A Lightweight SMTP Server for High-Performance Email Processing with AMQP Integration

Nikhil Raj
Independent Researcher, Bengaluru, Karnataka, India
ORCID: 0009-0008-3435-5674
Email: `nikhil@nekonik.com`

***Abstract*—This paper introduces the Lightweight SMTP Server (LSMTP), a minimalistic SMTP server designed for modern distributed systems that require high scalability and efficiency. LSMTP is optimized to handle the essential SMTP commands—`HELO/EHLO`, `MAIL FROM`, `RCPT TO`, `DATA`, and `QUIT`—while excluding advanced features like encryption, spam filtering, and email storage. This streamlined approach makes LSMTP ideal for use cases where basic email reception and forwarding are needed, such as in microservices architectures. The server is implemented using the Rust programming language and the Tokio asynchronous runtime, ensuring optimal performance and minimal resource consumption. Integrated seamlessly with AMQP (Advanced Message Queuing Protocol), LSMTP forwards incoming emails to message brokers like RabbitMQ for further processing, such as analytics or archiving. The results of our performance benchmarks highlight LSMTP's superior efficiency compared to traditional mail servers, making it a highly suitable solution for lightweight, high-performance email processing in distributed environments.**

I. Introduction

The Simple Mail Transfer Protocol (SMTP) is defined in RFC 5321 [1] and serves as the foundation for email communication, providing the framework for the exchange of messages across networks. Over decades, traditional mail servers like Postfix [2] and others have been developed to support comprehensive email handling capabilities, including advanced routing, filtering, and storage mechanisms. However, these systems often come with significant complexity that requires meticulous configuration, maintenance, and resource allocation.

In the era of micro-services and distributed architectures, this complexity becomes a bottleneck. Modern applications demand streamlined workflows, high scalability, and modularity. Traditional SMTP servers were not designed to integrate seamlessly into these architectures or forward emails to modern processing pipelines, such as message brokers. Additionally, their all-encompassing design can lead to over provisioning and inefficiency for use cases that require only a subset of email processing features.

To address these challenges, this paper introduces **Lightweight SMTP Server (LSMTP)**, a minimalistic SMTP server optimized for receiving emails and forwarding them to an AMQP (Advanced Message Queuing Protocol) [3] broker for further processing [4]. By focusing on essential functionality, LSMTP implements only core SMTP commands: `HELO/EHLO`, `MAIL FROM`, `RCPT TO`, `DATA`, and `QUIT`. By omitting advanced features to provide a lightweight, high-performance alternative that is simple to deploy and manage.

The server leverages the Rust programming language and the Tokio asynchronous runtime to achieve exceptional performance and reliability. Rust's focus on safety and concurrency, combined with Tokio's efficient event-driven model, ensures that the server can handle a high volume of connections with minimal resource usage. This design makes it an ideal choice for microservices environments where modularity, scalability, and fault tolerance are paramount.

A. Scope and Limitations

This paper focuses solely on creating a minimalistic SMTP server to receive and forward emails to an AMQP broker. It is designed to be lightweight, efficient, and easy to integrate into modern distributed systems. It does not aim to replace full-featured mail servers but rather to provide a specialized solution for specific use cases. The server has several limitations, including:

- **TLS Encryption**: Secure email transmission is outside the scope of this work.
- **Spam Filtering**: Mechanisms for identifying or handling spam are not implemented.
- **Email Storage**: The system does not store emails; all messages are forwarded to RabbitMQ or any AMQP-compatible broker for further processing.
- **Advanced Command Support**: Only basic SMTP commands are implemented, making it unsuitable for scenarios requiring extended features like authentication or batching.

These limitations are deliberate and aligning with the goal of achieving simplicity, modularity, and high performance for specific use cases. The intent is to provide a foundation for further customization and integration rather than a comprehensive mail server solution.

B. Motivation and Use Cases

Practical use cases for LSMTP include email archiving, lightweight notification systems, and email content forwarding for further processing, such as analytics or spam detection. By integrating with RabbitMQ or any AMQP-compatible message broker, LSMTP enables seamless downstream processing and enhances the overall flexibility of email workflows.

The contributions of this paper include the following.

1) **Design and Implementation**: A lightweight SMTP server optimized for modern architectures.
2) **Integration with AMQP**: Enabling email forwarding for further processing.
3) **Performance Analysis**: Demonstrating the efficiency and scalability of the server.

The rest of the paper is organized as follows: Section II provides background on SMTP, AMQP, and the technologies used in the implementation. Section III reviews related work and highlights the gaps in traditional SMTP solutions. Section IV details the proposed system design, focusing on architecture and integration with RabbitMQ. Section V presents the implementation details and performance results. Section VI concludes the paper with insights and future directions.

The source code for this project is available on GitHub [4].

II. Background on SMTP, AMQP, and Rust with Tokio

This section discusses the fundamental concepts behind Simple Mail Transfer Protocol (SMTP) and Asynchronous Messaging Queues (AMQP), as well as a review of relevant literature and existing solutions. Various SMTP servers and frameworks are explored, highlighting their limitations in modern, distributed applications.

A. SMTP (Simple Mail Transfer Protocol)

Simple Mail Transfer Protocol (SMTP) is the foundational protocol used for email communication over the internet [1]. It governs how email messages are composed, transmitted, and delivered across networks. This protocol operates in a client-server architecture, where the sending system functions as the SMTP client, and the receiving system acts as the SMTP server [1].

The communication between the client and server is structured around a series of commands and responses. Each command initiates a specific step in the email transfer process, while the server's responses provide feedback on the success or failure of each operation. This interaction ensures reliable synchronization and error handling during email transmission [1].

1) SMTP Workflow: The SMTP session is depicted in Figure 1, which illustrates the typical workflow for transmitting an email through a series of bidirectional exchanges. This process involves several stages:

1) **Establishing a Connection:** The client establishes a connection with the SMTP server, typically over TCP port 25. The server responds with a greeting message indicating its readiness to proceed.
2) **Introducing the Client (HELO/EHLO):** The client introduces itself to the server using the `HELO` or `EHLO` command, followed by its domain name. The server acknowledges this handshake with a success message.
3) **Sender Identification (MAIL FROM):** The client specifies the sender's email address using the `MAIL FROM` command. The server confirms receipt of this information.
4) **Recipient Identification (RCPT TO):** The client identifies one or more recipients using the `RCPT TO` command. The server validates the recipients and confirms each one individually.
5) **Transmitting the Email Content (DATA):** The client sends the email content, including headers (e.g. `From`, `To`, `Date`, `Subject`) and the message body. The `DATA` command signals the server that the content transfer is beginning, and the transmission ends when the client sends a single dot (`.`) on a new line.
6) **Closing the Connection (QUIT):** After successfully transferring the message, the client terminates the session with the `QUIT` command. The server confirms the closure.

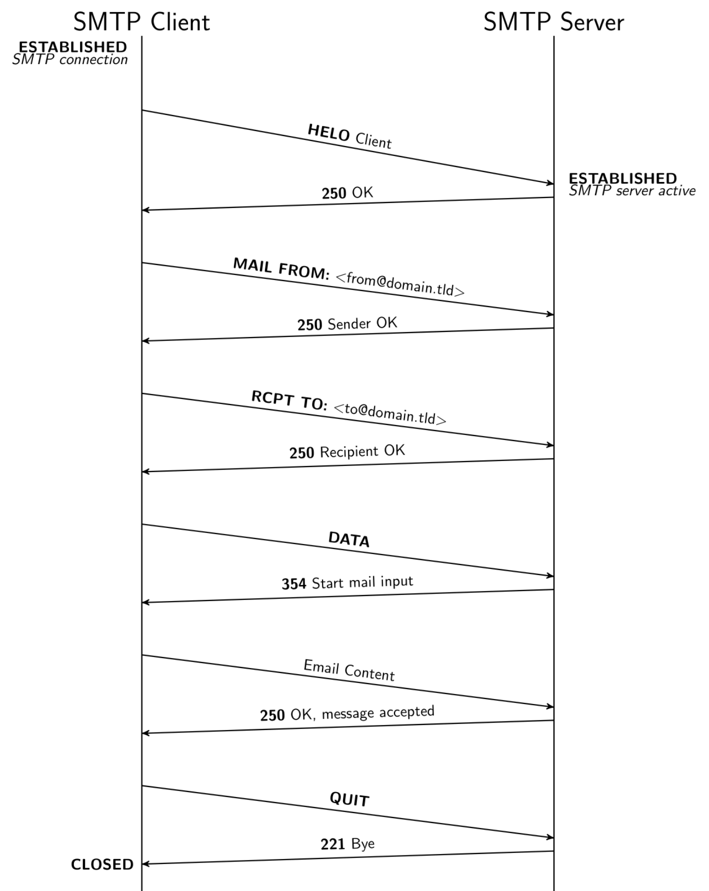

Fig. 1. Illustration of a typical SMTP session workflow, showing the exchange of commands and responses between client and server.

2) Importance and Limitations of SMTP: SMTP provides a reliable mechanism for email transfer, but was designed for simpler, text-based communication in the early days of the internet. Although it effectively handles the transport layer, modern use cases often require additional features such as encryption, authentication, and multimedia handling. Extensions like STARTTLS for secure connections and MIME for rich content have been introduced to address these limitations.

3) Relevance to This Paper: This paper focuses on a minimalistic implementation of the SMTP protocol. Unlike

traditional implementations that support a wide range of features and extensions, this work streamlines the process to support only essential commands (`HELO/EHLO`, `MAIL FROM`, `RCPT TO`, `DATA`, and `QUIT`). By reducing complexity, this implementation targets specific use cases, such as archiving emails or forwarding them to be processed further via AMQP brokers.

B. AMQP (Advanced Message Queuing Protocol)

AMQP (Advanced Message Queuing Protocol) is a communication protocol designed for message-oriented middleware. It enables reliable, asynchronous messaging between distributed applications, facilitating decoupling between producers and consumers. This decoupling enhances system scalability, reliability, and flexibility, especially in large-scale distributed systems. By providing a standardized wire-level protocol, AMQP ensures that messages can be reliably transmitted across heterogeneous platforms and technologies.

The protocol supports multiple messaging patterns, including point-to-point, publish-subscribe, and request-response. This flexibility allows AMQP to handle a variety of communication models depending on system requirements. It also provides features like message persistence, delivery acknowledgment, and routing, ensuring reliable delivery even in the presence of network failures or system crashes.

AMQP brokers, such as RabbitMQ, Apache ActiveMQ, and Apache Qpid, play a central role in message delivery. These brokers handle the routing and storage of messages, ensuring that they are delivered reliably to consumers. They support advanced features such as message queuing, filtering, and routing, making them ideal for high-performance, scalable, and fault-tolerant systems.

1) Core Features of AMQP: AMQP provides several key features that are essential for modern messaging systems:

- **Reliability and Persistence:** AMQP brokers ensure that messages are stored reliably in queues until consumed, even during system crashes or network issues. This feature guarantees that messages are not lost.
- **Asynchronous Communication:** Producers can send messages without waiting for an immediate response from consumers, enabling efficient, non-blocking communication.
- **Message Routing:** AMQP brokers route messages to appropriate queues based on predefined routing rules. This enables the system to direct messages to specific consumers based on content or other criteria.
- **Acknowledgments:** Consumers acknowledge receipt of messages after processing them, allowing the broker to track message delivery and avoid message loss.
- **Scalability:** AMQP brokers allow horizontal scaling by distributing messages between multiple queues and consumers, handling large volumes of messages with high throughput.

2) AMQP Brokers: Several popular AMQP brokers are available, each with distinct strengths and features. Notable examples include RabbitMQ, Apache ActiveMQ, and Apache Qpid. Among these, RabbitMQ stands out due to its extensive feature set, reliability, and performance. It is widely used in microservice architectures and large-scale applications where high availability and fault tolerance are critical.

RabbitMQ is known for its ease of use, robust community support, and scalability. It provides a wide range of messaging patterns, including direct, fan-out, and topic exchanges, which enable flexible message routing. RabbitMQ's rich feature set makes it a preferred choice for systems requiring high-performance, reliable messaging.

3) AMQP Workflow: The typical message flow in an AMQP-based system involves several steps:

1) **Producer Sends a Message:** A producer application sends a message to an exchange within the AMQP broker.
2) **Exchange Routes the Message:** The exchange processes the incoming message and routes it to one or more queues based on routing rules.
3) **Consumer Subscribes to the Queue:** A consumer subscribes to a queue and retrieves messages for processing.
4) **Message Processing:** The consumer processes the message and sends an acknowledgment back to the broker to confirm successful processing.
5) **Message Acknowledgment:** The consumer acknowledges the message, signaling to the broker that the message has been handled and can be removed from the queue.
6) **Persistence and Delivery Guarantees:** The broker ensures that the message is stored reliably in the queue and delivers it to the consumer even in the case of system failures.

4) Relevance to This Paper: In this paper, the integration of an SMTP server with an AMQP broker, specifically RabbitMQ, allows for efficient forwarding of received emails to downstream services for processing. By utilizing AMQP's messaging capabilities, the system can be seamlessly integrated with other services, such as email archiving systems, analytics engines, or notification services.

The use of AMQP in this context offers several advantages: it provides a scalable architecture, allows asynchronous processing, and decouples the email reception process from subsequent processing stages. By forwarding email content to an AMQP broker, the system can easily integrate with a range of microservices, each responsible for handling specific tasks such as spam filtering, archiving, or analytics. This enhances the flexibility and modularity of the email processing workflow, making it suitable for modern cloud-native architectures.

C. Rust Programming Language and Tokio Runtime

Rust is a systems programming language that focuses on performance, safety, and concurrency. [5] It is designed to overcome the limitations of C and C++ while providing memory safety guarantees without a garbage collector. Rust achieves this through its ownership system, which ensures that memory is managed efficiently, preventing common bugs such

as null pointer dereferencing, data races, and buffer overflows. [5]

Rust's key features include:

- **Memory safety without garbage collection:** The ownership system ensures that memory is automatically freed when no longer in use, preventing memory leaks and dangling references. [5]
- **Concurrency and parallelism:** Rust's model of ownership and borrowing allows for safe concurrency, where multiple threads can run simultaneously without causing data races. [5]
- **Performance:** Rust's performance is on par with C and C++, making it ideal for applications where low-level control and speed are critical. [5]
- **Rich ecosystem and tooling:** The Rust ecosystem, including Cargo (the package manager) and Rust's powerful compiler, enables developers to efficiently manage dependencies, run tests, and build applications. [5]

Rust is particularly suitable for building high-performance network services and systems that require safe memory management and efficient concurrency handling, which makes it a compelling choice for our lightweight SMTP server implementation. [5]

1) Tokio Runtime: Tokio is an asynchronous runtime for Rust, designed to handle I/O-bound operations efficiently. [6] Provides a foundation for building reliable and scalable network applications. Tokio's key features include asynchronous I/O, concurrency support, and lightweight task scheduling, making it an ideal choice for server applications that need to handle a large number of concurrent connections. [6]

Tokio is built around the 'async'/'await' syntax in Rust, which allows developers to write asynchronous code that looks and behaves like synchronous code, making it easier to read and maintain. [5], [6] The Tokio runtime manages the execution of asynchronous tasks, ensuring that they are scheduled and executed efficiently across available threads. [6]

Key features of Tokio:

- **Asynchronous I/O:** Tokio uses non-blocking I/O, allowing it to handle multiple tasks concurrently without blocking the thread. [6]
- **Lightweight concurrency:** Tokio's task model allows applications to scale efficiently without the overhead of creating and managing many threads. [6]
- **Integration with other libraries:** Tokio integrates seamlessly with Rust libraries for networking, file I/O, and database access, providing a complete toolset for building asynchronous applications. [6]
- **Ecosystem support:** With libraries such as 'tokio-tungstenite' for WebSockets and 'tokio-smtp' for SMTP, Tokio offers an extensive ecosystem that simplifies network programming in Rust. [5], [6]

In the context of this project, Tokio enables the efficient handling of multiple simultaneous SMTP connections. [6] The asynchronous nature of Tokio ensures that the server can handle many connections concurrently without blocking the main thread, making it a perfect fit for the lightweight SMTP server that is required for high performance and scalability.

III. Related Work

SMTP servers have been a foundational component of email communication for many years. The traditional SMTP server solutions, such as Postfix [2] and others, are widely used for email handling but come with significant complexity and resource consumption. These full-featured servers are often overkill for lightweight applications that require only basic email forwarding and handling capabilities.

Several lightweight SMTP server implementations have emerged as alternatives to the heavyweight solutions, such as 'msmtp' and 'Nullmailer'. These servers are designed for simplicity and ease of configuration, focusing on sending emails with minimal resource usage. However, they do not support advanced features like message queuing or integration with message brokers for further processing, making them unsuitable for modern, scalable email systems.

The integration of email systems with message brokers like RabbitMQ has been explored in the context of microservices and email processing. AMQP-based brokers enable email data to be decoupled and forwarded to other systems for further processing, archiving, or database storage. However, while RabbitMQ has been widely adopted in microservice architectures, there is limited research on lightweight SMTP servers that directly forward email content to RabbitMQ for processing.

IV. Proposed System Design

The proposed system design consists of two main components: the Lightweight SMTP Server (LSMTP) and the RabbitMQ [7] message broker. LSMTP is responsible for receiving emails via the SMTP protocol and forwarding them to RabbitMQ for further processing. RabbitMQ acts as the central message broker, routing email messages to downstream services based on predefined rules.

A. LSMTP Architecture

The architecture of the Lightweight SMTP Server (LSMTP) is designed to be minimalistic and efficient, focusing on core SMTP functionality. The server is implemented in Rust using the Tokio asynchronous runtime to handle concurrent connections and optimize resource utilization. The key components of the LSMTP architecture are as follows:

- **TCP Listener**: LSMTP listens for incoming SMTP connections on port 25, the standard SMTP port. It establishes a TCP listener using Tokio's asynchronous I/O capabilities to handle multiple concurrent connections efficiently.
- **SMTP Handler**: Upon accepting a new connection, LSMTP spawns a new asynchronous task to handle the SMTP session. The handler processes SMTP commands (HELO, MAIL FROM, RCPT TO, DATA, QUIT) and forwards the email content to RabbitMQ.

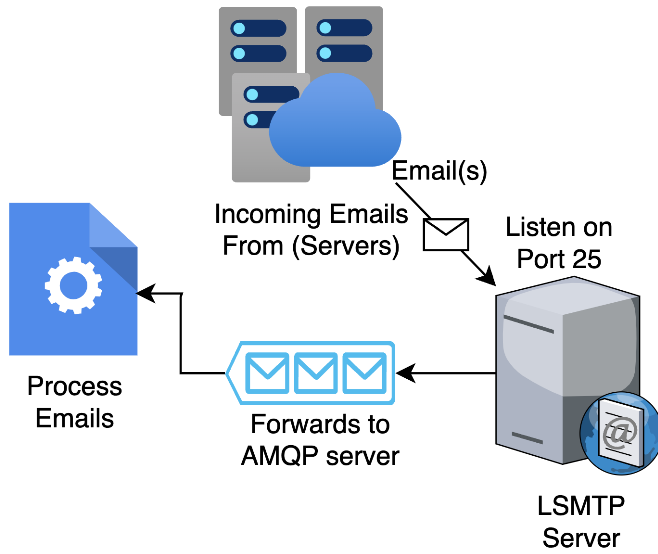

Fig. 2. System Architecture: SMTP Server receiving emails and forwarding them to RabbitMQ.

- **Email Forwarding**: The SMTP handler extracts the email content received over the DATA command and forwards it to RabbitMQ for further processing. The email headers and body are encapsulated in an AMQP message and sent to a predefined exchange.
- **Error Handling**: LSMTP includes robust error handling mechanisms to manage connection failures, protocol violations, and other exceptional conditions. It gracefully closes connections and logs errors for debugging and monitoring.

The LSMTP architecture is designed to be modular and extensible, allowing easy integration with other services and protocols. By focusing on core SMTP functionality and leveraging Rust's safety features, LSMTP provides a reliable and efficient solution for email reception and forwarding.

B. Integration with RabbitMQ

The integration of LSMTP with RabbitMQ enables seamless email forwarding and processing within a microservices architecture. RabbitMQ acts as the central message broker, receiving email messages from LSMTP and routing them to downstream services based on predefined rules. The integration workflow is as follows:

1) **Email Reception**: LSMTP receives an email message from an SMTP client and extracts the email content. [1]
2) **AMQP Message Creation**: LSMTP encapsulates the email content in an AMQP message, including headers and body.
3) **Message Routing**: LSMTP publishes the AMQP message to a predefined exchange in RabbitMQ. [3], [7]
4) **Consumer Subscription**: Downstream services subscribe to the RabbitMQ exchange to receive email messages for processing.
5) **Message Processing**: Consumers process email messages according to predefined rules, such as archiving, filtering, or forwarding.

Integration with RabbitMQ enhances the flexibility and scalability of the email processing workflow. By leveraging RabbitMQ's advanced messaging features, LSMTP enables seamless communication between email reception and downstream processing services, making it suitable for modern distributed systems.

C. System Assumptions

The proposed system design makes the following assumptions:

- **Network Connectivity**: The system assumes stable network connectivity between LSMTP and RabbitMQ for message forwarding.
- **AMQP Configuration**: RabbitMQ is pre-configured with the necessary exchanges, queues, and bindings to route email messages to downstream services.
- **Email Content Parsing**: LSMTP assumes that the email content received over SMTP is well-formed and adheres to standard email formatting rules.
- **Concurrent Connections**: LSMTP is designed to handle multiple concurrent SMTP connections efficiently using Tokio's asynchronous runtime.

These assumptions guide the design and implementation of the system, ensuring that it meets the requirements of modern email processing workflows.

The proposed server operates as a lightweight layer between email senders and RabbitMQ. Figure 2 illustrates the architecture of the system.

V. Implementation Results

The implementation of the Lightweight SMTP Server (LSMTP) in Rust using the Tokio asynchronous runtime demonstrates high performance and efficiency in handling email reception and forwarding. The server successfully processes incoming SMTP connections, extracts email content, and forwards it to RabbitMQ for further processing. The key results of the implementation are as follows.

A. Performance Benchmarks

- **Connection Handling**: LSMTP efficiently handles multiple concurrent SMTP connections, demonstrating low latency and high throughput.
- **Resource Utilization**: The server optimizes resource consumption, utilizing minimal CPU and memory for email processing.
- **Error Handling**: LSMTP gracefully manages connection errors, protocol violations, and other exceptional conditions, ensuring robustness and reliability.

Performance benchmarks highlight the efficiency and scalability of LSMTP in processing email messages and forwarding them to RabbitMQ for further processing. The server's lightweight design and asynchronous architecture make it well-suited for modern microservice architectures requiring high throughput and low latency.

Table I presents a comparison of different configurations, showing how throughput and latency vary based on concurrency and client count. The results demonstrate that increasing concurrency significantly improves throughput, although it may also impact average latency.

TABLE I
PERFORMANCE COMPARISON OF LSMTP RELAY SERVER

Clients	Concurrency	Total Emails	Throughput (emails/sec)	Latency(s)
100	10	14,074	505	2.03
200	20	23,276	1,880	2.08
300	30	54,350	4,385	2.03
400	4	15,445	918	1.76
400	10	44,178	1,740	2.19
800	5	20,513	1,697	2.14

Note: All tests were conducted on a small 2 vCPU, 4GB RAM server running a fresh Debian installation with default, non-optimized settings. The client and server experienced an approximate network latency of 315ms.

VI. CONCLUSION

This paper introduces a lightweight SMTP server optimized for high-performance email handling in scenarios where simplicity and resource efficiency are key. By focusing on a minimal subset of the SMTP protocol and integrating with an AMQP broker, the system provides a scalable solution for microservices, email archiving, and forwarding use cases. Its design results in significant improvements in performance and resource usage compared to traditional SMTP solutions like Postfix. [1], [2]

However, this simplicity comes with trade-offs. The server omits advanced features such as TLS encryption, authentication, and spam filtering, making it unsuitable for environments where security is critical. Furthermore, the lack of built-in failover mechanisms presents limitations for highly critical applications.

Future work will focus on addressing these limitations by implementing security features (e.g., TLS), introducing traffic management tools (e.g., throttling), and supporting additional message brokers beyond AMQP. These enhancements aim to broaden the applicability of the lightweight SMTP server for more complex use cases.

The implementation of the Lightweight SMTP Server (LSMTP) is available on GitHub as an open-source project, enabling further development and contributions from the community. [4]

VII. ACKNOWLEDGMENT

I would like to thank my peers and mentors who provided valuable feedback during the development of this project. Special thanks to the developers of the Rust programming language, Tokio, and RabbitMQ, whose tools made this project possible.

REFERENCES

[1] J. Klensin, "Simple mail transfer protocol," https://tools.ietf.org/html/rfc5321, 2008, rFC 5321, October 2008, Accessed: 2025-01-15.
[2] P. Team, "Postfix: The mail transfer agent," http://www.postfix.org/, 2025, accessed: 2025-01-15.
[3] OASIS, "Amqp: Advanced message queuing protocol," https://www.amqp.org/, 2025, accessed: 2025-01-15.
[4] Nikhil, "Lightweight smtp server for high-performance mail processing with amqp integration," https://github.com/Neko-Nik/LSMTP, 2025, accessed: 2025-02-12.
[5] R. P. Language, "The rust programming language," https://www.rust-lang.org/, 2025, accessed: 2025-01-15.
[6] T. Team, "Tokio: A rust asynchronous runtime," https://tokio.rs/, 2020, accessed: 2025-01-15.
[7] P. M. Graff, "Rabbitmq: A message broker," https://www.rabbitmq.com/, 2018, accessed: 2025-01-15.