

Informe Técnico: Revisión Conceptual y Análisis de la Transición a RabbitMQ

Presentado por

**Santiago Vargas Martinez
Jhonatan Steven Barón**

**Presentado a:
Ing. Martín Santiago Chiquillo**

**Curso:
Sistemas Distribuidos**

**Universidad Pedagógica y Tecnológica de Colombia.
Facultad de Ingeniería**

1. Introducción.....	3
2. Revisión Conceptual:.....	3
3. Análisis del Sistema Original.....	5
4. Transición a RabbitMQ: Cambios Realizados.....	5
5. Justificación del Rediseño con RabbitMQ.....	11
6. Lecciones Aprendidas.....	11
7.Diagrama de Arquitectura.....	12
8. Conclusiones:.....	13

Ingeniería de Sistemas y Computación

Tunja-2025

1. Introducción

Este informe está hecho como una revisión conceptual sobre RabbitMQ y su implementación en un ejercicio práctico de clase; además brinda un análisis detallado de la transición del sistema desde lo realizado previamente en el parcial 2 con una arquitectura basada en comunicación en HTTP a una arquitectura orientada a mensajería mediante el uso de RabbitMQ. Dicha transformación en este caso responde a necesidades de desacoplamiento, escalabilidad y resiliencia del sistema.

2. Revisión Conceptual:

- ¿Qué es RabbitMQ?

RabbitMQ según la documentación es un “message broker” de código abierto, que se usa ampliamente en arquitecturas distribuidas para habilitar la comunicación asíncrona entre varios servicios. Utiliza el protocolo avanzado de cola de mensajes ó AMQP, usado como estándar abierto para enviar mensajes entre aplicaciones y organizaciones, aunque también soporta otros protocolos como lo son MQTT (MQ Telemetry Transport) ó STOMP(Protocolo Simple de Mensajería Orientada a Texto).

- **Función en una Arquitectura Distribuida**

RabbitMQ tiene funciones interesantes ya que permite que los servicios no se comuniquen directamente entre sí, sino a través de colas y mensajes. Esto ofrece ventajas como lo son el desacoplamiento, ya que los servicios no necesitan conocer la ubicación ni el estado de otros servicios; la escalabilidad pues se pueden escalar servicios productores y consumidores de forma independiente, también tenemos la resiliencia donde los mensajes se almacenan en colas si un consumidor falla temporalmente y la distribución de carga donde varias instancias de consumidores pueden trabajar en paralelo.

- **Comparativa: RabbitMQ vs. HTTP Directo**

Característica	Comunicación HTTP Directo	Comunicación con RabbitMQ
Acoplamiento	Alto	Bajo
Sincronía	Síncrona	Asíncrona
Resiliencia	Baja	Alta
Escalabilidad	Limitada	Independiente
Disponibilidad	Requiere ambos servicios	Funciona si uno está caído
Flujo de trabajo	Simple	Ideal para eventos y tareas asíncronas

- **Elementos Clave de RabbitMQ**

Los principales elementos que se presentan dentro de la herramienta se muestran gráficamente y describen a continuación.

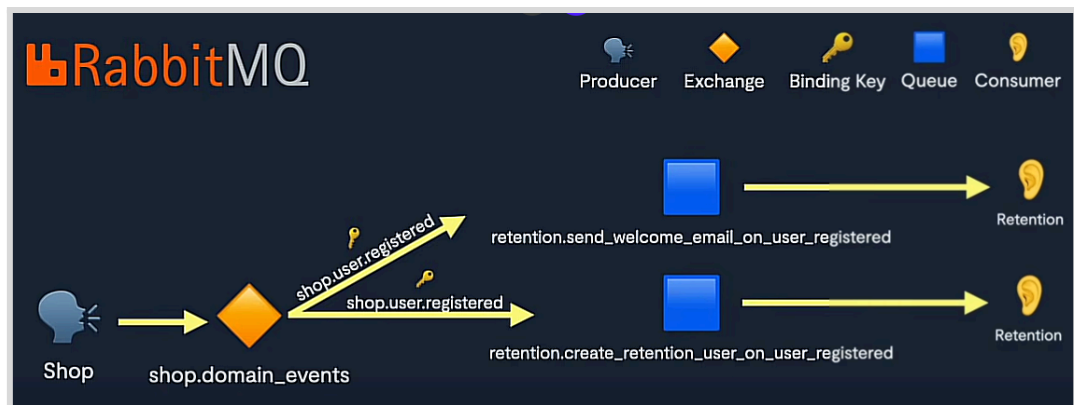


Figura 1. Elementos principales en RabbitMQ

- **Publisher:** Servicio que publica mensajes.
- **Exchange:** Recibe mensajes del publisher y los enruta según reglas hacia colas.
- **Queue:** Almacena mensajes hasta que sean procesados por consumidores.
- **Consumer:** Servicio que consume y procesa mensajes desde una cola.
- **Binding:** Regla que conecta un exchange con una queue, basada en claves de enrutamiento (routing keys).

3. Análisis del Sistema Original

- En cuanto al sistema original tenemos que los servicios cliente-uno y cliente-dos actuaban como productores de eventos, enviando periódicamente solicitudes HTTP tipo POST. El receptor de estos eventos era el servicio api-reporte, encargado de procesar directamente cada solicitud que llegaba.

```
const sendRequest = async () => {
  try {
    const response = await axios.post('http://analitica:3000/panel', {}, {
      headers: { 'X-Service-ID': CLIENT_ID },
      auth: { username: 'admin', password: 'password' }
    });
    console.log(`Response from analitica: ${response.data}`);
  } catch (error) {
    console.error(`Error sending request: ${error.message}`);
  }
};
```

Figura 2. Caso complementario de implementación original de petición tipo POST

Con lo anterior teníamos que los clientes estaban estrechamente conectados con el api-reporte, ya que conocían su URL y manejaban directamente su autenticación; Esto generaba una dependencia fuerte, donde cualquier falla o cambio en api-reporte afectaba de inmediato el funcionamiento de los clientes; Además si api-reporte estaba caído o lento, las solicitudes de los clientes también fallarían o se retrasarían, afectando su operación (aunque en este caso simple, solo se logueaba un error).

4. Transición a RabbitMQ: Cambios Realizados

- **Incorporación de RabbitMQ**

Se integró un contenedor de RabbitMQ en docker-compose.yml, y añadidamente se rediseñaron varios servicios para usarla comunicación por modo mensajería:

```
rabbitmq:
  image: rabbitmq:3-management-alpine
  container_name: rabbitmq
  ports:
    - "5672:5672" # AMQP protocol port
    - "15672:15672" # Management UI port
  environment:
    - RABBITMQ_DEFAULT_USER=user
    - RABBITMQ_DEFAULT_PASS=password
  networks:
    - web
  healthcheck:
    test: ["CMD", "rabbitmqctl", "status"]
    interval: 5s
    timeout: 10s
    retries: 5

networks:
  web:
    driver: bridge
```

Figura 3. Vista de integración de RabbitMQ en Docker-compose.yml

Se hicieron también modificaciones al código dentro de Cliente-App/ cliente-uno y Cliente-App/ cliente-dos

```
> Run Service
cliente-app1:
  build: ./cliente-app
  container_name: cliente-app1
  environment:
    - CLIENT_ID=app1
  labels:
    - "traefik.enable=true"
    - "traefik.http.routers.cliente-app1.rule=Host(`localhost`) && PathPrefix(`/cliente/app1`)"
    - "traefik.http.routers.cliente-app1.entrypoints=web"
    - "traefik.http.middlewares.cliente-app1-strip.stripprefix.prefixes=/cliente/app1"
    - "traefik.http.routers.cliente-app1.middlewares=cliente-app1-strip"
    - "traefik.http.services.cliente-app1.loadbalancer.server.port=3000"
  networks:
    - web
```

Figura 3. Vista de modificación de código en cliente-app1

Adicionalmente se eliminaron las llamadas HTTP con axios dentro de los archivos json en cliente-app

```
{
  "name": "cliente-app",
  "version": "1.0.0",
  "main": "server.js",
  > Debug
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "amqplib": "^0.10.7",
    "axios": "^1.0.0",
    "express": "^4.18.2"
  }
}
```

```
1 {
2   "name": "cliente-app",
3   "version": "1.0.0",
4   "main": "server.js",
5   > Debug
6   "scripts": {
7     "start": "node server.js"
8   },
9   "dependencies": {
10     "express": "^4.18.2",
11     "amqplib": "^0.10.3"
12   }
13 }
```

Figura 4. Versión preliminar Vs versión actual confirmando eliminación de axios.

Se ha logrado adicionalmente la inclusión de *amqplib* para publicar eventos al exchange *events_exchange* haciendo uso de la clave *event.analytics*.

```
1 const express = require('express');
2 const amqp = require('amqplib');
3
4 const app = express();
5 const PORT = 3000;
6 const CLIENT_ID = process.env.CLIENT_ID || 'unknown-client';
7 const RABBITMQ_URL = 'amqp://user:password@rabbitmq:5672';
8 const EXCHANGE_NAME = 'events_exchange';
9 const ROUTING_KEY = 'event.analytics';
10
11 let channel = null;
```

Figura 5. Visualización de inclusión de amqplib en cliente-app

Por otra parte dentro del servicio de Analíticas ó api-reporte se realiza la eliminación del endpoint POST /reporte además se realiza la configuración para consumir mensajes desde la queue analytics_queue.

```
8
9  const RABBITMQ_URL = 'amqp://user:password@rabbitmq:5672';
10 const EXCHANGE_NAME = 'events_exchange';
11 const QUEUE_NAME = 'analytics_queue';
12 const ROUTING_KEY = 'event.analytics';
13
```

Figura 6. Imagen de implementación de analytics_queue

Dentro del desarrollo del presente taller también se implementa la lógica para procesar eventos y mantener contadores en memoria.

```
channel.consume(q.queue, (msg) => {
  if (msg !== null) {
    const rawMessage = msg.content.toString();
    try {
      const event = JSON.parse(rawMessage);

      // Store recent message
      recentMessages.push({ timestamp: new Date().toISOString(), content: event });
      if (recentMessages.length > MAX_RECENT_MESSAGES) {
        recentMessages.shift(); // Keep only the last N
      }

      if (event.serviceId) {
        requestCounts[event.serviceId] = (requestCounts[event.serviceId] || 0) + 1;
        console.log(`Processed event from ${event.serviceId}. Current count: ${requestCounts[event.serviceId]}`);
      } else {
        console.warn('Received event without serviceId:', event);
      }
    } catch (error) {
      console.error('Error parsing message:', error);
    }
  }
});
```

Figura 7. Fragmento de la lógica de procesamiento de eventos

De manera añadida se implementan nuevos endpoints específicamente para los reportes solicitados como se muestran a continuación:

GET /reporte: Muestra estadísticas acumuladas.

```
// Endpoint to view the current status (protected by basic auth)
app.get('/reporte', auth, (req, res) => {
  // Format the counts into a plaintext string
  let report = 'Reporte de Accesos:\n';
  for (const [serviceId, count] of Object.entries(requestCounts)) {
    report += `${serviceId}: ${count}\n`;
  }
  res.type('text/plain');
  res.send(report || 'No hay datos aun.');
```

Figura 8. Fragmento de código GET/report

GET /reporte/recent: Lista los últimos eventos recibidos.

```
// Add a new endpoint to get recent messages (could be protected too if needed)
app.get('/reporte/recent', auth, (req, res) => {
  res.json(recentMessages);
});
```

Figura 9. Fragmento de código GET/report/recent

Dentro del panel visual ó Panel se hizo el reemplazo de Nginx por un servidor dinámico en [Node.js/Express](https://nodejs.org/en/express/).

```
panel > JS server.js > [x] express

1  const express = require('express');
2  const axios = require('axios');
3
4  const app = express();
5  const PORT = 3000;
6  const API_REPORT_URL = 'http://api-reporte:3000/report';
7  const API_RECENT_URL = 'http://api-reporte:3000/report/recent';
8  const AUTH = {
9    username: 'admin',
10   password: 'password'
11 };
12
13 let latestReportData = "Aun no hay datos...";
14 let latestRecentMessages = [];
15
16 // Function to fetch data from api-reporte
17 async function fetchData() {
18   try {
19     // Fetch main report (plaintext)
20     const reportResponse = await axios.get(API_REPORT_URL, {
```

Figura 10. Fragmento de código implementación de express

De manera gráfica se logró también la creación de la página HTML que se actualiza automáticamente para mostrar reportes en tiempo real usando los endpoints del servicio api-reporte.



Figura 11. Visual de la página HTML

Se ha creado también un nuevo servicio “logger-central” adicional para centralizar registros del sistema , sin embargo su correcta implementación sigue siendo potencialmente incremental.

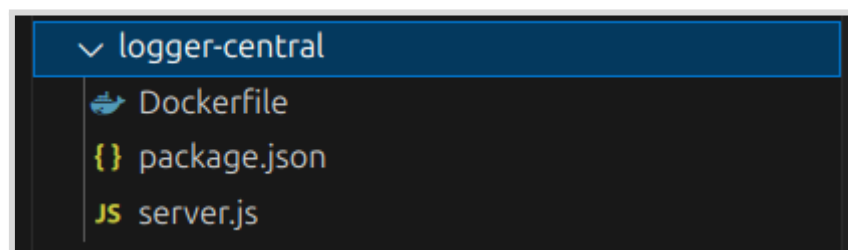
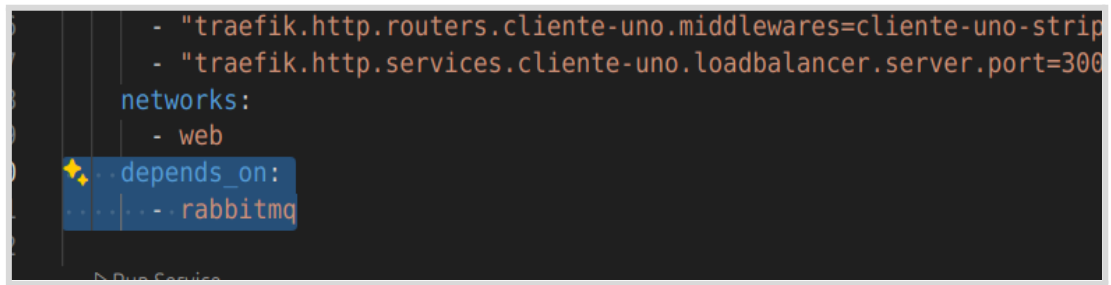


Figura 12. Servicio logger-central

Por último se destaca que se actualizó el docker-compose y se establecieron las respectivas dependencias entre contenedores con `depends_on`.



```
    - "traefik.http.routers.cliente-uno.middlewares=cliente-uno-strip"
    - "traefik.http.services.cliente-uno.loadbalancer.server.port=3000"
  networks:
    - web
  depends_on:
    - rabbitmq
```

Figura 13. implemen de `depends_on` en docker-compose.yml

5. Justificación del Rediseño con RabbitMQ

El rediseño mejora significativamente la arquitectura ya que permite una mayor disposición a ser un sistema distribuido mucho más robusto donde se presentan características como el desacoplamiento ya que se eliminan dependencias directas entre servicios, permitiendo su evolución o reemplazo sin afectar otros módulos; o también el asincronismo que mejora el rendimiento de los productores, que ya no dependen de la disponibilidad del consumidor. Se destaca también la resiliencia ya que RabbitMQ asegura la entrega de mensajes, incluso si el consumidor no está disponible temporalmente y su escalabilidad pueden añadir múltiples consumidores para balancear la carga en el sistema.

6. Lecciones Aprendidas

Hay varios conceptos importantes aprendidos a destacar al momento de usar RabbitMQ que se detallan de la siguiente manera:

- **Visibilidad del flujo:** Aunque agrega componentes, RabbitMQ clarifica el flujo de eventos haciéndolo más fluido .
- **Configuración crítica:** La correcta definición de exchanges, queues y bindings es esencial para el correcto enrutamiento al momento de implementar la herramienta.
- **Manejo de canales y conexiones:** Es Importante manejar reconexiones ante fallos que pueda llegar a presentar el broker.
- **Acuses de recibo (ack/nack):** Se puede hacer uso de claves para asegurar un procesamiento confiable de los mensajes.

- **Herramientas de monitoreo:** La consola de administración de RabbitMQ es valiosa para diagnóstico y seguimiento al momento de usar la herramienta.
- **Complejidad y Beneficio:** RabbitMQ introduce complejidad, pero los beneficios superan los costos en sistemas distribuidos que requieren tolerancia a fallos y escalabilidad.

7. Diagrama de Arquitectura

Se aprecia de acuerdo a la implementación el diagrama de arquitectura correspondiente.

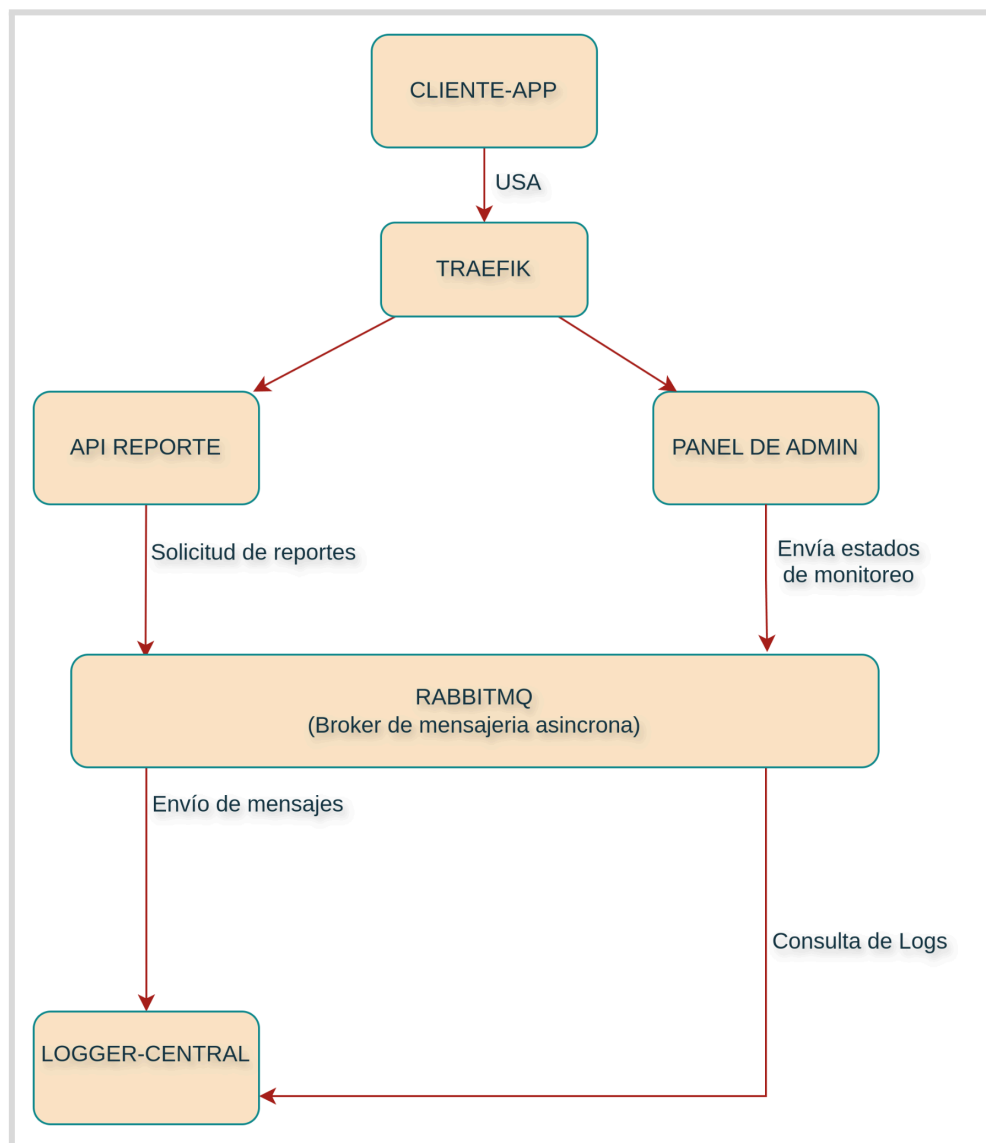


Figura 14. Diagrama de arquitectura ; disponible en :

https://drive.google.com/file/d/1dmUZtSDJzMTG02dHZHwkvdcgN_W9Xcae/view?usp=sharing

8. Conclusiones:

- Adoptar RabbitMQ ha mejorado nuestra arquitectura inicialmente planteada para el ejercicio , lo que nos ha permitido lograr una comunicación entre servicios mucho más clara, flexible y resistente a fallos. Adicionalmente tenemos ahora el reto adicional de configurar y monitorear este sistema de mensajería, donde los beneficios que aporta el uso de RabbitMQ superan con creces ese esfuerzo de implementación. En el contexto de sistemas distribuidos, RabbitMQ se convierte en una herramienta clave para escalar con confianza y orden.
- RabbitMQ no solo mejora la forma en que nuestros servicios interactúan, sino que también nos prepara para enfrentar futuros retos técnicos con una base más robusta y desacoplada. su implementación además marca el inicio de una arquitectura más moderna, mantenible y preparada para crecer en otro ejercicio futuro.