UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO

Daniel Lombardi de Oliveira

OPTIMIZING CLOUD IOT APPLICATIONS FOR SCALABILITY: LEVERAGING DISTRIBUTED QUEUES FOR SEAMLESS OPERATIONS

SÃO CARLOS - SP

2025

Daniel Lombardi de Oliveira

OPTIMIZING CLOUD IOT APPLICATIONS FOR SCALABILITY: LEVERAGING DISTRIBUTED QUEUES FOR SEAMLESS OPERATIONS

Course Conclusion Work presented to the Computer Engineering course at the Federal University of São Carlos, for the purpose of obtaining the Bachelor's degree in Computer Engineering.

Advisor: Professor Dr. Fredy João Valente

SÃO CARLOS - SP

2025

To everyone that supported me in the journey of seeking my passion, even the ones that get on my nerves sometimes.

ACKNOWLEDGEMENTS

To Fredy Jõao Valente for giving me the chance to perform at my best and propel my studies in distributed systems.

To all my other mentors during my undergraduate years.

To my parents, Ana and Cesar, and to everyone else in my family who supported my education. A special thank you to my sister, Natalia, for her great encouragement throughout my programming journey.

To my girlfriend, Letícia, for all her support and affection, as well as the patience during the tumultuous and often chaotic journey of my undergraduate years.

To all my friends for their companionship and late night talks.

To PATOS, for giving me a platform to talk about what I love.

“The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair.”

(Douglas Adams)

ABSTRACT

The rapid expansion of the Internet of Things (IoT) has necessitated the development of robust and scalable cloud applications to manage the vast influx of data and ensure reliable communication between devices. This work presents a comprehensive study backed by the practical experience gained during the development of large-scale networks with IoT actuators. We begin by exploring the unique challenges posed by IoT ecosystems, including real-time processing implementations and scaling for cloud applications and their difficulties. The study delves into the utilization of in-memory databases, queues and RPC methods to act as shared-memory, creating a vastly scalable distributed system that enables the utilization of traditional scaling methodologies for large MQTT based IoT environments.

Keywords: IoT, Distributed Systems, Publish/Subscribe, Distributed Architecture, RPC, MQTT, ESL, Signal Strength-based routing.

FIGURES LIST

Figure 1: Message Queue example..... 15
Figure 2: RPC Example.....16
Figure 3: System Simplified Architecture.....21
Figure 4: Pseudo-code for Routing Manager..... 23
Figure 5: RPC via HTTP.....24
Figure 6: Simplified Sequence Diagram of the Architecture..... 25
Figure 7: Round Robin demonstration of MQTT message delivery.....26
Figure 8: Sequence Diagram of RPC HTTP Implementation..... 27
Figure 9: RPC via Distributed Queues..... 29
Figure 10: Complete Architecture with Custom Router Backbone..... 29
Figure 11: Sequence Diagram of the FWD Router..... 30
Figure 12: Sequence Diagram of the BCK Router..... 30
Figure 13: Pseudo-code for task_worker.....31
Figure 14: Pseudo-code for fwd_router..... 32
Figure 15: Pseudo-code for bck_router..... 32
Figure 16: Pseudo-code for task_worker with telemetry..... 33
Figure 17: Histogram of the Baseline Backbone with 1 router replica for 1k tasks.....35
Figure 18: Histogram of the Custom Backbone with 1 router replica for 1k tasks.....36
Figure 19: Histogram of the Baseline Backbone with 2 router replicas for 1k tasks..... 37
Figure 20: Histogram of the Custom Backbone with 2 router replicas for 1k tasks..... 38
Figure 21: Histogram of the Custom Backbone with 2 router replicas for 10k tasks..... 39
Figure 22: Histogram of the Custom Backbone, 20 router replicas for 10k tasks.....39
Figure 23: Histogram of the Custom Backbone, 200 router replicas for 10k tasks.....40

SUMMARY

Optimizing IoT Cloud Applications for Scalability: Leveraging RPC with Distributed Queues for Seamless Operations..... 0
ACKNOWLEDGEMENTS..... 2
ABSTRACT..... 4
FIGURE LIST..... 5
SUMMARY..... 6
INTRODUCTION..... 8
OBJECTIVES..... 10
General Objective..... 10
Specific Objectives..... 10
Expected Results..... 10
THEORETICAL BASIS..... 11
The Internet of Things Ecosystem..... 11
Scalability..... 12
Cloud Computing Fundamentals..... 13
Message Queues and Asynchronous Communication..... 14
Remote Procedure Call (RPC)..... 15
In-Memory Data Stores / Databases..... 15
Kubernetes (and Container Orchestration)..... 16
METHODOLOGY..... 17
Flexible Scalability..... 17
Testing Methodologies..... 17
Quantitative Metrics..... 18
RPC Latency..... 18
Throughput..... 18
Qualitative Observations..... 19
DEVELOPMENT..... 20
General Architecture..... 20
Task Queue..... 20
Business Layer (Task Worker)..... 21
Communication Layer (Router)..... 21
Communication Backbone: RPC via HTTP..... 22
Limitations of this system..... 23
Communication Backbone: RPC via Distributed Queues..... 26
HOMOLOGATION..... 31
Functional Tests..... 32
Performance Tests..... 32

RESULTS.....32
Functional Tests.....32
Performance Tests.....33
CONCLUSIONS.....38
FUTURE WORKS.....39
REFERENCES.....40

INTRODUCTION

The last decade has witnessed a revolution driven by the Internet of Things (IoT), transforming how we interact with the physical and digital worlds. The proliferation of connected devices, from environmental sensors to industrial actuators, has generated an unprecedented volume of data, opening new frontiers for automation, process optimization, and intelligent decision-making. (MOTTA, R. et al. 2019)

This exponential growth in connected devices and the ensuing data deluge inherently necessitate robust and scalable underlying infrastructure. Cloud computing has emerged as the indispensable backbone for IoT ecosystems, providing the computational power, storage capabilities, and network resources required to ingest, process, analyze, and act upon this vast influx of information. However, the sheer volume, velocity, and variety of data generated by millions, or even billions, of distributed IoT devices present significant challenges for traditional cloud application architectures. Ensuring real-time processing, maintaining low latency, and achieving horizontal scalability to accommodate dynamic workloads become critical factors for the successful deployment and operation of large-scale IoT solutions, especially when considering IoT actuators.

Unlike traditional IoT sensors, which primarily collect and transmit data, IoT actuators introduce a new layer of complexity to these distributed systems. These devices are designed to execute physical actions or trigger processes based on received commands, effectively closing the loop between the digital and physical worlds. Their integration demands not only robust data ingestion and processing capabilities but also rely on components where faulty connections are much more common than on traditional systems. This new paradigm of bidirectional communication and real-time control places unprecedented demands on the scalability, reliability, and architectural design of cloud-based IoT applications, often extending beyond the capabilities of pre-existing system designs focused solely on data collection.

Given the already specific demands of IoT applications, we must also consider the heterogeneity of tasks to be executed. Depending on the type of IoT device and its function, the data and processing volume can be highly heterogeneous throughout the day. For instance, a smart lighting system might have minimal activity during daylight hours but experience peak command and telemetry traffic at dusk and dawn. Similarly, industrial sensors might burst data during specific operational cycles.

To effectively manage these fluctuating and diverse workloads, cloud computing providers rely on vast data center infrastructures with virtualized resources. They are made available to clients on a pay-as-you-go model, where customers pay according to their resource consumption, this is often offered as managed components such as Kubernetes clusters, load balancers and databases, constituting what is known as Infrastructure as a Service (IaaS). This model's inherent elasticity, particularly the horizontal scalability offered by platforms like Kubernetes, allows for the dynamic allocation and deallocation of resources in response to fluctuating demand, ensuring optimal performance during peak loads while minimizing costs during periods of low activity, making it ideal for modern IoT systems. (POURMAJIDI, W. et al. 2017).

Drawing from practical experience, our study examines a core problem arising from the implementation of an IoT solution in a hardware store, emphasizing the difficulties in achieving

real-time processing and effective scaling in a cloud scenario. This problem became acutely evident during the development and deployment of Electronic Shelf Labels (ESLs) in a large-scale hardware store chain in Brazil. This particular implementation presented unique challenges, given the potential for deploying around 50 thousand ESLs per store, across dozens of stores nationwide, leading to a vast IoT ecosystem. We'll dive into the scaling limitations of traditional broker architectures as applied to our specific use case, particularly when managing the bidirectional communication and real-time control demands of large-scale Electronic Shelf Label (ESL) deployments. This includes a detailed examination of how the MQTT protocol, while effective for general load balancing, encounters significant hurdles in maintaining the request-reply pattern necessary for Remote Procedure Calls (RPCs) when horizontal scaling is implemented, as replies can be misrouted to different nodes than their originating requests. We propose and evaluate a novel approach leveraging distributed queues to overcome these architectural constraints, ensuring seamless, scalable, and reliable operations for IoT cloud applications involving actuators.

OBJECTIVES

General Objective

The main objective of this work is to create a Minimum Viable Product (MVP) of a communication backbone for large-scale IoT device networks, being implemented and tested with the use of Electronic Shelf Labels (ESLs) in a large hardware-store chain in Brazil.

Specific Objectives

1. Analyze and define project requirements, identifying the main pitfalls and downsides of the pre-existing architectures and traditional models of communication with such devices.
2. Elaborate an adequate architecture that allows for high elasticity in terms of horizontal scaling.
3. Build the MVP.
4. Test and assess the proposed architecture with consideration for performance benchmarks and also a qualitative analysis.

Expected Results

1. A well-defined and robust architecture that prioritizes horizontal scalability, demonstrating high elasticity to accommodate varying loads throughout the day.
2. Functional Minimum Viable Product (MVP): A working MVP that embodies the core functionalities of the proposed solution and serves as a tangible demonstration of the architectural design.
3. Validated Performance and Quality: Quantitative data from performance benchmarks and a qualitative analysis confirming the efficacy and efficiency of the proposed architecture. This will include insights into its strengths and areas for further optimization, ensuring it meets or exceeds predefined performance criteria.

On top of this, it is expected that this project can be used as a base and reference for future projects that demand a quickly deployable IoT communication backbone with minimal configuration. The developed example of ESLs can quickly be adapted to other product niches.

THEORETICAL BASIS

This chapter will define the core concepts necessary to understand our solution, including the nuances of IoT ecosystems, the principles of cloud scalability, and the challenges of implementing Remote Procedure Calls (RPCs) over the MQTT protocol in horizontally scaled environments. It will further elaborate on how distributed queues and in-memory data stores are critical for overcoming these challenges, enabling seamless and reliable operations for large-scale IoT applications. This foundational knowledge is essential for appreciating the architectural design presented in this thesis. We will also present a quick analysis of used tools and technologies.

The Internet of Things Ecosystem

As defined by Atzori et al., Internet of Things can be defined as three main paradigms: internet-oriented (middleware), things oriented (sensors) and semantic-oriented (knowledge). It is a network of physical objects embedded with technologies such as sensors, actuators and software that allows it to connect to external systems.

Sensors are devices that act as an intake point of data, they measure physical phenomena (i.e. temperature) and convert them to digital data. Actuators are devices that receive commands from an external control system and converts them into physical actions (i.e. switching a light, changing a value etc.), they allow us to close the loop between the digital and physical worlds, commonly requiring bidirectional control to acknowledge the commands. Our work focuses on the latter. Both sensors and actuators commonly operate via a GW (gateway) that act as a middleman from our servers to the devices (this allows us to have use different protocols such as bluetooth while still connecting them via other protocols to the servers, such as MQTT).

Considering the particularities of IoT systems and Systems Engineering, there is a clear and pressing need for advanced development in this field, especially when considering large-scale applications. IoT systems inherently involve a vast array of interconnected components, ranging from sensors and actuators to complex cloud platforms and data analytics engines. This heterogeneity, combined with the need for real-time processing in large-scale environments, presents unique and new challenges for Software Engineering.

A significant hurdle in advancing this area is the current lack of comprehensive bibliography and standardized methodologies specifically tailored for large-scale IoT systems engineering. While there's a growing body of work on individual IoT technologies and general systems engineering, dedicated literature on the systematic design, development, and deployment of IoT at scale remains scarce. This gap creates a void in best practices, reference architectures, and proven approaches for tackling issues like managing interconnected "systems of systems" where constituent elements may have their own independent life cycles and stakeholders. Consequently, organizations embarking on large-scale IoT initiatives often find themselves navigating uncharted territory, relying on ad-hoc solutions or adapting frameworks from other domains, which may not fully capture the nuances of IoT's distributed, event-driven, and often highly autonomous nature. Filling this bibliographic void is paramount to establishing a robust baseline for development, fostering knowledge sharing, and accelerating the successful realization of ambitious, impactful IoT applications (MOTTA, R. et al. 2019).

As such, this study shares a proven, practical method for developing and deploying large-scale IoT networks that communicate via MQTT following a scalable request-reply RPC model for IoT architectures.

Scalability

Scalability is a primary concern for any robust system, especially in the context of dynamic environments like the Internet of Things (IoT). As client loads increase, whether from a surge in active users, a higher volume of data processing, or more frequent device interaction, a system must demonstrate its ability to cope with these elevated demands. This is not a binary state, but rather a spectrum defined by how effectively computing resources can be added to maintain performance as the "load parameters" (e.g., requests per second, read-to-write ratios, number of concurrent connections) evolve. The challenge often lies not in handling individual operations, but in managing the "fan-out" effect, as seen in the famous Twitter Timeline (KRIKORIAN R. 2012) example where a single tweet from a celebrity needs to be delivered to millions of followers, creating a vastly multiplied write burden, requiring special case for some users that would appen non-ideal in other scenarios. Transforming the multiple write burden in simple cache reads, combining the two methods of writing to each user's timeline and having dedicated caches for users with a large number of followers.

To address these escalating demands, systems typically employ two fundamental scaling strategies: vertical and horizontal. Vertical scaling, or "scaling up," involves enhancing the capabilities of a single machine by adding more CPU, RAM, or storage. While straightforward, this approach is severely limited by the physical constraints of hardware and the exponential cost associated with high-end, specialized servers. Crucially, it also introduces a single point of failure, meaning that the entire system's availability hinges on that one machine, which is unacceptable for large-scale, mission-critical applications where continuous operation is a must.

Conversely, horizontal scaling, or "scaling out," offers a more adaptable and resilient solution by distributing the workload across multiple machines within a cluster. This method provides virtually limitless expansion potential; as load increases, more commodity servers can be added to the pool, and a load balancer can distribute incoming requests among them. This not only inherently builds in fault tolerance—the failure of one node does not incapacitate the entire system—but also aligns perfectly with the "pay-as-you-grow" model prevalent in cloud computing. Technologies like Kubernetes exemplify this, dynamically adjusting the number of running application instances (pods) and even the underlying infrastructure nodes to match fluctuating demand, thereby optimizing resource utilization and ensuring continuous performance while managing costs effectively. This can also be referred to as increasing the number of replicas of a deployment or system.

However, the advantages of horizontal scaling come with their own set of architectural complexities. Ensuring data consistency across numerous distributed nodes, especially for stateful applications, requires careful design and the implementation of sophisticated mechanisms like distributed queues (e.g., RabbitMQ, Apache Kafka), which manage message streams asynchronously and reliably even if individual components fail. Furthermore, the operational overhead of managing a large, distributed cluster necessitates advanced tools for monitoring, deployment, and orchestration, while developers must design "cloud-native"

applications that are inherently suited to a distributed, stateless, or externally-state-managed environment to truly leverage horizontal scalability.

Cloud Computing Fundamentals

Cloud computing, as defined by the National Institute of Standards and Technology (NIST), is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. Essentially, it provides the on-demand availability of computer system resources, particularly data storage and computing power, without direct active management by the user. This paradigm has revolutionized how organizations deploy and manage their IT infrastructure, offering unprecedented flexibility, efficiency and security. Cloud services are offen offered as: Infrastructure as a Service (IaaS), Platform as a Service (Paas) and SaaS (System as a Service).

Having the flexibility and using PaaS to develop with Kubernetes truly unlocks powerful capabilities, especially when it comes to harnessing the core cloud benefit of elasticity. Elasticity in cloud computing refers to the ability of a system to rapidly and automatically adjust its computing resources (like CPU, memory, storage, network bandwidth and node count) to match fluctuating workloads and demands without human intervention. This dynamic scaling up or down ensures optimal performance and cost efficiency. This is particularly important when considering heterogeneous loads in IoT systems like the ESL described previously.

Scalability refers to a system's ability to receive more resources to increase its performance and capacity to handle demands. There are two basic types of scalability: vertical and horizontal (AAQIB, S, 2019).

Vertical Scaling (Scale-Up) is increasing the capacity of a single machine (i.e. adding more RAM, CPU). For truly large-scale systems, this is a great limiter when considering simple physical limits as to how many RAM slots or CPU sockets a machine has. Furthermore, a single powerful machine represents a single point of failure, meaning any outage takes the entire system offline.

Horizontal Scaling (Scale-Out) is adding more machines to a computing cluster. This approach overcomes the limitations of vertical scaling by allowing for virtually limitless expansion. This not only provides superior fault tolerance – if one server fails, the others can continue operating – but also enables a "pay-as-you-grow" model, where resources are added incrementally as demand increases, leading to more efficient resource utilization and significant cost savings over time. Modern cloud architectures and technologies like Kubernetes are built fundamentally around horizontal scalability, enabling applications to dynamically scale out and in as demand fluctuates, ensuring high availability and optimal performance.

Message Queues and Asynchronous Communication

Message Brokers (Queues) are essentially a kind of database that is optimized for handling message streams (GRAY, J. 1995). The Queue is the server, where producers and consumers connect to it as clients. Producers write messages to the broker, and the consumers recieve them. They allow easy communication between parties and act as a load balancer and buffer, they handle bursts of traffic by storing messages, preventing consumers from being overwhelmed.

Queues store messages (tasks to be processed) until workers (consumers) are ready for them in a First-In, First-Out (FIFO) matter. This also means that consumers and producers can operate independently, without direct contact or knowledge of each other.

With the decoupling of producers and consumers, this means the tasks can be executed completely independent of when it was generated. This is called Asynchronous Communication. The producer only waits for the broker to confirm that it has buffered the message, not waiting for it to be executed. The delivery to consumers will happen at some undetermined future point in time—often within a fraction of a second, but sometimes significantly later if there is a queue backlog (KLEPPMANN, M. 2007).

The Advanced Message Queuing Protocol (AMQP) is an open standard application-layer protocol for message-oriented middleware. Unlike proprietary messaging systems, AMQP provides a vendor-neutral, interoperable framework for reliable, asynchronous communication between applications. It defines the mechanics for securely and efficiently transferring messages, often through a broker-based architecture where producers send messages to an intermediary (the broker), which then routes them to consumers. Key features include message orientation, queuing, flexible routing, and robust delivery guarantees (like at-least-once or at-most-once delivery), ensuring messages reach their intended recipients even if systems are temporarily unavailable.

Message Queuing Telemetry Transport (MQTT) is a lightweight, publish-subscribe messaging protocol designed for constrained devices and unreliable networks, making it ideal for the Internet of Things (IoT). Instead of devices communicating directly, they connect to a central MQTT broker. Devices that want to send data become publishers, sending messages to specific "topics" on the broker. Devices that want to receive data become subscribers, indicating interest in certain topics. The broker then efficiently delivers messages from publishers to all interested subscribers, decoupling senders from receivers and allowing for highly scalable and asynchronous communication. For load balancing, MQTT offers the round-robin algorithm for delivering messages inside listeners of a specific topic.

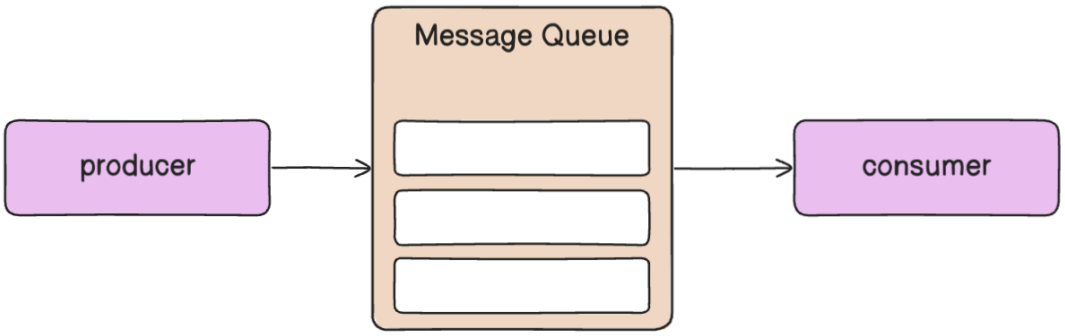

Figure 1: Message Queue example - Source: Author (eraser.io)

Remote Procedure Call (RPC)

A Remote Procedure Call (RPC) is a proposal to allow programs to call procedures located on other machines, with the message passing part of it not being visible to the programmer. The main idea is that a process that may be complex is called externally from the main machine, allowing the programmer to not care about the address spaces, parameters and results marshalling. In the context of IoT, a common use case for an RPC is for when an actuator is called to execute a task. The client (programmers code) calls the server (IoT system) to toggle a light switch, and then replying if the task was successful or not.

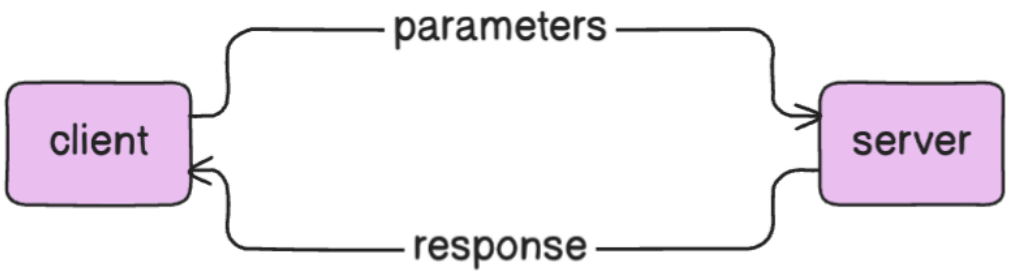

Figure 2: RPC example - Source: Author (eraser.io)

In-Memory Data Stores / Databases

In-memory datastores like Redis play a crucial role in modern distributed systems, particularly within the IoT ecosystem, by offering extremely fast data access and manipulation. Unlike traditional disk-based databases, Redis stores its data set primarily in RAM, significantly reducing latency for read and write operations, which is critical for real-time applications and rapidly changing data that doesn't require long-term persistence. This allows for rapid retrieval and updates of frequently accessed, volatile information, directly enhancing the responsiveness and efficiency of control systems or RPC calls by avoiding slower disk I/O. Its ability to handle high-throughput, low-latency operations for temporary data makes in-memory datastores invaluable for managing the vast and often dynamic information generated by IoT devices. Redis in particular stores data in a key-value storage model, a non-relational database that stores data as a collection of unique keys, each paired with an associated value, offering a simple yet highly efficient model for data retrieval. The inherent simplicity of the key-value model, coupled with Redis's in-memory architecture, makes it highly amenable to horizontal scalability through sharding or clustering, where the dataset is partitioned and distributed across multiple Redis instances. This allows for near-linear scaling of throughput and capacity by simply adding more nodes, ensuring that as the volume of ephemeral data or the number of read/write operations grows, the system can expand to meet demand without hitting the limitations of a single machine.

Kubernetes (and Container Orchestration)

Kubernetes is an open-source container orchestration platform that inherently delivers high scalability and elasticity for applications. It achieves this by managing containerized workloads and services, abstracting away the underlying infrastructure. Its core features, like the Horizontal Pod Autoscaler (HPA), automatically adjust the number of running application instances (pods) based on metrics such as CPU utilization or custom application-specific demands, ensuring that enough capacity is always available to handle fluctuating user traffic without manual intervention. Complementing this, the Cluster Autoscaler works at a lower level, dynamically adding or removing worker nodes (the virtual machines that run the pods) from the underlying cloud provider's infrastructure. This intelligent, automated scaling ensures optimal resource utilization, as the system can rapidly expand during peak loads and contract during off-peak periods, directly translating to cost efficiency by only paying for the resources actively consumed, making it ideal for the dynamic and often unpredictable workloads found in large-scale IoT systems.

METHODOLOGY

Considering the importance and challenges of achieving true scalability in large distributed IoT systems, this chapter describes the development of a new method of communication for RPC for IoT Cloud Applications, one that leverages the elasticity of cloud environments with kubernetes to achieve true horizontal scalability in an MQTT IoT actuator network. This will be called our Communication Backbone.

On top of that, we also discuss methodologies to test, measure and validate the proposed system, as well as comparing it to the de facto method for such cases, both in scalability as well as complexity of configuration and Time-to-Deploy/Onboard new devices.

Flexible Scalability

The dynamic nature of large-scale IoT deployments, where command and data volumes can shift dramatically throughout the day, renders static, peak-capacity provisioning economically unfeasible and technically inefficient. Such an approach inevitably leads to substantial resource waste during periods of low activity and eventual unavailability on partial system failures. To overcome this, our Communication Backbone is fundamentally designed for flexible scalability.

This means the system is built to intelligently and automatically adapt to match the actual, real-time demand. By designing the architecture with a Cloud-Native solution in mind, we leverage many toolings that inherently provide the elasticity and resilience required for dynamic IoT environments. This approach allows our system to seamlessly scale resources up or down, ensuring optimal performance during peak loads and maximizing cost-efficiency during periods of lower activity (INDRASIRI, K. SUHOTHAYAN S. 2021).

Testing Methodologies

To rigorously evaluate the proposed Communication Backbone, we will employ a comprehensive testing strategy that combines both quantitative performance measurements and qualitative operational observations. A core tenet of our methodology is the isolation of the Communication Backbone's performance by emulating external dependencies. This means that both the IoT Gateways (and, by extension, the ESLs they manage) and the task producers will be simulated. This controlled environment allows us to specifically assess the impact and capabilities of our solution, minimizing external variability and noise.

The comparison will take in count both Quantitative Metrics and Qualitative Observations. For the Quantitative Metrics, we'll dive into the RPC Latency and the Throughput. For the Qualitative Observations, we will consider the complexity of deploying and managing the systems, ease of debugging and the operational overhead. For the experiments, we'll take into consideration the case where the system has reached its operating limits on each node, and see how it's elasticity for horizontal scalability.

All testing will be conducted within Docker containers (DOCKER, 2025). This approach offers several critical advantages:

- Reproducibility: Docker ensures that the testing environment is consistent across all runs, eliminating "it worked on my machine" scenarios and guaranteeing that results are comparable.
- Isolation: Each component of our test setup (e.g., emulated gateways, task producers, the Communication Backbone itself) can run in its own isolated container, preventing interference and simplifying debugging.
- Scalability of the Test Environment: Docker allows for easy scaling of the test infrastructure itself. We can quickly spin up numerous instances of emulated gateways and task producers to simulate large-scale scenarios without needing extensive physical hardware.
- Efficiency: Containerization reduces setup time and overhead, enabling rapid iteration of tests and scenarios.

Our evaluation will focus on the following key aspects:

Quantitative Metrics

RPC Latency

This is a critical measure of the responsiveness of our system, especially for actuator commands where real-time feedback is crucial. We will measure:

- End-to-End Latency: The total time elapsed from the moment a task is dispatched by the emulated task producer until a success or failure acknowledgment is received from the emulated ESL/Gateway via the Communication Backbone. This will be measured at various load levels.
- Communication Layer Latency: The time taken for a request to traverse from the task_worker through the Communication Layer (Router) and reach the emulated Gateway, and for the response to return. This helps pinpoint bottlenecks within our proposed solution.
- Latency Distribution: Beyond just averages, we will analyze latency distribution (e.g., percentiles like P90, P95, P99) to understand tail latencies, which are crucial for real-world user experience and system reliability under stress.

Throughput

This metric quantifies the volume of operations our system can handle within a given timeframe, directly assessing its processing capacity. We will measure:

- RPCs per second: The number of successful RPC requests processed by the Communication Backbone per second.
- Task completion rate: The rate at which the system successfully processes and acknowledges commands from the task_queue to the emulated ESLs.

- Messages processed per second: The total volume of messages (requests and responses) flowing through the MQTT layer, indicating the underlying messaging infrastructure's efficiency.

The experiments will specifically focus on scenarios where the system components are pushed to their operational limits on individual nodes. This will allow us to observe and quantify how effectively the Communication Backbone leverages horizontal scalability to maintain performance and reliability when new nodes (Docker containers representing scaled instances of the Communication Layer and Task Workers) are introduced, demonstrating its elasticity under increasing load. We will gather data on how throughput increases and latency remains stable (or degrades gracefully) as the number of concurrent emulated devices and command rates escalate.

Qualitative Observations

Beyond raw numbers, the practical implications of implementing and managing such a system are crucial. Our qualitative observations will encompass:

- Ease of setup: How straightforward is it to deploy the Communication Backbone and its dependencies?
- Configuration overhead: The effort required to configure and integrate new components or devices.
- Operational simplicity: How easy is it to monitor, update, and manage the running system in a production-like environment? This includes aspects like ease of scaling up and down instances.

By combining these quantitative and qualitative measures, we aim to provide a holistic evaluation of our Communication Backbone, not only demonstrating its technical superiority in terms of scalability and performance for IoT cloud applications but also highlighting its practical advantages for real-world deployments.

DEVELOPMENT

General Architecture

The Communication Backbone developed for this thesis consists of several components:

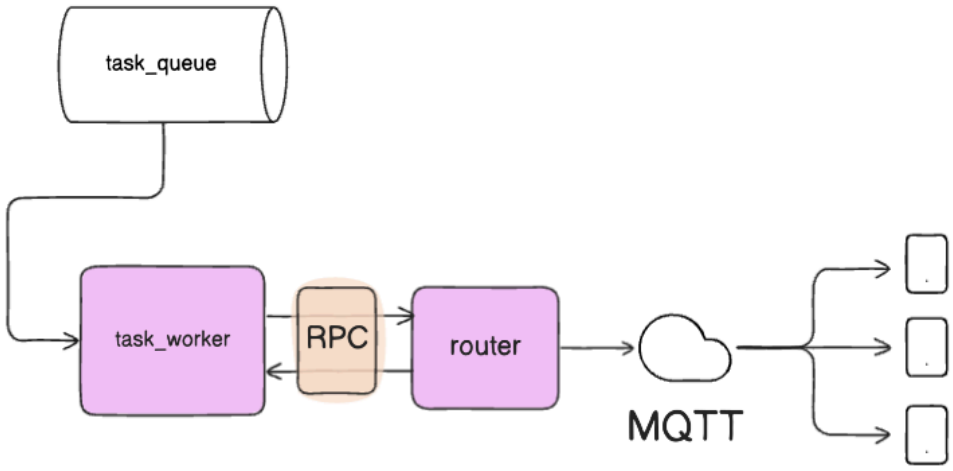

Figure 3: System Simplified Architecture - Source: Author (eraser.io)

Task Queue

Firstly, the task_queue is responsible for delivering tasks to our task_worker. Tasks are actions to be executed in the IoT devices (ESLs, in our case). The utilization of a queue here is paramount, primarily because these actions can be inherently long-running, often with an estimated timeout of around three minutes per task.

This potential for delays makes a synchronous request-reply model impractical and inefficient for the initial command dispatch. If the calling application were to wait synchronously for three minutes for each ESL action to complete, its threads would be tied up, quickly exhausting resources and severely limiting the system's overall throughput and responsiveness. This is particularly problematic in a large-scale IoT environment where thousands or tens of thousands of simultaneous commands might be issued (e.g., updating prices across an entire store).

By leveraging asynchronous communication via the task_queue, the producer (application that dispatches the tasks) can immediately hand off the task to the queue and consider its part of the transaction complete. It does not need to block and wait for the ESL to process the command. The queue then acts as a durable buffer, ensuring that the task is persisted and will eventually be delivered to an available task_worker, even if the worker or the IoT device experiences temporary disconnections or processing delays. This decoupling allows the producer to quickly move on to process other commands or serve other users, dramatically improving the perceived responsiveness and overall system capacity. It also provides inherent resilience: if a task_worker fails, the tasks remain in the queue to be picked up by another worker once available, preventing data loss and ensuring eventual consistency.

Furthermore, a key advantage of using a robust message queue is its ability to guarantee specific delivery semantics. In our model, this is extremely important, as we want to guarantee at-least-once execution semantics for critical operations like price updates. This means that once a price update command is sent to the task_queue, the system ensures it will be processed and executed by a task_worker at least one time. This guarantee is vital for business-critical operations like price synchronization, as it prevents scenarios where a price update might be lost due to transient network issues, worker failures, or device unresponsiveness. The queue achieves this through mechanisms such as message acknowledgments: a message is only considered processed and removed from the queue after the task_worker explicitly acknowledges its successful handling, or after a configurable timeout, upon which it can be redelivered to another worker (KLEPPMANN, M. 2017). This robustness is non-negotiable for ensuring that all prices displayed on ESLs are consistently and accurately updated, directly addressing a core business requirement.

Business Layer (Task Worker)

The task_worker is responsible for consuming messages (tasks) from the task_queue. Crucially, the task_worker encapsulates the core business logic associated with processing these commands. This includes interacting with databases (e.g., to fetch detailed product information for a price update, log the command's status, and update the device's state) and any other external services required to fulfill the task. Upon receiving a task and initial checks and processing is completed, the task_worker makes an RPC request to the Communication Layer of our system. After the replies from the Communication Layer, the task_worker treats the results accordingly by notifying clients and updating the databases.

Communication Layer (Router)

The Communication Layer (Router) is responsible for everything regarding the addressing and messaging of the individual devices. It connects to the subsequent MQTT Layer and holds the Communication Backbone logic. It allows the task_worker to be completely technology and vendor independent (when referring to the IoT devices).

Since the designed IoT devices communicate in Bluetooth Low Energy (BLE), there are intermediate Gateway Devices to "translate" requests to BLE. To reach a target device (ESL), the intermediate gateway hop also needs to be addressed, as such, the Communication Layer is also responsible for identifying the correct gateway that the ESL is currently addressable to. This identification is dynamic and crucial for reliable communication; for any given ESL, there might be multiple gateways within its BLE range. The Communication Layer must intelligently determine the optimal gateway to route the request through, often by selecting the gateway reporting the strongest Received Signal Strength Indicator (RSSI) from the ESL. This ensures the command is sent via the most stable and reliable path, minimizing signal loss and improving command success rates. This also saves us from needing more complex methods for reaching BLE signal stability, which would be too complex considering the dynamic environment inside the construction stores (THALJAOUI, A. et al. 2015).

To manage the RSSI and ESL routing addresses that link directly to the Gateway Device, we use a simple Load Balanced Subscriber to MQTT status topic, that forwards the ESLs BLE advertising packet. This ensures the Gateway Device MAC address is always up-to-date in our routing table in the database.

```
// routing_manager.go

func onMsgCallback(mqttPacket models.FullBLEAdvPacket) {
   currPacket := dbService.GetRoute(mqttPacket.EslMac)

   // Was not yet registered
      if currPacket == nil {
            dbService.SetPacket(mqttPacket)
            return
      }

      // Signal incresed
      if mqttPacket.Rssi > currPacket.Rssi {
            dbService.SetPacket(mqttPacket)
            return
      }

      // Signal decreased on same GW
      if mqttPacket.Rssi < currPacket.Rssi && mqttPacket.GwMac == currPacket.GwMac {
            dbService.SetPacket(mqttPacket)
            return
      }
}

func main() {
   topic := "/gw/+/status"
   mqtt.SetCallback("$share/routing-manager-group/", onMsgCallback)
   mqtt.Start() // Hangs
}
```

Figure 4: Pseudo-code for routing manager - Source: Author

This Communication Layer inherently represents the primary scalability challenge for the entire system. While the task_queue and task_workers can scale horizontally with relative ease by adding more instances, the Communication Layer faces unique challenges in maintaining device connectivity and command routing efficiency across a massive and dynamic network of IoT gateways and devices. In traditional architectures, the load-balancing done by MQTT would not allow a system to simply scale horizontally, as the nodes that receive the RPC request may not be the same ones that receive the response from the device. This "misrouting" of replies would lead to responses being discarded as unrelated or "junk messages," effectively breaking the critical request-reply pattern essential for reliable actuator control and feedback. Our Communication Layer's specialized logic directly addresses this limitation by ensuring response correlation and proper routing, which is fundamental to overcoming this scalability hurdle.

Any bottleneck or inefficiency within this intricate layer can propagate throughout the entire system, severely hindering the overall scalability, responsiveness, and reliability of the large-scale IoT application.

Communication Backbone: RPC via HTTP

By default, the exemplified architecture will make a simple RPC via an HTTP method, this will be our baseline system. The router entity exists to enable the task_worker to be completely device vendor/technology agnostic. It (in our system) communicates with the gateways via MQTT, but this separation allows us to quickly modify it in the future.

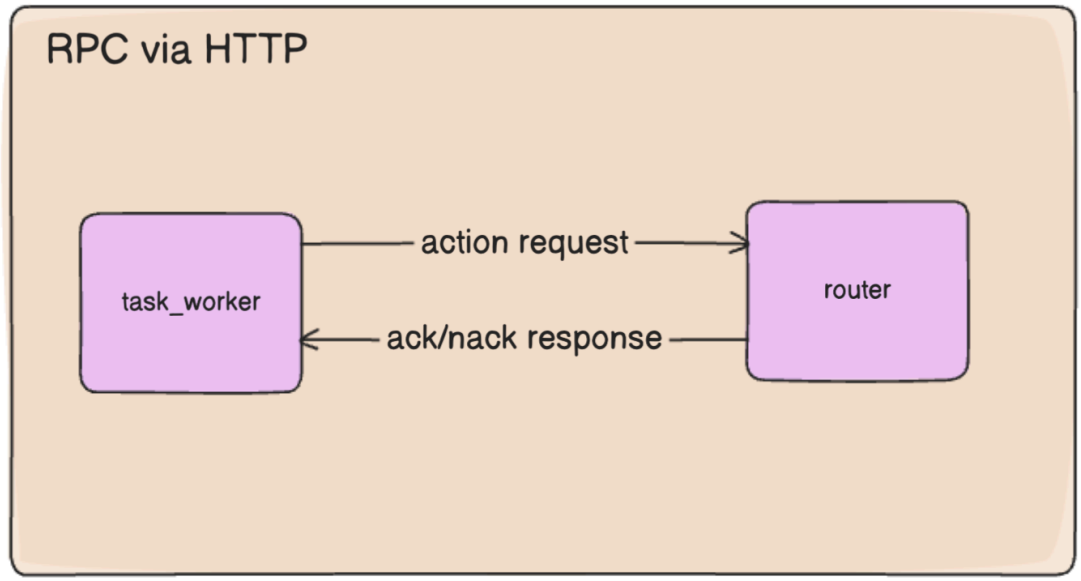

Figure 5: RPC via HTTP - Source: Author (eraser.io)

With the following sequence diagram:

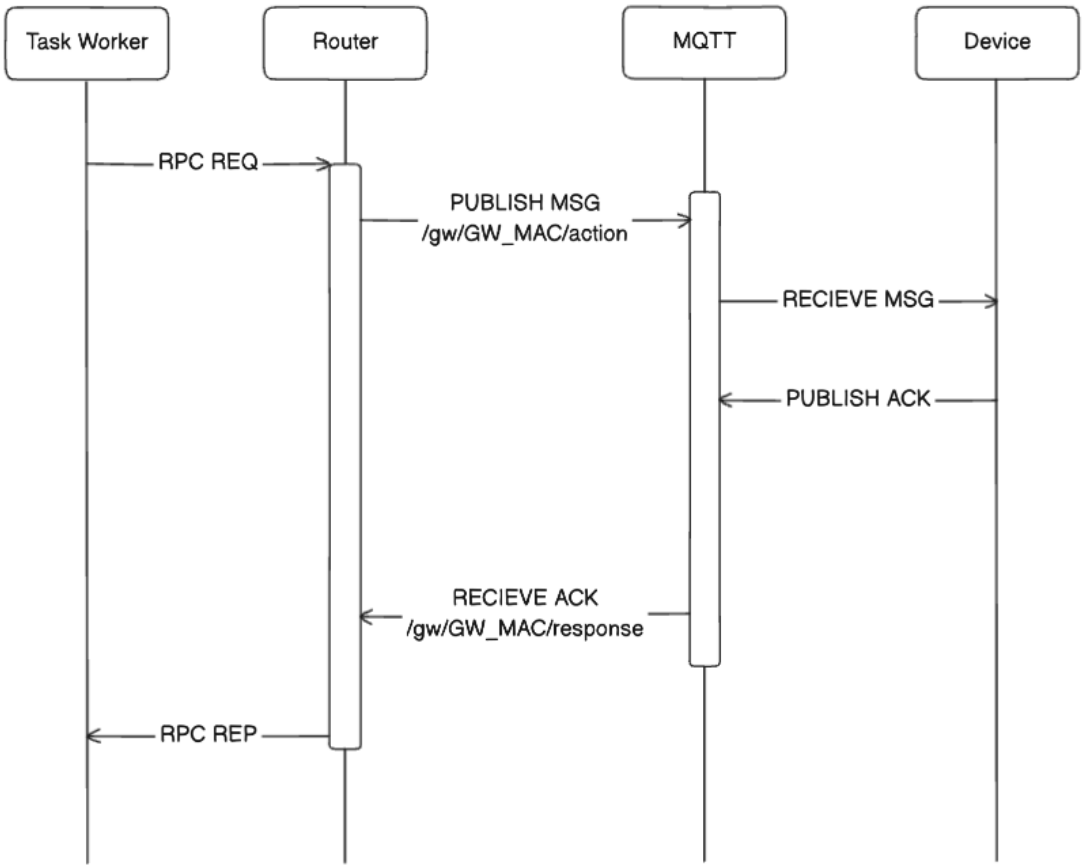

Figure 6: Simplified Sequence Diagram of the architecture - Source: Author (eraser.io)

Note that since the router can communicate with MQTT, it needs the GW_MAC. This means we would need a single router per gateway (with a ratio of around 1 gateway to every 5 thousand ESLs). This poses as the first limitation of scalability.

Limitations of this system

Although the router routines communicate with simple Inter-Process Communication (IPC) and are faster than network calls, the issue of not being able to communicate with multiple nodes is the main pitfall of this "traditional" system.

A first-look alternative to this would be to simply listen on all gateway MACs for the ACK Response. However this is too unreliable, let's take a look:

First, since MQTT load-balancing is offered in a Round-Robin manner, all listeners to a topic share the messages one-by-one.

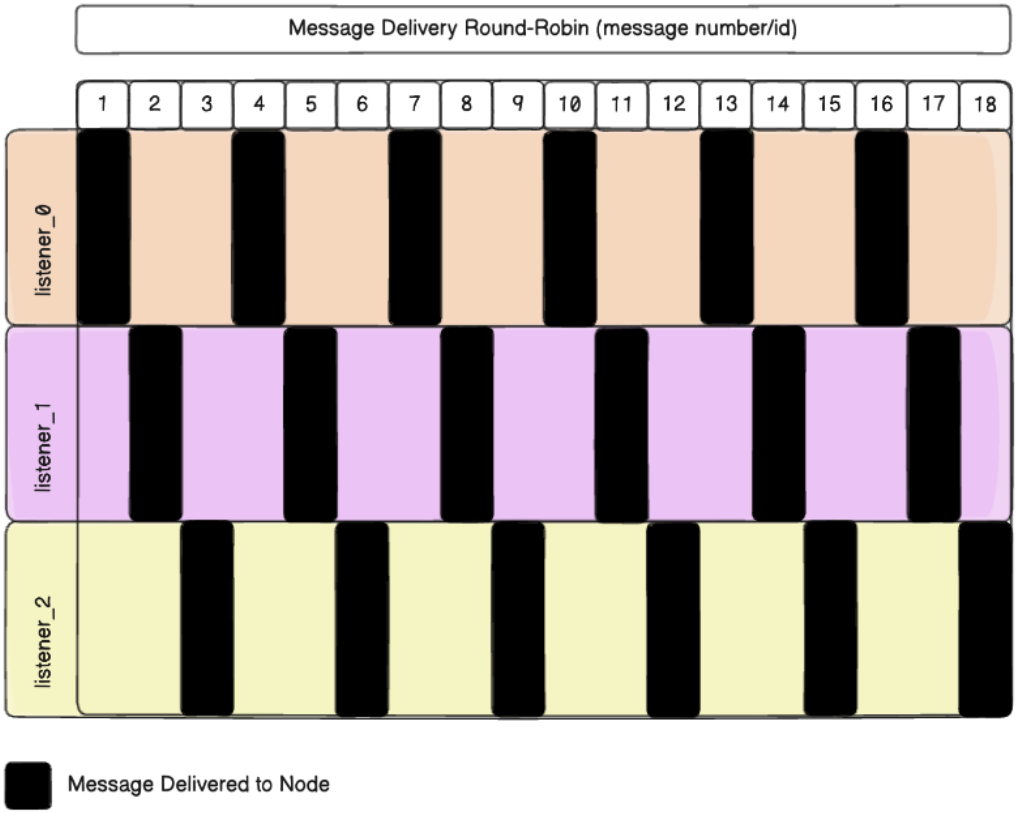

Figure 7: Round Robin demonstration of MQTT message delivery - Source: Author (eraser.io)

The problem arises from the fact that there is no guarantee that therouter_node that made the action request will be the same one that receives the response message.

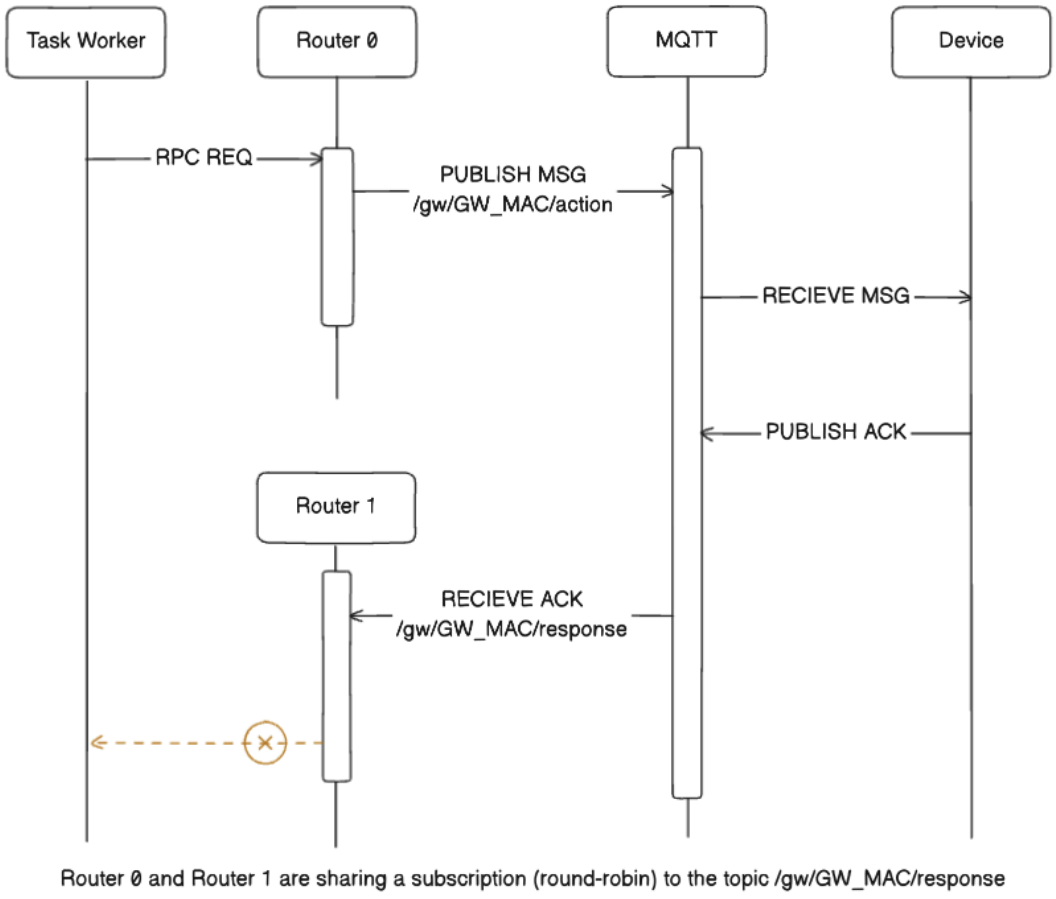

Figure 8: Sequence Diagram of RPC HTTP implementation - Source: Author (eraser.io)

Walking through this sequence diagram we see:

1. The Task Worker establishes a connection to Router 0 and sends the RPC request.
2. Router 0 publishes the message on the /gw/GW_MAC/action topic in MQTT.
3. MQTT forwards the reply to the Device Gateway.
4. The Device Gateway responds with the ACK to MQTT.
5. MQTT returns the ACK to Router 1 (as it is load-balanced).
6. Router 1 has no way of replying to the Task Worker as there is no established connection.

This poses as the first limitation of scalability and redundancy, locking the elasticity of the system as there must always be exactly one instance of the router running per gateway.

The current system faces a critical scalability limitation due to the tight coupling between router nodes and device gateways, where HTTP/RPC over MQTT relies on the same router instance processing both the request and response. Since MQTT load-balances responses across routers in

a round-robin fashion, there is no guarantee that the router receiving the reply will be the one that initiated the request, leading to dropped acknowledgments and forcing a 1:1 router-gateway binding. This constraint severely restricts horizontal scaling, as each gateway must be statically assigned to a single router, during both traffic spikes and at idling.

Communication Backbone: RPC via Distributed Queues

By considering the limitations in the initially described system, we must analyse both the system requirements and the issues encountered in the baseline backbone.

First we need to resolve the tight coupling of the systems. To resolve this, a message queue should be introduced to decouple request handling from response routing. In this model, task workers publish requests to a dedicated request queue, which any available router can consume. Responses are then directed to a reply queue using a unique correlation ID, ensuring they return to the originating worker—regardless of which router processed the request. This approach eliminates the need for sticky sessions, allows multiple routers to share the load for a single gateway, and provides buffering during traffic surges, preventing message loss. Additionally, queues enable auto-scaling based on backlog depth, further improving elasticity.

By adopting this architecture, the system gains true horizontal scalability, fault tolerance, and resilience against burst workloads, while removing the restrictive 1:1 router-gateway dependency. Industry best practices, such as AWS's recommendation for decoupling microservices with SQS and Microsoft's asynchronous request-reply pattern, strongly support this design for high-throughput, distributed systems. As such, the brittle system now becomes resilient and scalable (KLEPPMANN, 2017).

The system is built using Redis and RabbitMQ (RBMQ). This is done so that the routing messages can be stored using only in-memory data structures to reduce the latency overhead from the added network calls. Furthermore, since each action request references a single ESL, redis allows up to lookup the correlation ID for each ESL's MAC in O(1) time complexity. Redis also has many built-in features that support short-lived, ephemeral data in a shared and easily scalable environment. Also importantly, using in-memory Redis also allows the system to better respond to sudden request spikes more effectively (MADAPPA, 2012).

Furthermore, Redis and RBMQ Time to Live (TTL) functionality fits perfectly the short-lived Correlation IDs. As they are inherently temporary; once a response has been successfully routed back to the originating worker or no response is generated, the Correlation ID serves no further purpose. This also makes the process of cleaning up the Correlation IDs much simpler on the router's part, as it only needs to set the appropriate TTL when the request is received. On the task worker's part, when making the RPC request, it needs to create an ephemeral queue in RBMQ, that will be purged when the connection is closed.

This creates a highly efficient and self-managing system for decoupling the task request handling from the response routing.

Now the task worker interacts with the RPC mechanism as:

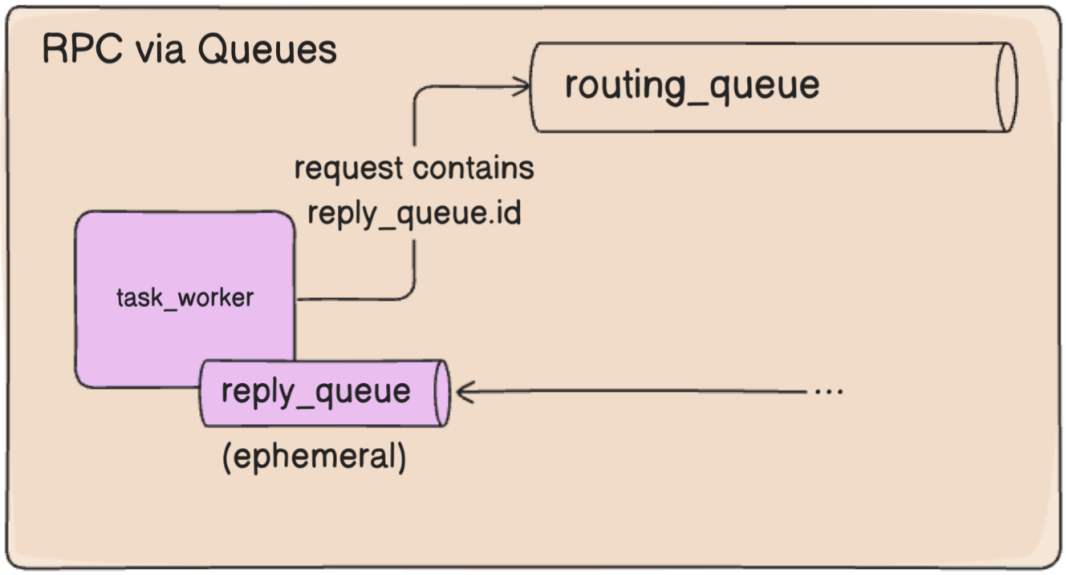

Figure 9: RPC via Distributed Queues diagram - Source: Author (eraser.io)

Changing the overall architecture to:

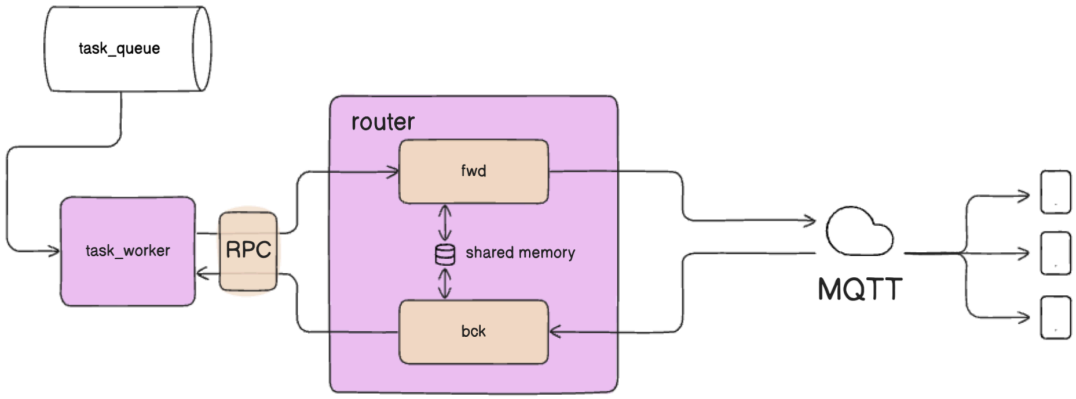

Figure 10: Complete Architecture with Custom Router Backbone - Source: Author (eraser.io)

By decoupling the task workers from the routers, we now can create a forward (fwd) router and a back (bck) router services to communicate with MQTT, simplifying the code and software architecture.

The routing tables referenced by the routers are also saved to Redis. The ESLs broadcast a signal to all Gateway Devices in range every couple of seconds. The Gateway Devices publish these messages to MQTT with the BLE RSSI of the ESL connection. With this, we are able to build a routing table in Redis to utilize the gateway with better device communication to the designated ESL. This is also saved as a Map of the Key being the ESL's MAC address and it's Value being the MAC address of the Gateway Device with the strongest connection to that ESL, also allowing up to efficiently manage all ESLs with Redis O(1) lookup complexity.

With this, we have the following two sequence diagrams, the former from the task worker utilizing the FWD Router, and the latter from the perspective of the BCK Router utilization.

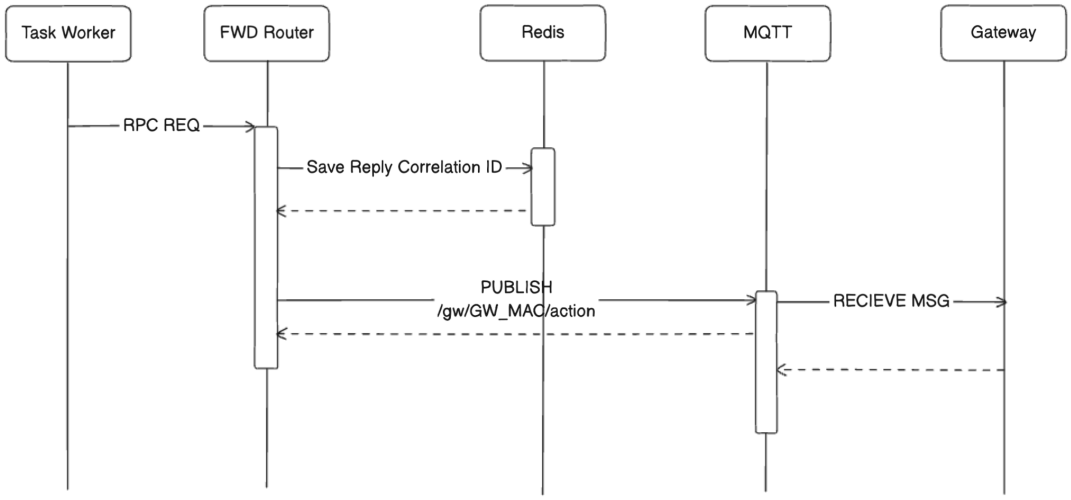

Figure 11: Sequence Diagram of the FWD Router - Source: Author (eraser.io)

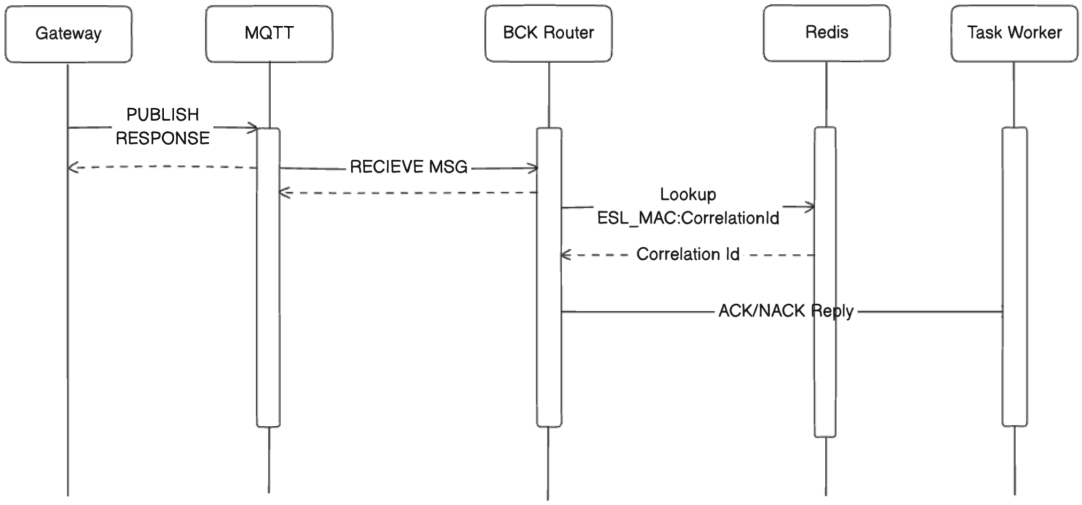

Figure 12: Sequence Diagram of the BCK Router - Source: Author (eraser.io)

With this, all messages are always routed back to the requesting node, independently of the number of replicas of the routers, using the Correlation ID stored in Redis as service discovery to correctly route the response to the task_worker, making for a scalable and reliable system.

By leveraging the proposed architecture, all messages are consistently routed back to their originating task worker, irrespective of the number of router replicas in the system. This reliable redirection is achieved by utilizing the Correlation ID, temporarily stored within Redis, which effectively acts as a dynamic service discovery mechanism. Consequently, this design establishes

a highly scalable and robust system capable of handling increased loads and maintaining operational integrity even with fluctuating service demands.

The system was completely developed in Golang (Go), as its design aimed to develop distributed systems brings in loads of built-in features that aid with IPC such as channels, Goroutines and more. Note that for load balancing, we used 2 different packages for MQTT interaction, as the official paho image does not support shared subscriptions as it's an MQTTv5 feature, where the paho package only supports MQTTv3.

```
// task_worker
func main() {
   for task := range rabbitMqMsgsChan {
      esls := dbService.BusinessLogic(tasks)

      var msgs []models.RoutingMessage
      for esl := range esls {
         m := models.RoutingMessage{}.FromEsl(esl)
         msgs = append(msgs, m)
      }

      reps := commsBackbone.Forward(msgs) // This method waits for replies
      if len(reps) != len(macs) {
         return errors.New("not all devices replied")
      }
   }
}
```

Figure 13: Pseudo-code for task_worker - Source: Author

```
// fwd_router
func main() {
   for routingMsg := range rabbitMqMsgsChan {
      sharedMem.Save(routingMsg.EslMac, routingMsg.CorrelationId)

      gwMac := dbService.GetRoute(routingMsg.EslMac)
      topic := "/gw/"+gwMac+"/action"

      mqtt.Publish(topic, routingMsg)
   }
}
```

Figure 14: Pseudo-code for fwd_router - Source: Author

```
// bck_router
func onMsgCallback(rep models.RoutingReply) {
   correlationId := sharedMemService.RepKey(rep.EslMac)
messagingService.Reply(correlationId, rep)
}

func main() {
   topic := "/gw/+/response"
   mqtt.SetCallback("$share/router-bck-group/", onMsgCallback)
   mqtt.Start() // Hangs
}
```

Figure 15: Pseudo-code for bck_router - Source: Author

HOMOLOGATION

After the MVP development, tests were employed to guarantee the working and efficiency of the system. Firstly, the baseline system performance numbers were established. All tests were run on a 12-core machine with 16GB of RAM. Resource limits were only applied to the router elements in the architecture, while all other systems (i.e., queues, databases, etc.) had no resource limits. The CPU of the router elements was capped at 100 mCPU per replica. All tests were executed by 50 workers simultaneously to increase the load on the routers, bringing it to a more realistic value. The request timeout was set to 60 seconds; if a reply was not received within this timeout, the reply window was closed, and the request was deemed unsuccessful.

The values of the tests were measured using a telemetry service that asynchronously saves the latency of the RPCs to MongoDB in a different coroutine to reduce the impact of the measurements. The MongoDB instance was running on a separate server to avoid interfering with the experiments.

To capture the complete system latency and execution time, the only data collected was on the worker-side as shown in the provided pseudo-code:

```
// task_worker with telemetry
func main() {
   for task := range rabbitMqMsgsChan {
      esls := dbService.BusinessLogic(tasks)

      var msgs []models.RoutingMessage
      for esl := range esls {
         m := models.RoutingMessage{}.FromEsl(esl)
         msgs = append(msgs, m)
      }

      t0 := time.Now()
      reps := commsBackbone.Forward(msgs) // This method waits for replies
      if len(reps) != len(macs) {
         return errors.New("not all devices replied")
      }
      delta := time.Since(t0)
      asyncTelemetryService.AddDelta(delta)
      // ...
   }
}
```

Figure 16: Pseudo-code for task_worker with telemetry - Source: Author

This telemetry implementation on the task_worker captured the time taken for the commsBackbone. Forward method, which waits for replies from the system.

Functional Tests

To guarantee and homologate the workings of the system, the initial functional tests aimed to verify if the system simply followed the sequence diagram of the architecture. For this purpose, 1,000 tasks were executed with only 1 router replica. After running the experiment, a histogram of the latency of each request was plotted to visualize the distribution of response times.

Performance Tests

For the performance tests, the task count was increased to 10,000 to significantly increase the bulk of concurrent operations, ensuring that router requests overlapped much more than in the functional tests, thereby stressing the router node. Additionally, the number of router replicas was increased in each experiment to evaluate horizontal scalability.

RESULTS

Functional Tests

Starting with the baseline system, we measure the execution times to assess request timeouts. Noting that all tests ran with 50 concurrent workers making requests (a small number, in our practical ESL case, it was expected to have around 700 workers concurrently at average working conditions, with peaks of up to 5 thousand workers).

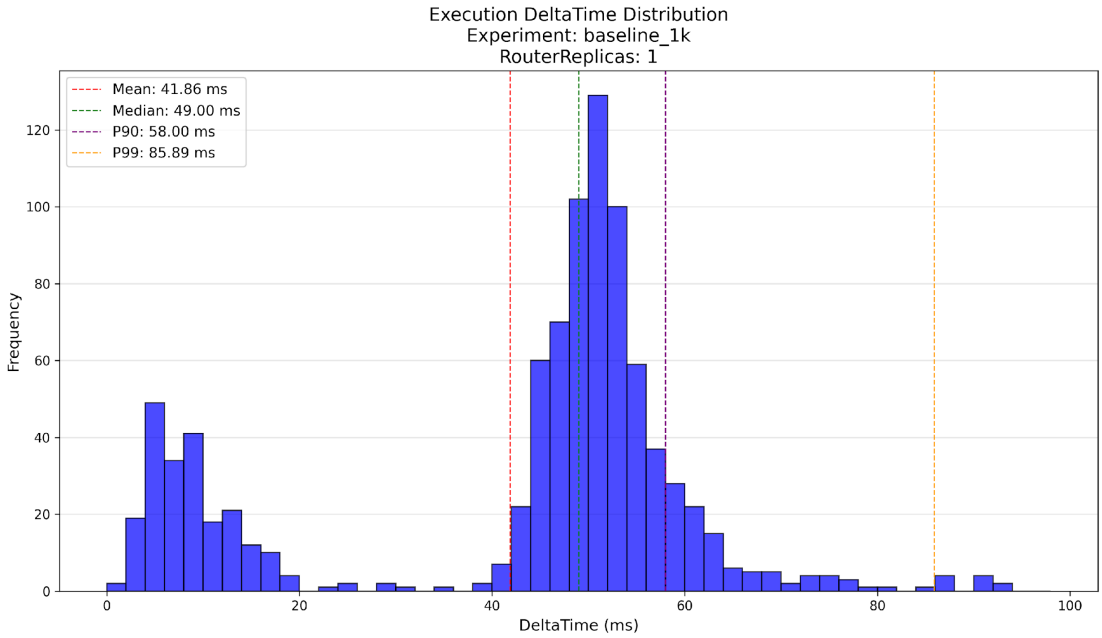

Figure 17: Histogram of the Baseline Backbone with 1 router replica for 1k tasks - Source: Author

The histogram for the baseline system with 1,000 tasks and 1 router replica showed a large grouping around the 50ms mark, originating from the bulk of concurrent code execution. A smaller peak of less than 10ms was observed, possibly from the first and last few requests where code was mostly not concurrent. With most results under 1 second, well below the 60-second timeout, it was confirmed that all requests reached the ESL emulator and received successful replies, regardless of ACK or NACK status.

Following with the Custom Backbone, the execution times:

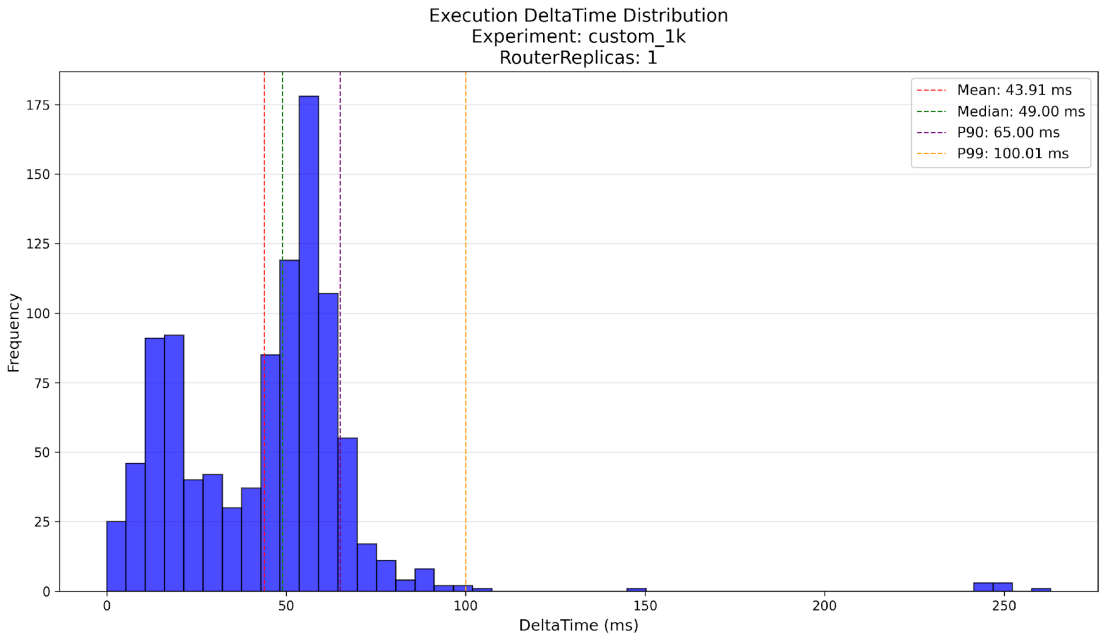

Figure 18: Histogram of the Custom Backbone with 1 router replica for 1k tasks - Source: Author

For the developed Custom Communication Backbone, with the same setup, the histogram also showed the router working as expected. There was a small increase in mean delay and an increase in p99 delay, but the maximum delay remained around 250ms, significantly below the 60-second timeout. This validated the proposed custom architecture for a single replica.

These initial functional tests established the base functionality of both the custom backbone and the baseline backbone, confirming their expected operation under basic circumstances (50 task workers executing in parallel with a single router replica routing requests to device gateways).

Performance Tests

To start the performance tests, we increase the router instances replicas to 2. This "starts" the scalability aspect of the systems.

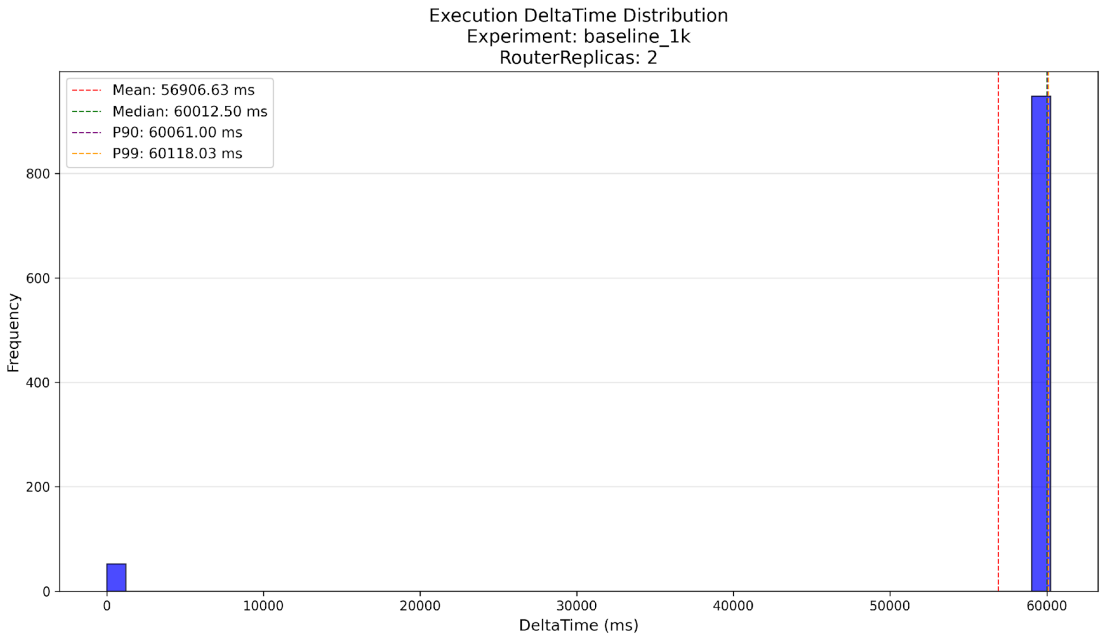

Figure 19: Histogram of the Baseline Backbone with 2 router replicas for 1k tasks - Source: Author

The first performance test with the baseline backbone, using 1,000 requests and 2 router replicas, revealed a significant pitfall: the misrouting issue. A large number of requests escaped the 60-second timeout window, indicating that they timed out and were unable to correctly reply to the worker. The few successful requests were likely randomly routed to the same node that made the request. With only 2 replicas, the ratio of misrouted to correctly routed messages already greatly surpassed the acceptable limit of eventual failures, a ratio expected to increase with more router replicas. Consequently, further tests with increased task counts and router replicas were not conducted on the baseline router case.

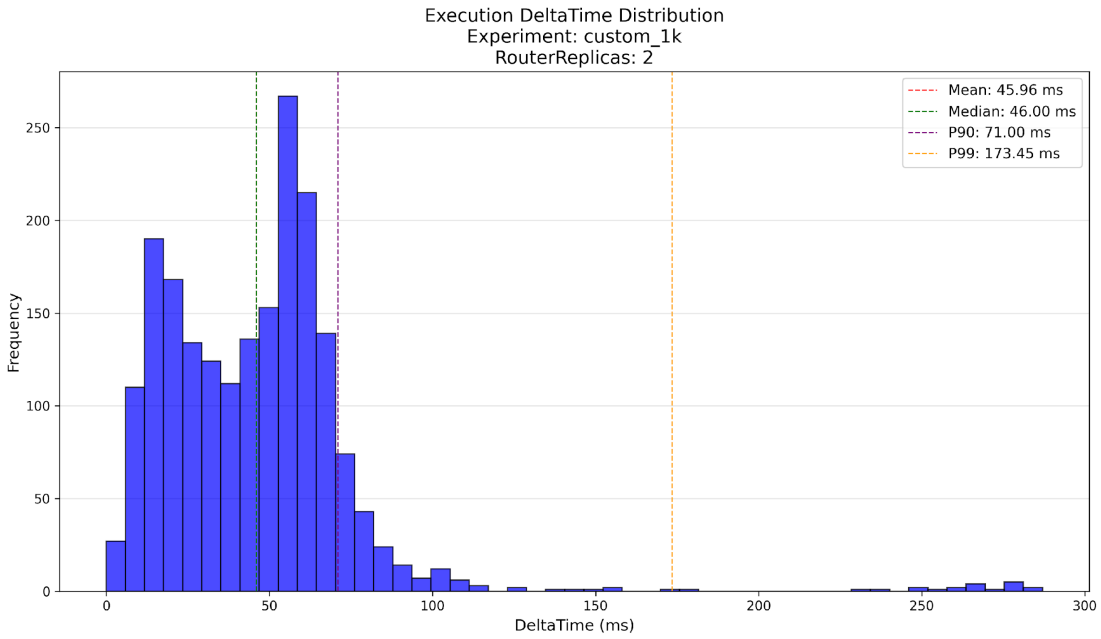

Figure 20: Histogram of the Custom Backbone with 2 router replicas for 1k tasks - Source: Author

With the custom backbone, using the same parameters (1,000 requests, 2 router replicas), the system continued with correct message routing, evidenced by no requests timing out. Importantly, a small increase in the median latency was observed compared to a single router replica.

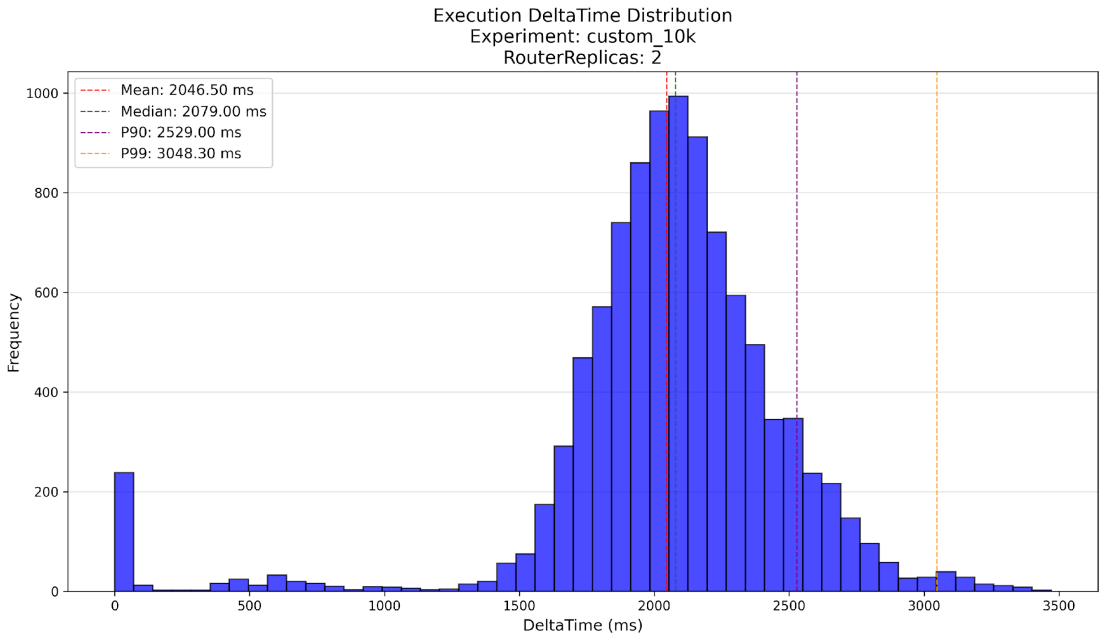

Figure 21: Histogram of the Custom Backbone with 2 router replicas for 10k tasks - Source: Author

A heavier experiment with 10,000 requests and 2 replicas using the custom backbone showed a great increase in the mean execution time, indicating that the system was beginning to bottleneck. To address this, the number of router replicas was increased to 20 to allow for better load balancing and to compare the results.

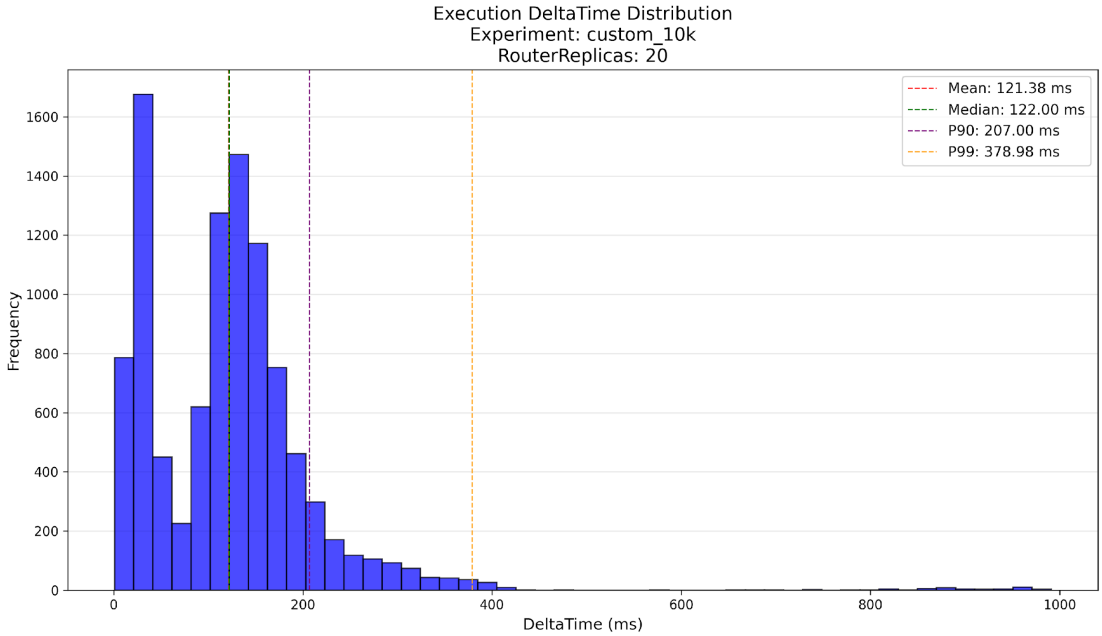

Figure 22: Histogram of the Custom Backbone with 20 router replicas for 10k tasks - Source: Author

Increasing the router replicas to 20 for 10,000 requests resulted in much better results, with both the mean and p99 significantly reduced (mean from 2000ms down to approximately 120ms, and p99 from 3000ms to 400ms). This demonstrated that when measuring the custom backbone with 20 router replicas, the system achieved a minimal increase in latency, showing a true increase in scalability and proven elasticity.

Finally, a "for fun" experiment was conducted with 200 router replicas, reserving a machine with 64vCPU and 256GB RAM on MagaluCloud. This experiment showed a great decrease in latency, down to around 6ms with a p99 of 57ms. This result indicates that increasing the number of router replicas indeed leads to greater scalability, as messages spend less time waiting in the RabbitMQ queue before being collected by the routers and forwarded to MQTT.

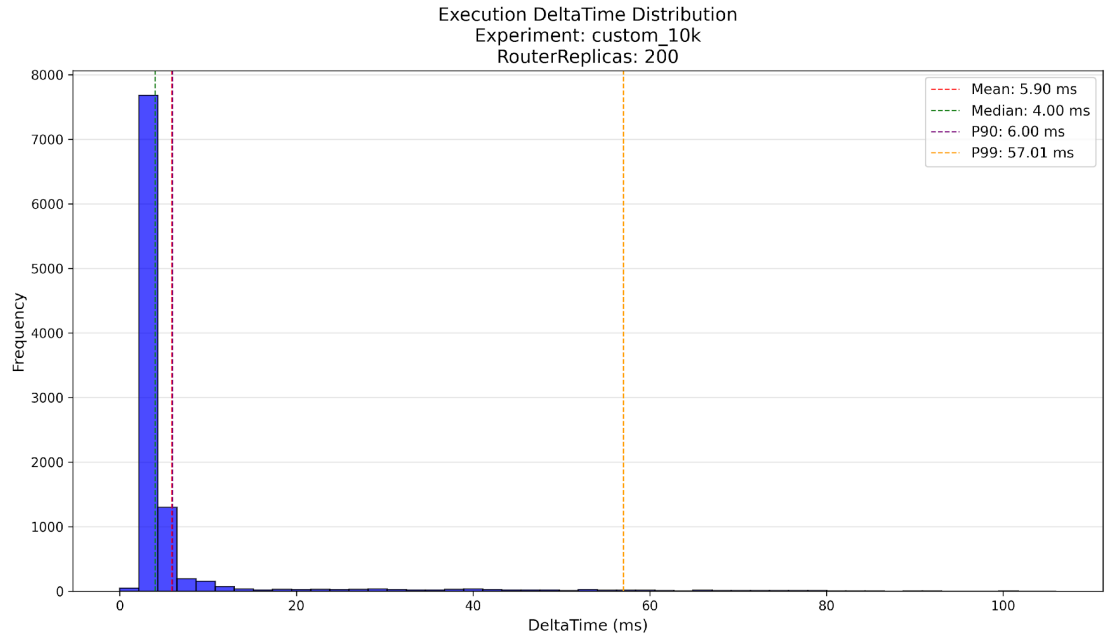

Figure 23: Histogram of the Custom Backbone with 200 router replicas for 10k tasks - Source: Author

It must be noted that the ideal configuration parameters for scalability using a platform such as KEDA for Kubernetes have not been developed in this study.

CONCLUSIONS

Based on the initial requirements and the comprehensive test results, we can confidently conclude that all objectives for the MVP of the Custom Communication Backbone were successfully achieved. The developed system fully embodies the characteristics of an MVP, demonstrating proficiency in the complete sequence of all described diagrams. Beyond merely meeting functional requirements, the system exhibits simple configuration and exceptional scalability. Increasing replicas introduces no additional complications beyond adjusting the replica count, making it ideal for cloud-native applications.

The entire backbone was developed using modern, well-established technologies such as Go, Redis, RabbitMQ, MQTT, and Docker. This strategic choice inherently makes the system Cloud Native by design, facilitating straightforward sequence flows for every software component within the backbone. This design choice contributes to easier debugging, future expansion, and enhanced adaptability. Furthermore, the complete system boasts extremely simple configuration, allowing for effortless deployment and testing on any machine.

FUTURE WORKS

Building upon the robust foundation established by this MVP, a significant future work direction would involve further evolving the Custom Communication Backbone into a truly "plug-and-play" solution. This would entail developing a more sophisticated and user-friendly configuration layer that allows for the explicit definition of the message structure and scalability parameters tailored for Kubernetes environments, specifically leveraging KEDA (Kubernetes Event-driven Autoscaling). Such an enhancement would enable users to easily configure the system's autoscaling behavior based on various metrics and event sources, further streamlining deployment and operational management within a cloud-native ecosystem. This would transform the current architecture into a highly adaptable and self-managing system, optimizing resource utilization and performance under fluctuating loads without manual intervention.

Another compelling area for future work involves enhancing the system's technology agnosticism. While currently leveraging a well-defined set of modern technologies, future iterations could explore abstracting away specific dependencies where feasible. This could involve developing standardized interfaces or adapting to widely adopted protocols that are not tightly coupled to a single vendor or technology stack for components like message queuing or data storage. The goal would be to maximize interoperability and provide greater flexibility for integration into diverse existing infrastructures, allowing organizations to adopt the backbone without significant refactoring of their current technology investments, thereby broadening its applicability and appeal across various enterprise environments.

REFERENCES

[1] Lombardi, D. (2025). Optimizing IoT Cloud Applications for Scalability: Leveraging RPC with Distributed Queues for Seamless Operations (Version 1.0.0) [Computer software]

[2] Motta, R. C., Silva, V., & Travassos, G. H. (2019). Towards a more in-depth understanding of the IoT Paradigm and its challenges. Journal of Software Engineering Research and Development, 7, 3:1 – 3:16. https://doi.org/10.5753/jserd.2019.14

[3] POURMAJID, W. et al. (2017). On Challenges of Cloud Monitoring. Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering

[4] Walter L. Heimerdinger and Charles B. Weinstock: "A Conceptual Framework for System Fault Tolerance," Technical Report CMU/SEI-92-TR-033, Software Engineering Institute, Carnegie Mellon University, October 1992.

[5] MOTTA, Rebeca; OLIVEIRA, Káthia; TRAVASSOS, Guilherme. IoT Roadmap: Support for Internet of Things Software Systems Engineering. arXiv preprint arXiv:2103.04969, 2021. https://arxiv.org/ftp/arxiv/papers/2103/2103.04969.pdf

[6] L. Atzori, A. Iera, G. Morabito. The Internet of Things: a survey. Computer Networks, 54 (2010), pp. 2787-2805

[7] MELL, P. M.; GRANCE, T. The NIST Definition of Cloud Computing. 2011, Available at: https://www.nist.gov/publications/nist-definition-cloud-computing.

[8] AAQIB, S. An Efficient Cluster-Based Approach for Evaluating Vertical and Horizontal Scalability of Web Servers using Linear and Non-Linear Workloads. In 2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI), Tirunelveli, India, 2019, pp. 287-291, doi: 10.1109/ICOEI.2019.8862561.

[9] Jim N. Gray: "Queues Are Databases," Microsoft Research Technical Report MSR-TR-95-56, December 1995.

[10] KLEPPMANN, Martin. Designing Data-Intensive Applications. Sebastopol, CA: O'Reilly Media, Inc., 2017.

[11] M. van Steen and A.S. Tanenbaum, Distributed Systems, 4th ed., distributed-systems.net, 2023.

[12] Raffi Krikorian: "Timelines at Scale," at QCon San Francisco, November 2012. Available at: https://www.infoq.com/presentations/Twitter-Timeline-Scalability/

[13] A. Thaljaoui, T. Val, N. Nasri and D. Brulin, "BLE localization using RSSI measurements and iRingLA," 2015 IEEE International Conference on Industrial Technology (ICIT), Seville, Spain, 2015, pp. 2178-2183, doi: 10.1109/ICIT.2015.7125418. keywords: {Receivers;Indexes;Accuracy;Bluetooth;Estimation;Distance measurement;Position measurement;Smarthome;Localization;BLE;RSSI},

[14] KASUN INDRASIRI; SRISKANDARAJAH SUHOTHAYAN. Design patterns for cloud native applications : patterns in practice using APIs, data, events, and streams. [s.l.] Sebastopol, Ca O'reilly, 2021.

[15] CLOUD NATIVE COMPUTING FOUNDATION. Who We Are. Available at: https://www.cncf.io/about/who-we-are/. Accessed on: 15 jun. 2025.

[16] KUBERNETES. Overview. 2024. Available at: https://kubernetes.io/docs/concepts/overview/. Accessed on: 16 jun. 2025.

[17] DOCKER. What is Docker. Available at: https://docs.docker.com/get-started/docker-overview/. Accessed on: 18 jun. 2025.

[18] KEDA. Available at: https://keda.sh/

[19] GO. Use Cases. Available at: https://go.dev/solutions/use-cases. Accessed on 18 jun. 2025.

[20] MARTINEKUAN. Queue-Based Load Leveling pattern - Azure Architecture Center. Available at: https://learn.microsoft.com/en-us/azure/architecture/patterns/queue-based-load-leveling.

[21] Decouple messaging pattern - AWS Prescriptive Guidance. Available at: https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-integrating-microservices/decouple-messaging.html.

[22] MADAPPA, S. Ephemeral Volatile Caching in the cloud. Available at: https://netflixtechblog.com/ephemeral-volatile-caching-in-the-cloud-8eba7b124589.