Distributed System with RabbitMQ

Student:

Daniela Cadena
Universidad Nacional de Ingeniería, Faculty of Sciences
e-mail: daniela.cadena.v@uni.pe

Student:

Eladio Huamani
Universidad Nacional de Ingeniería, Faculty of Sciences
e-mail: eladio.huamani.v@uni.pe

Course:

CC4P1 Concurrent Programming and Distributed Systems
Laboratory 03

Abstract

This report presents the design, implementation, and deployment of a distributed system that integrates multiple programming languages, databases, and services through a message-oriented middleware architecture using RabbitMQ. The system is designed to process user data in a decentralized manner, validating identity information before persisting it across distinct data sources.

The project consists of three main services: a Node.js client application that collects user input and publishes it to a message queue; a Java-based service that validates the provided DNI (national ID) against a MariaDB database; and a Python-based service that persists validated user information into a PostgreSQL database, including the validation of friend IDs. These services communicate asynchronously through RabbitMQ, using two message queues to separate concerns and ensure a reliable, decoupled workflow.

Each service is containerized using Docker, and all containers are orchestrated via Docker Compose, enabling consistent deployment, modular development, and network isolation. SQL initialization scripts are executed automatically when the containers start, ensuring that database schemas are ready for immediate operation.

The system leverages durable queues, persistent messages, and explicit acknowledgements to achieve fault tolerance and message reliability. Multithreading is employed in the Python service to improve throughput and handle concurrent user insertions efficiently. A stress-testing script was developed to simulate 1000 user entries, allowing performance analysis under load.

This implementation reflects real-world distributed system challenges such as service coordination, message validation, data consistency, and container orchestration. The report concludes by evaluating the system's robustness and discussing potential improvements in security, monitoring, and scalability.

Keywords: parallelism, Java, multithreading, matrix multiplication, Jacobi method.

Contents

1 Introduction

This report documents the development of a distributed system using RabbitMQ as middleware. The system is composed of three main components:

- A **Node.js client** that sends user data.
- A **Java service** that validates the DNI in MariaDB.
- A **Python service** that saves validated users in PostgreSQL.

The architecture simulates a real-world scenario involving asynchronous message validation and persistence across independent nodes.

2 System Architecture

2.1 Components

- **SO1**: Linux system running PostgreSQL and the Python service.
- **SO2**: Linux system with the Java validator and MariaDB.
- **SO3**: Client system using Node.js for user input.

2.2 Languages and Roles

- **LP1**: Python (Data persistence in BD1).
- **LP2**: Java (DNI validation in BD2).
- **LP3**: Node.js (Client UI/interaction).

3 Database Schemas

3.1 BD1 (PostgreSQL)

Stores validated user information:

- `id`
- `nombre`
- `correo`
- `clave`
- `dni`
- `telefono`
- `amigos` (array of IDs)

3.2 BD2 (MariaDB)

Holds reference records for DNI validation:

- `id`
- `dni`
- `nombre`
- `apellidos`
- `ubigeo`
- `direccion`

4 Communication Protocol

4.1 Message Flow

1. The Node.js client sends user data to RabbitMQ queue `validar_dni`.
2. The Java service validates the DNI using BD2.
3. If valid, the message is published to `validar_guardar_usuario`.
4. The Python service consumes the message and saves the user in BD1.

4.2 Protocol Diagram

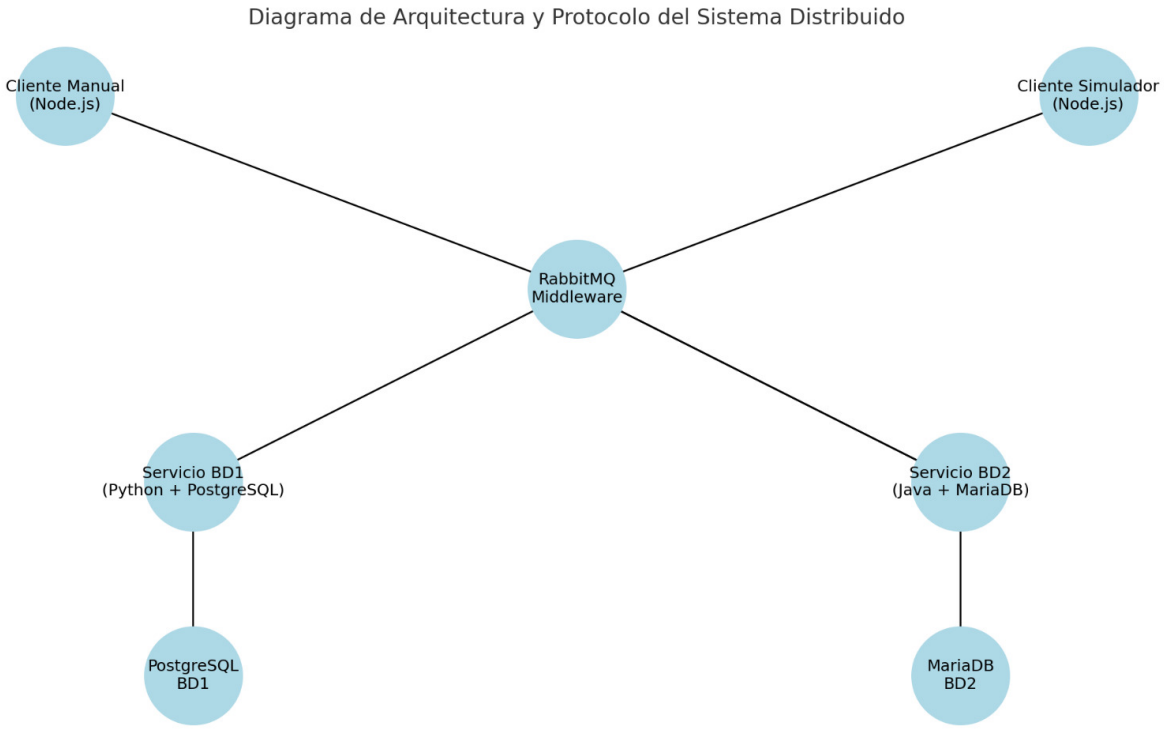

Figure 1: Message Flow and Validation Protocol

5 Middleware: RabbitMQ in Our Architecture

RabbitMQ is an open-source message broker that acts as a middle layer between producers (senders of data) and consumers (receivers of data). In this project, it enables asynchronous, decoupled communication between the client, validator, and database writer services.

5.1 Basic Concepts of RabbitMQ

RabbitMQ is based on the Advanced Message Queuing Protocol (AMQP), and uses the following core components:

- **Producer**: The component that sends messages (our Node.js client).
- **Queue**: A buffer that stores messages until they are processed.
- **Consumer**: The component that receives and processes messages (Java and Python services).
- **Exchange (default in our case)**: Routes messages to queues. We use the default exchange for simplicity.

5.2 Queues Used in This Project

1. `validar_dni`: Receives messages from the Node.js client. Consumed by the Java service to validate DNI.
2. `validar_guardar_usuario`: Receives messages only after successful validation. Consumed by the Python service to save users.

5.3 Message Flow with RabbitMQ

The workflow is as follows:

1. The Node.js client gathers user data and publishes it to the `validar_dni` queue.
2. The Java service connects to RabbitMQ, consumes the message, and extracts the DNI.
3. If the DNI exists in MariaDB (BD2), the service republishes the message to the `validar_guardar_usuario` queue.
4. The Python service listens on this second queue and saves the data into PostgreSQL (BD1).

5.4 Connecting to RabbitMQ

Each service connects to RabbitMQ using its respective client library:

- **Node.js**: Uses `amqplib` to connect via `amqp://guest:guest@rabbitmq:5672`.
- **Java**: Uses `com.rabbitmq.client` and establishes a connection using a retry loop to ensure RabbitMQ is available.
- **Python**: Uses `pika` and `BlockingConnection` to listen and acknowledge messages.

5.5 Durability and Acknowledgements

To ensure fault tolerance and avoid message loss:

- All queues are declared as `durable=true`.
- Messages are published as `persistent=true`.
- Acknowledgements (`ack`) are used in Java and Python services to confirm successful processing.

5.6 Benefits of RabbitMQ in This System

- **Decoupling**: Each service operates independently of the others.
- **Scalability**: Consumers can be scaled horizontally (e.g., multiple Python consumers).
- **Fault Tolerance**: Messages are not lost even if one service is temporarily down.
- **Asynchronous Processing**: Improves responsiveness and throughput.

5.7 Diagram of RabbitMQ Integration

6 Implementation Details

6.1 Node.js Client

Collects user input via `inquirer` and sends it to RabbitMQ.

```
channel.sendToQueue(queue, Buffer.from(JSON.stringify(datos)), { persistent: true
    });
```

Listing 1: send.js - Node.js Client

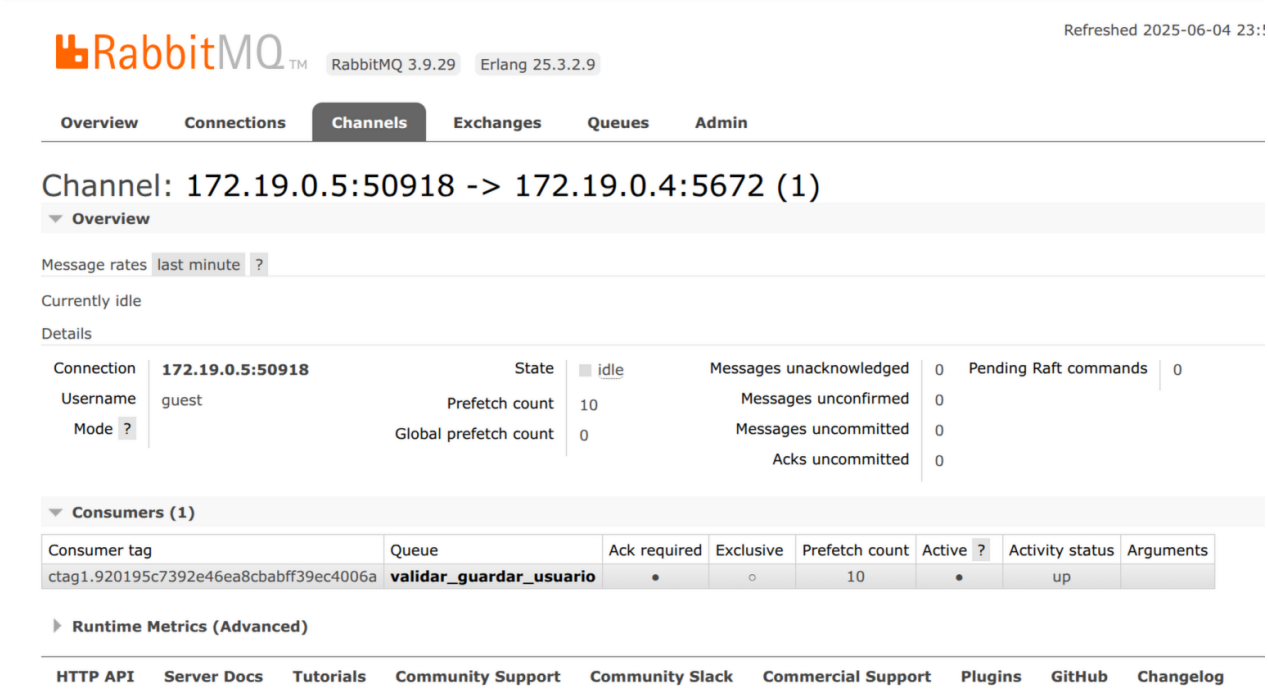

Figure 2: RabbitMQ Message Flow and Service Integration

6.2 Java DNI Validator

Consumes from `validar_dni`, validates the DNI using MariaDB, and publishes to `validar_guardar_usuario`.

```
Matcher m = Pattern.compile("\"dni\"\\s*:\\s*\"(\\d{8})\"").matcher(json);
...
if (existe) {
    ch.basicPublish("", "validar_guardar_usuario", null, message.getBody());
}
```

Listing 2: ValidadorDNI.java

6.3 Python Consumer

Consumes from `validar_guardar_usuario`, validates friends, and saves to PostgreSQL.

```
cur.execute("INSERT INTO usuarios (...) VALUES (...)", (...))
```

Listing 3: consumer_bd1.py

6.4 Concurrency

The Python consumer uses threads to improve throughput and avoid data corruption:

```
hilo = threading.Thread(target=guardar_usuario, args=(data,))
hilo.start()
```

7 Containerization with Docker and the Role of Dockerfiles

The development and deployment of this distributed system heavily rely on Docker, an open-source platform designed to automate the deployment of applications as portable, self-sufficient containers.

7.1 Why Docker?

Distributed systems often involve services written in different languages, running on different environments, and depending on a multitude of software components. Docker solves these complexities by offering:

- **Consistent Environments**: Every container encapsulates the runtime, libraries, and dependencies required by the service.
- **Service Isolation**: Each service runs in its own container, eliminating side effects and port conflicts.
- **Easy Scaling**: Containers can be replicated or replaced quickly, making load balancing and scaling easier.
- **Simplified Onboarding and Deployment**: With Docker, the full system can be started with a single command, ensuring all team members or machines run exactly the same stack.

7.2 Docker Compose Orchestration

To simplify service orchestration, `docker-compose` was used. The configuration file defined six services:

1. `rabbitmq`: Message broker used for decoupled communication.
2. `bd1 (PostgreSQL)`: Stores validated users. Initialized via SQL script.
3. `bd2 (MariaDB)`: Stores official DNI records. Also initialized via script.
4. `cliente-simulador (Node.js)`: CLI client that gathers user input and pushes messages into RabbitMQ.
5. `servicio_bd1 (Python)`: Listens to validation-passed messages and saves them into BD1.
6. `servicio_bd2 (Java)`: Listens for DNI validation requests and checks them in BD2.

Each service runs in an isolated container, yet communicates through an internal Docker network. This closely mimics real-world microservice deployments.

7.3 The Role of Dockerfiles

Each custom service was built from a `Dockerfile`, which defines how the container is constructed. The Dockerfile serves as a blueprint, specifying base images, dependencies, working directories, and the command to run.

Node.js Client - `cliente-simulador`

- Based on the `node:18-alpine` image.
- Installs project dependencies using `npm`.
- Sets the working directory and the default command to run the interactive script.

Java Validator - `servicio_bd2`

- Based on an OpenJDK image (`openjdk:17`).
- Compiles the Java source code.
- Uses a loop to retry RabbitMQ connection, improving reliability in containerized environments.

Python Consumer - `servicio_bd1`

- Based on `python:3.11-slim`.
- Installs required libraries like `psycopg2` and `pika`.
- Executes a threaded consumer script to allow concurrent data processing.

Each Dockerfile ensures that the final container image is lightweight, secure, and built precisely for its service's role.

7.4 Lifecycle and Execution Flow

When running `docker-compose up`, the following execution flow occurs:

1. RabbitMQ, BD1 (PostgreSQL), and BD2 (MariaDB) containers start with exposed ports and environment variables.
2. Initialization SQL scripts are executed automatically inside BD1 and BD2, creating the necessary tables.
3. The Java validator waits for RabbitMQ availability, then starts listening to the `validar_dni` queue.
4. The Python consumer does the same, but listens to `validar_guardar_usuario`.
5. The Node.js container remains interactive, allowing for manual user data entry and message dispatch.

7.5 Dockerfile Robustness in Distributed Systems

The use of well-structured Dockerfiles played a crucial role in the reproducibility and reliability of the system. For example:

- By explicitly defining retry logic inside the Java and Python services, the system becomes resilient to race conditions during startup.
- Volume mounting of SQL files allows database initialization without manual intervention.
- Docker's network bridge facilitates secure and seamless communication between services.
- Keeping the Dockerfiles minimal and clean helped reduce image sizes and improved build times.

7.6 Final Thoughts

Without Docker, managing the interplay between three different languages, two databases, and a message broker would require significant manual configuration and coordination. Docker abstracts these complexities, making the system modular, scalable, and production-ready.

8 Testing and Evaluation

8.1 Performance Evaluation

A stress test script was developed to send 1000 random entries to RabbitMQ, evaluating end-to-end throughput and resilience.

8.2 Validation

Both BD1 and BD2 were queried to verify correct data insertion and validation logic.

8.3 Manual Input

The system also supports interactive user registration from the client terminal.

9 Deployment

9.1 Local Setup

Each service was containerized and run using Docker Compose. Network links simulate distributed communication.

10 Conclusion

This distributed system successfully demonstrates middleware-based communication and asynchronous data processing using RabbitMQ, Java, Python, and Node.js. It validates real-world scenarios of data integrity, inter-service messaging, and multithreaded persistence.

11 Appendices

11.1 Source Code Files

- Node.js client: `send.js`
- Java validator: `ValidadorDNI.java`
- Python consumer: `consumer_bd1.py`
- Docker compose: `docker-compose.yml`

```
const amqp = require('amqplib');

const inquirer = require('inquirer');

async function main() {
    // Pedimos datos al cliente
    const datos = await inquirer.prompt([
        { name: 'nombre', message: 'Nombre: ' },
        { name: 'correo', message: 'Correo:' },
        { name: 'clave', message: 'Clave:' },
        { name: 'dni', message: 'DNI (8 digitos):' },
        { name: 'telefono', message: 'Telefono:' },
        { name: 'amigos', message: 'IDs de amigos (coma-separaods, opcional):' },
    ]);

    // Normalizar amigos
    datos.amigos = datos.amigos
    ? datos.amigos.split(',').map(a => a.trim()).filter(a => a !== '')
    : [];

    // conectar con RabbitMq
    const connection = await amqp.connect('amqp://guest:guest@rabbitmq:5672');
    const channel = await connection.createChannel();

    // garantizar que la cola exista y enviar mensaje persistente
    const queue = 'validar_dni';
    await channel.assertQueue(queue, { durable: true });

    // enviar mensaje
    console.log('Enviando mensaje...');
    channel.sendToQueue(queue, Buffer.from(JSON.stringify(datos)), { persistent:
        true });
    console.log('Mensaje enviado');

    console.log('Datos enviados a la cola: ', queue);
```

```
    setTimeout(() => {
        connection.close();
    }, 500);
}

main().catch( err => {
    console.error('Error enviando: ', err);
    process.exit(1);
});
```

Listing 4: send.js

```
package sd.middleware;

import com.rabbitmq.client.*;
import java.sql.*;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.regex.*;

public class ValidadorDNI {

    public static void main(String[] args) throws Exception {
        // Conexi n a RabbitMQ
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("rabbitmq");
        factory.setPort(5672);

        com.rabbitmq.client.Connection rabbitConn = null;
        int retries = 60;
        while (retries-- > 0) {
            try {
                rabbitConn = factory.newConnection();
                break;
            } catch (Exception e) {
                System.out.println("RabbitMQ no disponible, reintentando...");
                Thread.sleep(5000);
            }
        }

        if (rabbitConn == null) throw new RuntimeException("No se pudo conectar a
            RabbitMQ");

        Channel ch = rabbitConn.createChannel();
        ch.queueDeclare("validar_dni", true, false, false, null);
        ch.queueDeclare("validar_guardar_usuario", true, false, false, null);

        // Conexi n a MariaDB
        java.sql.Connection db =
            DriverManager.getConnection("jdbc:mariadb://bd2:3306/bd2", "root",
            "root");

        // Callback
        DeliverCallback callback = (consumerTag, message) -> {
            new Thread(() -> {
                String json = new String(message.getBody(),
                    StandardCharsets.UTF_8);
                long tag = message.getEnvelope().getDeliveryTag();

                // Extraer DNI con expresi n regular
                Matcher m =
                    Pattern.compile("\"dni\"\\s*:\\s*\"(\\d{8})\"").matcher(json);
```

```
                if (!m.find()) {
                    System.out.println("DNI no encontrado en: " + json);
                    ack(ch, tag);
                    return;
                }

                String dni = m.group(1);
                boolean existe = false;

                try (PreparedStatement ps = db.prepareStatement("SELECT 1 FROM
                    personas WHERE dni = ?")) {
                    ps.setString(1, dni);
                    existe = ps.executeQuery().next();
                } catch (SQLException e) {
                    System.err.println("Error al validar DNI " + dni);
                    e.printStackTrace();
                }

                if (existe) {
                    try {
                        ch.basicPublish("", "validar_guardar_usuario", null,
                            message.getBody());
                        System.out.println("DNI v lido     reenviado: " + dni);
                    } catch (IOException e) {
                        System.err.println("Error reenviando mensaje: " +
                            e.getMessage());
                    }
                } else {
                    System.out.println("DNI no existe: " + dni);
                }

                ack(ch, tag);
            }).start();
        };

        ch.basicConsume("validar_dni", false, callback, tag -> {});
        System.out.println("Validador DNI escuchando ");
    }

    // Asegura que el ack se haga incluso si hay errores
    private static void ack(Channel ch, long tag) {
        try {
            ch.basicAck(tag, false);
        } catch (IOException e) {
            System.err.println("Error haciendo ACK del mensaje:");
            e.printStackTrace();
        }
    }
}
```

Listing 5: ValidadorDNI.java

```
const amqp = require('amqplib');

const inquirer = require('inquirer');

async function main() {
    // Pedimos datos al cliente
    const datos = await inquirer.prompt([
        { name: 'nombre', message: 'Nombre: ' },
        { name: 'correo', message: 'Correo:' },
        { name: 'clave', message: 'Clave:' },
```

```
        { name: 'dni', message: 'DNI (8 digitos):' },
        { name: 'telefono', message: 'Telefono:' },
        { name: 'amigos', message: 'IDs de amigos (coma-separaods, opcional):' },
    ]);

    // Normalizar amigos
    datos.amigos = datos.amigos
    ? datos.amigos.split(',').map(a => a.trim()).filter(a => a !== '')
    : [];

    // conectar con RabbitMq
    const connection = await amqp.connect('amqp://guest:guest@rabbitmq:5672');
    const channel = await connection.createChannel();

    // garantizar que la cola exista y enviar mensaje persistente
    const queue = 'validar_dni';
    await channel.assertQueue(queue, { durable: true });

    // enviar mensaje
    console.log('Enviando mensaje...');
    channel.sendToQueue(queue, Buffer.from(JSON.stringify(datos)), { persistent:
        true });
    console.log('Mensaje enviado');

    console.log('Datos enviados a la cola: ', queue);
    setTimeout(() => {
        connection.close();
    }, 500);
}

main().catch( err => {
    console.error('Error enviando: ', err);
    process.exit(1);
});
```

Listing 6: consumer_bd1.py

```
services:
  rabbitmq:
    image: rabbitmq:3.9-management
    container_name: rabbitmq
    ports:
      - "5672:5672"
      - "15672:15672"
    environment:
      RABBITMQ_DEFAULT_USER: guest
      RABBITMQ_DEFAULT_PASS: guest

  bd1:
    image: postgres:15
    container_name: postgres
    environment:
      POSTGRES_DB: bd1
      POSTGRES_USER: user
      POSTGRES_PASSWORD: pass
    volumes:
      - ../sql/bd1_postgres.sql:/docker-entrypoint-initdb.d/init.sql
    ports:
      - "5432:5432"

  bd2:
    image: mariadb:10.11
```

```
    container_name: mariadb
    environment:
      MARIADB_DATABASE: bd2
      MARIADB_ROOT_PASSWORD: root
    volumes:
      - ../sql/bd2_mariadb.sql:/docker-entrypoint-initdb.d/init.sql
    ports:
      - "3306:3306"

  cliente-simulador:
    build:
      context: ../cliente-nodo
    container_name: cliente-simulador
    depends_on:
      - rabbitmq
    tty: true
    stdin_open: true

  servicio_bd1:
    container_name: service-bd1
    build:
      context: ../servicio-bd1-python
    depends_on:
      - rabbitmq
      - bd1
    command: python consumer_bd1.py

  servicio_bd2:
    container_name: service-bd2
    build:
      context: ../servicio-bd2-java
    depends_on:
      - rabbitmq
      - bd2
```

Listing 7: docker-compose.yml