levels shorter. Meanwhile, all subscribers subscribe to the evaluation/test topic. In contrast to the previous baseline, in this scenario, all subscribers still receive all messages, but the filtering process now depends on the location information found within the payload rather than utilizing the topic. Also in this scenario, we compute metrics only when a message is accepted, emulating the rejection of messages at the subscriber side before reaching the application layer.

In these two baselines, it is interesting to analyze how performance evolves when we introduce two levels of complexity into the topic, as opposed to incorporating the same information into the payload. Ultimately, we compare these results to the same scenarios executed using Zenoh Location-aware to see how our custom implementation natively performs compared to the original version.

Testing scenarios vary; indeed, we assess our system using three distinct network configurations. Each configuration features a unique topology and parameters, specifically addressing the number of messages sent per second by the publishers and the link speed between nodes.

7.2. Results Overview

Before we delve into individual experiments and their outcomes, we present a summary of the typical characteristics exhibited by encoding techniques, focusing on system and network properties.
In our evaluation, we examine three network topologies. In the first topology, we partition the network into two distinct geographical interconnected zones. In each zone, there is a publisher focused on reaching subscribers within that specific area. Through the use of location-aware encoding techniques, messages are restricted from traversing between the two zones, unlike in the baseline scenario. We design this topology to assess the extent to which the introduction of messages from different zones influences performance in a target zone. The second topology, referred to as *Near-field*, aims to measure the overhead that routers face in handling Zenoh location keys, particularly when subscribers are in close proximity to the publisher. Additionally, we place additional subscribers configured with non-matching locations further in the network with the purpose of generating traffic in the routers, which occurs only with the baselines. The third topology, known as *Far-field*, seeks to measure the impact of non-matching subscribers along the network path. In this topology, we position our location-matching subscriber far away from the publisher

and, for each one, we place an additional subscriber in the middle of the path, which is a receiver target only for baselines, increasing the outbound traffic of the router.
In all three topologies, we configure the network to emulate a 4G/LTE connection for both publishers and subscribers, with a slowed connection of 5Mbps between routers. This configuration enables us to simulate, in conjunction with a high message rate from the sender, a congested router backbone infrastructure. This choice aims to emphasize the behavior of encoding techniques, as differences may not be visible in a high-performance infrastructure. Furthermore, we perform tests in all the three network topologies using UDP as transport protocols instead of TCP. We conduct this test to observe how performance is affected and how the encoding techniques react eliminating the overhead associated with the reliable communication provided by TCP. In conclusion, we deliberately reduce the computational capacity of router nodes to underline the effort required to route messages.

In Table 7.2, we present a summary of the results obtained, which are thoroughly explained in subsequent sections. We use + and - symbols to indicate respectively a good

Scenario	**Baseline**		**Location-aware**		
	Topic	Payload	Base64	MGRS	Bloom Filter
Zoned Location-aware					
*	- -	- -	+ +	+ +	+ +
Near-field					
TCP + RH	+ +	+ +	- -	-	- -
UDP + RH	+ +	+ +	+	+ +	+
TCP + RL	+	+ +	- -	+	- -
UDP + RL	+	+ +	+	+ +	+
Far-field					
TCP + RH	- -	+ +	+ +	+ +	+ +
UDP + RH	+ +	+ +	+ +	+ +	+ +
TCP + RL	+	+ +	-	+ +	+
UDP + RL	+	+	+	+ +	+

Table 7.2: Results Overview.

or a bad performance level on each encoding technique in the specified scenario. We organize the table into the three proposed network topologies, and for each, we indicate the scenario by mentioning the transport protocol used, along with the computational capacity of the routers. We describe the computational capacity using the labels RH for high-performance routers and RL for low-performance routers.

As indicated in Table 7.2, regardless of the transport protocol or router computational capacity, the location-aware encoding techniques consistently outperform the baseline in a full location-aware scenario. This improved performance depends on the enhanced routing capability of the location-aware implementation, ensuring that messages are confined within their respective zones. This, in turn, mitigates unnecessary traffic, minimizes network bandwidth usage, and reduces computational resource wastage. On the other hand, opting for using a location-aware encoding in a network topology where there is no location-based distinction, may not always represent the most efficient choice. The additional overhead on routers to manage location keys could represent a useless operation when dealing with a limited number of subscribers or a small set of distinct topics. A slightly different case is the MGRS encoding technique, which, thanks to its lightweight nature in terms of both the length of the location key and the required resources, almost always represents a good choice. Therefore, the selection between different encoding methods is closely tied to the specific characteristics of the network topology and the performance capabilities of the system.

7.3. Zoning Network

The zoning network topology aims to underline the differences between the three location awareness methods we develop and the two baselines in a full location-aware scenario with well defined geographical zones of interest.

7.3.1. Network Layout

The experimental setup we show in Figure 7.1 consists in 17 nodes configured with the following roles:

- Nodes 1 to 5 and 14 to 17: Zenoh routers
- Nodes 6 to 11: Zenoh subscribers
- Nodes 12 and 13: Zenoh publishers

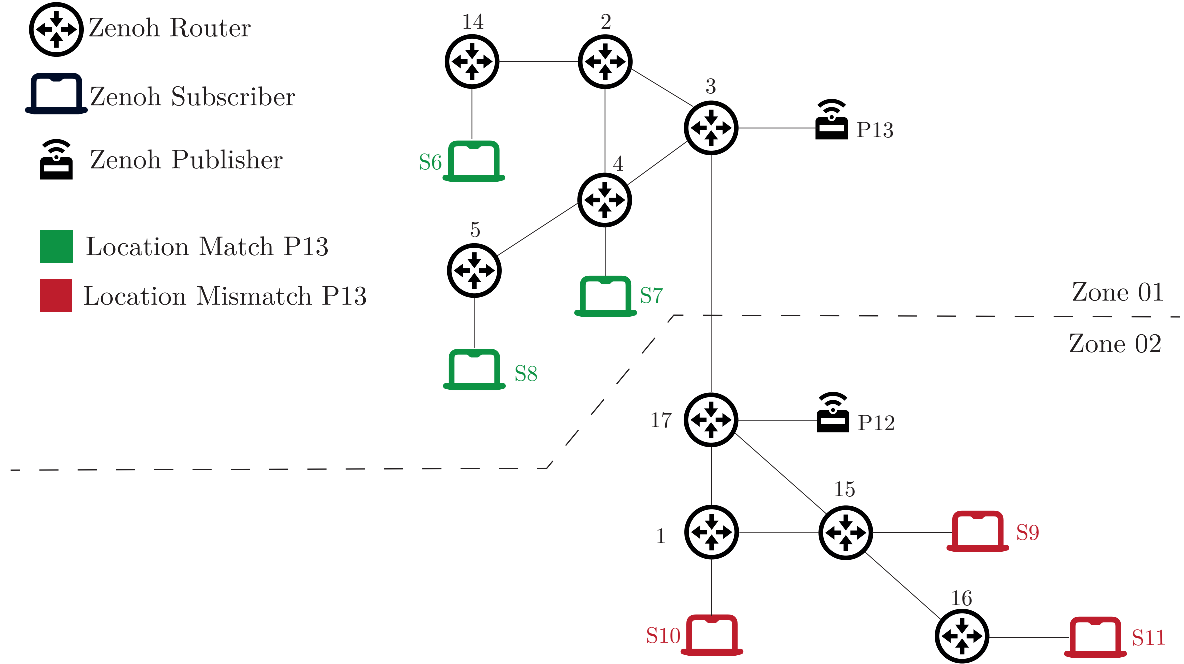


Figure 7.1: Zoning network topology.

We partition the network into two clearly defined interconnected zones, with each publisher utilizing distinct location data. Subscribers within a given zone only consider messages originating from the publisher within that same zone. For instance, subscribers S6, S7, and S8 exclusively consider messages from publisher P13.

In this network configuration, we employ the MahiMahi trace file named *TMobile-LTE-driving.down* to set up the outbound links for publishers and the inbound links for subscribers. This configuration ensures a connection speed of around 100Mbps, emulating mobile devices connected to a 4G/LTE mobile network. Instead, the router links are fixed at a constant speed of 5Mbps, which simulates a network infrastructure that requires a limited number of messages per second to be congested. Although Zenoh has the capability to dynamically build its network by discovering nearby nodes, we opt for a static configuration. This choice ensures to maintain the defined network topology throughout different executions of the same experiment.

Moreover, we configure publishers and subscribers as clients. By doing so, they function as end devices incapable of routing messages or extending the network. Therefore, they only depend on the Zenoh router they are connected to.

Next, we establish a message publishing rate of 20 messages per second for `publisher P13` to replicate a realistic scenario. Concurrently, we set a rate of 3200 messages per second for `publisher P12` with the intention of simulating a broad network and inducing

congestion in the router links. Given that our primary focus in this topology is to measure the impact of extraneous messages in a specific zone, we configure the 20 messages per second on publisher P13 to ensure that the network does not become congested. Subscribers S6, S7, and S8 exclusively capture messages from P13, allowing them to measure latency and throughput using only those messages. Publisher P12, on the other hand, serves the purpose of simulating a large volume of messages in a wide network. Its high rate contributes to link congestion. This behavior should have a notable impact on Zone 01 in baseline experiments, as there are no mechanisms in place to isolate the two zones. In this test, we employ TCP as the transport protocol to enhance communication reliability. Additionally, we configure the Bloom Filter with a 25% probability of false positives for a domain comprising 100 elements. We discuss the choice of this probability in Section 7.8.

7.3.2. Results

We conduct the experiment in Zone 01, gathering data from subscribers S6, S7, and S8. The experiment is repeated 12 times, with each iteration lasting 3 minutes. During each run, we collect and analyze all the messages received by each subscriber.

In Figure 7.2, we observe the average latency recorded for all messages received during

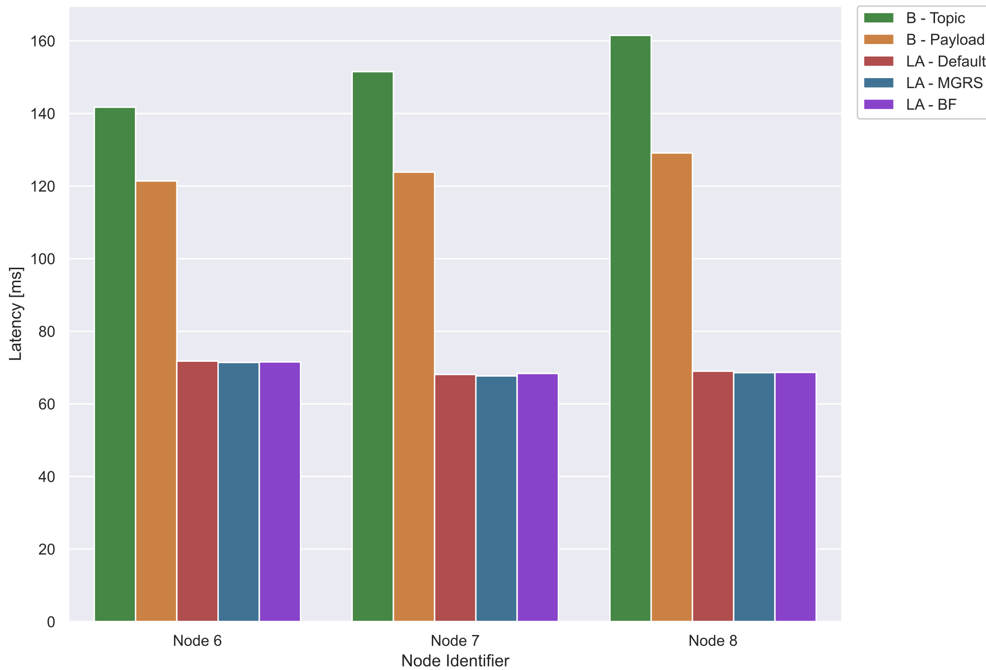


Figure 7.2: Zoned network: latency.

all 12 runs for each subscriber of Zone 01. Messages sent by Zone 02 heavily impact the latency on the baselines, since they are received also by the subscribers in Zone 01. In contrast, the location-aware mechanism prevents the unnecessary routing of messages outside their respective zones, effectively partitioning the network.

In Figure 7.3, a consistent pattern emerges in terms of mean throughput. The baselines show lower throughput in comparison to the location-aware technology. This discrepancy arises from the baselines receiving a reduced number of messages from Publisher 13 within the same timeframe. The reason behind this is that subscribers must handle and discard messages originating from Zone 02 as well, since there are no mechanisms to prevent message routing. In this chart, we zoom in on the y-axis, starting from 19.5ms instead of 0, to emphasize and accentuate this behavior.

In Figure 7.4, we observe a notable difference in the standard deviation calculated across all latency values for each node. The baselines show greater variability, due to the queuing phenomenon on inbound links of each node. That means extraneous messages from Zone 02 bring to congestion links in Zone 01, slowing down message processing and increasing latency also for messages from P13, which are the ones we measure, impacting the overall performance.

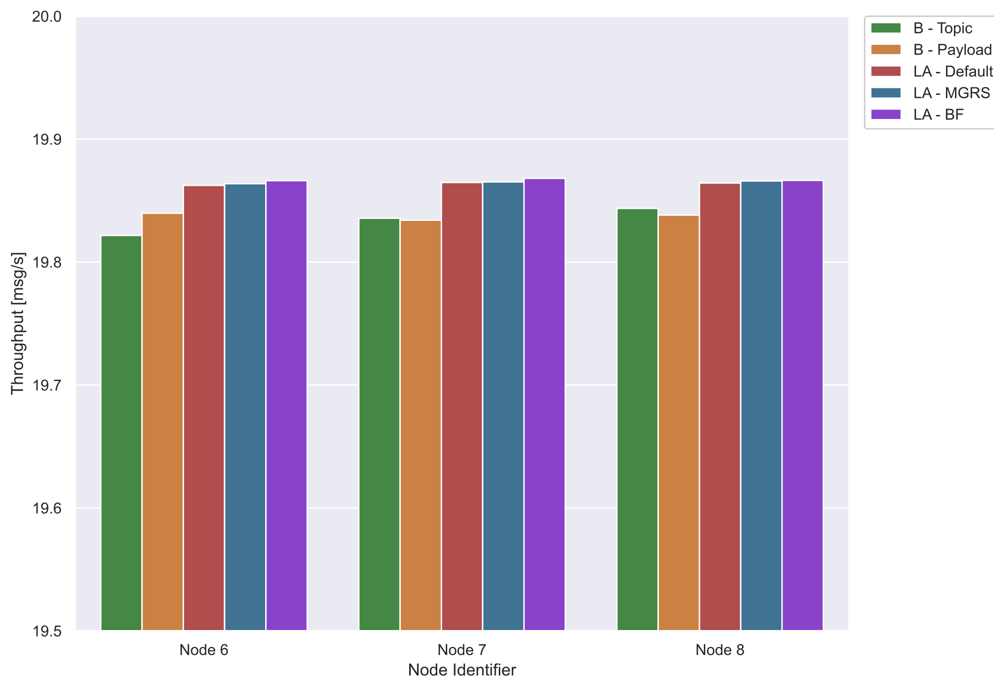


Figure 7.3: Zoned network: throughput.

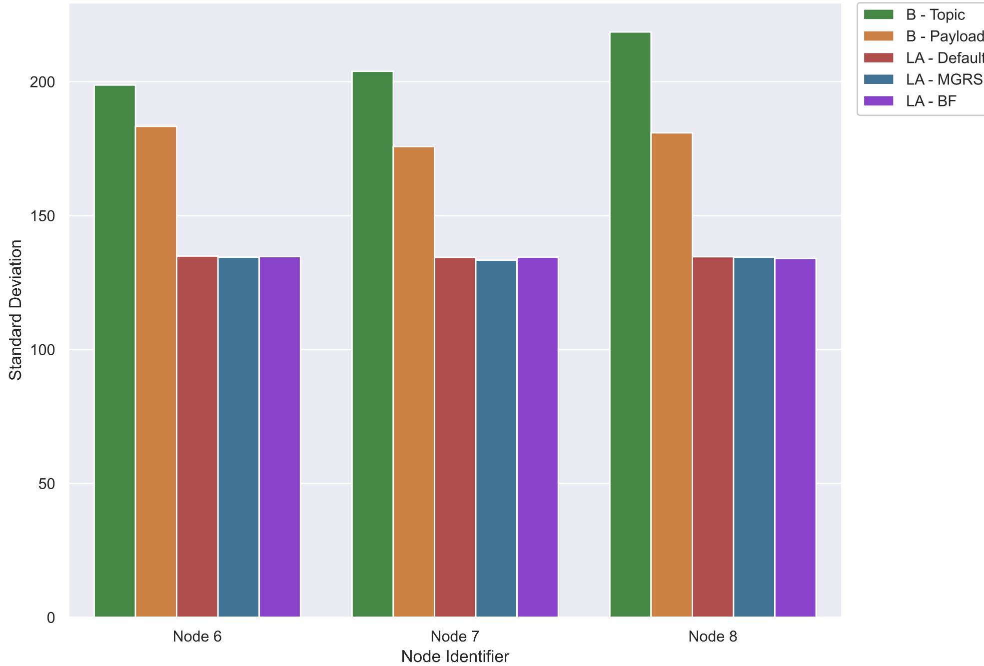


Figure 7.4: Zoned network: latency standard deviation.

7.4. Near-field Location-aware Subscribers

The near-field location-aware subscriber experiment aims to assess performance variations among the three location-aware technologies and highlight the additional overhead placed on routers in comparing location keys.

7.4.1. Network Layout

The experimental setup we show in Figure 7.5 consists in 15 nodes configured with the following roles:

- Nodes 1 to 5 and 14 to 16: Zenoh routers
- Nodes 6 to 11: Zenoh subscribers
- Node 13: Zenoh publisher

In this experiment, we do not partition the network into distinct geographical zones. Instead, we position subscribers that match published messages closer to the unique publisher. Opting for a single publisher is our choice to prevent biases from extraneous messages, ensuring an isolated test environment. Node P13 serves as the publisher, configured with a message rate of 3200 messages per second, a quantity sufficient to induce

congestion in the links. A congested network deliberately slows down messages, accentuating performance differences. In a less congested or more efficient infrastructure, these differences would not be as significant.
Subscribers S6, S7, and S8 match the location configuration of the publisher, while the remaining subscribers do not record any messages, if received. Placing matching subscribers closer to the publisher allows us to measure latency before encountering other client nodes, whereas non-matching subscribers are positioned at the network's edge, with the sole purpose of representing possible outbound links for routers 14, 4, and 5.

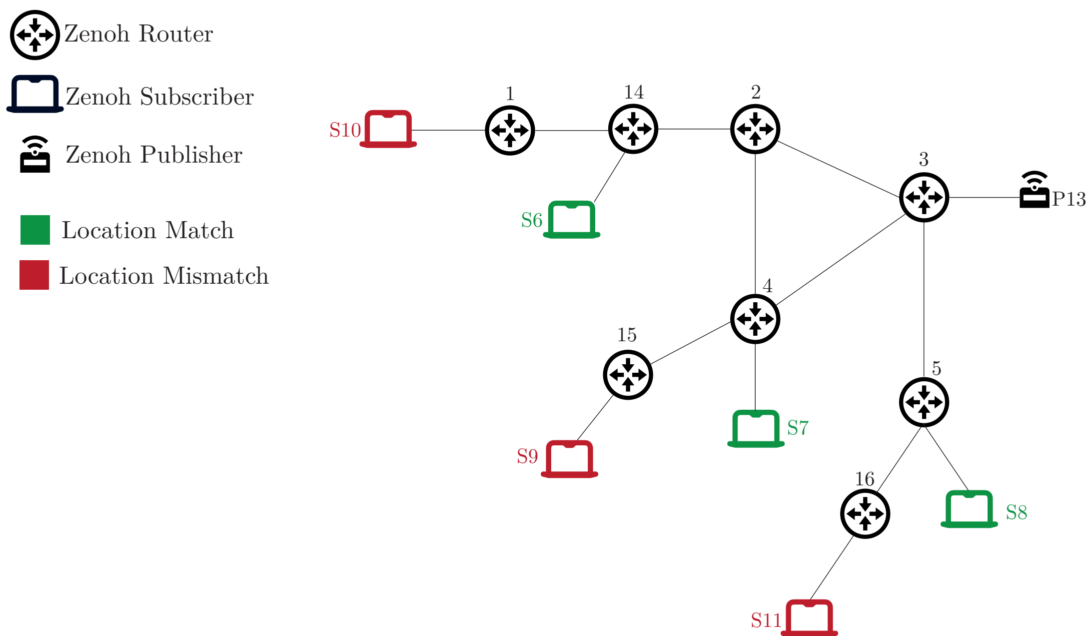


Figure 7.5: Near-field location-aware subscribers topology.

In this configuration, our objective is to evaluate the overhead that routers encounter in handling the location key. As the matching subscribers constitute the first point where a message intersects a client in the path from the subscriber to the network edges, we gauge the time it takes for a message to be routed up to that point, ensuring the exclusion of any alternative paths. In the baseline scenarios, the subscribers still retain their position as the primary points in the path, but we introduce an additional outbound link, representing the connection with a broader network. This is achieved by placing S9, S10, and S11 after our target nodes. Of course, these last nodes still exist in all the experiments, but the location-aware mechanism prevents messages from overcoming routers 14, 4, and 5.

The remaining experimental parameters remain consistent with the previous experiment for a better comparison. We use TCP as transport protocol. Then, both the publisher

uplink and subscribers downlink utilize the MahiMahi trace file, specifically *TMobile-LTE-driving.down*, ensuring an average speed of 100Mbps. Similarly, Wondersharper decelerates links between routers to 5Mbps. Furthermore, we maintain a false positive probability of 25% for the Bloom Filter within the same domain of 100 entities.

7.4.2. Results

We conduct the experiment 12 times, with each iteration lasting 3 minutes. During each run, we collect and analyze the messages received by subscribers S6, S7, and S8.

In Figure 7.6, we can observe the influence of the location matching mechanism. The congested network and the proximity of the subscribers to the publisher highlight the distinctions among the three location awareness methods and the baselines. In fact, the latency of the three location-aware encoding is much higher w.r.t. the baselines due to the presence of an additional key to handle, while the baselines just match the topic and, in this experiment, dispatch messages to all the subscribers.
In Figure 7.7 we zoom on the three location-aware methods. The best performing is MGRS encoding. It has a shorter representation, 15 characters maximum, resulting in a smaller message size and requiring less computational time for matching.
The worst-performing method is the Base64 one. Due to the possibility of representing detailed areas, it requires a more complex encoding. In fact, the data representation as

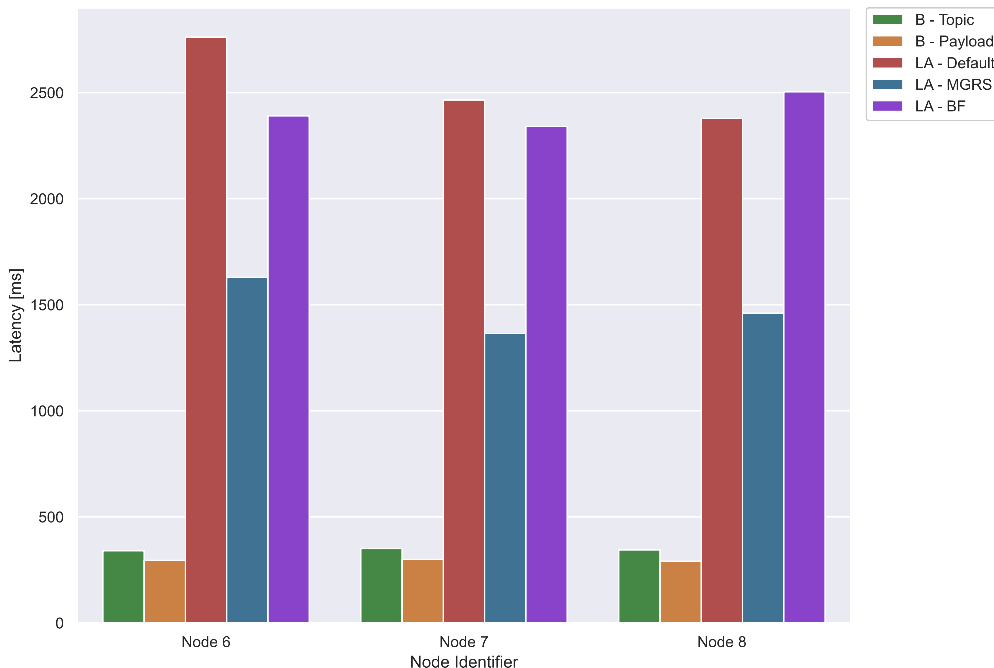


Figure 7.6: Near-field location-aware subscribers network: latency.

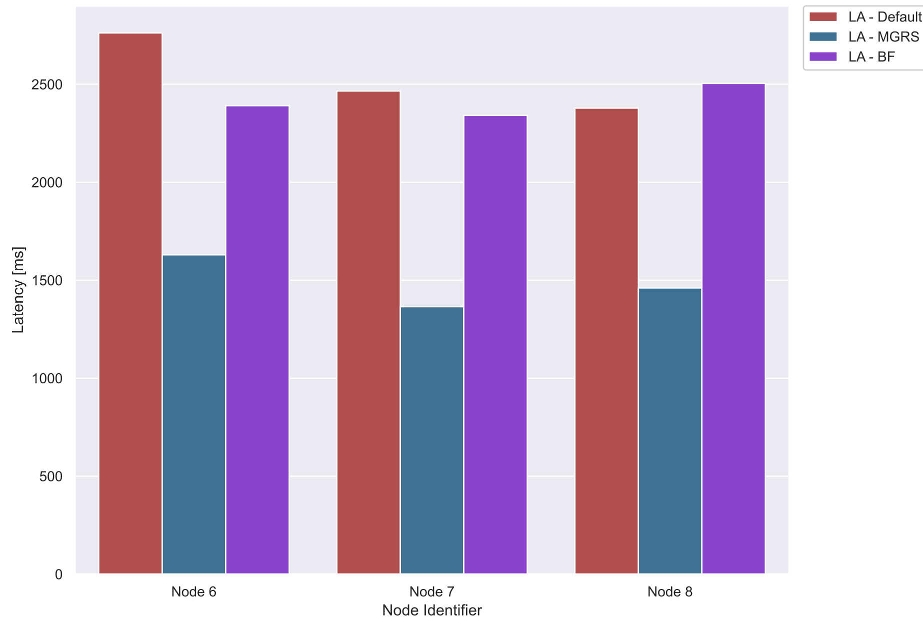


Figure 7.7: Latency zoom on location-aware methods.

a JSON object and the subsequent encoding in Base64 string generates a longer location key. In addition, routers must decode the base64 string and recreate the objects to handle them, resulting in longer and more complex operations.
Finally, between the two, we find the Bloom Filter. It has the ability to describe complex areas but it adds the possibility of false positives during the match. The length of its encoding is comparable to the one of the Base64 technique. The Bloom Filter uses the bitwise operations to match filters, which requires fewer computational resources compared to the decoding and parsing operations necessary in the Base64 technique.

In Figure 7.8, we observe the effects on the throughput. Despite the increased latency of location-aware methods, the throughput remains good. In fact, as evident from the chart in Figure 7.9, the higher variability observed in location-aware methods indicates a significant queuing phenomenon. This phenomenon, indeed, is responsible for an increase in the average latency due to a slowdown in processing most of the messages, an effect that does not impact the average throughput. Let us take, for example, a data-set obtained from subscriber S6 on the location-aware Base64 technique experiment. The graph of the latency of individual messages, shown in Figure 7.10, illustrates how the latency of a single message increases initially, then stabilizes, and increases again towards the end. The central part of the graph, where stabilization occurs, represents the network in a steady state, meaning we are in a network congestion condition and the queues remain stable. With queue stability messages still be processed at a constant rate but the longer wait in both outbound and inbound queues in interconnected routers, increases the

overall latency shifting the reception of the same amount of messages, compared with the baselines, forward in time.

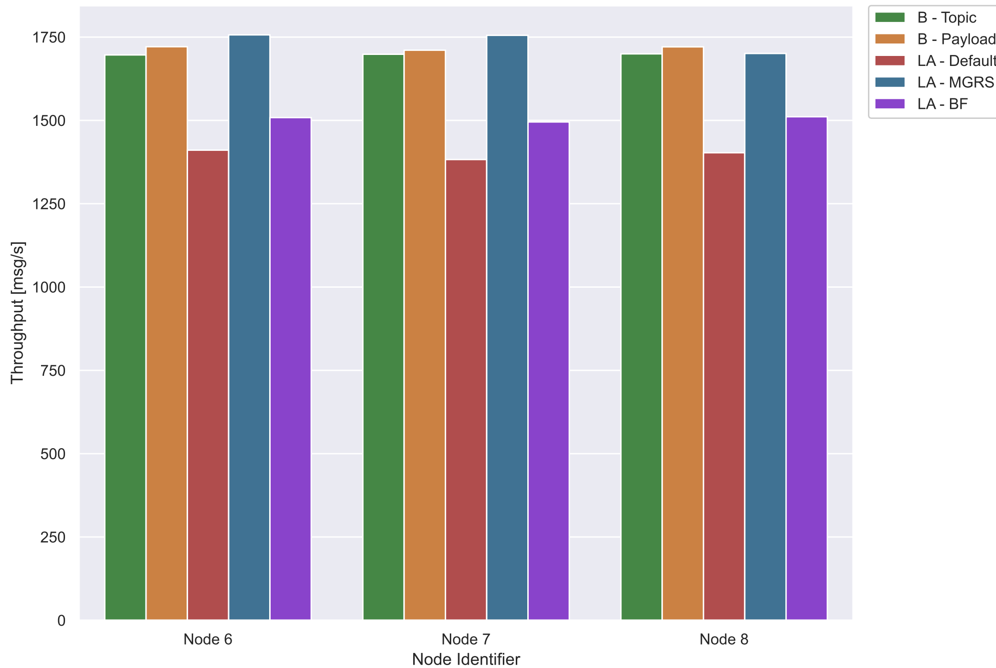


Figure 7.8: Near-field location-aware subscribers network: throughput.

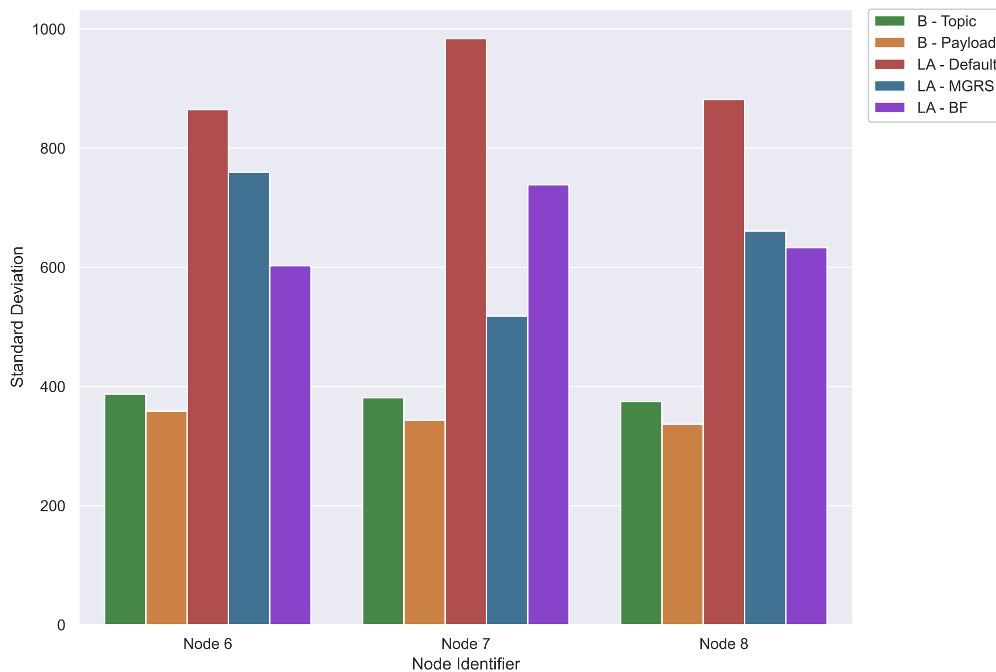


Figure 7.9: Near-field Location-aware subscribers network: latency standard deviation.

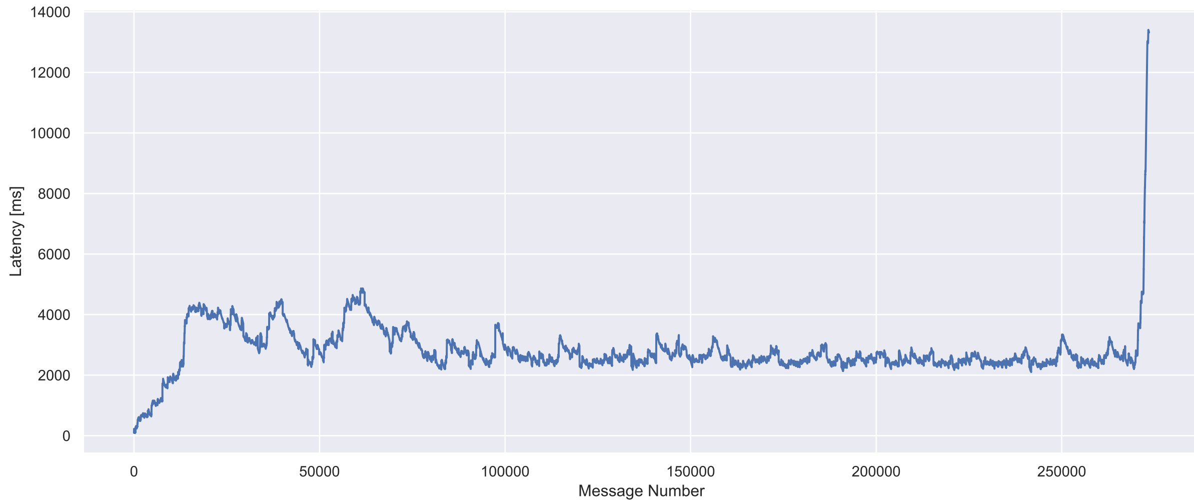


Figure 7.10: Base64 location-aware single experiment latency on node 6.

7.5. Far-field Location-aware Subscribers

The topology of the far-field location-aware subscribers aims to evaluate the influence of subscribers positioned halfway along a path between a publisher and a matching subscriber. This is done to emphasize distinctions between the baseline, where messages are sent to all subscribers, and the location-aware techniques, which directly select target subscribers on routers.

7.5.1. Network Layout

The experimental configuration we show in Figure 7.11 presents the same topology we use in the near-field subscriber setup. The only variation lies in the placement of subscribers S6 and S7. We place them at the edges of the network, while we move nodes S9 and S10 in the middle of the path. This arrangement allows us to assess the performance of the location-aware version in a network where there are the presence of non-relevant subscribers.
Message rate, links speed, transport protocol, and Bloom Filter false positive probability remain unchanged for a better comparison.

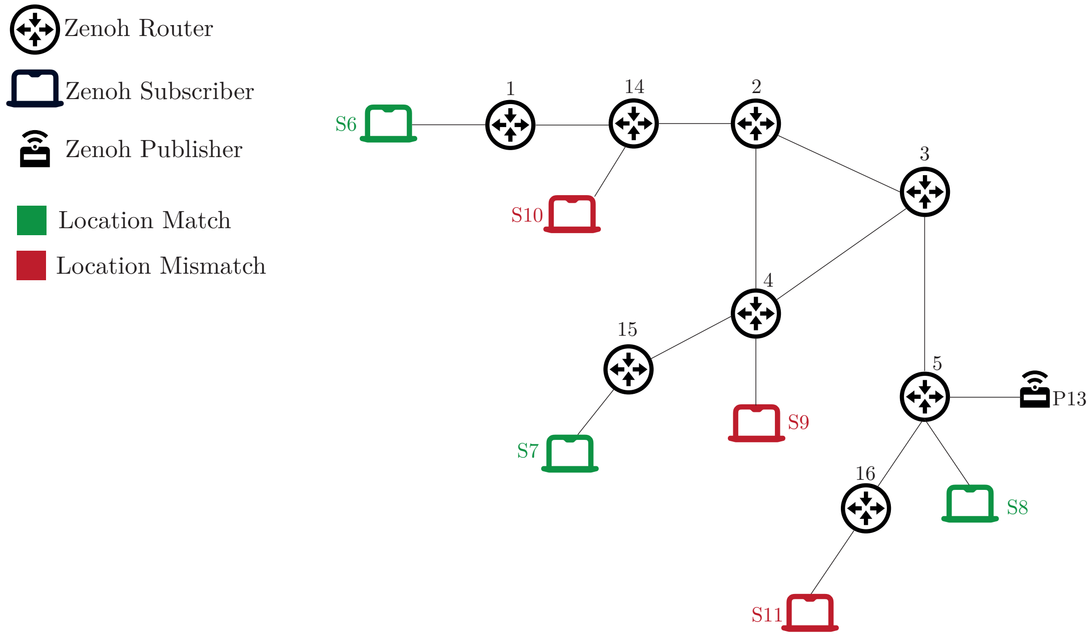


Figure 7.11: Far-field location-aware subscribers topology.

7.5.2. Results

We conduct the experiment 12 times, with each iteration lasting 3 minutes. During each run, we collect and analyze all the messages received by subscribers S6, S7, and S8.

Shifting S6 with S10, and S7 with S9, moving subscribers to the edges of the network induces a shift in the dynamics of the proposed methods. Figure 7.12 illustrates the better performance of location-aware methods in terms of latency when compared to the baselines, particularly when juxtaposed with the Near-field experiment introduced in Section 7.4. The higher network hops and the presence of extraneous subscribers along the route affect the efficiency of message routing in the baseline scenarios. This is attributed to the higher traffic on the links and an increased workload on routers. For example, router 14, referring to the topology in Figure 7.11, initially dispatches a message to subscriber 10, which rejects it, before forwarding the same message to router 1. In contrast, location-aware methods avoid this behavior. Router 14 considers only router 1 as next step since subscriber 10 does not meet the location criteria and it is considered unrelated to the current topic.

Regarding throughput, as depicted in Figure 7.13, we observe the same trend as in the Near-field experiment. The increased latency characteristic of the baselines does not result in a reduction of throughput; in fact, the queuing phenomenon reoccurs. In Figure 7.14, the standard deviation calculated in latency reveals a greater variability in values for the

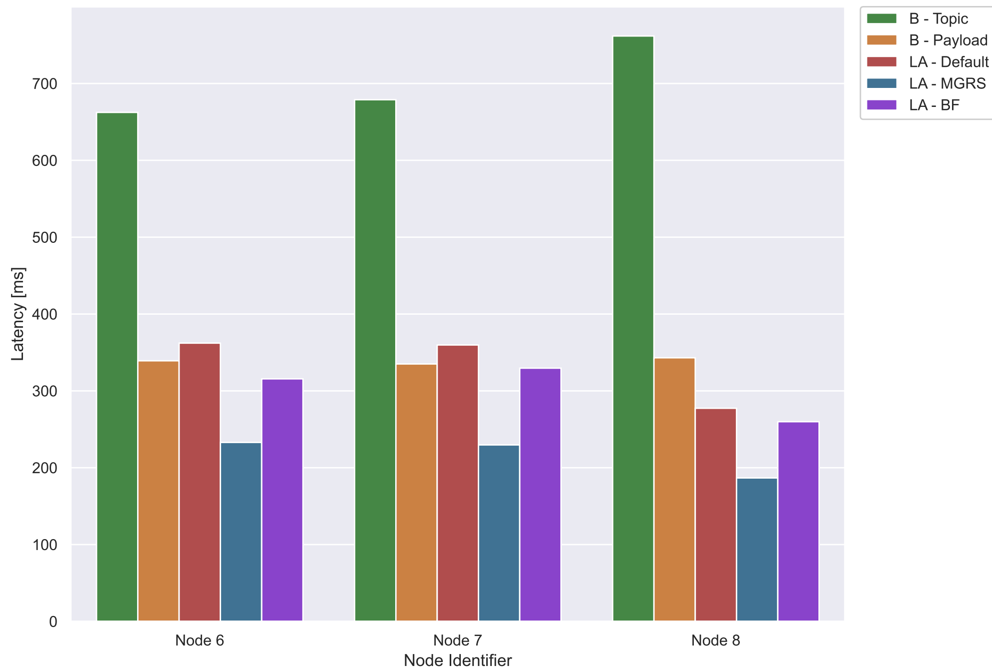


Figure 7.12: Far-field location-aware subscribers network: latency.

baselines compared to location-aware methodologies, which confirms the behavior we see in Section 7.4 but affecting the baselines instead of the location-aware techniques.
Lastly, looking at the three location-aware encoding techniques, the difference among them confirms the previously described results, ranking MGRS first, followed by Bloom Filter, and positioning the Base64 technique as the slowest. This also confirms that the mutual performances among the three methods are maintained across different topologies, since they reflect the results we find in the experiment in Section 7.4.

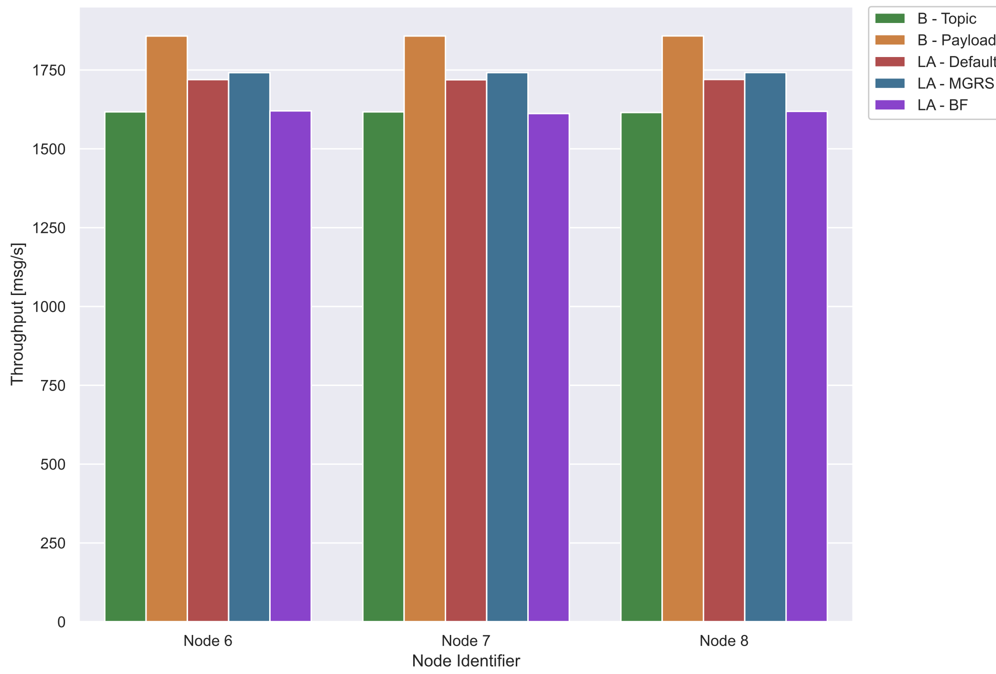


Figure 7.13: Far-field location-aware subscribers network: throughput.

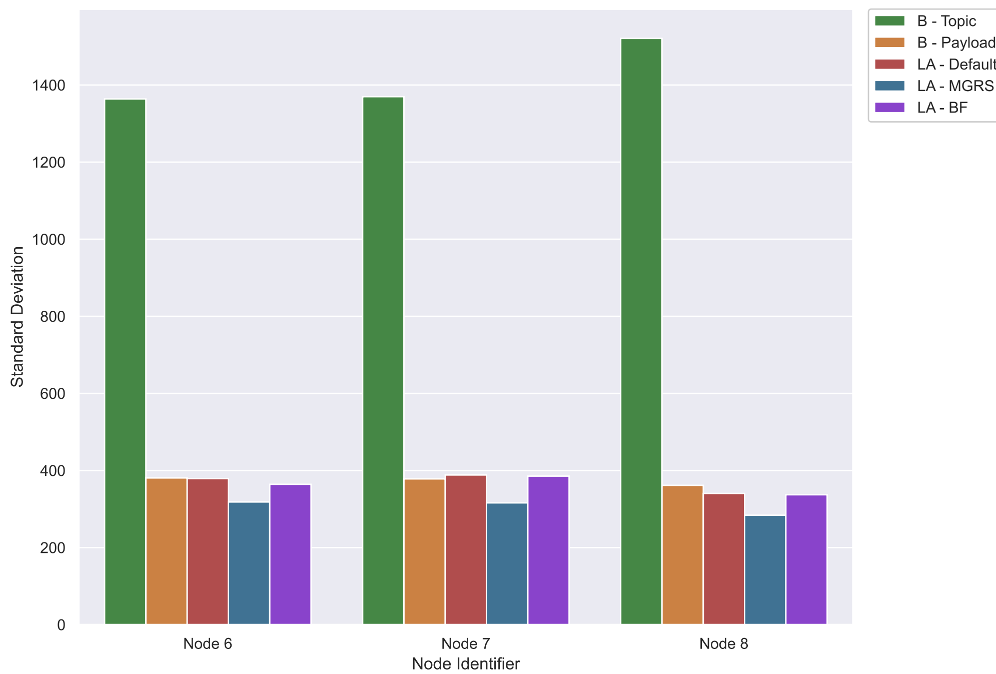


Figure 7.14: Far-field location-aware subscribers network: latency standard deviation.

7.6. Computing Capacity

Along with the performance the network infrastructure provides, another aspect influencing the system's performance is the computational capacity of the routers responsible for message distribution. While we take into account the first aspect by inducing congestion on the links, for the computational capacity, we conduct specific tests.
Thanks to the Proxmox configurations, we limit the vCPUs to 20% of their actual capacity in all virtual machines serving as routers. By reducing the computational capacity we want to evaluate how our system performance changes when routers require more time to process messages and how this impacts in particular on the encoding techniques.
Let us take the Near-field subscribers experiment we discuss in Section 7.4. Conducting the identical experiment with a limited CPU capacity, for which we report the latency chart in Figure 7.15 along with its standard deviation in Figure 7.17, results in an increase in the average latency accompanied by a rise in the standard deviation calculated on the latency values. This is attributed to the increased time messages require for processing. However, this combination does not impact the throughput as we can see from Figure 7.16, which does not show significant variations compared to what we achieve using the full computational capacity of the vCPUs, shown in Figure 7.21. We observe this trend in the majority of our experiments conducted with limited computational capacity. This indicates that when we decrease the processing speed of the router, the entire system experiences an identical slowdown.

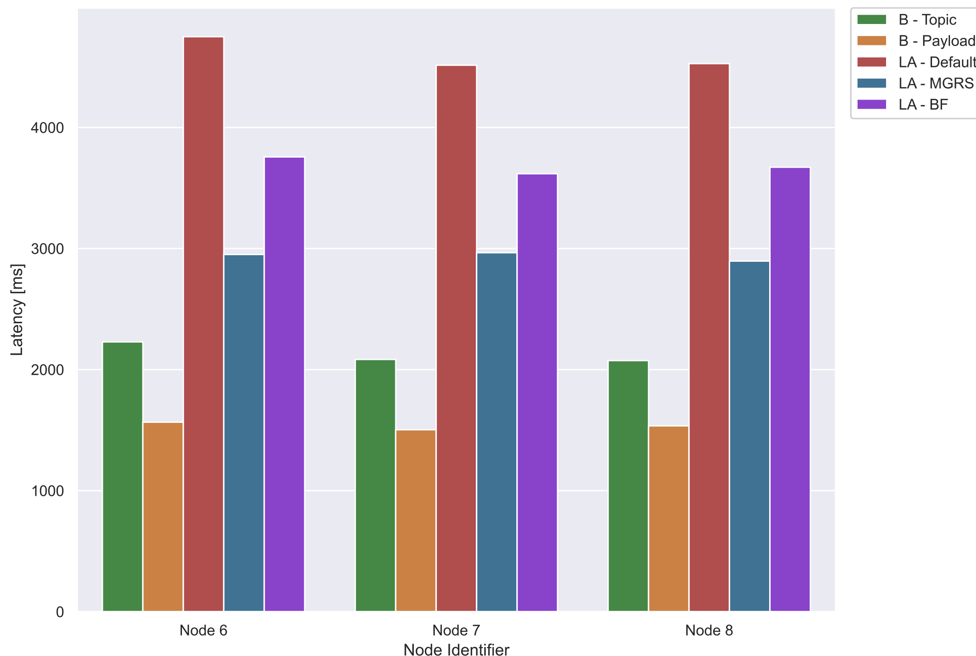


Figure 7.15: Near-field location-aware subscribers network: CPU 20% latency.

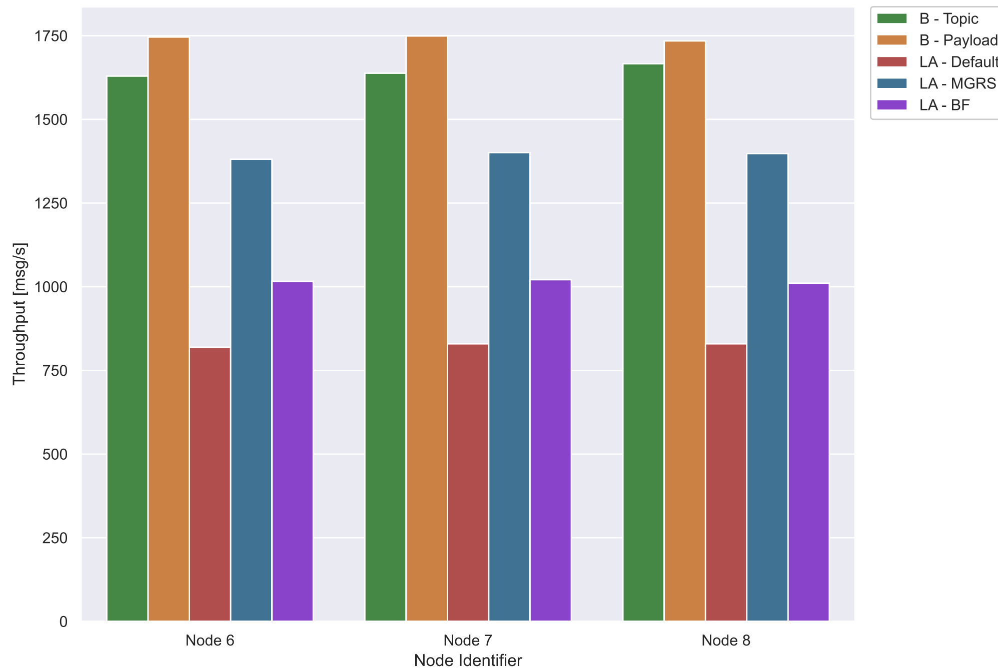


Figure 7.16: Near-field location-aware subscribers network: CPU 20% throughput.

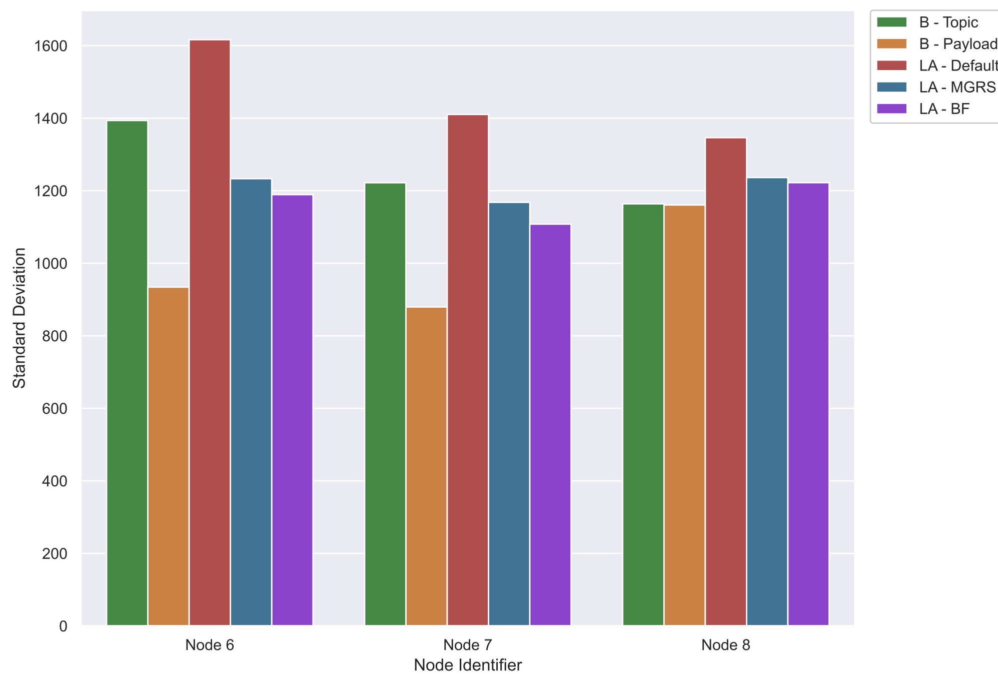


Figure 7.17: Near-field location-aware subscribers network: CPU 20% latency standard deviation.

A different scenario concerns the trend among the encoding techniques related to the same experiment. In the case of the Near-field Location-aware subscriber experiment, the proximity of subscribers to the publisher is sufficient to ensure that there is no alteration in the trend among the different methods since the number of routers that can affect the performance are limited.
A different situation occurs, for example, in the Far-field experiment that we discuss in Section 7.5. By reducing the computational capacity of routers, there is also a variation in the mutual trend among the different methodologies. In the chart we present in Figure 7.18, the higher number of routers between the publisher and the subscribers more significantly impacts the latency in the location-aware Base64 and Bloom Filter methods if compared to other methodologies, due to the longer encoding each router has to process. At the same time, the standard deviation computed over latency values, as we can see in Figure 7.20, shows a lower value considering the latency increasing. This means that routers process messages more slowly but also more consistently.
The higher latency and the lower standard deviation are also evident from the decrease in throughput, which we report in Figure 7.19, which characterizes the two techniques.
However, considering that we reduce the computational capacity by 80% in an already congested network, the observed difference is not significant enough to characterize computational capacity as the primary influencing factor on the performance of location-aware methods compared to our baselines.

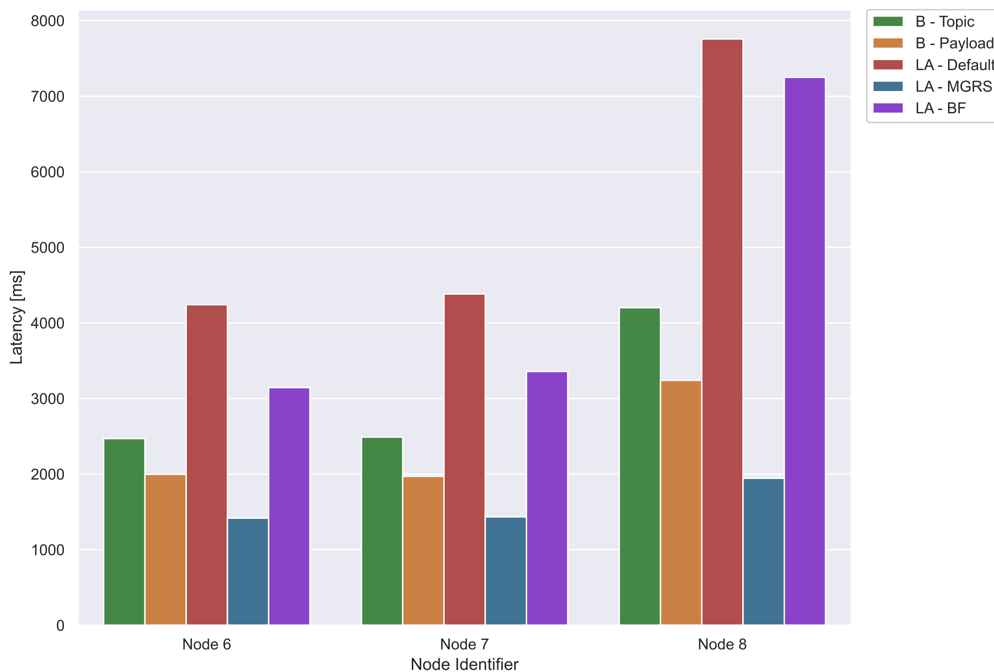


Figure 7.18: Far-field location-aware subscribers network: CPU 20% latency.

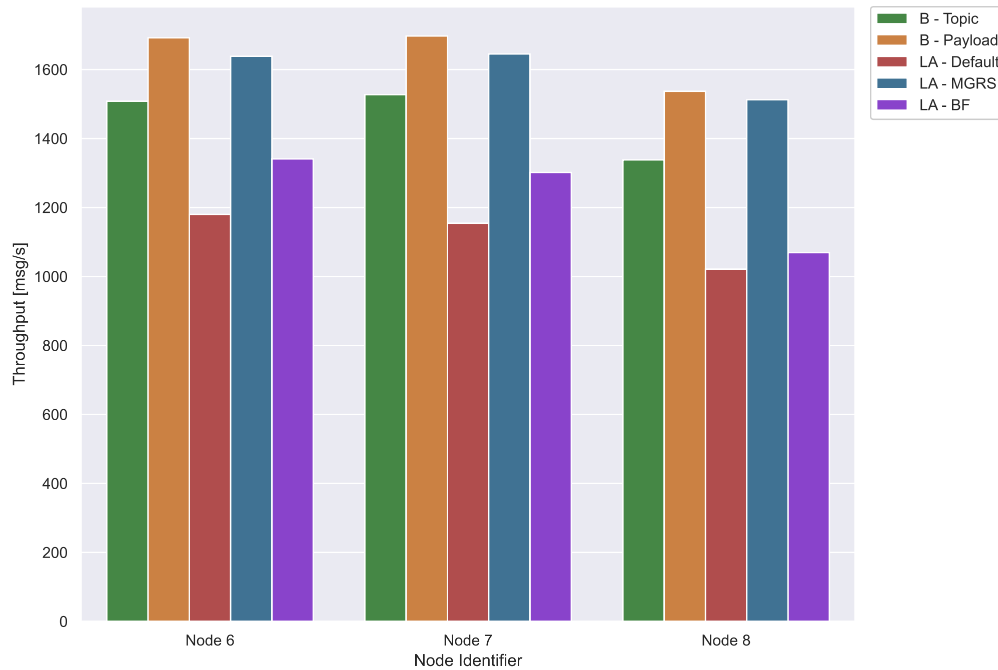


Figure 7.19: Far-field location-aware subscribers network: CPU 20% throughput.

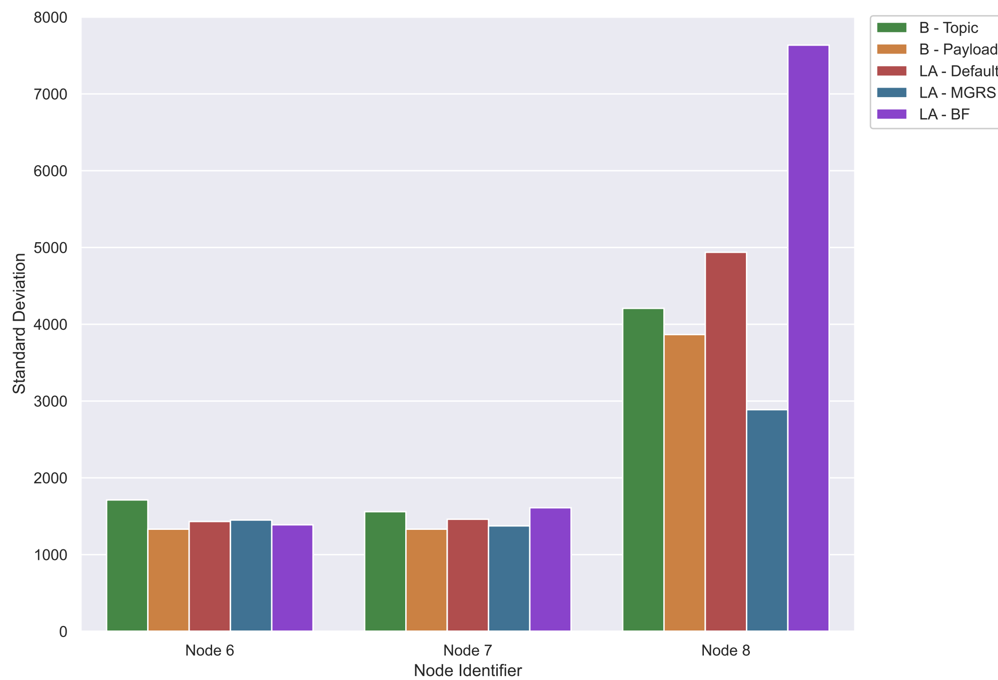


Figure 7.20: Far-field location-aware subscribers network: CPU 20% latency standard deviation.

7.7. Transport Protocol

Zenoh does not only work with TCP as a transport protocol but it can be configured with different types of transport protocols, as we mentioned in Section 2.3.1.

To assess the system's response to a diverse transport protocol, we opt to employ UDP. Unlike TCP, UDP is a connectionless protocol that does not ensure packet delivery. The absence of these features makes UDP a lightweight and faster transport protocol when compared to TCP. In our experiments, we execute all the experiments with the usual configurations described above, conducting each test 12 times to obtain data directly comparable to that already discussed.

In all the results we obtain, we observe a similar behavior. Specifically, a significant decrease in the average latency of each experiment due to the absence of the multiple overheads that distinguish TCP from UDP, resulting in a reduction in the standard deviation calculated on latency values. This indicates a more consistent message rate transmission over time. However, the trend among the various encodings remains the same, as the change in transport protocol does not impact the computational load that routers have to handle.

On the other hand, we observe a decrease in the average throughput. This is due to the fact that UDP does not guarantee message delivery, lacking retransmission mechanisms. In the congested network condition in which we conduct the tests, the lost packets using UDP are significantly higher than those lost with TCP, resulting in a lower number of received packets and consequently, a lower throughput.

For illustration purposes, let us examine the Near-field Subscriber experiment we discuss in Section 7.4. In this experiment, all configuration parameters remain unchanged except for the switch of the transport protocol from TCP to UDP. In that way, we can compare results directly.

As we can see from Figure 7.21 compared to the latency chart using TCP in Figure 7.6, we have a huge reduction of the mean latency computed over the 12 runs.

In Figure 7.22, we present the throughput chart. When comparing it with the TCP experiment chart in Figure 7.8, we observe a decrease in average throughput, particularly noticeable in the Location-aware Base64 and Bloom Filter experiments. These two methods exhibit greater weight in terms of topic length, consequently making them more susceptible to packet loss in a congested network. Similarly, despite an overall reduction in standard deviation across all latency values, as depicted in Figure 7.23, there is a slight elevation in values for the Location-aware Base64 and Bloom Filter techniques, a direct result of the longer encoding requiring messages to wait in the queue for a longer period, increasing queue length over time.

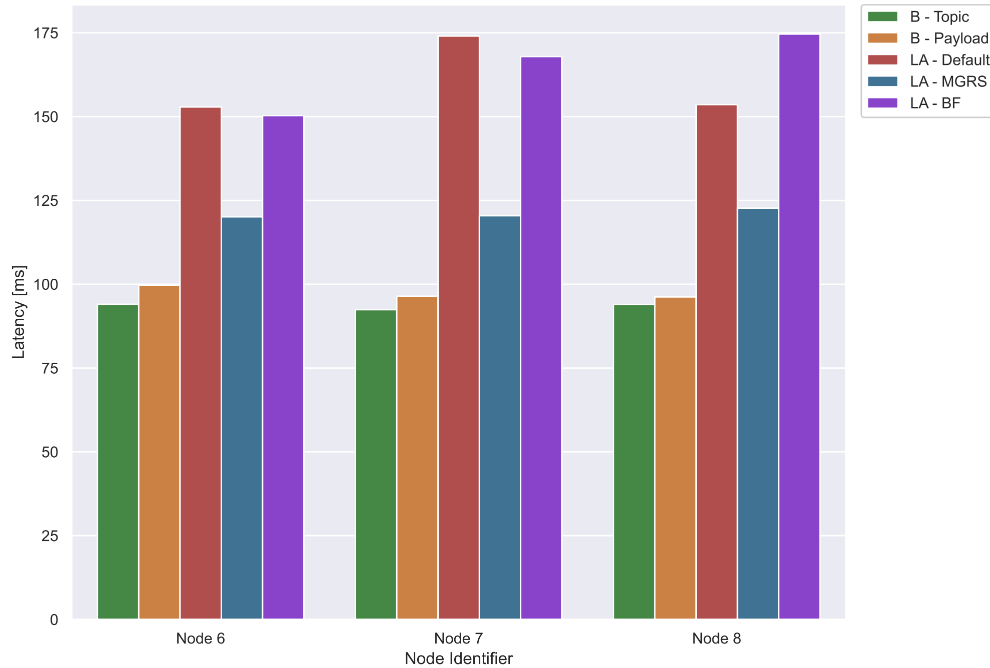


Figure 7.21: UDP near-field location-aware subscribers network: latency.

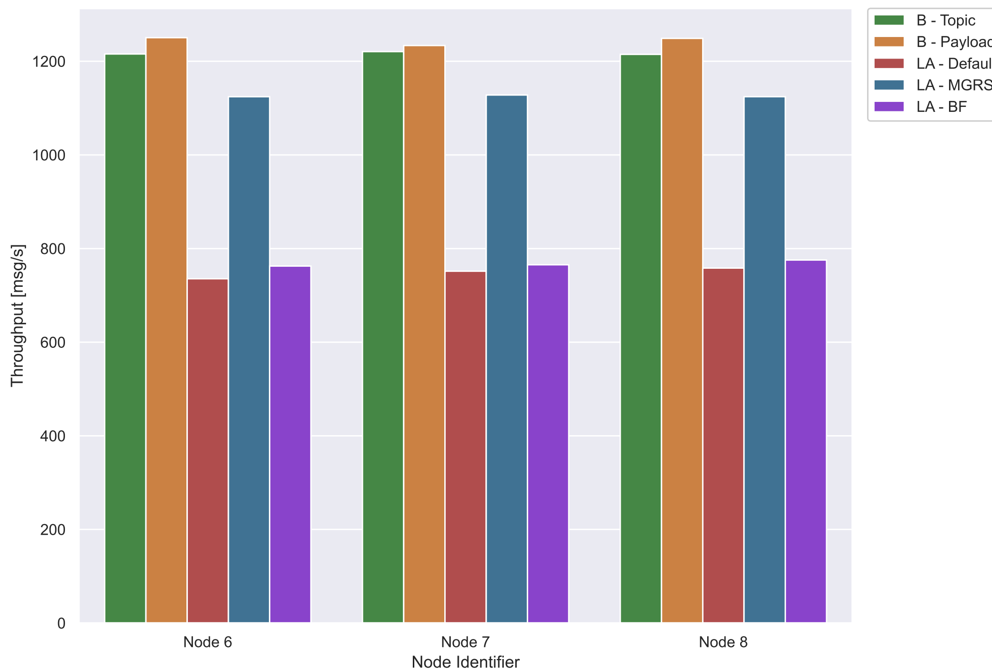


Figure 7.22: UDP near-field location-aware subscribers network: throughput.

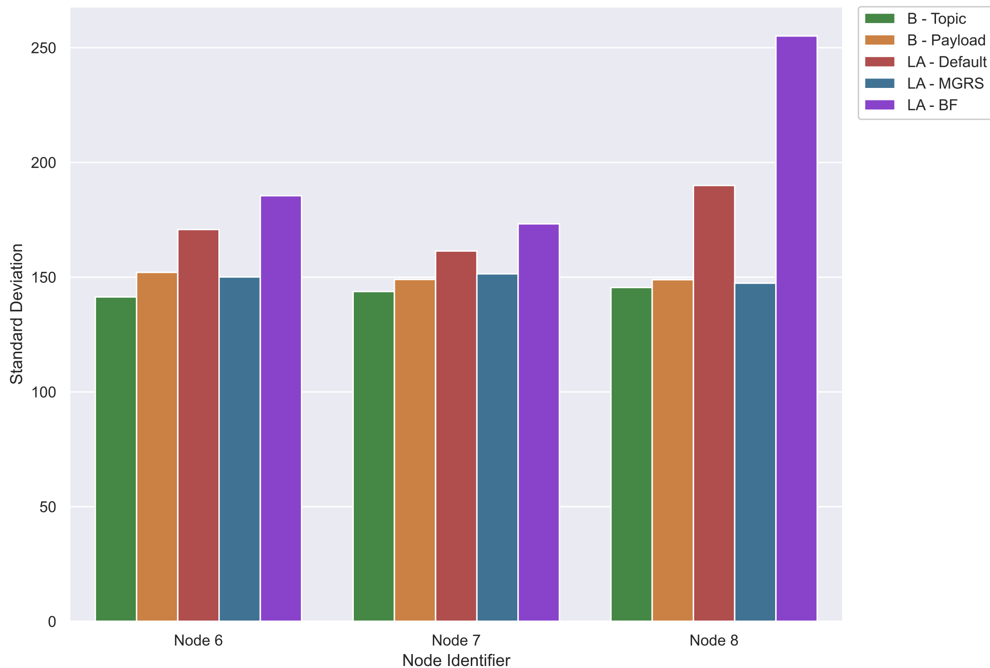


Figure 7.23: UDP near-field location-aware subscribers network: latency standard deviation.

Even more interesting is what happens when considering the use of UDP with a simultaneous reduction in the computational capacity of the routers. Let us take, for example, the Far-field Location-aware subscribers experiment with the vCPUs of routers limited to the 20% of their capacity, whose results are reported in Figures 7.18 - 7.20.
Along with the latency and its standard deviation reduction reported respectively in Figure 7.24 and in FIgure 7.26, we observe convergence in the performance of individual encodings when compared to each other. Indeed, if the reduction in computational capacity highlights differences in both the Location-aware Base64 and Bloom Filter methods, the use of UDP eliminates these distinctions, lightening the message routing at the transport protocol level. Regarding throughput in Figure 7.25, as we already mentioned, the increased number of lost messages results in a general decrease in throughput.
MGRS encoding shows an interesting behavior. It is the lightest location-aware method we implement, and by removing the TCP overhead using UDP, it is able to keep a higher throughput compared with the other methods despite the reduced computational capacity and the network distance from the publisher. In fact, the shorter encoding combined with a lower number of target subscribers thanks to the location-aware mechanism, makes the MGRS encoding technique less susceptible to packet loss.

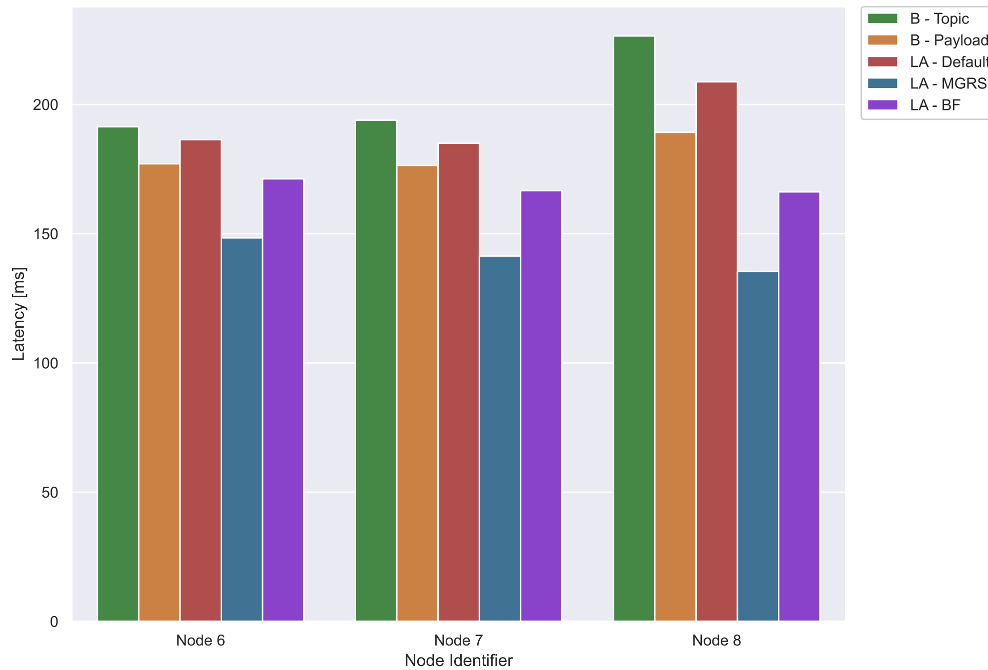


Figure 7.24: UDP far-field location-aware subscribers network: latency.

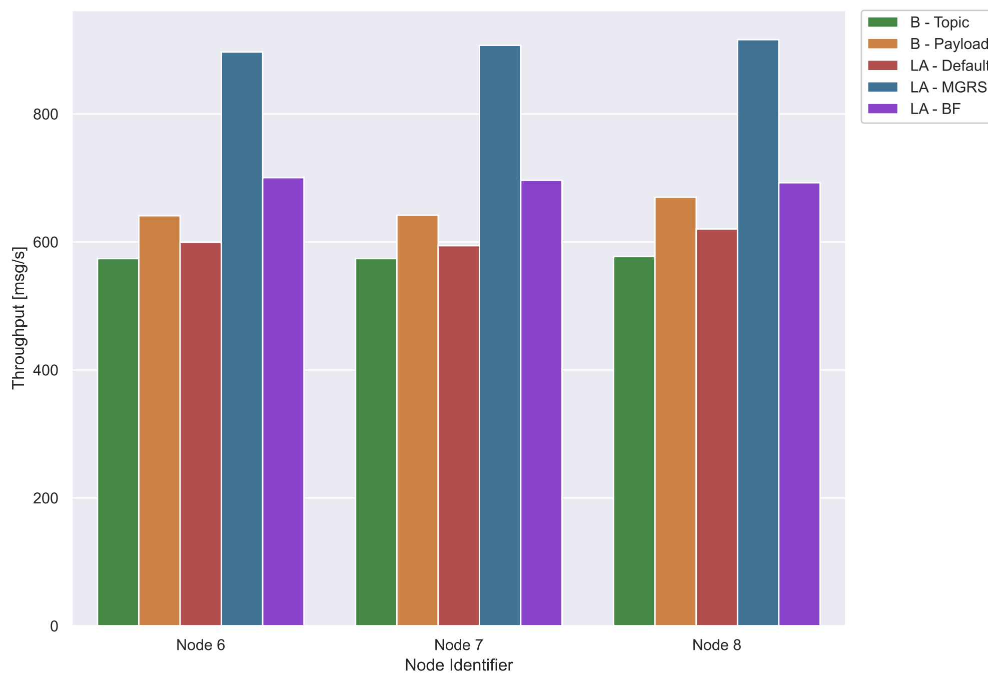


Figure 7.25: UDP far-field location-aware subscribers network: throughput.

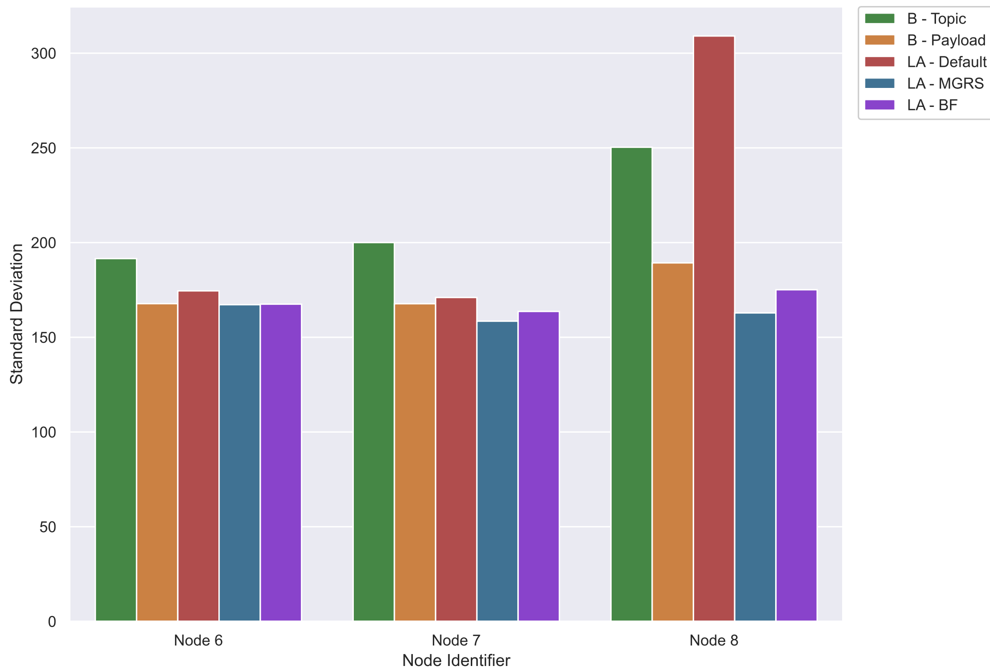


Figure 7.26: UDP far-field location-aware subscribers network: latency standard deviation

7.8. Bloom Filter Configuration

Up to now, we use a false positive probability of 25% for the Bloom Filter. We do not choose this percentage randomly but it comes from an additional test we conduct to determine the threshold beyond which no false positives occur within the defined domain. The false positive parameter together with the number of elements in the domain, as we explain in Section 5.3, contributes to defining the length of the bit-array that forms the Bloom Filter, impacting linearly on processing time.

7.8.1. Setup

In Figure 7.27 we report the domain within 6 subscribers configuration we use to conduct this test. Using a 5×5 units grid with a resolution of half a unit, we derive 100 elements for our testing domain.

In this test, the network topology, message rate, transport protocol, or computational capacity are not relevant, as our focus is to understand the threshold beyond which no false positives occur.

In practice, we configure a single publisher to send a unique message for each element in the domain. On the subscriber side, we record every message each subscriber receives within the identifier of the domain element. In each run, we incrementally increase the

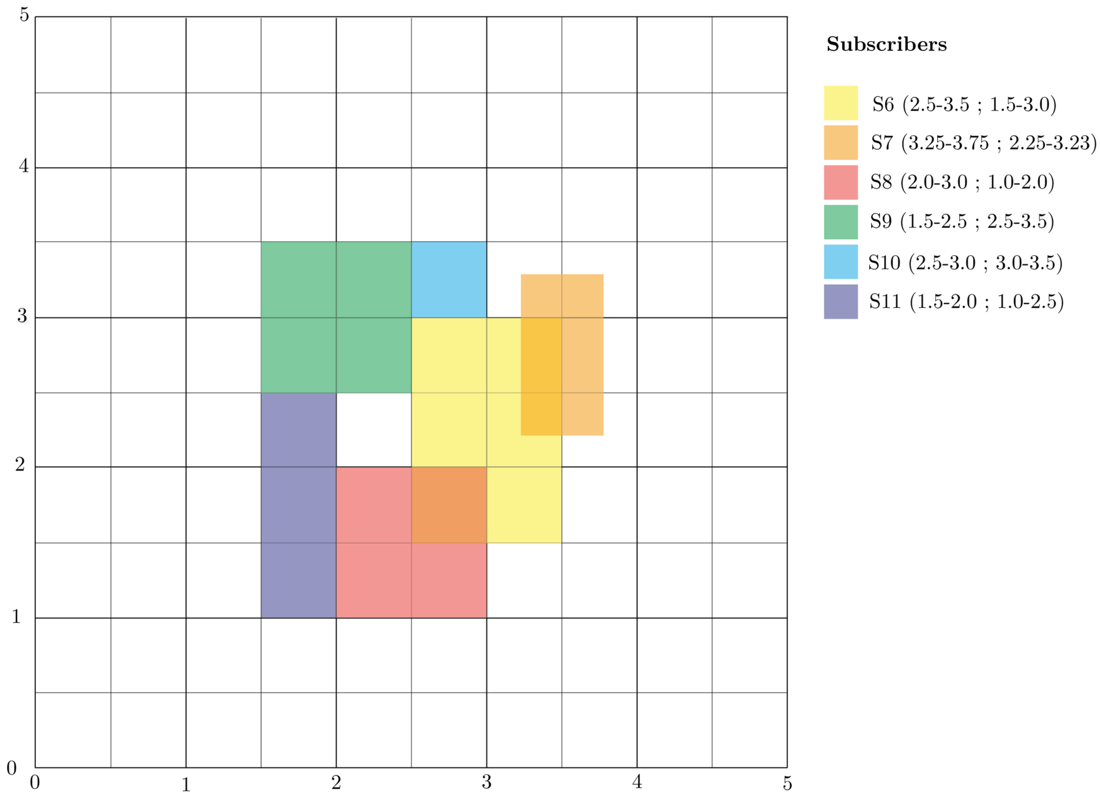


Figure 7.27: Bloom Filter test domain.

false positive percentage by 10%, observing, at the end of the test, the output the subscribers produce and comparing it with the expected results we report in Table 7.3, which represents the outcome of a correct execution.

Subscriber Identifier	**Expectation: correct execution outcome**
S6	6
S7	6
S8	4
S9	4
S10	1
S11	3

Table 7.3: Expected Results.

7.8.2. Results

In Table 7.4 we report the result we obtain.
Up to 20%, we do not encounter any false positives. Upon setting the percentage to 30%, we observe the first false positive occurring in S9, related to a point located at $(1.25, 0.75)$, which is outside the subscriber range $(1.5 - 2.5, 2.5 - 3.5)$.
To conduct further verification, we also perform the test with a 40% false positive probability. As expected, we obtain a total of 5 false positives recorded across different subscribers, indicating that as the percentage increases, once it surpasses the threshold, false positives also increase.
Finally, we also test 25% as a value, and after ensuring that it does not produce any false positives, we choose it as the value for our experiments.

False positive percentage	**Result**
1%	No false positives
10%	No false positives
20%	No false positives
25%	No false positives
30%	S9 registers 1 false positive
40%	S6 register 1 false positive, as well as S7 and S9. S11 registers 2 false positives.

Table 7.4: Obtained results.

To show how the false positive percentage impacts the Bloom Filter performances, in Figure 7.28 we present the latency chart derived from the Far-field Subscriber experiment with the Bloom Filter false positive probability set to 1%. The experiment shares the other configuration outlined in Section 7.5. From the chart, we observe that the latency associated with Bloom Filters is significantly higher, both compared to other methodologies and in comparison with the values we obtain with a 25% false positive probability, as seen in Figure 7.12. This difference is due to the length of the bit-array describing the filter. For the same number of domain elements, in the case of a 25% probability, the bit-array has a length of 320 bits, while at 1%, we obtain a length of 960 bits, which is much higher. Indeed, establishing an accurate false positive probability in the Bloom Filter encoding technique is a crucial factor for system performance, with potentially significant effects on message latency.

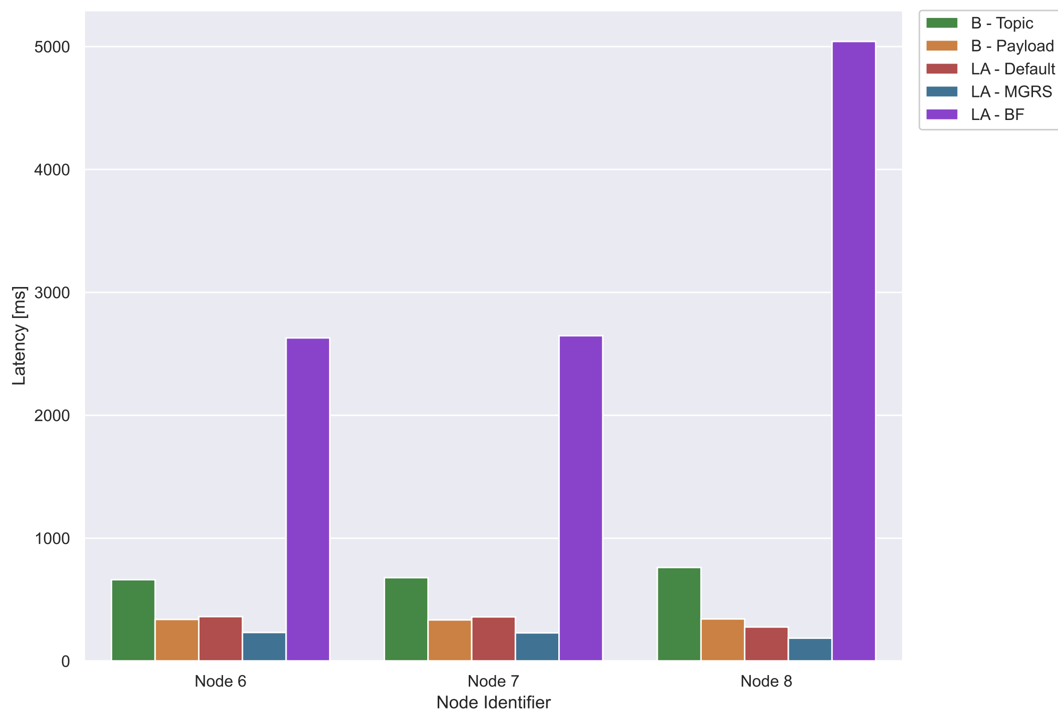


Figure 7.28: Far-field subscribers network latency with 1% false positive.

8 | Conclusion

In this thesis, we address the challenge of directly associating data with its location while transmitting messages. This proves valuable in large-scale IoT systems as it allows for linking message senders with interested receivers not solely based on the data type, which may be identical for multiple senders, but also by considering the location where the data is generated, maintaining the separation of these two aspects. This enables tasks like coordinating and controlling mobile robots in space, collecting data depending on geographical location, monitoring value changes for moving sensors, and various other applications where location is a valuable attribute.
We address this issue by enhancing the Zenoh protocol with location-aware capabilities. Since Zenoh operates as a topic-based Pub/Sub protocol, we opt to use a single topic level for conveying geographic information, encapsulating it within an identifiable key, in an encoded manner. The proposed approach allows us to alter the routing of messages from publishers to subscribers based on positional information, all without the necessity of internal router changes, requiring no more than 6 lines of code in the protocol's core and relocating the handling of positional information external to the routing module, injecting, only at the end, the location-based match result into Zenoh's topic matching interface. Furthermore, this approach ensures the preservation of the original matching interface, thereby maintaining compatibility with all devices using the native version of the protocol. In addition, our proposed solution lets us to develop multiple ways to encode information within the same key, providing different encoders each of them representing a trade-off between performance and expressiveness.
We introduce three encoding techniques for representing geographical data. The first method utilizes a full-featured Base64 representation of a JSON object, allowing us to depict complex areas with punctual precision. The MGRS encoding, on the other hand, employs a lighter approach in terms of both length and computational resources, at the expense of precision. Lastly, the Bloom Filter encoding enables us to approximate complex shapes and areas, introducing a probability of false positives during matching.

We assess our system using two key metrics: latency and throughput. We establish two baselines where messages are discarded upon reception. These baselines involve transfer-

ring location data initially through two topic levels and then via message payload. We test three distinct network topologies:

- **Zoning network:** This configuration represents an example of a network requiring location-awareness. We delineate two logically distinct geographical zones with the purpose of evaluating the system's performance by excluding irrelevant messages from a designated zone of interest. This experiment effectively generates a logically split network topology and it aims to evaluate a concrete location-aware scenario.
- **Near-field subscriber network:** This topology aims to evaluate the overhead that routers face in decoding and handling position-based matching. We strategically place subscribers close to the publisher, extracting differences in latency and throughput between location-aware techniques and the established baselines. Although this network does not simulate a location-aware scenario, we leverage it to comprehend the boundaries within which our solution can demonstrate effectiveness in diverse contexts.
- **Far-field subscriber network:** In this scenario, we evaluate the consequences of positioning an additional subscriber along the communication path between the publisher and the target subscriber. The objective is to determine whether optimizing routers by reducing outbound paths to only the essential ones, even with the added overhead of location-based matching, is compensated by the longer distance also in a non-location-based network. We utilize this network to gain insights into the trade-offs involved in such a configuration.

For each of the three network topologies, we conduct experiments employing either TCP or UDP as transport protocols. These experiments involve utilizing the complete computational capacity of the router as well as constraining it to operate at only 20% of its full capacity. Additionally, in the Near-field and Far-field experiments, we intentionally induce network congestion by limiting the network bandwidth and generating a high volume of messages per second. This deliberate congestion aims to stress the distributed system to underscore differences that might not be apparent in a well-performing network.

Based on the obtained results, we argue that location-aware techniques outperform baselines in the Zoning network. Location-aware encoding techniques result in a mean latency reduction of more than 50%. In fact, limiting message routing to the zone of interest avoids resource waste both in terms of computation and network traffic. Furthermore, in a large-scale system with numerous publishers and subscribers, location-awareness prevents message flooding events. In simpler terms, it helps avoid the unnecessary flow of a large number of messages throughout the entire network. However, in a mixed, non-

logically-split network, performance variations are evident. The Near-field experiment underscores the overhead associated with location-aware messages compared to the baselines. This overhead depends on the longer topic generated by the location key and the time required for decoding and processing the key. As expected, computing a second match, where there are no needs to filter messages based on location, leads to a latency increase, reaching up to 70% in the worst scenarios.
Furthermore, we assess the differences among the three location-aware encoding techniques. The Default method, owing to its complexity, exhibits inferior performance as it requires more processing time. In contrast, the MGRS encoding performs significantly better, gaining a 40% margin in latency, approaching the baselines due to its concise encoding and straightforward matching process. The Bloom Filter demonstrates an intermediate performance, featuring a longer representation offset by lightweight matching through bitwise operations. Additionally, the performance of Bloom Filter is influenced by the desired false positive probability, which significantly impacts the length of the location key.
Ultimately, the constrained computational capacity and the adoption of UDP as the transport protocol do not consistently alter the pattern among the five distinct techniques. The only noteworthy observation is the resilience of the MGRS encoding in congested, slow, and resource-constrained networks, rendering it suitable for such scenarios.

By separating the topic match from the location match, opportunities for improvement and extensions emerge, including:

- Investigating innovative and more efficient encoding techniques aimed at mitigating latency and augmenting system throughput. This involves a deeper examination of encoding methodologies that have the potential to optimize data processing in routers and reduce encoding length, contributing to improved overall performance.
- Exploring and integrating alternative approaches in describing location, introducing logical overlays. This entails going beyond conventional geographic-like coordinates and considering the incorporation of custom domain definitions across all encoding techniques. This exploration could open avenues for a more versatile and tailored representation of location information, enhancing the adaptability of the system to diverse scenarios and potentially improving its overall effectiveness in handling location-based data.

In summary, we have effectively integrated location-aware functionality into the Zenoh protocol, introducing the ability to subscribe using both topic and location in a novel, in-

dependent manner while maintaining compatibility with the original version of the protocol. Our implementation encompasses three encoding techniques, each tailored to specific properties, making them suitable for diverse situations. We assess the performance of our implementation across three distinct network topologies and four different configuration combinations, achieving a 50% reduction in latency in best scenarios.

Bibliography

[1] V. Antipov, O. Antipov, and A. Pylkin. Mobility support in publish/subscribe systems. *ITM Web of Conferences*, 6:03001, 2016. ISSN 2271-2097. doi: 10.1051/itmconf/20160603001. URL `http://www.itm-conferences.org/10.1051/itmconf/20160603001`.

[2] P. Bellavista, A. Corradi, and A. Reale. Quality of Service in Wide Scale Publish—Subscribe Systems. *IEEE Communications Surveys & Tutorials*, 16(3):1591–1616, 2014. ISSN 1553-877X. doi: 10.1109/SURV.2014.031914.00192. URL `http://ieeexplore.ieee.org/document/6803100/`.

[3] M. Bishop. Rfc 9114: Http/3, Jun 2022. URL `https://datatracker.ietf.org/doc/rfc9114/`.

[4] C. Bormann, A. P. Castellani, and Z. Shelby. CoAP: An application protocol for billions of tiny internet nodes. 16(2):62–67. ISSN 1089-7801. doi: 10.1109/MIC.2012.29. URL `http://ieeexplore.ieee.org/document/6159216/`.

[5] G. Cugola and J. Munoz de Cote. On Introducing Location Awareness in Publish-Subscribe Middleware. In *25th IEEE International Conference on Distributed Computing Systems Workshops*, pages 377–382, Columbus, OH, USA, 2005. IEEE. ISBN 978-0-7695-2328-6. doi: 10.1109/ICDCSW.2005.101. URL `http://ieeexplore.ieee.org/document/1437200/`.

[6] J. Dizdarevic and A. Jukan. Experimental Benchmarking of HTTP/QUIC Protocol in IoT Cloud/Edge Continuum. In *ICC 2021 - IEEE International Conference on Communications*, pages 1–6, Montreal, QC, Canada, June 2021. IEEE. ISBN 978-1-72817-122-7. doi: 10.1109/ICC42927.2021.9500675. URL `https://ieeexplore.ieee.org/document/9500675/`.

[7] P. Eugster. Type-based publish/subscribe: Concepts and experiences. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(1):6–es, 2007.

[8] M. Hazas, J. Scott, and J. Krumm. Location-aware computing comes of age. 37(2):

95–97. ISSN 0018-9162. doi: 10.1109/MC.2004.1266301. URL `http://ieeexplore.ieee.org/document/1266301/`.

[9] U. Hunkeler, H. L. Truong, and A. Stanford-Clark. MQTT-S — A publish/subscribe protocol for Wireless Sensor Networks. In *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE '08)*, pages 791–798, Bangalore, India, Jan. 2008. IEEE. ISBN 978-1-4244-1796-4. doi: 10.1109/COMSWA.2008.4554519. URL `http://ieeexplore.ieee.org/document/4554519/`.

[10] S. Khare and M. Totaro. Big data in iot. In *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1–7. IEEE, 2019.

[11] R. B. Langley. The utm grid system. *GPS world*, 9(2):46–50, 1998.

[12] W.-Y. Liang, Y. Yuan, and H.-J. Lin. A Performance Study on the Throughput and Latency of Zenoh, MQTT, Kafka, and DDS.

[13] J. E. Luzuriaga, M. Perez, P. Boronat, J. C. Cano, C. Calafate, and P. Manzoni. A comparative evaluation of AMQP and MQTT protocols over unstable and mobile networks. In *2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC)*, pages 931–936, Las Vegas, NV, USA, Jan. 2015. IEEE. ISBN 978-1-4799-6390-4. doi: 10.1109/CCNC.2015.7158101. URL `http://ieeexplore.ieee.org/document/7158101/`.

[14] M. Milošević, V. Mladenović, and U. Pešović. Evaluation of HTTP/3 Protocol for Internet of Things and Fog Computing Scenarios. *Studies in Informatics and Control*, 30(3):75–84, Sept. 2021. ISSN 12201766, 1841429X. doi: 10.24846/v30i3y202107.

[15] N. Naik. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In *2017 IEEE International Systems Engineering Symposium (ISSE)*, pages 1–7, Vienna, Austria, Oct. 2017. IEEE. ISBN 978-1-5386-3403-5. doi: 10.1109/SysEng.2017.8088251. URL `http://ieeexplore.ieee.org/document/8088251/`.

[16] A. Prajapati. Amqp and beyond. In *2021 International Conference on Smart Applications, Communications and Networking (SmartNets)*, pages 1–6, 2021. doi: 10.1109/SmartNets50376.2021.9555419.

[17] A. Rawat and A. Pandey. Recent trends in iot: A review. *Journal of Management and Service Science (JMSS)*, 2(2):1–12, 2022.

[18] T. R. Sheltami, A. A. Al-Roubaiey, and A. S. H. Mahmoud. A survey on developing publish/subscribe middleware over wireless sensor/actuator networks. *Wireless Networks*, 22(6):2049–2070, Aug. 2016. ISSN 1022-0038, 1572-8196. doi: 10.1007/s11276-015-1075-0. URL http://link.springer.com/10.1007/s11276-015-1075-0.

[19] D. Soni and A. Makwana. A SURVEY ON MQTT: A PROTOCOL OF INTERNET OF THINGS(IOT).

[20] S. Tarkoma and J. Kangasharju. Mobility and completeness in publish/subscribe topologies. In *IASTED International Conference on Networks and Communication Systems, ACTA Press*. Citeseer, 2005.

[21] T. Z. Team. Zenoh overhead: a story from our community. https://zenoh.io/blog/2021-07-05-zenoh-overhead/. [Accessed 31-May-2023].

[22] T. Z. Team. Zenoh reliability, scalability and congestion control. https://zenoh.io/blog/2021-06-14-zenoh-reliability/, 2021. [Accessed 31-May-2023].

[23] T. Z. Team. Taming the dragon: Get started with zenoh. https://www.linkedin.com/smart-links/AQEuTXQcy3TLag/6f6c5594-3452-4572-9f92-26e78098c9d5, 2022. [Accessed 28-Feb-2023].

[24] R. I. Vitanov and D. N. Nikolov. A state-of-the-art review of ultra-wideband localization. In *2022 57th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST)*, pages 1–4. IEEE. ISBN 978-1-66548-500-5. doi: 10.1109/ICEST55168.2022.9828723. URL https://ieeexplore.ieee.org/document/9828723/.

[25] X. Xiong and J. Fu. Active status certificate publish and subscribe based on amqp. In *2011 International Conference on Computational and Information Sciences*, pages 725–728, 2011. doi: 10.1109/ICCIS.2011.63.

[26] ZettaScale. Zenoh - the zero overhead, pub/sub, store, query, and compute protocol., . URL https://zenoh.io.

[27] ZettaScale. Zettascale product: Zenoh, . URL https://www.zettascale.tech/product/zenoh.

List of Figures

1.1 High level architecture. 3

2.1 Pub/Sub message pattern. 10
2.2 Pub/Sub network topology 11
2.3 MQTT-S gateways 14
2.4 Connection establishment differences in HTTP. 18
2.5 Zenoh network topologies. 21

3.1 Example of data interest in Milano areas. 28
3.2 Location as a topic. 30
3.3 Location in the payload. 31
3.4 Zenoh location awareness example. 35
3.5 Zenoh location awareness example with explicit key. 35

4.1 Zenoh location-aware intersection example. 42

5.1 MGRS Example. 49
5.2 MGRS matching example. 51
5.3 Insertion on bit-array. 52
5.4 Query on Bloom filter. 53
5.5 Bloom filter domain definition. 54
5.6 Example of Bloom filter domain representation. 55
5.7 Query on Bloom filter in Zenoh router. 56

6.1 Zenoh internals structure. 60
6.2 General structure of the Location Key. 61
6.3 Zenoh REST Flow. 70

7.1 Zoning network topology. 77
7.2 Zoned network: latency. 78
7.3 Zoned network: throughput. 79
7.4 Zoned network: latency standard deviation. 80

7.5 Near-field location-aware subscribers topology. 81
7.6 Near-field location-aware subscribers network: latency. 82
7.7 Latency zoom on location-aware methods. 83
7.8 Near-field location-aware subscribers network: throughput. 84
7.9 Near-field location-aware subscribers network: latency standard deviation. 84
7.10 Base64 location-aware single experiment latency on node 6. 85
7.11 Far-field location-aware subscribers topology. 86
7.12 Far-field location-aware subscribers network: latency. 87
7.13 Far-field location-aware subscribers network: throughput. 88
7.14 Far-field location-aware subscribers network: latency standard deviation. . 88
7.15 Near-field location-aware subscribers network: CPU 20% latency. 89
7.16 Near-field location-aware subscribers network: CPU 20% throughput. . . . 90
7.17 Near-field Location-aware subscribers network: CPU 20% latency standard deviation. 90
7.18 Far-field location-aware subscribers network: CPU 20% Latency. 91
7.19 Far-field location-aware subscribers network: CPU 20% throughput. 92
7.20 Far-field location-aware subscribers network: CPU 20% latency standard deviation. 92
7.21 UDP near-field location-aware subscribers network: latency. 94
7.22 UDP near-field location-aware subscribers network: throughput. 94
7.23 UDP near-field location-aware subscribers network: latency standard deviation. 95
7.24 UDP far-field location-aware subscribers network: latency. 96
7.25 UDP far-field location-aware subscribers network: throughput. 96
7.26 UDP far-field location-aware subscribers network: latency standard deviation. 97
7.27 Bloom Filter test domain. 98
7.28 Far-field subscribers network latency with 1% false positive. 100

List of Tables

2.1 MQTT properties. 12
2.2 AMQP properties. 15
2.3 HTTP properties. 17
2.4 CoAP properties. 19
2.5 Protocol comparison. 24

4.1 Publisher characteristics. 38
4.2 Subscriber characteristics. 38

7.1 ProxMox node configuration. 71
7.2 Results Overview. 75
7.3 Expected Results. 98
7.4 Obtained results. 99