PROPUESTA DE MODELOS DE COMUNICACIÓN DESACOPLADOS PARA SISTEMAS DE INFORMACIÓN CON ARQUITECTURA ORIENTADA A MICROSERVICIOS

ANDRÉS JULIÁN GARCÍA RUEDA
JEFREY STEVEN TORRES GARZÓN

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERÍAS FISICOMECÁNICAS
ESCUELA DE INGENIERÍA DE SISTEMAS E INFORMÁTICA
INGENIERÍA DE SISTEMAS
BUCARAMANGA
2024

PROPUESTA DE MODELOS DE COMUNICACIÓN DESACOPLADOS PARA SISTEMAS DE INFORMACIÓN CON ARQUITECTURA ORIENTADA A MICROSERVICIOS

ANDRÉS JULIÁN GARCÍA RUEDA
JEFREY STEVEN TORRES GARZÓN

Trabajo de grado para optar al título de Ingeniero de Sistemas

Director
Henry Andres Jiménez Herrera
MsC en Ingeniería de Sistemas e Informática

Codirector
Jathinson Meneses Mendoza
Magíster en Gestión, Aplicación y Desarrollo de Software

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERÍAS FISICOMECÁNICAS
ESCUELA DE INGENIERÍA DE SISTEMAS E INFORMÁTICA
INGENIERÍA DE SISTEMAS
BUCARAMANGA
2024

DEDICATORIA

Dedicamos este logro a nuestras familias, cuyo amor, apoyo y comprensión han sido nuestro pilar durante todo este viaje.

Al universo por las oportunidades y los desafíos que nos ha presentado, permitiéndonos crecer y aprender en cada paso del camino.

A nuestro yo del pasado, por su perseverancia, esfuerzo y determinación, que nos han llevado a alcanzar esta meta tan anhelada.

AGRADECIMIENTOS

Expresamos nuestro más sincero agradecimiento a nuestras familias, cuyo apoyo incondicional y comprensión han sido fundamentales a lo largo de este arduo proceso. Extendemos nuestro agradecimiento al director de Proyecto de Grado, Henry Andrés Jiménez Herrera, y al Codirector de Proyecto de Grado, Jathinson Meneses Mendoza, por su invaluable apoyo, orientación y dedicación durante todo el desarrollo del proyecto. Su experiencia y compromiso han sido esenciales para alcanzar el éxito. Además, queremos agradecer también al Tutor y Líder BackEnd de RSI, el Ingeniero de Sistemas Danny Vergel, cuya disposición para brindar asistencia han sido de apoyo para la implementación efectiva de nuestras soluciones. A todos ustedes, les agradecemos profundamente por su contribución y confianza en nuestro trabajo.

CONTENIDO

pág.

LISTA DE FIGURAS

pág.

LISTA DE TABLAS

pág.

LISTA DE ANEXOS

Ver anexos adjuntos

Anexo A. Análisis tecnologías de brokers de mensajería
Anexo B. Diagrama de proceso de flujo de proceso
Anexo C. Documentación previa a la implementación
Anexo D. Diagrama de árbol de configuración
Anexo E. Código implementación plantillas para el broker de mensajería
Anexo F. Explicación del código de la implementación de las plantillas en RSI
Anexo G. Vídeo escenarios de prueba

RESUMEN

TÍTULO: PROPUESTA DE MODELOS DE COMUNICACIÓN DESACOPLADOS PARA SISTEMAS DE INFORMACIÓN CON ARQUITECTURA ORIENTADA A MICROSERVICIOS*

AUTOR: ANDRES JULIAN GARCIA RUEDA, JEFREY STEVEN TORRES GARZÓN**

PALABRAS CLAVE: MICROSERVICIOS, COMUNICACIÓN ASÍNCRONA, BROKERS DE MENSAJERÍA, RABBITMQ, ARQUITECTURA ORIENTADA A SERVICIOS, ESCALABILIDAD, INTEGRACIÓN DE SISTEMAS.

DESCRIPCIÓN:

Este proyecto se enfoca en resolver los desafíos de comunicación que surgen en sistemas desacoplados, basados en arquitecturas orientadas a microservicios (MSA), mediante la implementación de un broker de mensajería adecuado para el proyecto de Renovación de Sistemas de Información (RSI).

Para alcanzar este objetivo, se llevaron a cabo varias etapas clave. En primer lugar, se identificaron las necesidades y requerimientos de comunicación específicos del proyecto. Luego, se realizó un análisis comparativo de las principales tecnologías de brokers de mensajería disponibles en el mercado, seleccionando la opción que mejor se adaptaba. A continuación, se llevó a cabo una investigación sobre las tecnologías elegidas, y se desarrolló una propuesta que abarcó tanto el diseño como la implementación personalizada de la solución para RSI. Adicionalmente, se aplicó en dos microservicios críticos para evaluar su efectividad en condiciones operativas reales. Posteriormente, se realizó una evaluación que incluyó pruebas funcionales en diferentes escenarios, así como pruebas de carga para verificar la escalabilidad y robustez de la comunicación. Además, para complementar todo lo realizado, se documentaron los aspectos del proceso y la tecnología empleada en la wiki de RSI.

En términos generales, del desarrollo de este proyecto se puede concluir que la solución implementada ilustra cómo una elección adecuada de tecnología puede mejorar la comunicación asíncrona en sistemas distribuidos, optimizando no solo la disponibilidad, sino también la escalabilidad, la eficiencia y la confiabilidad del sistema en conjunto.

---

*Trabajo de grado

** Facultad de ingenierías fisicomecánicas. Escuela de ingeniería de sistemas e informática. Ingeniería de sistemas. Director: Henry Andres Jiménez Herrera. MsC en Ingeniería de Sistemas e Informática. Codirector: Jathinson Meneses Mendoza. Magíster en Gestión, Aplicación y Desarrollo de Software.

ABSTRACT

TITLE: PROPOSAL FOR DECOUPLED COMMUNICATION MODELS FOR INFORMATION SYSTEMS WITH ARCHITECTURE ORIENTED TO MICROSERVICES*

AUTHOR: ANDRES JULIAN GARCIA RUEDA, JEFREY STEVEN TORRES GARZON**

KEY WORDS: MICROSERVICES, ASYNCHRONOUS COMMUNICATION, MESSAGE BROKERS, RABBITMQ, SERVICE-ORIENTED ARCHITECTURE, SCALABILITY, SYSTEM INTEGRATION.

DESCRIPTION:

This project focuses on solving the communication challenges that arise in decoupled systems, based on microservice oriented architectures (MSA), by implementing a message broker suitable for the Renewal of Information Systems (RSI) project.

To achieve this objective, several key steps were carried out. First, the specific communication needs and requirements of the project were identified. Then, a comparative analysis of the main messaging broker technologies available in the market was carried out, selecting the most suitable option. Next, research was conducted on the chosen technologies, and a proposal was developed that encompassed both the design and custom implementation of the solution for RSI. Additionally, it was applied in two critical microservices to evaluate its effectiveness in real operating conditions. Subsequently, an evaluation was performed that included functional tests in different scenarios, as well as load tests to verify the scalability and robustness of the communication. In addition, to complement everything that was done, aspects of the process and the technology used were documented in the RSI wiki.

In general terms, from the development of this project it can be concluded that the implemented solution illustrates how an appropriate choice of technology can improve asynchronous communication in distributed systems, optimizing not only availability, but also scalability, efficiency and reliability of the system as a whole.

---

*Degree work

**Faculty of physicomechanical Engineering. School of Systems and computer engineering. Systems engineer. Director: Henry Andres Jiménez Herrera. MsC in Systems Engineering and Computer Science. Codirector: Jathinson Meneses Mendoza. Master's Degree in Management, Application and Software Development.

INTRODUCCIÓN

En la actualidad, cuando abordamos el tema de los sistemas de información, es imprescindible no obviar su base fundamental: la arquitectura. Esta debe ser definida mediante el debido proceso de planeación, teniendo un especial cuidado en los requerimientos del software, la escalabilidad, el mantenimiento, así como los posibles desafíos y necesidades que puedan surgir durante su desarrollo.

En este contexto, la arquitectura orientada a microservicios (MSA) se destaca como una de las más usadas en la actualidad, gracias a su flexibilidad y estructura, no solo mejora, sino que facilita la implementación y el funcionamiento de muchos procesos de negocio. En el mercado actual, existen múltiples opciones tecnológicas, ya sea de pago o de código libre, que ofrecen soluciones a problemas que se pueden encontrar desde la documentación de código hasta la integración o comunicación asincrónica en los servicios. Esto da lugar a ecosistemas diversos, únicos, complejos y adaptados a las necesidades de la organización que los requiere.

El presente proyecto se enfoca en generar una propuesta con base en las diferentes opciones de comunicación desacoplada disponibles en el mercado. Estas opciones, ofrecen una experiencia de uso única y adaptable a diferentes requerimientos. Además, se llevó a cabo la implementación de dicha tecnología junto con la realización de una guía de uso que se dará a conocer por medio de un proceso de difusión en el proyecto de Renovación de Sistemas de Información (RSI).

Este proyecto, gracias a su arquitectura y constante estado de crecimiento, se posiciona como el candidato idóneo para la ejemplificación y enfoque del estudio. La intención es no solo explorar las tecnologías disponibles, sino también demostrar su aplicabilidad en un contexto práctico, contribuyendo así al avance y la eficiencia en el ámbito de los sistemas de información.

1. PLANTEAMIENTO Y JUSTIFICACIÓN

Los sistemas de información tradicionalmente son diseñados con base en arquitecturas monolíticas, estas arquitecturas pueden presentar desafíos notables, estos incluyen la dificultad de acoplamiento a otros aplicativos existentes y la integración de nuevas tecnologías a los mismos, dificultando el desarrollo y el crecimiento de las organizaciones que los emplean. Como respuesta a esta problemática, surge la arquitectura orientada a servicios (SOA) y posteriormente, su derivada más usada la arquitectura orientada a microservicios (MSA) caracterizada por ser más flexible, fácil de mantener y capaz de adaptarse a nuevos requerimientos emergentes. No obstante, esta transición implica abordar retos como la comunicación entre microservicios, la disponibilidad y los tiempos de respuesta ante grandes volúmenes de datos.

Para ejemplificar lo antes dicho, se presenta el caso del proyecto de Renovación de Sistemas de Información (RSI). En el que, debido a la falta de disponibilidad de un servicio al momento de realizar una petición, el aplicativo responderá con un error, provocando retrasos, limitaciones y afectando de manera significativa la experiencia del usuario, especialmente en periodos de alta demanda. Por lo cual, se destaca la importancia de garantizar la disponibilidad y la comunicación efectiva en los servicios presentes, un ejemplo perfecto de la importancia de esta transición y de los retos que conlleva podemos encontrarlo en la propuesta realizada por Arboleda.[1]

Partiendo de esta premisa, este proyecto propone investigar formas de comunicación entre servicios, con un enfoque particular en la mensajería y las tecnologías disponibles del mercado que puedan abordar los retos mencionados con anterioridad.

[1] ARBOLEDA COLA, Carlos Augusto. Propuesta metodológica para migración de sistemas web con arquitectura monolítica hacia una arquitectura basada en microservicios. Trabajo de grado Ingeniero en sistemas informáticos y de computación. Quito: Escuela Politécnica Nacional. Facultad de Ingeniería de sistemas, 2017. p. 1.

Afortunadamente, cuando se trata de este tipo de soluciones, se puede encontrar una gran cantidad de información y estudios al respecto que respaldan el uso de estas tecnologías, como por ejemplo lo desarrollado por Bonhomme y Camejo[2]. Por consiguiente, podemos hablar de los brokers de mensajería, como una solución, destacando algunos de estos nos encontramos con Apache Kafka, RabbitMQ, Mosquitto y ActiveMQ, los cuales presentan diferencias en su arquitectura, funcionamiento, terminología y características. La elección de la tecnología a implementar dependerá de las necesidades del proyecto RSI, teniendo en cuenta factores como la compatibilidad con la arquitectura y la adaptabilidad a corto y largo plazo. Esto permitirá asegurar la escalabilidad y una respuesta satisfactoria a los problemas de comunicación, disponibilidad y tiempos de respuesta de los servicios.

Se espera que la solución propuesta logre asegurar la efectividad en los procesos de negocio en situaciones críticas, como la integridad del flujo de información en un proceso de liquidación o incluso en el envío de correos informativos, mejorando la experiencia de usuario y adicionalmente, generando nuevo conocimiento que puede implementarse en diferentes desarrollos e incluso facilitar el mantenimiento de estos, aportando un valor considerable a todo lo realizado en el proyecto RSI.

[2] BONHOMME, Carlos Augusto y CAMEJO, Enrique. Plataforma de Integración basada en Microservicios. Trabajo de grado Ingeniero en Computación. Montevideo: Universidad de la República. Facultad de Ingeniería, 2019. p. 7.

2. OBJETIVOS

2.1 Objetivo general

Proponer mecanismos de integración para microservicios desacoplados basados en brokers de mensajería con el fin de implementar el más apropiado al marco del proyecto de renovación de sistemas de información (RSI).

2.2 Objetivos especificos

- Identificar las necesidades y requerimientos de comunicación presentes en la arquitectura del proyecto RSI.
- Comparar las características y capacidades de las tecnologías que actualmente lideran el mercado y seleccionar el bróker de mensajería más acorde a las necesidades del proyecto RSI.
- Identificar los microservicios críticos en la arquitectura del proyecto RSI e implementar en estos la comunicación mediante el bróker de mensajería seleccionado.
- Evaluar la implementación mediante la construcción de un marco de evaluación, considerando diversos criterios que permitan medir la efectividad del broker de mensajería utilizado.
- Registrar el desarrollo del proyecto mediante la documentación de la tecnología implementada, junto a una guía que facilite la correcta identificación de escenarios de uso de los brokers de mensajería como mecanismo de comunicación.

3. MARCO DE REFERENCIA

En esta sección, se recopila la información necesaria y se presentan de manera detallada los conceptos fundamentales que resultan esenciales para la correcta ejecución del trabajo presentado. Cada uno de estos conceptos ha sido cuidadosamente seleccionado para dar una explicación de los agentes que incurren en la sustentación de este proyecto garantizando un enfoque fundamentado del desafío abordado.

3.1 Arquitectura de software

La base de esta disciplina se remonta a 1968, cuando Edsger Dijkstra[3] planteó la necesidad de establecer una estructura adecuada en los sistemas de software antes de iniciar el proceso de programación. Con el tiempo, la idea fue tomando forma gracias al trabajo de diversos pensadores e ingenieros. En 1992, Perry y Wolf,[4] en uno de sus estudios, mencionaron y compararon la arquitectura de software con la de un edificio, marcando así el primer acercamiento a lo que conocemos en la actualidad. No obstante, fue gracias al desarrollo tecnológico e intelectual en los años siguientes que evolucionó hasta convertirse en una disciplina bien establecida.

En términos generales, se puede decir que esta disciplina determina la estructura del software, establece cómo deben ensamblarse sus componentes y cómo deben interactuar entre sí [5]. Sin embargo, esta definición superficial no captura completamente

[3] DIJKSTRA, Edsger. The Structure of the The Multiprogramming System. Citado por ESPINAL, Yanet. *Arquitectura de software. Arquitectura orientada a servicios*. 2012, nro. 5, p.1. ISSN-e 2306-2495

[4] PERRY, and WOLF. Worldwide Institute of Software Architects, ESPINAL. Citado por Yanet. *Arquitectura de software. Arquitectura orientada a servicios*. 2012, nro. 5, p.1. ISSN-e 2306-2495

[5] SHAW, Mary. and GARLAN, David. An Introduction to Software Architecture, Citado por VIGIL, Yamila. *La arquitectura de software como disciplina científica*. 2010, nro 3, p.2. ISSN-e 2306-2495

el impacto y la importancia que tiene en el mundo que nos rodea. Por lo tanto, es esencial comprender un poco de la trayectoria de la arquitectura de software y explorar algunos de los esquemas o arquitecturas que fueron surgiendo con ella.

3.1.1 Arquitectura monolítica. Este tipo de arquitectura se encuentra entre las primeras propuestas en la historia, surgiendo como respuesta al flujo de ideas derivadas del planteamiento realizado por Perry y Wolf,[6] quienes compararon la arquitectura de software con la de un edificio. Varios ingenieros se inspiraron en estas ideas para concebir estructuras de software que ofrecieran control y previsión desde el inicio de un desarrollo. En este contexto, surge lo que conocemos como arquitectura monolítica, un término que adquiere más significado al considerar lo mencionado anteriormente. Adicionalmente, el término "mono", epistemológicamente, hace referencia a uno, y podemos vincular esto con la idea presentada por Newman,[7] según él, cuando nos referimos a esta arquitectura, hablamos de un software o unidad de único despliegue, con un repositorio único o, en otras palabras, un programa o sistema en el cual todas sus funciones fueron implementadas conjuntamente.

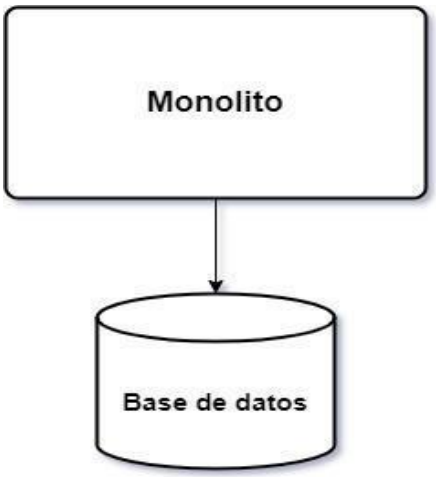
Sin embargo, esta definición resulta demasiado general y permitiría que casi cualquier cosa fuese considerada arquitectura monolítica. Por esta razón, se llegaron a definir algunas variaciones de este tipo de arquitectura, que fueron dando forma poco a poco a lo que conocemos hoy en día, como lo son el monolito modular, el monolito distribuido o el monolito de caja negra o de terceros, no obstante, el precursor de estos y más conocido es el monolito de único proceso, sobre el cual indagaremos más a continuación.

[6] PERRY, and WOLF. Worldwide Institute of Software Architects, Citado por ESPINAL, Yanet. *Arquitectura de software. Arquitectura orientada a servicios*. 2012, nro. 5, p.1. ISSN-e 2306-2495

[7] NEWMAN, Sam. Monolith to Microservices. Evolutionary Patterns to Transform Your Monolith. 2 ed. United States of America: O'Reilly, 2019. p. 12. ISBN 9781492047841

3.1.2 El monolito de proceso único. Siguiendo lo planteado por Newman,[8] este sería el tipo de monolito más común y, a menudo, el que genera más desafíos para las organizaciones y sus ingenieros informáticos. Esta estructura presenta dificultades tanto en escalabilidad como en mantenimiento. Este tipo de monolito es aquel en el que todo su código se encuentra empaquetado e implementado como un solo proceso, de ahí su nombre. Su estructura podría visualizarse de manera similar a como se presenta en la figura 1.

Figura 1. Representación de monolito de único proceso



3.1.2.1 Ventajas o beneficios de los monolitos. Si bien es cierto que esta es una de las arquitecturas más antiguas y que ha experimentado algunos cambios, conserva sus ventajas frente a otras arquitecturas. Algunas de ellas son las siguientes:

- **Autonomía del proceso único:** Al tratarse de un único proceso en el cual se implementan todas sus funcionalidades, suele tener autonomía total o parcial, dependiendo del tipo de monolito del que estemos hablando. Por lo tanto, se evitan gran parte de los problemas de comunicación o dependencias que se pueden presentar en otros tipos de arquitecturas.

[8] Ibid., p. 12

- **Facilita el trabajo para equipos pequeños:** Al encontrarse todo implementado en un único lugar, puede facilitar el flujo y tiempo de trabajo a equipos de desarrollo pequeños, dado que no tienen que preocuparse por cosas externas o fuera de las funcionalidades del monolito. Claramente, esto depende del tipo de monolito del que se esté hablando.
- **Estabilidad de plataforma y tecnologías:** Todo el sistema suele correr sobre una misma plataforma, y las tecnologías suelen estar bien establecidas desde el principio del desarrollo. Además, no suelen cambiar en el proceso, proporcionando estabilidad y consistencia en el entorno de desarrollo.

3.1.2.1 Desventajas o dificultades de los monolitos. Con el tiempo, las organizaciones han venido creciendo y, por consiguiente, evolucionando a un ritmo acelerado, lo que ha hecho que sus procesos de negocio sean cada vez más grandes y complejos. Este crecimiento ha resaltado las deficiencias y desventajas de los sistemas monolíticos. Entre estas encontramos las siguientes:

- **Flujo de trabajo y tiempo de implementación:** Al tratarse de grupos de desarrollo grandes, el flujo de trabajo y el tiempo de implementación tienden a aumentar. Esto se debe a que, al ser un único bloque donde se encuentra todo el código, pueden surgir múltiples problemas de versiones, ya que todos están modificando el mismo código.
- **Escalamiento de aplicaciones demasiado grandes:** Cuando se trata de escalar una aplicación demasiado grande, se hace necesario escalar todo el código, lo cual representa un mayor gasto de tiempo y recursos para la organización, ya que implica una modificación general en todos los aspectos.
- **Acoplamiento de nuevas tecnologías o funciones:** Introducir nuevas tecnologías o funciones puede ser complicado, ya que pueden ser incompatibles o de difícil integración. Esto puede resultar en algunas de las variaciones de monolitos antes mencionadas.

- **Mantenimiento complejo:** El mantenimiento puede volverse complejo debido a la fuerte interconexión de todo en el monolito. Esto lleva a la necesidad de modificar grandes cantidades de código incluso si el cambio inicial era pequeño.
- **Fallos generalizados:** En los monolitos, cuando algo falla, por más pequeño que sea, puede hacer que todo falle simultáneamente, lo cual puede tener un impacto significativo en el rendimiento y la estabilidad del sistema.

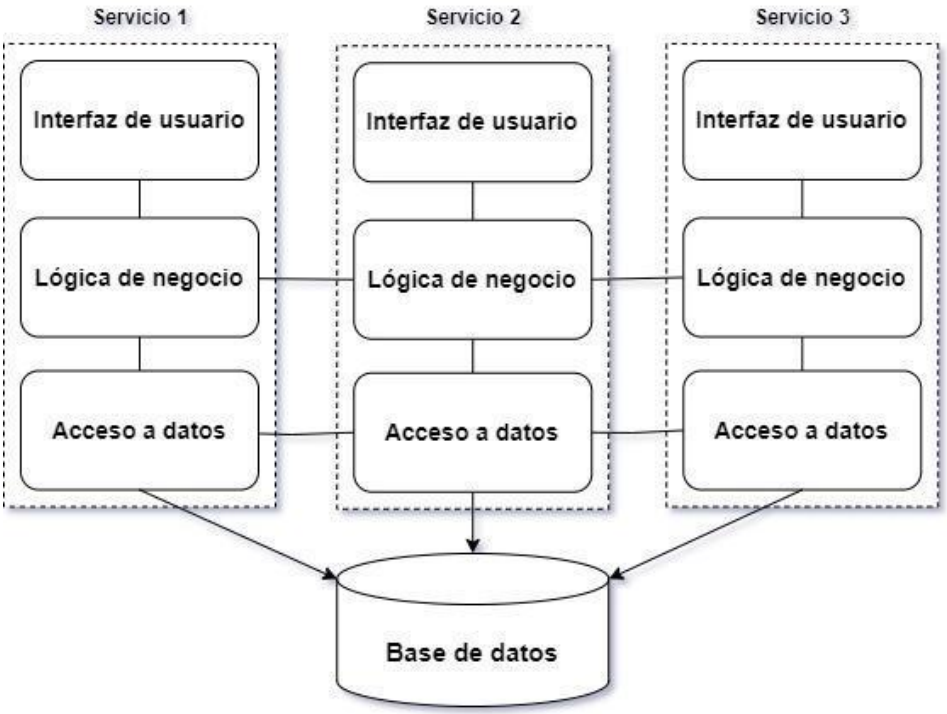
3.1.3 Arquitectura de servicios (SOA). Como se ha destacado, la arquitectura monolítica ha experimentado una disminución en su capacidad para satisfacer los crecientes y cambiantes requisitos de los sistemas modernos. En respuesta a esta necesidad de flexibilidad y adaptabilidad se ideó la arquitectura basada en servicios.

Pero ¿en qué consiste esta arquitectura? Según Erl,[9] sería aquella que se construye a partir de un conjunto de componentes y configuraciones. Por ejemplo, a diferencia de la arquitectura monolítica, que unifica todas las funciones en un mismo proceso, la arquitectura basada en servicios hace uso de diferentes procesos para llevar a cabo cada función. Además, a pesar de trabajar en conjunto, cada componente es casi independiente y reutilizable. Esto no solo simplifica ciertos aspectos del desarrollo del sistema, sino que de esta forma proporciona la flexibilidad necesaria para adaptarse a los requisitos cambiantes.

Esta interpretación se puede apreciar de manera más precisa al observar la figura adjunta figura 2, donde se define como una interfaz de usuario, una lógica de negocio y un acceso a datos para cada uno de los componentes, o, mejor dicho, servicios, sin obviar que hacen uso de un modelo de datos compartido.

[9] ERL, Thomas. Service-Oriented Architecture: Concepts, Technology and Design. United States of America: Prentice Hall, 2005. p. 32. ISBN: 0131858580

Figura 2. Representación de la arquitectura de servicios (SOA)

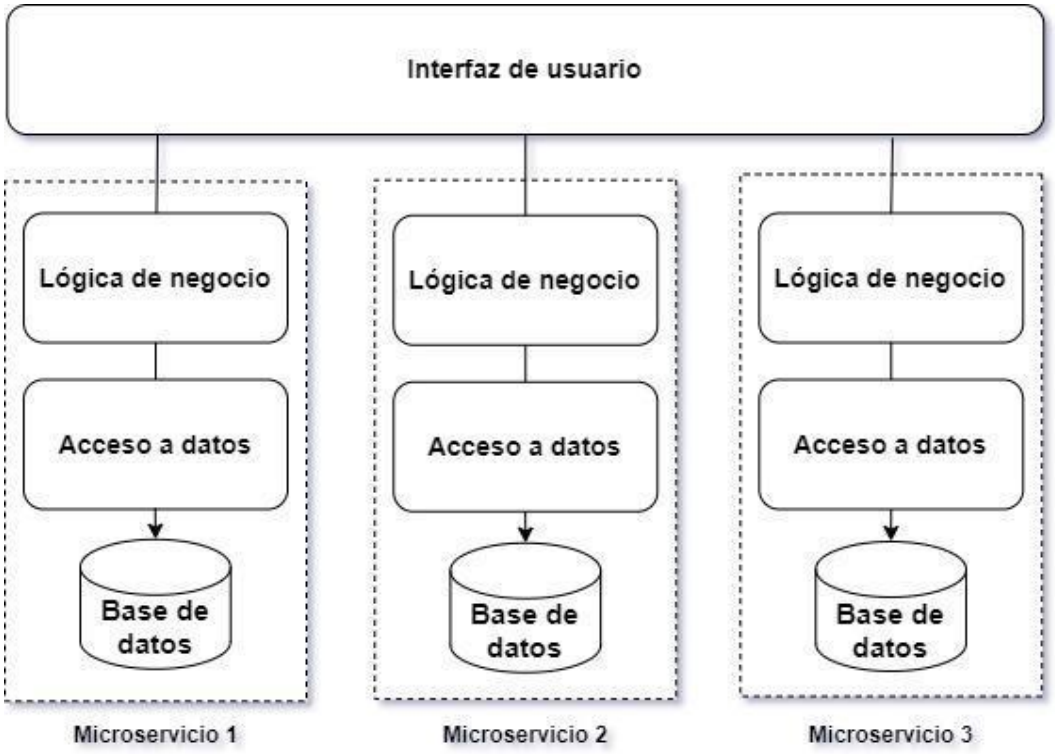


3.1.4 Arquitectura de microservicios (MSA). Cuando se habla de arquitectura de microservicios, es común que se confunda con la de servicios o se piense erróneamente que son idénticas. A pesar de compartir algunas similitudes, los microservicios podrían considerarse un derivado que eleva la idea de servicios a un nivel superior. Según De la Torre,[10] la propuesta de los microservicios se centra en desarrollar sistemas como un conjunto de servicios más pequeños. Entonces, ¿cuál es la diferencia fundamental? Si retomamos la figura adjunta, figura 2, se pueden observar las interconexiones entre los componentes o servicios que pueden darse según se requiera en cualquiera de los niveles. Por otro lado, los microservicios toman los beneficios preexistentes y agregan total autonomía y desacople, como se muestra en la figura 3, sólo utilizan una interfaz de

[10] MICROSOFT DEVELOPER DIVISION. [.NET Microservices: Architecture for Containerized .NET Applications] United States of America: Microsoft Corporation. [Consulta: 22 de diciembre 2023]. Disponible en: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/

usuario en común y el resto de los niveles o capas no cuentan con interconexiones, e incluso cada uno maneja un modelo de datos diferente. Esto permite modularizar y lograr que todo siga trabajando en conjunto para el sistema.

Figura 3. Representación de la arquitectura de microservicios (MSA)



3.1.4.1 Ventajas o beneficios de las arquitecturas de microservicios: Al explorar las ventajas y beneficios de las arquitecturas de servicios y microservicios, es fundamental comprender lo que las hace tan significativas en comparación con las arquitecturas monolíticas. Lo siguiente es una interpretación basada en las ideas propuestas por Vilajosa y Navarro: [11]

[11] VILAJOSANA, Xavier and NAVARRO, Leandro. Arquitectura de aplicaciones web. Barcelona, España: Universitat Oberta de Catalunya, 2001. p. 36-37. PID_00184783

- **Distribución:** La autonomía y facilidad de desacople permiten que el conjunto de servicios o microservicios esté distribuido en diferentes ubicaciones e incluso con cada parte de ellos repartida según sea necesario.
- **Diversidad:** Las tecnologías empleadas en los servicios o microservicios son independientes, brindando flexibilidad para operar en sistemas operativos distintos, equipos diversos e incluso en diferentes lenguajes de programación.
- **Interoperabilidad:** Dada su amplia adopción, se han desarrollado estándares como los de OASIS o WSDL, facilitando la operación conjunta de servicios o microservicios creados por diferentes desarrolladores o compañías.
- **Desacoplado:** Los servicios o microservicios pueden replicarse o distribuirse en diversas máquinas, simplificando su uso remoto, mantenimiento y funcionamiento continuo.
- **Flexibilidad:** El uso de interfaces estándar facilita el desarrollo y la incorporación de servicios o microservicios existentes, ya sea desarrollados internamente o por terceros, permitiendo la adaptación de diversas tecnologías según los requerimientos.
- **Escalabilidad y mantenimiento:** La división en diferentes servicios o microservicios agiliza los flujos de trabajo, la documentación y el desarrollo, ya sea para nuevas implementaciones o para mantener las existentes. Mientras que la distribución en diversos equipos mejora la disponibilidad y la tolerancia a volúmenes elevados de usuarios.
- **Disminución de fallos:** La división de funcionalidades permite que, en caso de una caída, la arquitectura siga funcionando. Además, mediante diferentes métodos, es posible prevenir y mitigar los impactos de fallos, salvaguardando así los procesos de negocio.

3.1.4.1 Desventajas o dificultades de las arquitecturas de microservicios: Se ha explorado la parte positiva o beneficiosa de lo que ofrece este tipo de arquitectura. Sin embargo, todos esos beneficios vienen acompañados de un precio, y con ellos surgen nuevas dificultades que deben abordarse si se busca no solo que funcione, sino que lo

haga de la mejor manera posible. Lo siguiente son algunas interpretaciones de lo proponen como dificultades los autores Söylemez, Tekinerdogan y Kolukısa: [12]

- **Administración de datos y consistencia:** Este problema suele manifestarse con mayor frecuencia en los microservicios, ya que cada uno de ellos es independiente, incluyendo sus datos. Podría decirse que estamos hablando de una base de datos distribuida. Por lo tanto, se vuelve más complejo mantener consistentes y sincronizados los datos de cada uno de los microservicios. Sin embargo, este problema también puede presentarse en los servicios, ya que, a pesar de utilizar una base de datos centralizada, si no se diseña de manera correcta, podría generar inconsistencias en los datos manejados.
- **Pruebas del sistema:** Al estar todo distribuido se evita en principio la necesidad de cambiar o reformar mucho código y funcionalidades, no obstante, agrega más trabajo al realizar pruebas de funcionamiento. Dado que estos servicios o microservicios pueden estar en diferentes tecnologías e incluso no estar totalmente desacoplados, por consiguiente, si se realizan nuevas implementaciones o cambios es necesario llevar a cabo pruebas completas para comprobar que todo funcione como debería.
- **Medición y predicciones:** Al diseñar un software y plantear una arquitectura, es preferible realizar mediciones o, mejor dicho, predicciones del funcionamiento de todo en conjunto antes de implementar, con el objetivo de prevenir errores y gastos mayores en caso de mal rendimiento. Sin embargo, al tratarse de servicios y microservicios, este trabajo se vuelve más complicado debido a la presencia de muchos más factores a tener en consideración y la variabilidad que puede llegar a ser considerable al tener todo en cuenta.

[12]SÖYLEMEZ, Mehmwt; TEKINERDOGAN, Bedir and KOLUKISA, Tarhan. Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review. *applied sciences* [en línea]. 2022, mayo, 12(11). p. 12-15.

- **Comunicación e integración:** Al diseñar un software y plantear una arquitectura, es preferible realizar mediciones o, mejor dicho, predicciones del funcionamiento de todo en conjunto antes de implementar, con el objetivo de prevenir errores y gastos mayores en caso de mal rendimiento. Sin embargo, al tratarse de servicios y microservicios, este trabajo se vuelve más complicado debido a la presencia de muchos más factores a tener en consideración y la variabilidad que puede llegar a ser considerable al tener todo en cuenta.
- **Seguridad:** En estas arquitecturas distribuidas, cada uno de sus servicios, componentes o infraestructura de comunicación puede representar una potencial vulnerabilidad externa ante amenazas de hackers, así como interna cuando se trata de personas poco confiables, siendo una situación agravante la complejidad que representa administrar y gestionar los controles de acceso. Por consiguiente, los esfuerzos y gastos en seguridad se incrementan.
- **Descomposición:** Al momento de diseñar el sistema, es crucial tener en cuenta que todos los servicios o microservicios deben ser lo más desacoplados y cohesivos posibles. No obstante, cuando se considera la lógica de negocio y la capacidad empresarial, se convierte en un desafío determinar el tamaño y la capacidad del sistema, por consiguiente, si no se lograse el equilibrio adecuado podrían surgir problemas significativos en escalabilidad, rendimiento, confiabilidad y disponibilidad. Por ende, no es suficiente simplemente adoptar este tipo de arquitectura; es necesario implementarla de manera efectiva, asegurando una descomposición adecuada y conexiones bien establecidas que garanticen el éxito a largo plazo del sistema.
- **Orquestación:** Este desafío es crucial en estas arquitecturas, ya que el término hace referencia a la coordinación y gestión de los servicios o microservicios, junto con la automatización de los procesos relacionados, como el despliegue y su funcionamiento completo todo el tiempo. Es decir, es una tarea esencial que conlleva gasto de recursos y mucho trabajo. Por lo tanto, si no se lleva a cabo de manera adecuada, puede presentar problemas en la disponibilidad y el funcionamiento correcto de los servicios, incluso si estos se encuentran

operativos. Es fundamental abordar la orquestación con precisión y eficiencia para garantizar un rendimiento óptimo del sistema.

3.2 Comunicación entre Microservicios

Tras explorar la arquitectura de servicios (SOA) y su derivado, la arquitectura de microservicios (MSA), nos enfocaremos en esta última, ya que, como se mencionó anteriormente, es la más competitiva en la actualidad e integra gran parte de la SOA. Continuaremos abordando la comunicación entre los microservicios, que representa la principal dificultad al implementar y utilizar esta arquitectura. No obstante, en la actualidad, se pueden encontrar múltiples soluciones cuando se comprende cómo funciona la comunicación entre estos. Según un artículo de Microsoft,[13] existen diferentes tipos según el caso y el propósito. Sin embargo, estos se pueden clasificar en dos categorías: la primera considera si la comunicación es síncrona o asíncrona, y la segunda analiza si se dirige a un solo receptor o a varios receptores.

3.2.1 Comunicación según sincronicidad.

- **Síncrona**: Según Viveiros, citado de la revista Espacios,[14] cuando nos referimos a la comunicación síncrona, estamos hablando de la interacción en tiempo real entre los participantes. Esto implica la emisión de un mensaje y la obtención de

[13] MICROSOFT DEVELOPER DIVISION. [Comunicación en una arquitectura de microservicio] United States of America: Microsoft Corporation. [Consulta: 22 de diciembre 2023]. Disponible en: https://learn.microsoft.com/es-es/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture

[14]VIVEIROS. Desarrollo de tecnologías y tecnologías para el desarrollo, Citado por LAY, Nelson, *et al*. Uso de las herramientas de comunicación asincrónicas y sincrónicas en la banca privada del municipio Maracaibo. En: *Revista ESPACIOS*. 2019, nro 4, vol 40, p.4. ISSN N 0798 1015

una respuesta de manera inmediata, ósea en tiempo real. Al aplicar esto a los microservicios, se podría describir como el cliente emitiendo una solicitud y el microservicio respondiendo de manera inmediata.

- **Asíncrona:** Según Araujo, citado de la revista Espacios,[15] cuando hablamos de comunicación asíncrona, nos referimos a un proceso que no sigue una correspondencia temporal precisa. Esto implica que los participantes interactúan, pero no necesariamente en tiempo real. En el contexto de los microservicios, podría ilustrarse como una suscripción a un tercero o un bróker de mensajería que almacena la solicitud del cliente para que el receptor la procese cuando esté disponible y devuelva una respuesta en un momento posterior.

3.2.2 Comunicación según cantidad de receptores.

- **Único receptor:** En este tipo de comunicación, se envía una solicitud a un único receptor. Por ejemplo, cuando desde una aplicación móvil se realiza una transacción bancaria a otro individuo. Relacionándolo con los microservicios, podríamos decir que el cliente emite una solicitud específica dirigida y procesada por un solo microservicio.
- **Varios receptores:** En este tipo de comunicación, se envía una solicitud a uno o varios receptores. Por ejemplo, cuando desde Gmail se envía un correo electrónico y se tiene la opción de enviarlo a múltiples usuarios. En el contexto de los microservicios, esto implica una solicitud enviada para ser procesada por varios microservicios.

[15]ARAUJO. Estrategias didácticas para la enseñanza para la enseñanza en entornos Virtuales, Citado por LAY, Nelson, *et al*. Uso de las herramientas de comunicación asincrónicas y sincrónicas en la banca privada del municipio Maracaibo. En: *Revista ESPACIOS*. 2019, nro 4, vol 40, p.3. ISSN N 0798 1015

3.2.2 Protocolos de comunicación.

Los diferentes patrones, arquitecturas o modelos de comunicación utilizados para conectar e integrar microservicios suelen estar basados en algún tipo de protocolo, los cuales a su vez se encuentran contenidos o relacionados con las dos categorías antes mencionadas

- **HTTP (Hyper Text Transport Protocol):** Este es el protocolo de mensajería web más influyente en la actualidad, perteneciendo al tipo de comunicación síncrona, específicamente en la categoría de petición/respuesta. Además, utiliza el protocolo TCP en la capa de transporte y TLS/SSL para garantizar la seguridad.[16]
- **CoAP (Constrained Application Protocol):** Según lo presentado por Al-Masri, Kalyanam, Batts, Kim, Singh, Vo, & Yan,[17] se trata de un protocolo de transferencia web diseñado específicamente para aplicaciones M2M, con un enfoque primordial en redes restringidas. Similar al protocolo HTTP, este sigue la estructura de petición/respuesta, es síncrono y utiliza conceptos familiares como identificadores de URL. El protocolo opera a través de un sistema donde un publicador comparte información mediante URLs específicas, permitiendo que un suscriptor se suscriba a un recurso deseado mediante esta URL. En consecuencia, si el publicador emite nueva información a una URL a la que el suscriptor está suscrito, este último recibirá una notificación.
- **MQTT (Message Queuing Telemetry Transport Protocol):** Según la descripción de Naik, [18] estamos ante un protocolo asíncrono de mensajería del tipo

[16] AL-MASRI, Eyhab, *et al*. Investigating Messaging Protocols for the Internet of Things (IoT). *IEEE Access* [en línea]. 2020 junio, 8(2020). p. 94887.

[17] Ibid., p. 94887.

[18] NAIK, Nitin. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. *IEEE International Systems Engineering Symposium (ISSE)* [en línea]. 2017 octubre. p. 2.

publicador/suscriptor, también diseñado para M2M, pero con un enfoque específico en comunicaciones livianas en redes restringidas. En este protocolo, el publicador comparte datos en un tema específico, los cuales son enviados a un bróker de mensajería encargado de distribuirlos a todos los suscriptores activos de dicho tema. [19]

- **AMQP (Advanced Message Queuing Protocol):** Este protocolo, al igual que el mencionado anteriormente, está diseñado para facilitar comunicaciones M2M de bajo peso. Este protocolo, comúnmente empleado en entornos corporativos, destaca por su capacidad para respaldar una amplia variedad de aplicaciones de mensajería. Se clasifica como asíncrono y adopta el modelo de comunicación publicador/suscriptor.

3.3 Brokers de mensajería

Para concluir el marco de comunicación entre microservicios, se toma como enfoque central la arquitectura proporcionada por brokers de mensajería, los cuales tiene como principal función mediar entre las solicitudes que pueden ser recibidas por una diversidad de microservicios. Tomando la definición formulada por IBM[20], un broker de mensajería puede describirse como un tipo de middleware que facilita el almacenamiento de datos para su transmisión entre los componentes de una aplicación, permitiendo el intercambio

[19] INTERNATIONAL CONFERENCE ON TELECOMMUNICATION, POWER ANALYSIS AND COMPUTING TECHNIQUES. (173, 2017 abril, 8: Chennai, India). A survey on MQTT: A protocolo of Internet of Things (IoT). Bharath Institute of Higher Education and Research, 2017, p. 3.

[20] INTERNATIONAL BUSINESS MACHINES CORPORATION. [What is a message broker?] United States of America: IBM. [Consulta: 20 de diciembre 2023]. Disponible en: https://www.ibm.com/topics/message-brokers

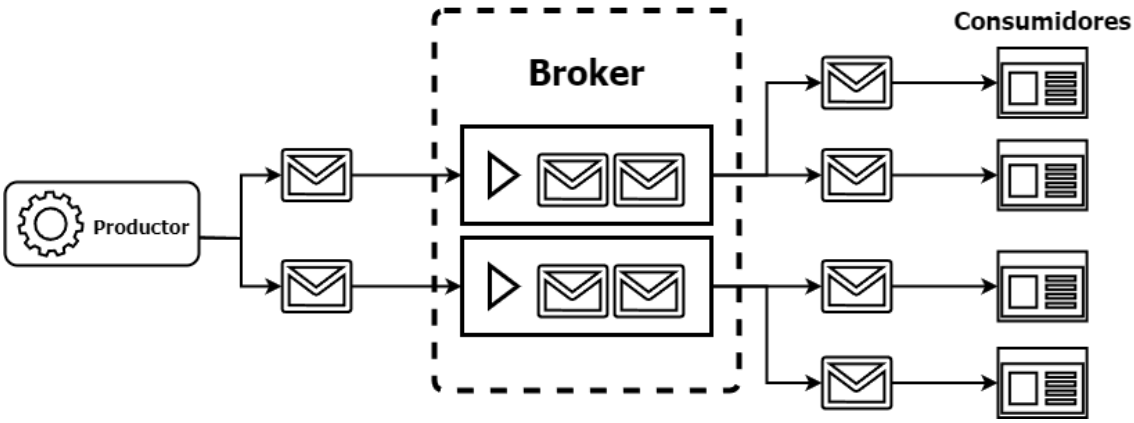
de información en forma estandarizada y confiable entre los componentes del sistema, los cuales reciben el nombre de “mensajes”.

Este papel intermedio es crucial para permitir la gestión asíncrona de las peticiones, otorgando a cada componente la libertad de operar de manera independiente, sin estar atado directamente a las respuestas de estos. Esta independencia funcional no solo mejora la eficiencia operativa, sino que también contribuye a la robustez y flexibilidad del sistema en su conjunto. De igual manera, otro aspecto destacado de los broker de mensajería es su capacidad para manejar patrones de comunicación complejos, como la mensajería uno a uno, uno a muchos y muchos a muchos, lo cual permite que los microservicios se comuniquen de manera eficiente incluso en entornos altamente distribuidos y heterogéneos.

3.3.1 Arquitectura del broker

Al explorar más a fondo la anatomía de los sistemas que integran broker de mensajería, se revela una estructura compuesta por tres componentes esenciales que en conjunto se encargan de realizar un proceso de intercomunicación de las peticiones del sistema, estos están representados en la figura 4.

Figura 4. Representación de la arquitectura de un bróker de mensajería



De acuerdo con VMware [21] la arquitectura de los sistemas basados en brokers de mensajería se constituye principalmente de tres componentes que en conjunto proporcionan los elementos necesarios para ejercer una transmisión de mensajes:

- Productor: Esta interfaz se encarga de iniciar el flujo de mensajes en el sistema al generar peticiones que contienen la información pertinente para los procesos de los que hará uso el consumidor. Desarrolla un papel crucial en la generación y transmisión eficiente de peticiones al intermediario o broker a través de mensajes.
- Intermediario (Broker de mensajería): Actuando como intermediario entre el productor y el consumidor, el broker procesa las peticiones. Posteriormente, mediante el uso de una entidad lógica se encarga de distribuir la información a manera de colas en el sistema, adaptándose a la arquitectura específica de cada tipo de broker y garantizando una gestión organizada y eficaz. Internamente, el broker presenta una estructura que varía según la implementación realizada, brindando así un manejo personalizado a las necesidades del sistema en el que está implementado.
- Consumidor: También conocido como consumidores, esta entidad concluye el proceso y se encarga de recibir el mensaje almacenado en el broker correspondiente e iniciar el flujo del proceso de la información de la petición solicitada, cerrando ciclo de comunicación.

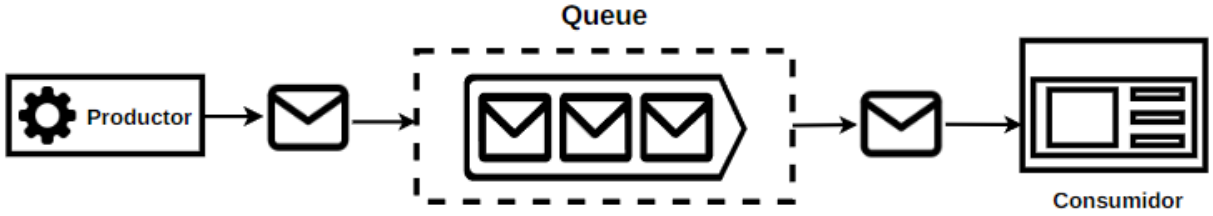
[21] VMWARE INC. [What is a message broker?] United States of America: VMware. [Consulta: 20 de diciembre 2023]. Disponible en: https://www.vmware.com/topics/glossary/content/message-brokers.html

3.3.2 Patrones de mensajería.

En el ámbito de la mensajería, los brokers ofrecen dos patrones fundamentales de distribución de mensajes o estilos de mensajería, cada uno diseñado para satisfacer necesidades específicas.

- **Mensajería punto a punto:** Este patrón de distribución se distingue por la utilización de colas de mensajes con una relación de uno a uno entre el productor y el consumidor. En este contexto, cada mensaje de la cola se envía a un consumidor único y se consume una sola vez. Este patrón se implementa de manera específica cuando es necesario que un proceso se ejecute de manera individual sin depender de iteraciones. Es particularmente útil cuando la ejecución de una tarea debe ser única y no puede depender de la intervención o procesamiento adicional por parte de otros consumidores.

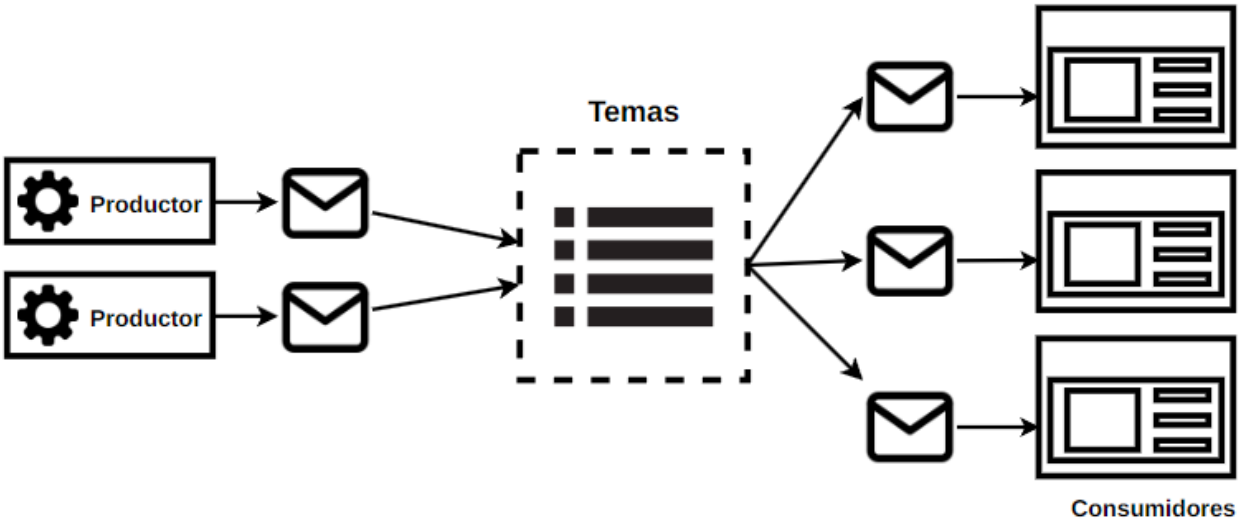
Figura 5. Representación de mensajería punto a punto



- **Mensajería publicación/suscripción:** Este patrón, ampliamente conocido como "pub/sus", establece una relación uno a muchos entre el productor y los consumidores. En este contexto, el productor emite mensajes en un tema determinado, mientras que los consumidores interesados en recibir dicha información se suscriben a dicho tema, donde todos los mensajes publicados sobre un tema se difunden a todas las aplicaciones suscritas. Este patrón demuestra ser particularmente efectivo en situaciones donde la información compartida tiene diversas aplicaciones y utilidades, permitiendo que múltiples

consumidores se beneficien simultáneamente de los mensajes emitidos en un tema específico. Además de su eficacia en la distribución de datos, la mensajería de publicación/suscripción se destaca por su capacidad para facilitar la comunicación fluida y oportuna entre los diferentes componentes de un sistema, mejorando así la coordinación y la eficiencia en diversas aplicaciones y entornos operativos.[22]

Figura 6. Representación de mensajería publicación/suscripción



Estos dos patrones de mensajería ofrecidos por los broker proporcionan flexibilidad y adaptabilidad a las diferentes necesidades y escenarios de implementación. Ya sea para ejecutar procesos de forma individual o para difundir información a una audiencia diversa, la elección del patrón adecuado juega un papel crucial en el diseño eficiente y la operación efectiva de sistemas basados en mensajería.

[22] INTERNATIONAL BUSINESS MACHINES CORPORATION. [What is a message broker?] United States of America: IBM. [Consulta: 20 de diciembre 2023]. Disponible en: https://www.ibm.com/topics/message-brokers

3.3.3 Conceptos de funcionamiento interno en los brokers

Tras el análisis realizado a la arquitectura base del sistema, pueden ser identificadas ciertas características internas fundamentales que implementan de los broker para su adecuado funcionamiento, con base a estas, podemos describir el comportamiento general que dirigen el proceso de información de estos intermediarios.

3.3.3.1 Protocolo de comunicación. En primer lugar, es esencial resaltar que los brokers de mensajería implementan un protocolo de comunicación que regula la interacción entre los clientes y el broker. Este protocolo especifica el formato de los mensajes, los comandos para enviar y recibir mensajes, así como los procedimientos para establecer y cerrar conexiones de red. La selección del protocolo varía dependiendo del broker instalado en el sistema de información. Ejemplos destacados de protocolos incluyen AMQP (Advanced Message Queuing Protocol), empleado por RabbitMQ, y MQTT (Message Queuing Telemetry Transport), utilizado por Mosquitto.

3.3.3.2 Gestión de enrutamiento de mensajes. Los brokers mantienen una estructura de datos la cual puede variar en su manejo interno según el tipo de especificaciones que tenga el broker, estas se encargan del manejo de suscripciones asociados al sistema. Esto puede implicar el uso de estructuras como árboles de tópicos para organizar los tópicos jerárquicamente o tablas de hash para almacenar las suscripciones de los clientes de manera eficiente.

Cuando un mensaje llega al broker, se determina a qué clientes suscritos debe ser entregado. Esto puede implicar la búsqueda de suscripciones relevantes en estructuras de datos como las ya mencionadas tablas de enrutamiento o árboles de suscripciones, y la distribución de copias del mensaje a través de canales de comunicación sujetos al sistema, según el tipo de arquitectura interna del broker el enrutamiento puede implicar el envío de mensajes a múltiples clientes simultáneamente, casos como este pueden presentarse al hacer uso de modelos de publicación-suscripción.

Una vez que se determina el destino de un mensaje, el broker lo entrega a los clientes suscritos utilizando el protocolo de comunicación adecuado. Esto puede implicar el enrutamiento de mensajes a través de redes de área local, con técnicas como encolado de mensajes en colas de salida y gestión de flujo de datos para garantizar la entrega confiable y eficiente.

3.3.3.3 Almacenamiento de mensajes. Algunos brokers pueden optar por almacenar temporalmente los mensajes entrantes antes de entregarlos a los clientes. Esto puede implicar el uso de caches en memoria para mensajes recientes y almacenamiento en disco para mensajes más antiguos, con políticas de expiración para eliminar mensajes después de un cierto período de tiempo o tamaño de almacenamiento con el fin de garantizar su durabilidad y disponibilidad incluso en caso de fallo del sistema.

3.3.3.4 Gestión de la calidad de servicio (QoS). Los brokers de mensajería admiten diferentes niveles de QoS para garantizar la fiabilidad de la entrega de mensajes. Esto implica implementar técnicas como confirmaciones de acuse de recibo, almacenamiento en búfer de mensajes salientes, y reintento de entrega en caso de fallo de la conexión o la entrega inicial. Los clientes pueden negociar el nivel de QoS deseado al establecer una conexión con el broker, lo que determina el nivel de garantía de entrega ofrecido por el broker para los mensajes enviados y recibidos.

3.3.4 Ventajas y beneficios de los broker de mensajería

La implementación de un broker de mensajería en una aplicación conlleva una serie de beneficios significativos, estos pueden variar ligeramente según el patrón que este esté implementando.

3.3.4.1 Beneficios de los brokers punto a punto: Según AWS, [23] esta arquitectura implementada de brokers que hacen uso de colas puede permitirnos mejorar características tales como:

- **Mejora en el rendimiento:** Las comunicaciones asíncronas se vuelven posibles mediante el uso de puntos de conexión que producen y consumen mensajes de manera interactiva con la cola, en lugar de interactuar directamente entre sí. En este escenario, los productores tienen la capacidad de agregar solicitudes a la cola sin necesidad de esperar su procesamiento, permitiendo un flujo de datos optimizado. Por otro lado, los consumidores procesan los mensajes únicamente cuando estos se encuentran disponibles. Esta característica evita que cualquier componente del sistema se detenga a la espera de otro, promoviendo una notable eficiencia en el rendimiento.
- **Mayor fiabilidad:** Estos sistemas garantizan la persistencia de los datos, reduciendo los errores que suelen originarse cuando diferentes partes del sistema se desconectan. La separación de componentes mediante colas de mensajes aumenta el nivel de tolerancia a fallos. En situaciones donde una parte del sistema no sea accesible, la otra puede seguir interactuando con la cola de manera ininterrumpida. Es importante destacar que la cola en sí misma puede ser replicada para mejorar aún más la disponibilidad.
- **Escalabilidad granular:** La capacidad de escalar de manera precisa donde sea necesario es una de las ventajas clave de estos sistemas. Cuando las cargas de trabajo alcanzan su punto máximo, múltiples instancias de la aplicación pueden agregar solicitudes a la cola sin riesgo de colisiones. A medida que las colas se extienden con las solicitudes entrantes, la distribución de la carga de trabajo entre

[23] AMAZON WEB SERVICES. [Benefits of Message Queues] United States of America: AWS. [Consulta: 20 de diciembre 2023]. Disponible en: https://aws.amazon.com/message-queue/benefits/

una flota de consumidores se vuelve posible. La capacidad de crecimiento o reducción de productores, consumidores y la propia cola en respuesta a la demanda permite una adaptabilidad excepcional.

- **Desacoplamiento simplificado:** Estos sistemas eliminan las dependencias entre los componentes y simplifican de manera significativa la escritura de código en aplicaciones desacopladas. Los componentes de software no se ven sobrecargados con el código de comunicación y pueden diseñarse para cumplir funciones empresariales distintas. Las colas de mensajes constituyen un método sencillo y organizado para desacoplar sistemas distribuidos, ya sea en el contexto de aplicaciones monolíticas, microservicios o arquitecturas sin servidor.
- **Migración a microservicios:** La adopción de patrones de integración de microservicios basados en eventos y mensajes asincrónicos ofrece mejoras significativas en términos de escalabilidad y resiliencia del sistema. Para coordinar la comunicación entre múltiples microservicios, se recomienda el uso de servicios de colas de mensajes. Estos servicios permiten notificar a los microservicios sobre cambios en los datos, actuar como un canal de eventos para el procesamiento de datos de IoT, redes sociales y otras aplicaciones en tiempo real. Esta arquitectura asegura una integración eficiente y robusta, facilitando la gestión de la carga y la recuperación ante fallos, lo que resulta crucial en entornos dinámicos y de alta demanda.

3.3.4.2 Beneficios de brokers publicación/suscripción: De igual manera la mensajería de tipo publicador/suscripción cuenta con ciertas ventajas que pueden ser de utilidad para su uso en aplicativos de microservicios. De acuerdo con AWS,[24] algunas de sus ventajas son:

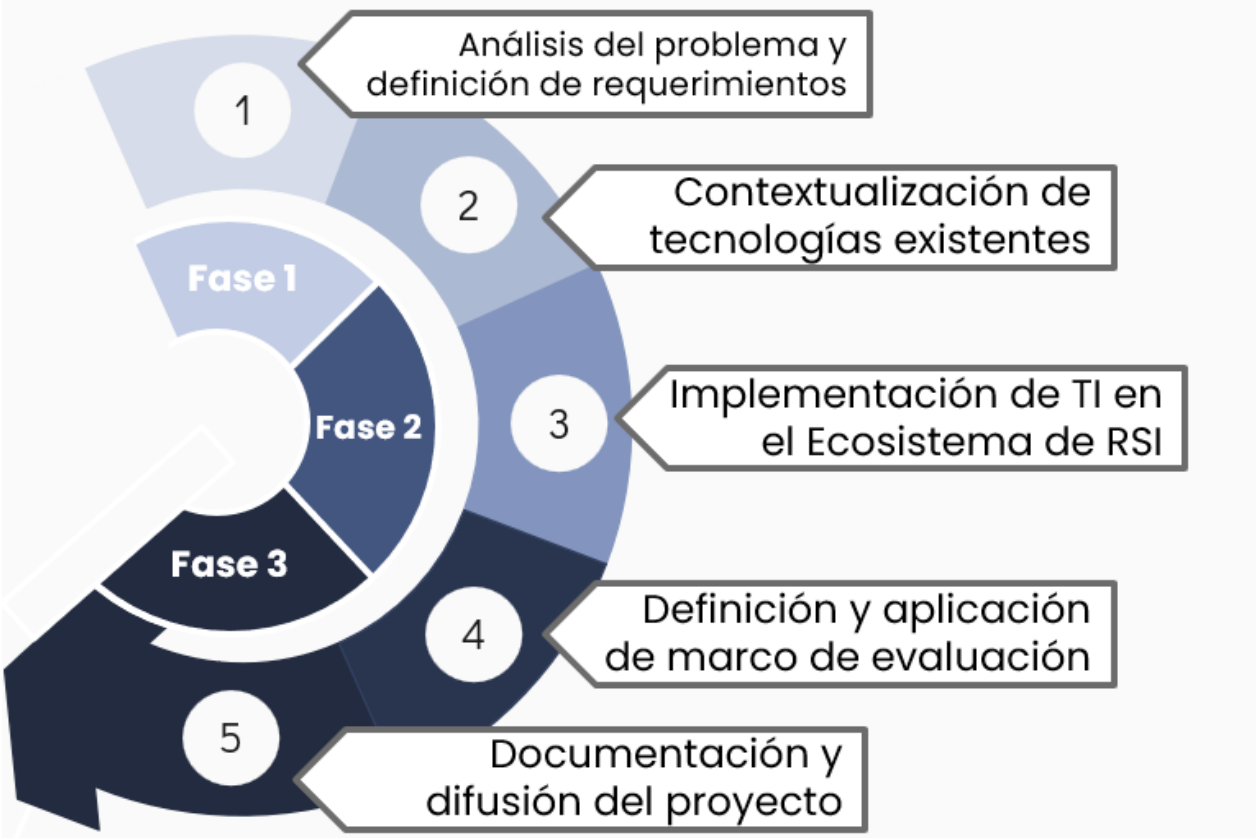
[24] AMAZON WEB SERVICES. [¿Qué son los mensajes de publicación y suscripción?] España: AWS. [Consulta: 20 de diciembre 2023]. Disponible en: https://aws.amazon.com/es/what-is/pub-sub-messaging/.

- **Eliminación de sondeos:** El uso de temas en los mensajes permite una entrega inmediata basada en push, eliminando la necesidad de que los destinatarios comprueben periódicamente la existencia de nueva información y actualizaciones. Esto acelera el tiempo de respuesta y reduce la latencia de entrega, eliminando así posibles retrasos en procesos internos del aplicativo.
- **Focalización dinámica:** El patrón objeto de estudio simplifica el proceso de descubrimiento de servicios, reduciendo la propensión a errores. Esto es gracias a que, en lugar de mantener una lista de pares para la comunicación, un emisor publica mensajes en un tema y cualquier parte interesada puede suscribirse al tema para empezar a recibir dichos mensajes. Los suscriptores pueden cambiar, actualizarse o desaparecer, y el sistema se ajusta dinámicamente a estos cambios.
- **Desacoplamiento y escalado independientes:** El modelo de publicación y suscripción incrementa la flexibilidad del software al desacoplar emisores y suscriptores, permitiendo que operen de manera independiente. Esto facilita el desarrollo y la ampliación modular, permitiendo cambios en la gestión de pedidos o funciones sin afectar al sistema en su totalidad.
- **Simplificación de la comunicación:** El código para las comunicaciones e integraciones es uno de los más complejos de desarrollar. El modelo de publicación y suscripción reduce esta complejidad al eliminar las conexiones directas entre puntos, sustituyéndolas por una única conexión a un tema de mensaje. El tema administra las suscripciones y decide qué mensajes se entregan a cada punto de conexión resultando igualmente en un acoplamiento más flexible.
- **Seguridad:** Los temas de los mensajes autentican las aplicaciones que intentan publicar contenidos y permiten el uso de puntos de conexión cifrados, protegiendo así los mensajes en tránsito a través de la red.

4. MARCO METODOLÓGICO

Para iniciar el desarrollo del proyecto, se propone adoptar un marco metodológico compuesto por cinco actividades principales, diseñadas específicamente para abordar distintos aspectos de la propuesta realizada usando una estructura secuencial. De igual manera, estas actividades se distribuyen, en tres fases, proporcionando una organización más efectiva de las temáticas tratadas y asegurando una estructura óptima para el correcto desarrollo del proyecto propuesto. Esta estructura propuesta se encuentra representada a detalle por la figura 7.

Figura 7. Propuesta de marco metodológico



A continuación, se explican cada una de las actividades propuestas en la metodología asociada, seguido de los resultados esperados de cada una de ellas.

1. Análisis del Problema y Definición de Requerimientos:

Para iniciar el proyecto de grado, se llevó a cabo un análisis de los problemas asociados con el uso de microservicios en el ecosistema de RSI. Se identificaron los principales riesgos y se evaluó cómo la implementación de brokers de mensajería podría mitigar estos riesgos y mejorar la confiabilidad del sistema. Este análisis no solo reveló los desafíos actuales, sino que también permitió establecer objetivos específicos que deben alcanzarse mediante la implementación de esta tecnología, definiendo así los requerimientos técnicos necesarios para asegurar una implementación exitosa en RSI.

2. Contextualización de las tecnologías existentes:

En esta etapa, se llevó a cabo una investigación de las tecnologías disponibles en el mercado con respecto a brokers de mensajería. Se compararon las ventajas y capacidades de las opciones más relevantes, evaluando su adecuación a los requerimientos definidos previamente. Este enfoque busca no solo adquirir conocimientos esenciales, sino también comprender en profundidad la dinámica de la nueva tecnología, estableciendo una base sólida escencial para realizar la implementación del broker de mensajería en los sistemas de RSI. Esta investigación fue documentada en un informe técnico que se añadió como anexo.

3. Implementación de TI en el Ecosistema de RSI:

Durante esta fase, se desarrolló el código necesario para la implementación propuesta, aplicando los conocimientos adquiridos en la etapa anterior y orientando el desarrollo hacia los objetivos propuestos en el proyecto. Este paso da inicio de la transformación tecnológica en RSI, proporcionando soluciones concretas a los problemas identificados en los módulos seleccionados para ejecutar la implementación.

4. Definición y aplicación de marco de evaluación:

Una vez completada la implementación de la tecnología, se ejecutaron una serie de pruebas para evaluar la correcta integración del broker de mensajería en el ecosistema

de RSI. Estas pruebas buscan demostrar la viabilidad de la tecnología para cumplir con los objetivos planteados en la tesis, especialmente en términos de cumplimiento de los requerimientos establecidos. Los resultados de las pruebas se documentaron en los anexos correspondientes para corroborar su realización y efectividad.

5. Documentación y difusión del Proyecto:

Finalmente, se llevó a cabo una documentación exhaustiva de todos los procesos relacionados con la instalación y ejecución del broker seleccionado, plasmada en la wiki de RSI que proporciona una guía detallada para futuras implementaciones. Esta documentación sirve tanto como referencia técnica como recurso para el aprendizaje continuo y la mejora de procesos.

Adicionalmente, tras la implementación exitosa, se realizó un proceso de difusión que incluye capacitaciones específicas para los miembros asociados a RSI. El objetivo es no solo presentar los resultados obtenidos, sino también familiarizar a la comunidad con el nuevo modelo de comunicación implementado y sus beneficios. Este proceso facilitará la aceptación y adopción de los cambios en el sistema de RSI.

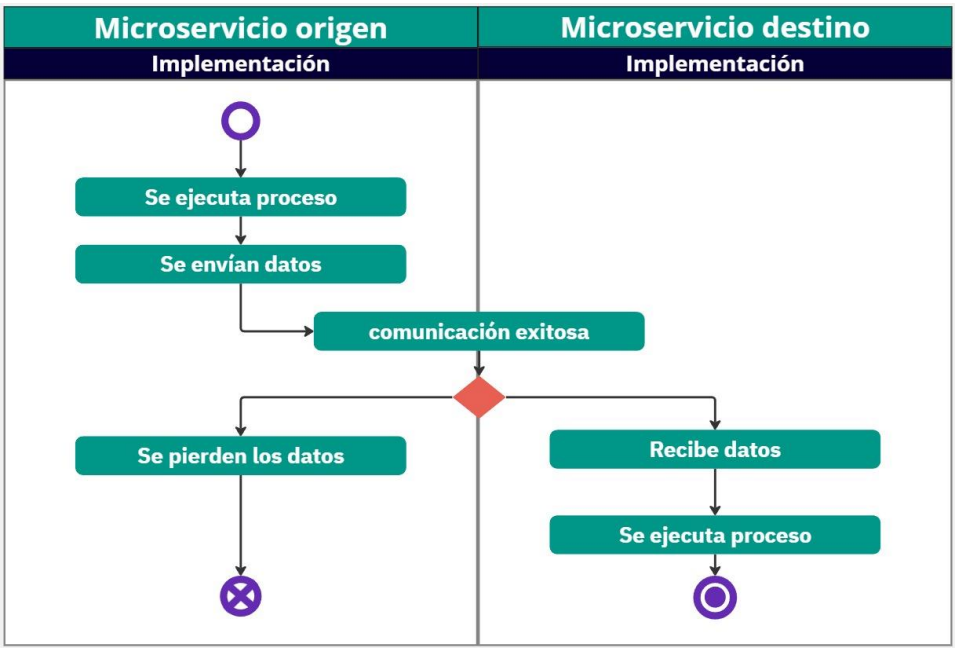
5. DISEÑO DE LA SOLUCIÓN

Para dar inicio al desarrollo del proyecto de grado se realizaron las actividades correspondientes al diseño de la propuesta para la implementación del broker dentro de los sistemas de RSI. A continuación, se describe detalladamente la definición de la problemática, los requerimientos a abordar, la selección de la tecnología elegida y la arquitectura de implementación propuesta.

5.1 Definición de problemática

Debido a la naturaleza de los sistemas basados en microservicios implementados en el Proyecto de Renovación de Sistemas (RSI), se identifican diversas desventajas inherentes a esta arquitectura, especialmente en términos de comunicación entre los componentes del software. En la actualidad, el ecosistema de RSI presenta vulnerabilidades relacionadas con la transmisión de datos entre los distintos componentes BackEnd, lo que puede resultar en la pérdida de información crucial durante diversas solicitudes, caso que se representa mediante la figura 8. Esta situación se torna particularmente problemática cuando los procesos involucran múltiples peticiones en único flujo, tales actividades no cuentan con un seguimiento adecuado de los procesos internos de estas, generando inconsistencias en el envío adecuado de datos.

Figura 8. Problemática en arquitectura de microservicios



Esta problemática ha captado la atención a los directivos de RSI, quienes buscan soluciones que mejoren el control sobre los fallos descritos y permitan un seguimiento efectivo de la información. En consecuencia, en colaboración con el líder BackEnd y especialista en gestión de proyectos, se exploró la implementación de brokers de mensajería como método de solución. La implementación de esta tecnología ofrece una alternativa efectiva para mitigar los fallos anteriormente explicados y asegurar una comunicación robusta entre los microservicios que componen el entorno de RSI, gracias a la naturaleza de los brokers, se permite brindar una mejor estabilidad sin comprometer la arquitectura usada.

Tras revisar a fondo los microservicios que componen el ambiente de RSI, se evidencia una notable falta de viabilidad en el desarrollo del microservicio de Notificaciones, responsable de la gestión de peticiones cruciales dentro del sistema, como el envío de correos electrónicos tanto de manera síncrona como asíncrona, este módulo es el encargado de recibir peticiones de todos los demás proyectos presentes, llevando una gran carga de manejo constantemente. Dada la importancia y criticidad de este módulo,

es crucial asegurar su sostenibilidad mediante la adopción de un sistema de brokers adecuado, garantizando así la calidad y estabilidad del proyecto en su totalidad, por lo cual, se ha decidido hacer uso de este como prueba piloto de la implementación realizada, estableciendo una base para futuras implementaciones en demás microservicios del proyecto donde el módulo de Notificaciones actúa como consumidor y recibe peticiones desde un servicio que realiza el papel de productor que para esta primera implementación se trata del microservicio HojaDeVida.

5.2 Definición de requerimientos

Tomando como base la problemática asociada se redactaron una serie de requerimientos con base en las necesidades del sistema solicitadas por el líder BackEnd del proyecto RSI. Estos requerimientos permiten evaluar las características de los brokers de mensajería a evaluar durante el desarrollo de la propuesta y ayudan a ejercer una elección más metódica de la tecnología.

5.2.1 Requerimientos funcionales

Tras el análisis de la problemática anteriormente revisada, se identificaron los siguientes requerimientos funcionales presentados a continuación:

- Comunicación Desacoplada de Microservicios: Implementar una solución basada en brokers de mensajería que permita la comunicación desacoplada entre microservicios, facilitando un entorno de desarrollo independiente y modular.
- Implementación en Infraestructura RSI: El sistema debe ser desplegado en la infraestructura de servidores de RSI para evitar la tercerización y garantizar el control total sobre los datos y operaciones.
- Mecanismos de Monitoreo y Notificación: El sistema debe incorporar funcionalidades de monitoreo y reporte del estado de los mensajes, proporcionando notificaciones en tiempo real sobre la entrega exitosa o fallida de los mismos.

- Soporte de Mensajería Síncrona y Asíncrona: La solución debe soportar tanto la mensajería asíncrona como la síncrona, permitiendo flexibilidad para satisfacer los requisitos específicos de diferentes tipos de solicitudes.
- Persistencia de Mensajes en Caso de Fallo: El sistema debe incluir mecanismos de persistencia de mensajes que aseguren el almacenamiento y recuperación de mensajes no procesados en situaciones de fallo del sistema.
- Manejo de Errores: El sistema debe contar con capacidades robustas de manejo de errores, minimizando la pérdida de mensajes y asegurando la integridad de los datos transmitidos.
- Seguridad y Autenticación: El sistema debe implementar mecanismos de autenticación y autorización robustos para asegurar que solo usuarios y servicios autorizados puedan acceder y enviar mensajes a través del broker.

5.2.2 Requerimientos no funcionales

De igual manera, se realiza la documentación de los requerimientos no funcionales a considerar dentro de la propuesta e implementación del broker de mensajería a seleccionar:

- Compatibilidad con los Patrones de Desarrollo RSI: El sistema debe ser plenamente compatible con los patrones y metodologías de desarrollo actualmente utilizados en el entorno de RSI, asegurando una integración sin fricciones.
- Escalabilidad para Grandes Volúmenes de Datos: La solución debe garantizar la capacidad de manejar grandes volúmenes de datos, escalando eficientemente para satisfacer las demandas crecientes.
- Alta Disponibilidad: El sistema debe proporcionar alta disponibilidad, asegurando la continuidad del servicio incluso en casos de fallos de componentes individuales.
- Facilidad de Implementación y Uso: La implementación del sistema debe ser intuitiva y accesible para los desarrolladores, facilitando un rápido despliegue y un mantenimiento sencillo, promoviendo la adopción y el uso eficiente del sistema.

5.3 Proceso de selección de tecnología

La selección de un broker de mensajería apropiado es esencial para la mejora de los procesos asociados al proyecto RSI, con el fin de garantizar incrementar la confiabilidad y eficiencia operativas a través de la implementación de este método en el actual aplicativo basado en una arquitectura de microservicios, se espera que el broker cumpla con los criterios más adecuados para esta implementación y brinde una solución que pueda satisfacer las necesidades del proyecto.

5.3.1 Comparativa brokers de mensajería

A través de un análisis detallado, descrito en detalle en el anexo A, se ha recopilado información sobre cuatro posibles opciones, las cuales representan las opciones vigentes con más relevancia en el mercado en tecnologías de brokers de mensajería, siendo estos: Apache Kafka, RabbitMQ, ActiveMQ y MosquittoMQTT. Cada una de estas tecnologías ha sido evaluada en función de características técnicas y operativas relevantes, como se detalla en la Tabla 1. Este análisis comparativo proporciona una base sólida para tomar una decisión fundamentada sobre la implementación más adecuada para el ecosistema de RSI.

Tabla 1. Comparativa Brokers de Mensajería

	Apache Kafka	***RabbitMQ***	***ActiveMQ***	***Mosquitto MQTT***
Modelo de Mensajería	Protocolo binario sobre TCP	AMQP con complementos para otros protocolos	AMQP, MQTT y STOMP entre otros	MQTT
Patrones	Publicación/Suscripción, procesamiento de flujos	Publicación/Suscripción, colas de trabajo, enrutamiento	Publicación/Suscripción, colas de trabajo	Publicación/Suscripción

Escalabilidad	Muy alta, diseñado para grandes volúmenes de datos	Buena, Soporte de clustering	Buena, Clustering disponible	Adecuado para redes IoT
Rendimiento	Mejor rendimiento en throughput y latencia para grandes volúmenes de datos	Rendimiento fiable con baja latencia y buen throughput para volúmenes medios	Rendimiento fiable baja latencia y buen throughput para volúmenes medios	Baja latencia y eficiencia de recursos en aplicaciones IoT No destaca en otro tipo de aplicaciones
Persistencia	Predeterminada	Configurable	Configurable	Limitada, Configurable
Tolerancia a Fallos	Alta disponibilidad	Alta disponibilidad	Alta disponibilidad	Básica
Facilidad de Uso	Requiere más configuración	Buena configuración y Soporte	Flexibilidad en protocolos y configuraciones	Simple de configurar y usar
Casos de uso	Análisis en tiempo real Big data Streaming	Sistemas empresariales Comunicación entre microservicios	Aplicaciones empresariales variadas	Dispositivos de baja potencia IoT

5.3.1.1 Modelo de Mensajería: Uno de los factores fundamentales a considerar en el desarrollo de esta propuesta es el modelo que pueda proporcionar interoperabilidad

versátil al sistema y adaptarse al entorno de microservicios establecido en el ecosistema de RSI. En este contexto, RabbitMQ, utilizando el protocolo AMQP, ofrece un sistema lo suficientemente robusto para gestionar la cantidad de peticiones realizadas en sistemas administrativos como el objetivo de la propuesta, siendo un protocolo que permite integrarse con los lenguajes presentes en los desarrollos del proyecto y asegurando una comunicación fluida y escalable.

Otra alternativa viable para esta propuesta podría ser ActiveMQ, conocido por su flexibilidad para adaptarse a diversas tecnologías y necesidades mediante sus protocolos, facilitando la integración de sistemas heterogéneos, lo cual presenta un beneficio significativo para el entorno en el que se implementa el broker.

Por otro lado, aunque Apache Kafka está diseñado para el procesamiento en tiempo real, su implementación conlleva una complejidad adicional de administración que puede no ser necesaria para este caso específico. En contraste, Mosquitto MQTT presenta limitaciones en su capacidad de adaptación al sistema debido a su enfoque primario en dispositivos IoT.

5.3.1.2 Patrones: Los patrones de mensajería son fundamentales para definir cómo se comunican los microservicios con el broker, esta característica permite adaptarse a las diversas necesidades de comunicación que requiere el ecosistema de RSI. Una vez más, RabbitMQ se presenta como una opción idónea al soportar patrones de publicación/suscripción, colas de trabajo y enrutamiento. Esto proporciona un entorno versátil para la gestión de mensajes según lo requieran los desarrollos, ofreciendo además opciones robustas para manejo de errores a través del enrutamiento.

En contraste, los demás brokers considerados ofrecen menos variedad en términos de patrones de mensajería, lo que destaca a RabbitMQ como la opción más adecuada en este ámbito específico.

5.3.1.3 Escalabilidad: En cuanto a la escalabilidad, factor destacado para asegurar un mejor rendimiento tras la ampliación de la tecnología sobre los microservicios de RSI, la opción más adecuada es indudablemente Apache Kafka. Este broker está diseñado para el procesamiento de datos en tiempo real y puede manejar grandes volúmenes de datos. Sin embargo, dado que el sistema objetivo del proyecto se centra en la gestión de datos dentro de un entorno universitario, la potencia ofrecida por Apache Kafka en este contexto puede resultar innecesaria y añadir complejidad adicional durante la implementación de los métodos que utilicen este broker.

En cambio, se considera que RabbitMQ y ActiveMQ son opciones más apropiadas para la implementación. Ambos brokers ofrecen un buen nivel de escalabilidad y soportan la configuración de clusters de brokers, lo cual es beneficioso para entornos que requieren crecimiento y distribución de carga controlada.

Por otro lado, Mosquitto MQTT vuelve a ser la opción menos favorable debido a su naturaleza orientada a dispositivos IoT, que puede no adaptarse eficientemente a las necesidades del sistema universitario en términos de escalabilidad y capacidad de procesamiento de datos.

5.3.1.4 Rendimiento: Al igual que en el caso anterior, Apache Kafka destaca como el broker más adecuado debido a su capacidad para manejar grandes volúmenes de datos en tiempo real. Sin embargo, como se mencionó anteriormente, esta potencia puede ser excesiva para la implementación requerida en el proyecto, lo que posiciona a RabbitMQ y ActiveMQ como opciones más apropiadas.

RabbitMQ y ActiveMQ ofrecen un rendimiento adecuado para aplicaciones empresariales y de volumen medio, equilibrando eficiencia y capacidad de procesamiento sin la complejidad adicional asociada con Apache Kafka. Por otro lado, Mosquitto MQTT se considera la opción menos favorable debido a su orientación hacia dispositivos IoT, lo

cual puede limitar su rendimiento en un entorno universitario que requiere una mayor capacidad de procesamiento y escalabilidad.

5.3.1.5 Persistencia: La persistencia de mensajes, es un factor crucial para garantizar un manejo adecuado de errores en los casos donde el envío de mensajes no sea efectivo. Tanto RabbitMQ como ActiveMQ ofrecen un entorno configurable para la gestión de mensajes, lo cual proporciona una flexibilidad significativa, especialmente en términos de manejo de errores, promoviendo un entorno adaptable a las necesidades de RSI. Estas características posicionan a ambos brokers como opciones viables para el desarrollo de la propuesta.

Por otro lado, Apache Kafka utiliza una estrategia de persistencia predeterminada, lo cual limita su flexibilidad, pero asegura un almacenamiento de datos duradero y confiable.
En contraste, Mosquitto MQTT se presenta como una opción limitada para la implementación, lo que lo convierte en la menos favorable entre las alternativas consideradas.

5.3.1.6 Tolerancia a fallos: La disponibilidad en cuanto a tolerancia de fallos es una característica crítica en los brokers de mensajería. Estos atributos permiten que el sistema continúe operando a pesar de posibles fallos o errores. En este contexto, Apache Kafka, RabbitMQ y ActiveMQ destacan por su alta tolerancia a fallos, lo que hace que cualquiera de estas opciones sea adecuado para la implementación del broker en el ecosistema de RSI.

5.3.1.7 Facilidad de uso: La facilidad de implementación es un factor relevante, aunque no decisivo, para asegurar que cualquier desarrollador dentro de RSI pueda utilizar el sistema de manera eficiente. En este ámbito, RabbitMQ destaca como una opción viable debido a su excelente documentación y soporte, lo que facilita significativamente el desarrollo e implementación del broker y permite una rápida adaptación al mismo. Por su lado, Mosquitto MQTT también presenta una alternativa fácil de usar y configurar,

proporcionando un proceso de adaptación sencillo al sistema a la cual pueden adecuarse los desarrolladores con gran facilidad.

ActiveMQ ofrece una buena facilidad de uso, aunque no supera a RabbitMQ, situándose como una opción intermedia en términos de facilidad de uso.

Finalmente, Apache Kafka es la opción con menor facilidad de uso entre las consideradas, ya que requiere una configuración más compleja y un mayor conocimiento para su manipulación. Esta tecnología presenta una curva de aprendizaje más pronunciada en comparación con las otras opciones, haciendo que su manejo sea más desafiante.

5.3.2 Selección del broker

La selección del broker de mensajería adecuado para el sistema de administración de datos de una universidad es crucial para mejorar la eficiencia y confiabilidad de los procesos del proyecto RSI. Tras la realización del análisis comparativo de las características de cada una de las opciones y la adecuación de los respectivos brokers, se concluye que RabbitMQ es la opción más apropiada para la implementación propuesta.

RabbitMQ, utilizando el protocolo AMQP, proporciona una robusta interoperabilidad y adaptabilidad al entorno de microservicios del RSI. Este protocolo facilita la integración con diversos lenguajes de programación utilizados en el proyecto, asegurando una comunicación fluida y escalable. De igual manera, RabbitMQ sobresale una mayor variedad de modelos de mensajería, incluyendo publicación/suscripción, colas de trabajo y enrutamiento, brindando una mayor cantidad de posibilidades de diseño de código. En términos de escalabilidad y rendimiento, aunque Apache Kafka es superior en teoría, su capacidad excede los requisitos del sistema universitario y añade una complejidad innecesaria. RabbitMQ, con su capacidad de clustering, proporciona un crecimiento

controlado y una distribución eficiente de la carga, siendo más adecuado para las necesidades de RSI.

Por otro lado, la persistencia de mensajes es un factor crítico para garantizar un manejo adecuado de errores. RabbitMQ ofrece una persistencia configurable, proporcionando flexibilidad y adaptabilidad, lo que es crucial para RSI. Aunque Apache Kafka es confiable, su persistencia predeterminada es menos flexible, limitando su adaptabilidad a las necesidades específicas del proyecto. En términos de tolerancia a fallos, RabbitMQ sobresale con alta disponibilidad y clustering, asegurando la continuidad operativa del sistema, comparable a Apache Kafka y ActiveMQ.

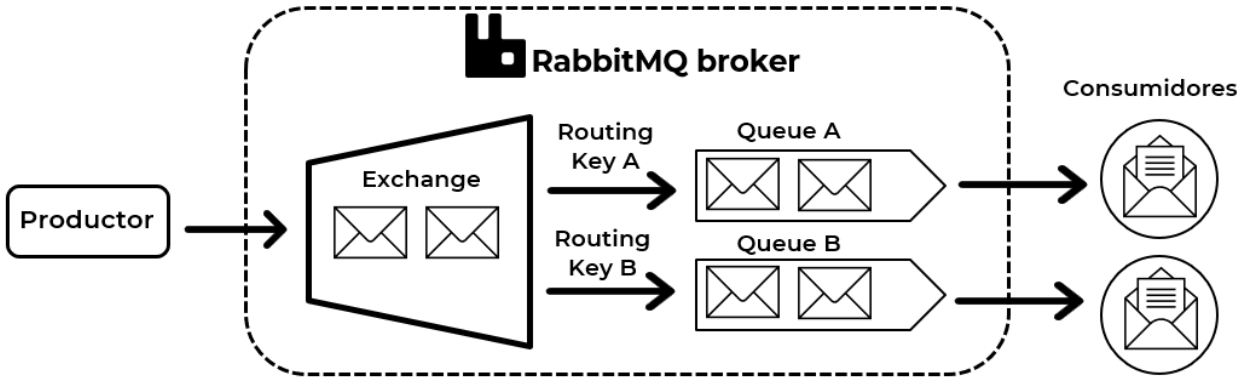
Finalmente, en cuanto a la facilidad de uso, RabbitMQ destaca notablemente por su excelente documentación y soporte, facilitando su implementación y gestión. Esta accesibilidad es crucial para los desarrolladores de RSI, permitiendo una rápida adaptación y uso eficiente del broker.

En conclusión, RabbitMQ es la opción más equilibrada y adecuada para el sistema de administración de datos de la universidad. Su robustez, flexibilidad, escalabilidad, rendimiento confiable, manejo configurable de persistencia, alta tolerancia a fallos y facilidad de uso lo convierten en la solución ideal para soportar los microservicios y procesos empresariales de una institución académica como la Universidad Industrial de Santander.

5.4 Arquitectura de comunicación propuesta

Antes de iniciar con la propuesta realizada, es necesario abordar el funcionamiento y arquitectura de RabbitMQ, este nos propone un modelo de mensajería asíncrona basado en colas, donde además de los componentes vistos en el desarrollo del marco de referencia como los productores y consumidores, cuenta con un sistema de manejo de mensajes internamente visto en la figura 9.

Figura 9. Arquitectura RabbitMQ



Estas partes cuentan con las siguientes funciones: [25]

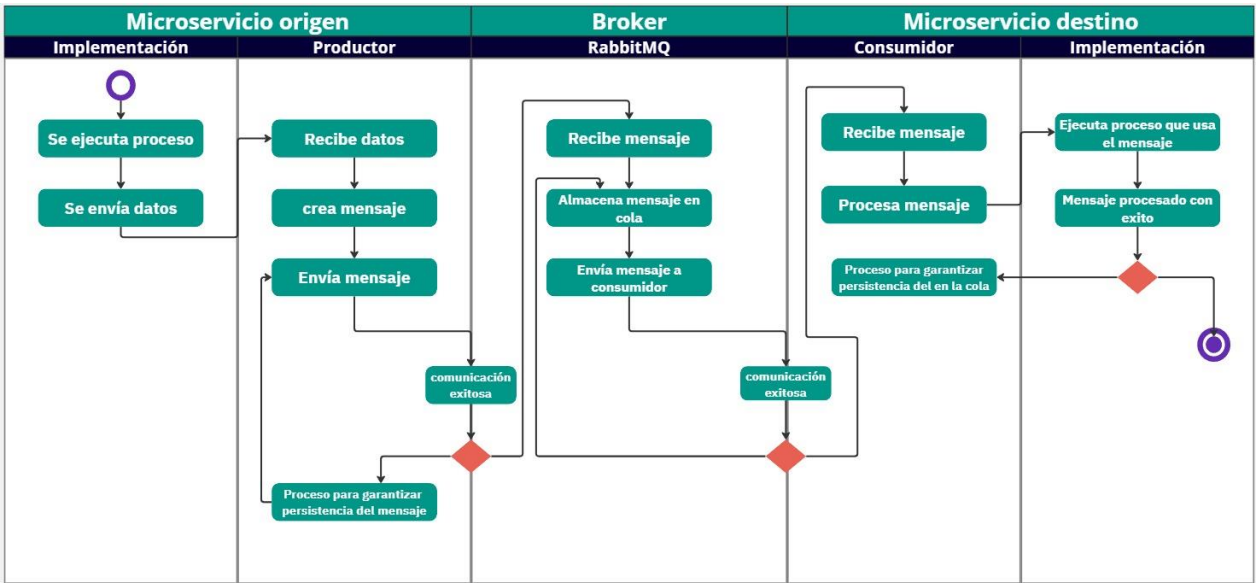
- **Intercambio/Exchange:** Actua como un intermediario que recibe mensajes de los productores y determina cómo deben ser enrutados hacia las colas apropiadas. Los exchanges son configurables y pueden seguir diferentes reglas de enrutamiento, como enrutamiento directo, basado en temas o enrutamiento con comodines.
- **Enlace/bindings:** Los bindings son configuraciones que determinan cómo los mensajes se enrutan desde los exchanges hacia las queues. Los bindings pueden utilizar diversas reglas de enrutamiento basadas en patrones o criterios específicos, como claves de enrutamiento.
- **Cola/Queue:** Una queue en el contexto de mensajería es un almacenamiento intermedio donde los mensajes se acumulan en espera de ser procesados por los consumidores. Las queues aseguran que los mensajes se manejen de manera ordenada y confiable, desacoplando la producción de mensajes de su consumo.

[25] ROY, Gabin. RabbitMQ in Deep. Nueva York: Manning Publications, 2017. ISBN 9781638353225

La documentación de la arquitectura de este broker, así como otras características relevantes puede encontrarse más detallada en el anexo A y las diferentes configuraciones que pueden tener estas se encuentran en el anexo B.

Como parte del diseño de la propuesta, se ha desarrollado el proceso que describe la operación del sistema de microservicios utilizando el broker, el cual se encuentra detallado en la figura 10. Este diagrama ilustra la arquitectura de comunicación para la implementación sugerida, que incluye la integración de los microservicios productor y consumidor del sistema, los cuales para la primera implementación realizada se tratan de Notificaciones y Hoja de Vida. Para revisar el diagrama completo de la propuesta de implementación dirigirse al anexo B.

Figura 10. Diseño de arquitectura propuesta



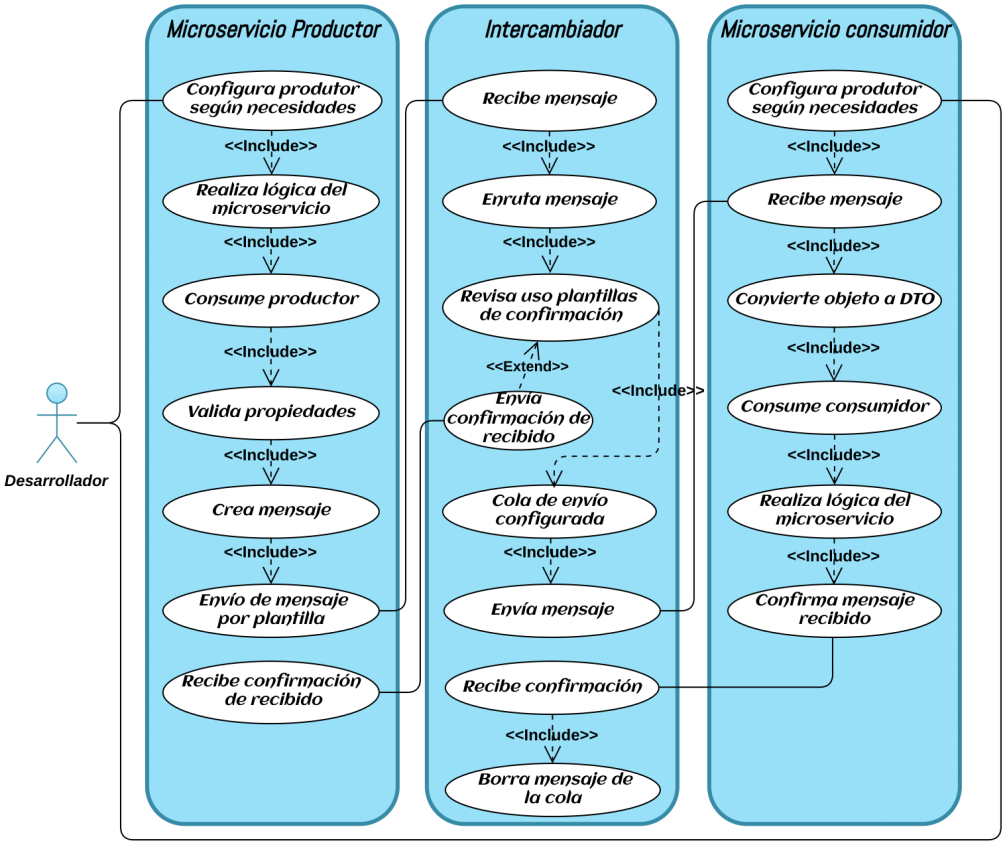
El diagrama propuesto cuenta también con la incorporación del intermediario o broker, componente fundamental en esta arquitectura, este componente además de recibir los mensajes se encarga de confirmar su entrega exitosa y en caso de fallos, el intermediario está diseñado para almacenar temporalmente los mensajes y garantizar la persistencia de la información y la integridad del proceso.

Para profundizar en la viabilidad de esta propuesta de implementación, se presentan seis casos de uso que demuestran los posibles resultados al enviar mensajes dentro de la arquitectura propuesta. Estos ilustran diversas situaciones y resultados esperados bajo el diseño propuesto, proporcionando una comprensión detallada del rendimiento y la fiabilidad del sistema.

1. Caso de uso I: Publicación y consumo de mensajes de manera correcta

Como primer caso revisado, se considera la posibilidad de que el envío se realice de manera correcta. Este sirve como base para todos los posteriores. En esta primera posibilidad, se muestra qué ocurriría al ejecutar el proceso adecuadamente. La figura 11 explica el proceso realizado detalladamente.

Figura 11. Diagrama caso I: Publicación y consumo de mensajes de manera correcta



El proceso ilustrado comienza cuando se configura el servicio productor con el método o endpoint que hará uso del broker para enviar una petición, internamente el microservicio valida las propiedades con las que se envió el mensaje. Con base en estas propiedades, el servicio crea el mensaje con la información proporcionada por el desarrollador y las propiedades adicionales necesarias para ser enviado a RabbitMQ.

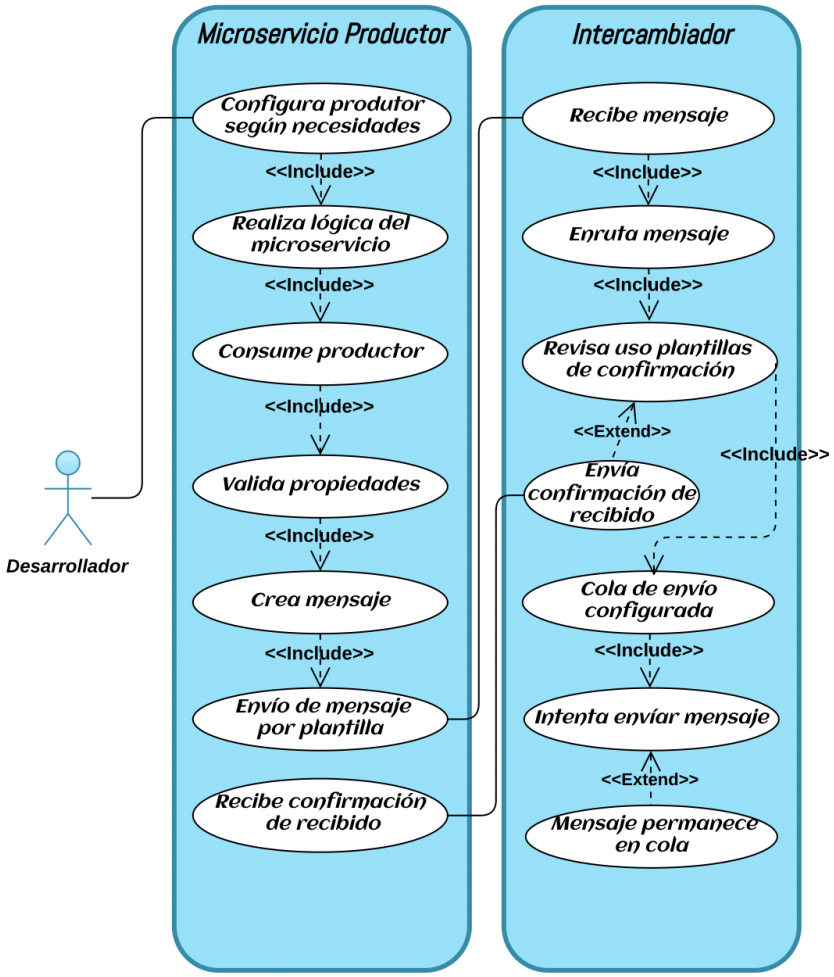
Tras el envío del mensaje, el broker se encarga de confirmar la recepción del mensaje, esto solo si se utiliza la plantilla con confirmación. Una vez asegurado que el mensaje puede ser enrutado correctamente, el broker lo dirige hacia el microservicio encargado de consumirlo.

Finalmente, tras ser correctamente recibido, el microservicio consumidor redirige la información hacia el método que el desarrollador ha definido que consumirá la información enviada, completando el proceso para el cual se creó la petición y finalizando con la confirmación del mensaje para el broker.

2.Caso de uso II: Publicación de mensajes con consumidor desconectado

El segundo caso para revisar aborda la situación en la que el consumidor no se encuentra conectado. Este es uno de los enfoques clave en el uso del broker para cumplir con los requisitos de persistencia de peticiones. Este caso ocurre, por ejemplo, cuando el servicio consumidor está caído. Este se representa con la figura 12.

Figura 12. Diagrama caso II: Publicación de mensajes con consumidor desconectado



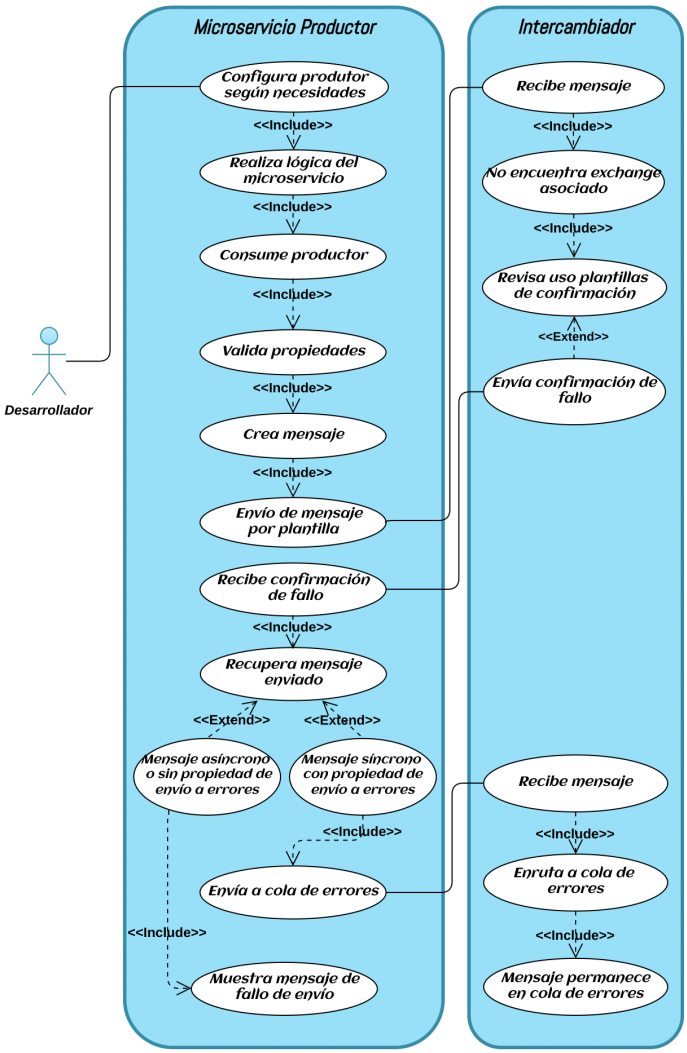
En esta ocasión el proceso en el productor es el mismo mostrado en el caso al anterior, donde el microservicio productor se encarga de la creación del mensaje y su posterior envío.

Como siguiente paso, tras la recepción del mensaje por parte del broker, este envía la confirmación al productor. En esta ocasión, el broker no puede enviar el mensaje al consumidor, ya que este no está conectado al sistema, por consiguiente, el mensaje se mantendrá en la cola hasta que pueda ser enviado o el desarrollador decida eliminarlo.

3. Caso de uso III: Fallo de publicación de mensajes por error del Exchange

Como siguiente caso, se pretende detallar el resultado necesario tras la generación de un fallo en el Exchange al que se dirige el mensaje. De acuerdo con la documentación anteriormente definida, este componente lógico permite determinar las reglas de enrutamiento, por lo que debe de contarse con una medida de precaución ante esta posibilidad. Este caso es representado con la figura 13:

Figura 13. Diagrama caso III: Publicación de mensajes en cola de errores por fallo en el Exchange



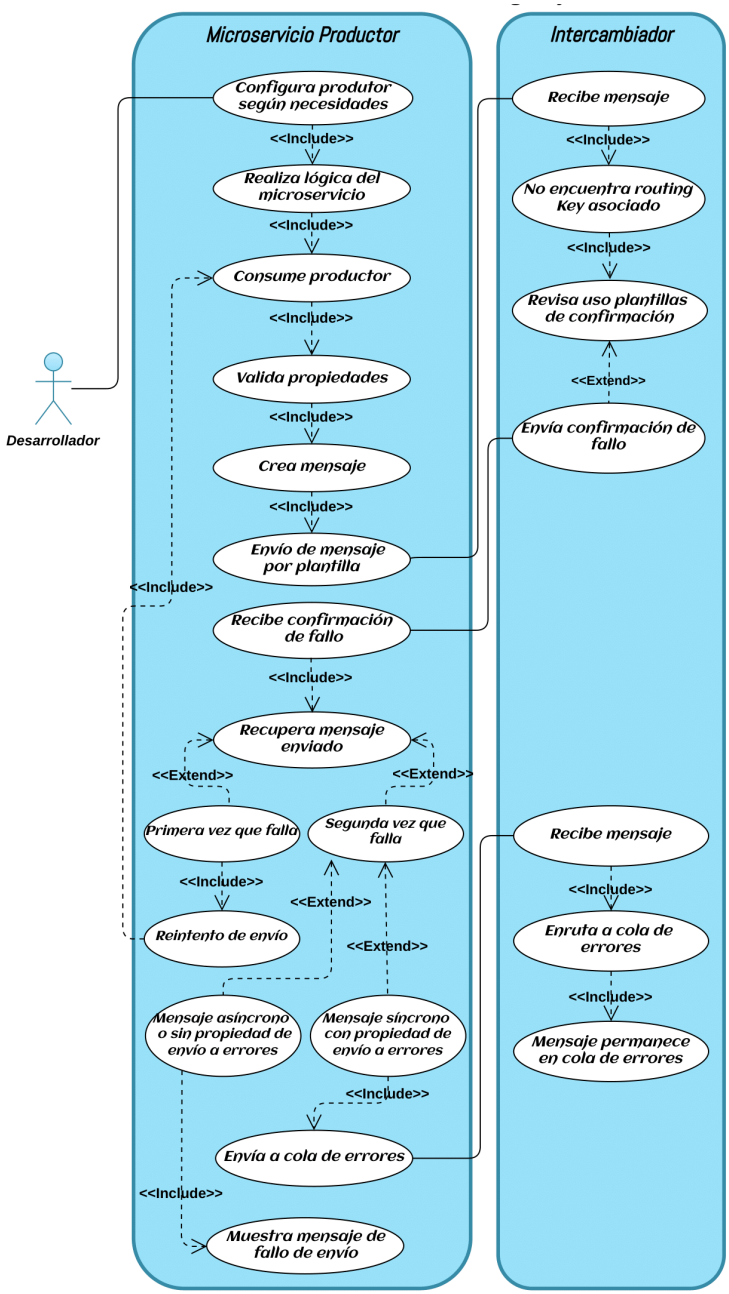
Tal como está definido en la figura 13, el proceso comienza de manera similar, desarrollando y ejecutando los procesos de envío de mensajes vistos en el caso de uso anterior, sin embargo,al fallar el envío por parte del microservicio productor, el broker no puede enrutar correctamente el mensaje y envía una confirmación de fallo, lo que activa inmediatamente un proceso de manejo de errores dentro del productor.

El proceso para manejo de errores consiste en la recuperación del mensaje y la posterior información al desarrollador sobre el fallo. De igual manera, si el mensaje cuenta con con la necesidad de almacenamiento de este mensaje, microservicio cuenta con capacidades de enrutamiento que reenvía al broker el mensaje fallido con el fin de ser enrutado hasta una cola que pueda contener los mensajes para ser revisados posteriormente por los desarrolladores con el fin de conservar la información.

4. Caso de uso IV: Fallo de publicación de mensajes por error del RoutingKey

Para continuar con la documentación de casos de uso se presenta la posibilidad donde se representa qué ocurre cuando el sistema presenta fallos con el routing key. La figura 14 representa el procedimiento realizado por el sistema de mensajería en el caso propuesto.

Figura 14. Diagrama caso IV: Publicación de mensajes en cola de errores por fallo con el RoutingKey



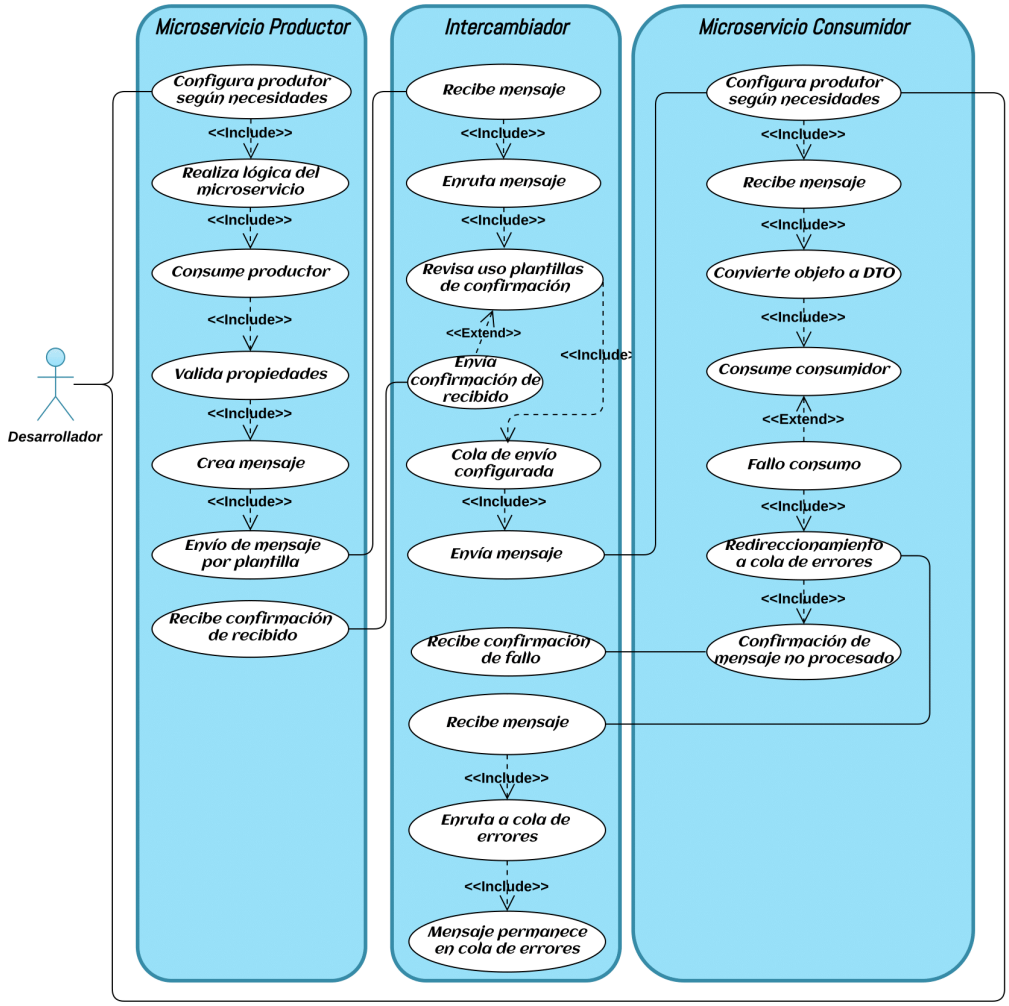
El sistema de mensajería tiene un funcionamiento similar al anteriormente estudiado, el microservicio productor se encarga de generar y enviar el mensaje hasta el broker, este devuelve una confirmación de fallo de entrega que desencadena una serie de procesos de manejo de errores.

A diferencia del proceso anterior, en esta ocasión el broker asume que el fallo ocurrido puede deberse a problemas internos del servicio, por lo que realiza un reintento de envío con la misma ruta destino. Si el envío falle nuevamente, el microservicio recupera internamente el mensaje y mostrará el error en la consola para informar al usuario, de igual manera, el mensaje se reenvía a la cola de errores predefinida para contener mensajes con problemas de envío.

5. Caso de uso V: Fallo de consumo de mensajes

A partir de este caso, se comienzan a revisar las pruebas pertenecientes a los procesos en los que los fallos ocurren en el microservicio consumidor. Este explica el proceso ocurrido dado que falle el consumo del mensaje. Esto puede darse cuando, por ejemplo, los datos enviados a través del broker de mensajería no correspondan a los solicitados por el consumidor. Este se representa en la figura 15.

Figura 15. Diagrama caso V: Fallo de consumo y posterior redireccionamiento a la cola de errores



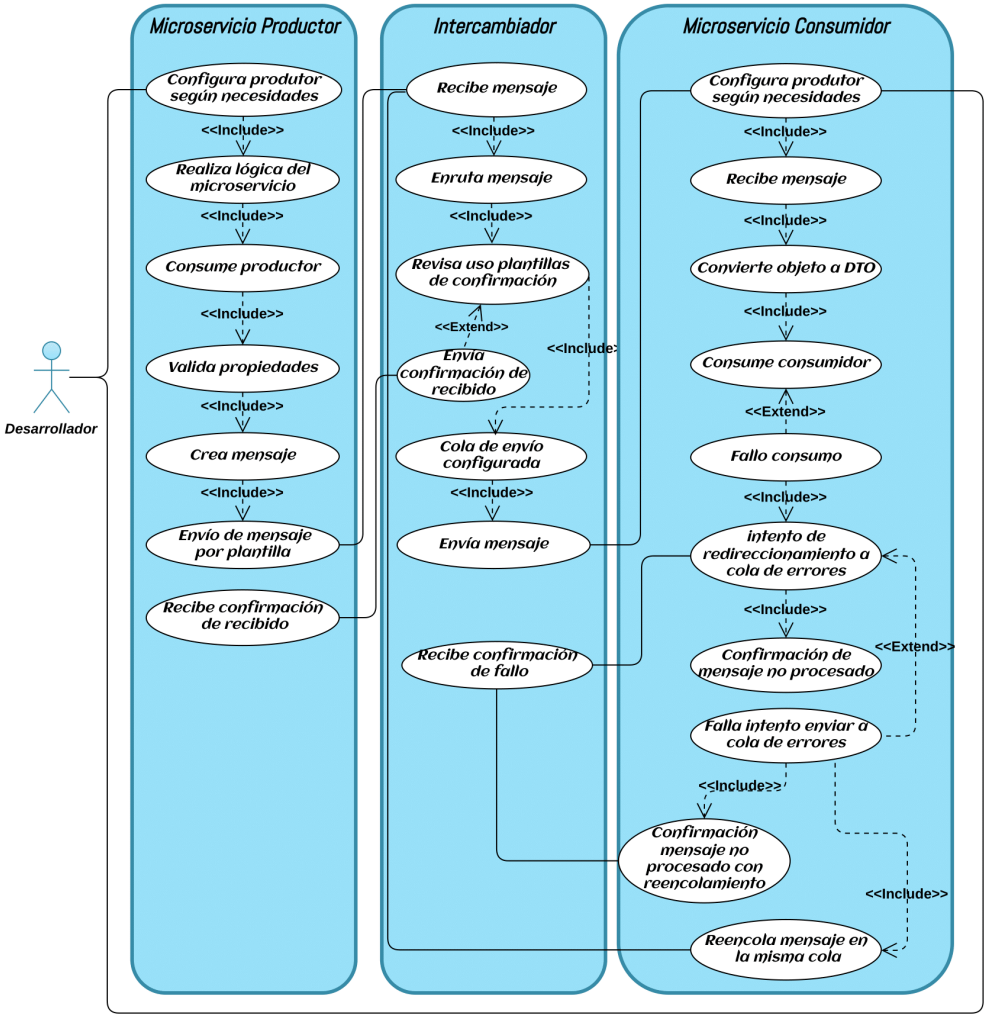
En esta ocasión se sigue el mismo procedimiento que el primer caso revisado, hasta el que el microservicio recibe el mensaje, en donde, el microservicio productor genera el mensaje con la información proporcionada por el desarrollador y hace uso de la plantilla para el envió al broker, que se encargará de redirigirlo al consumidor. Sin embargo, cuando el mensaje se intenta procesar en el consumidor y fallar, se inicia el proceso de manejo de errores correspondiente, este muestra el motivo que causó el error para

notificar al desarrollador y reenvía el mensaje al broker, que lo coloca en la cola de errores predefinida.

6. Caso de uso VI: Fallo del consumo de mensajes y fallo de tratamiento de errores

Como caso final, además de fallar el consumo del mensaje, también falla el envío a la cola de errores. Este fallo ocurre, por ejemplo, cuando las claves predeterminadas correspondientes a la dirección de la cola de errores definidas en el proyecto no son adecuadas. El caso representado en esta fase se planea como una medida de seguridad para evitar la pérdida de mensajes y se ilustra la figura 16.

Figura 16. Diagrama caso VI: Reencolamiento de respaldo posterior al fallo de consumo y de redireccionamiento a cola de errores



Como se demuestra en la figura 16, los procesos hasta el fallo se mantienen igual al del caso anterior, con diferencia que, en esta instancia, falla el enrutamiento realizado en el sistema del microservicio consumidor hacia la cola de errores. Mediante un mensaje, el sistema informa al desarrollador sobre el fallo. En esta ocasión, el broker genera una confirmación del fallo de encolamiento del mensaje, por lo que este vuelve a la ruta desde la cual ingresó al consumidor. El mensaje permanecerá encolándose y fallando en su consumo indefinidamente hasta que el desarrollador elimine el mensaje o corrija las rutas de la cola de errores.

6. IMPLEMENTACIÓN

Una vez finalizada la contextualización del problema, se llevó a cabo la implementación de la tecnología propuesta en el entorno RSI. Esta sección detalla el proceso integral de la implementación, incluyendo la explicación y documentación del código desarrollado para adaptar la arquitectura de RabbitMQ a los requisitos específicos solicitados.

6.1 Documentación previa y propuesta

Previo a la implementación en los microservicios de RSI, se realizó una investigación sobre las tecnologías, la cual puede encontrarse en el anexo C. Para tener una vista rápida de ellas, se elaboró la figura 17, donde se puede apreciar la relación de estas con la parte del proceso en la que están implicadas.

Figura 17. Tecnologías asociadas

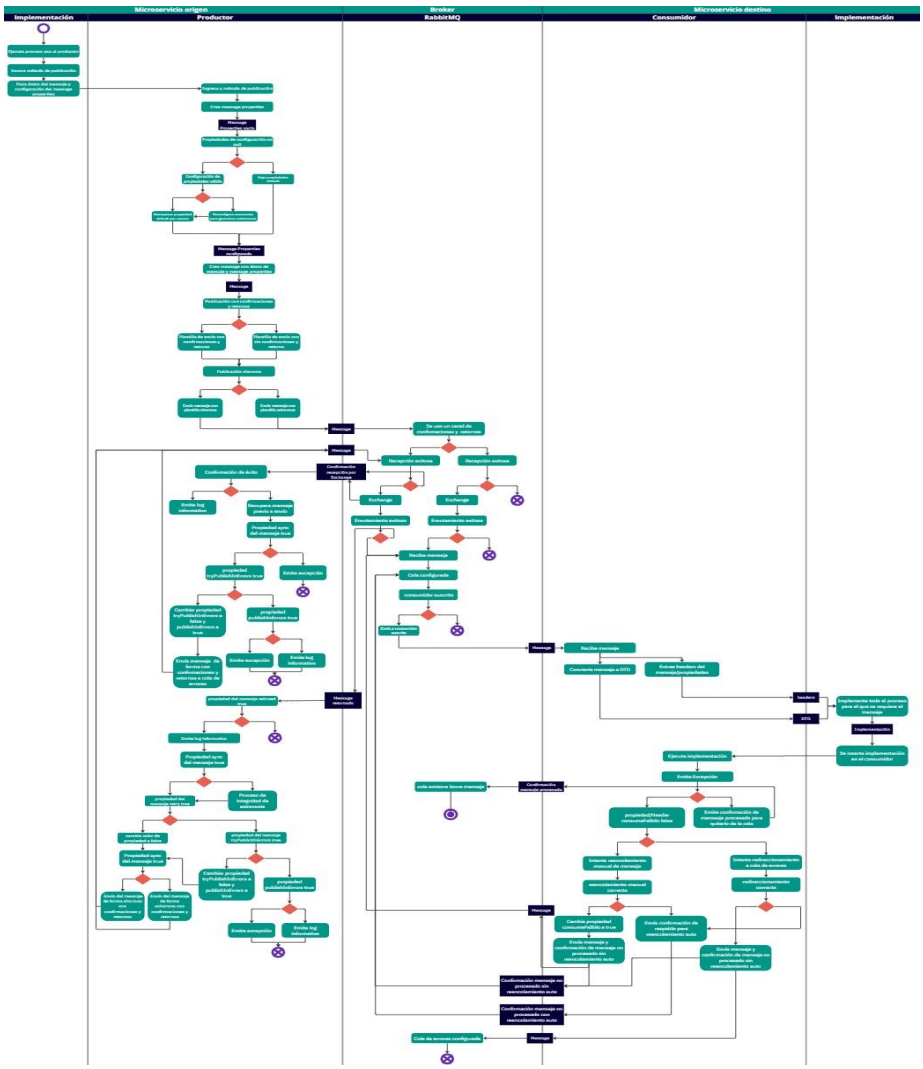
Con base en la documentación existente sobre RabbitMQ y Spring Boot, y considerando el potencial conjunto de estos, se plantea implementar una solución que aproveche al máximo sus beneficios y, además, facilite su uso en futuros desarrollos de RSI. La propuesta se centra en realizar la mayor parte del trabajo de configuración y codificación, generando así una implementación que permite al desarrollador desentenderse del trabajo específico de utilizar el broker de mensajería. De esta manera, se busca crear una forma estándar y flexible de uso que, además, proporcione una base que no solo permita centralizar al máximo el trabajo de mantenimiento, sino también explotar en el futuro virtudes de RabbitMQ no abordadas.

Por consiguiente, al igual que la librería Spring AMQP de Spring Boot ofrece diversas plantillas que facilitan la implementación de RabbitMQ según las necesidades específicas, se propone la creación de unas plantillas generales para RSI. Estas usan internamente de las plantillas de Spring AMQP, abarcando los diferentes escenarios posibles y aprovechando la documentación previamente investigada. Además, se enfocarán en aplicar estrategias garantizando la integridad y persistencia de los mensajes en caso de fallos mediante confirmaciones, retornos de mensajes y reencolamiento.

6.2 Diagrama de proceso y código desarrollado

A continuación, se presenta en la figura 18 el diagrama de flujo del proceso implementado, el cual también se puede encontrar en el anexo B.

Figura 18. Diagrama de flujo de procesos



Con el fin de facilitar la visualización y explicación del diagrama, se ha realizado una versión seccionada, la cual se estará usando más adelante en la explicación. Primero, es necesario entender algunos aspectos del proceso.

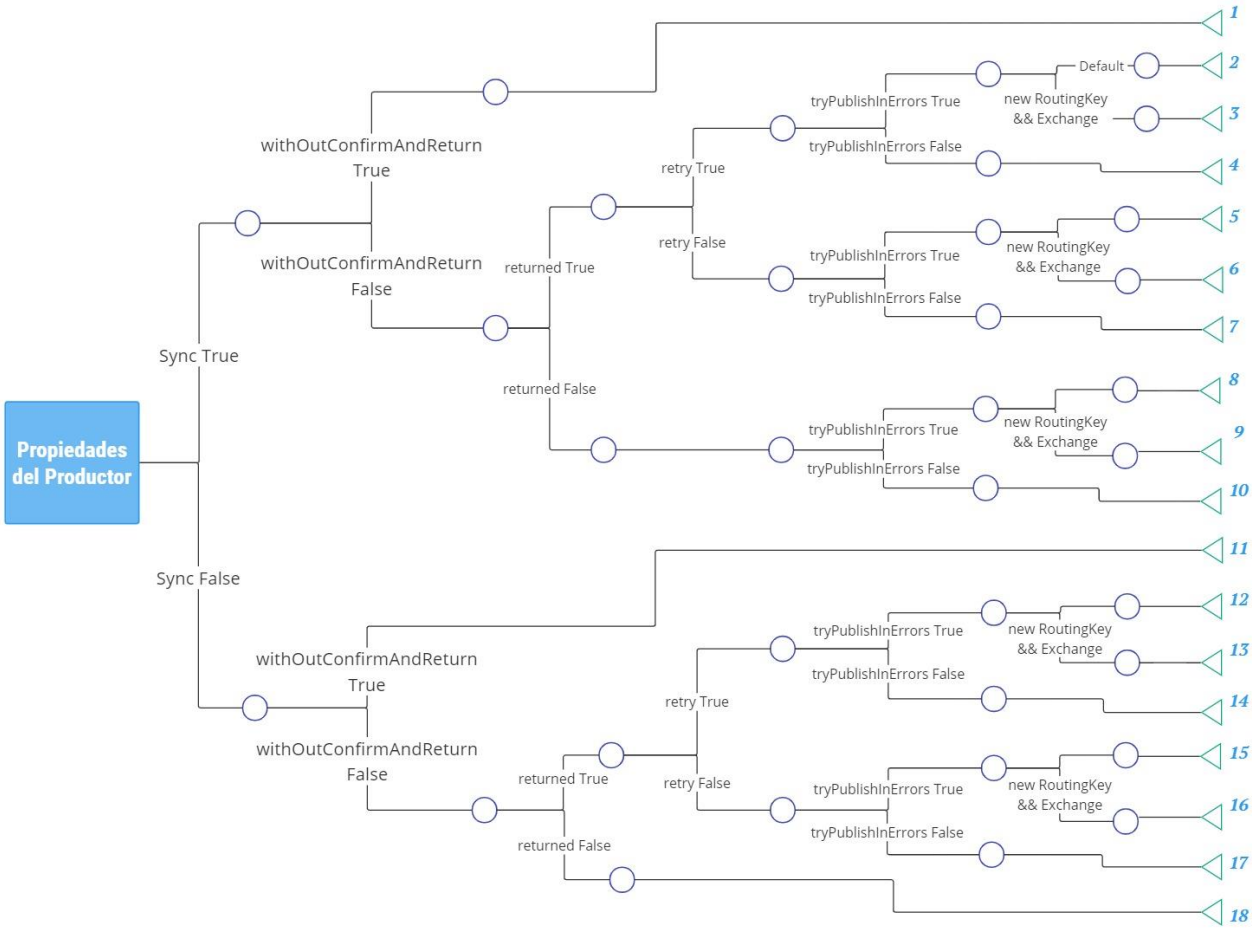
Respecto a su composición, se encuentra dividido en tres áreas generales. El origen contiene una sección denominada productor, que hace referencia a la plantilla propuesta para manejar la publicación de mensajes. El destino, por su parte, incluye una sección llamada consumidor, que se refiere a la plantilla propuesta para la recepción de los

mensajes. Además, tanto el origen como el destino comparten una sección denominada implementación, que se refiere al código desarrollado por el programador para hacer uso de las plantillas respectivamente.

Adicionalmente, el diagrama de procesos tiene en cuenta una serie de configuraciones que nos ayudan a abordar diferentes situaciones con el mensaje. Por consiguiente, se ha realizado un diagrama de árbol que nos facilita entender, según la configuración seleccionada, cuál es el resultado.

La figura 19 es una representación modificada del diagrama de árbol para facilitar su visualización. En el anexo D podrás encontrar la versión completa del mismo.

Figura 19. Configuración de propiedades del productor



Los siguientes numerales hacen referencia a los diferentes resultados obtenidos según la configuración seleccionada en la figura 19. No obstante, no es necesario entender cada uno de los resultados para entender el diagrama de la figura 18.

1. Se publica el mensaje de forma síncrona, sin hacer uso del canal de confirmaciones, sin redireccionamiento a cola de errores activo en el método confirm y sin el método return activo con redireccionamiento a errores y sin reintento en la misma cola.
2. Se publica el mensaje de forma síncrona, haciendo uso del canal de confirmaciones, con redireccionamiento a cola de errores activo en el método confirm y con el método return activo con redireccionamiento a errores y con reintento en la misma cola.
3. Se publica el mensaje de forma síncrona, haciendo uso del canal de confirmaciones, con redireccionamiento a cola de errores activo en el método confirm y con el método return activo con redireccionamiento a cola personalizada y con reintento en la misma cola.
4. Se publica el mensaje de forma síncrona, haciendo uso del canal de confirmaciones, con redireccionamiento a cola de errores activo en el método confirm y con el método return activo sin redireccionamiento a errores y con reintento en la misma cola.
5. Se publica el mensaje de forma síncrona, haciendo uso del canal de confirmaciones, con redireccionamiento a cola de errores activo en el método confirm y con el método return activo con redireccionamiento a errores y sin reintento en la misma cola.
6. Se publica el mensaje de forma síncrona, haciendo uso del canal de confirmaciones, con redireccionamiento a cola de errores activo en el método confirm y con el método return activo con redireccionamiento a cola personalizada y sin reintento en la misma cola.
7. Se publica el mensaje de forma síncrona, haciendo uso del canal de confirmaciones, con redireccionamiento a cola de errores activo en el

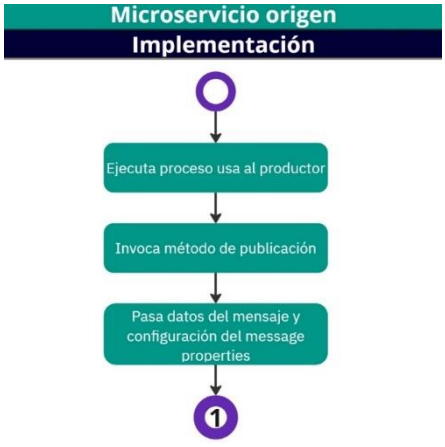
método confirm y con el método return activo sin redireccionamiento a errores y sin reintento en la misma cola.

8. Se publica el mensaje de forma síncrona, haciendo uso del canal de confirmaciones, con redireccionamiento a cola de errores activo en el método confirm y sin el método return activo con redireccionamiento a errores y sin reintento en la misma cola.
9. Se publica el mensaje de forma síncrona, haciendo uso del canal de confirmaciones, con redireccionamiento a cola de errores activo en el método confirm y sin el método return activo con redireccionamiento a cola personalizada y sin reintento en la misma cola.
10. Se publica el mensaje de forma síncrona, haciendo uso del canal de confirmaciones, con redireccionamiento a cola de errores activo en el método confirm y sin el método return activo sin redireccionamiento a errores y sin reintento en la misma cola.
11. Se publica el mensaje de forma síncrona, sin hacer uso del canal de confirmaciones, sin redireccionamiento a cola de errores activo en el método confirm y sin el método return activo con redireccionamiento a errores y sin reintento en la misma cola.
12. Se publica el mensaje de forma asíncrona, haciendo uso del canal de confirmaciones, con redireccionamiento a cola de errores activo en el método confirm y con el método return activo con redireccionamiento a errores y con reintento en la misma cola.
13. Se publica el mensaje de forma asíncrona, haciendo uso del canal de confirmaciones, con redireccionamiento a cola de errores activo en el método confirm y con el método return activo con redireccionamiento a cola personalizada y con reintento en la misma cola.
14. Se publica el mensaje de forma asíncrona, haciendo uso del canal de confirmaciones, con redireccionamiento a cola de errores activo en el método confirm y con el método return activo sin redireccionamiento a errores y con reintento en la misma cola.

15. Se publica el mensaje de forma asíncrona, haciendo uso del canal de confirmaciones, con redireccionamiento a cola de errores activo en el método confirm y con el método return activo con redireccionamiento a errores y sin reintento en la misma cola.
16. Se publica el mensaje de forma asíncrona, haciendo uso del canal de confirmaciones, con redireccionamiento a cola de errores activo en el método confirm y con el método return activo con redireccionamiento a cola personalizada y sin reintento en la misma cola.
17. Se publica el mensaje de forma asíncrona, haciendo uso del canal de confirmaciones, con redireccionamiento a cola de errores activo en el método confirm y con el método return activo sin redireccionamiento a errores y sin reintento en la misma cola.
18. Se publica el mensaje de forma asíncrona, haciendo uso del canal de confirmaciones, sin redireccionamiento a cola de errores activo en el método confirm y sin el método return activo con redireccionamiento a errores y sin reintento en la misma cola.

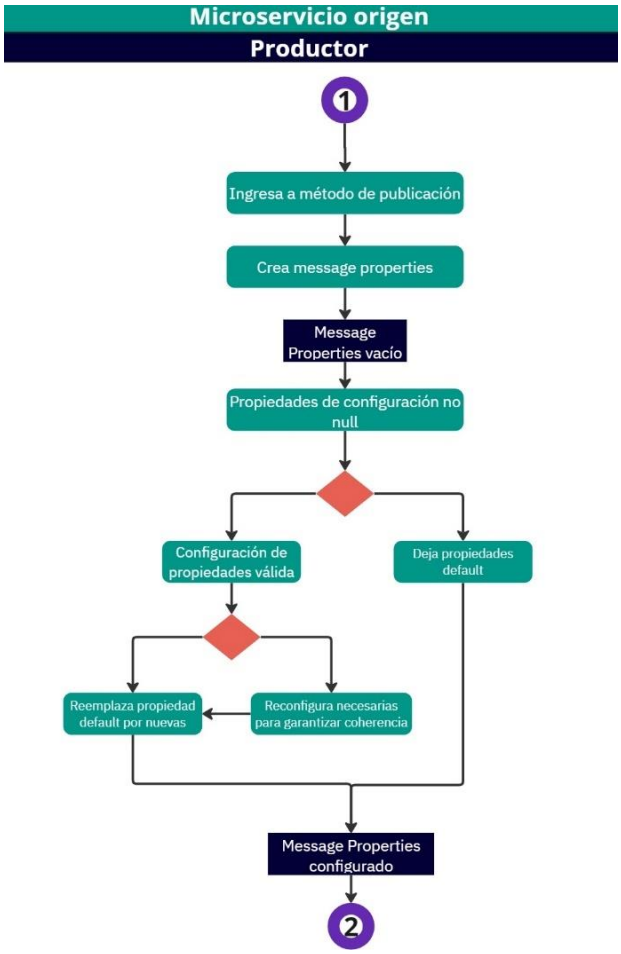
Una vez explicado lo anterior, podemos hablar del diagrama de procesos. En la figura 20 se puede observar la parte inicial en la que se hace la llamada a la plantilla propuesta para el uso del productor y, por consiguiente, la publicación del mensaje.

Figura 20 Diagrama de flujo de proceso del microservicio origen



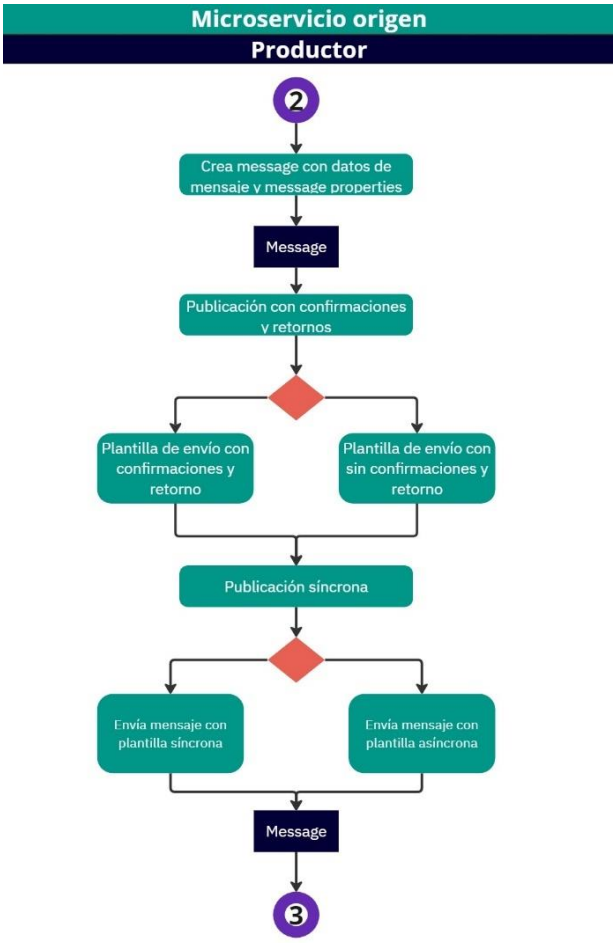
Una vez se pasan los datos del mensaje y la configuración respectiva, se puede observar en la figura 21 que se lleva a cabo todo el proceso en el que se validan las configuraciones proporcionadas y se utilizan para crear las propiedades del mensaje configuradas (messageProperties).

Figura 21. Primer diagrama de flujo de proceso del productor



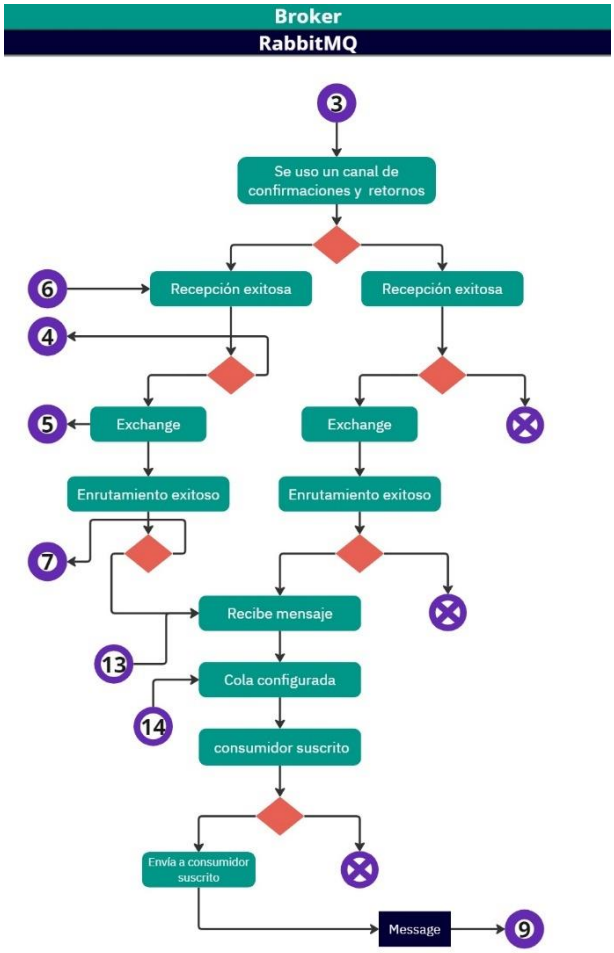
Posteriormente, en la figura 22, se observa que las propiedades del mensaje creadas (messageProperties) se utilizan junto con los datos a publicar para crear el mensaje (message). A continuación, se decide, según la configuración proporcionada, cuál plantilla de las ofrecidas por Spring AMQP usar: RabbitTemplate o AsyncRabbitTemplate.

Figura 22. Segundo diagrama de flujo de proceso del productor



A continuación, como se observa en la figura 23, esta es la parte en la que se conectan tanto el productor (en el microservicio origen) como el consumidor (en el microservicio destino) con RabbitMQ. Dependiendo del caso, RabbitMQ realiza la acción respectiva, como enviar las señales correspondientes de confirmación o retorno de mensaje al productor y redireccionar el mensaje recibido a la cola designada para su posterior consumo.

Figura 23. Primer diagrama de flujo de proceso del Broker



Antes de avanzar con la parte del proceso en la que se involucra el consumidor, terminemos de ver las partes que se desarrollan en el productor. Según las confirmaciones emitidas por RabbitMQ, en la figura 24 se visualiza el proceso tanto para las confirmaciones positivas como para las negativas, así como lo que se ejecuta según la configuración de las propiedades del mensaje (messageProperties). Por otro lado, en la figura 25 se visualiza el proceso que se lleva a cabo en el caso en que el mensaje se retorne al productor por no poder ser direccionado a una cola específica.

Figura 24. Tercer diagrama de flujo de proceso del productor

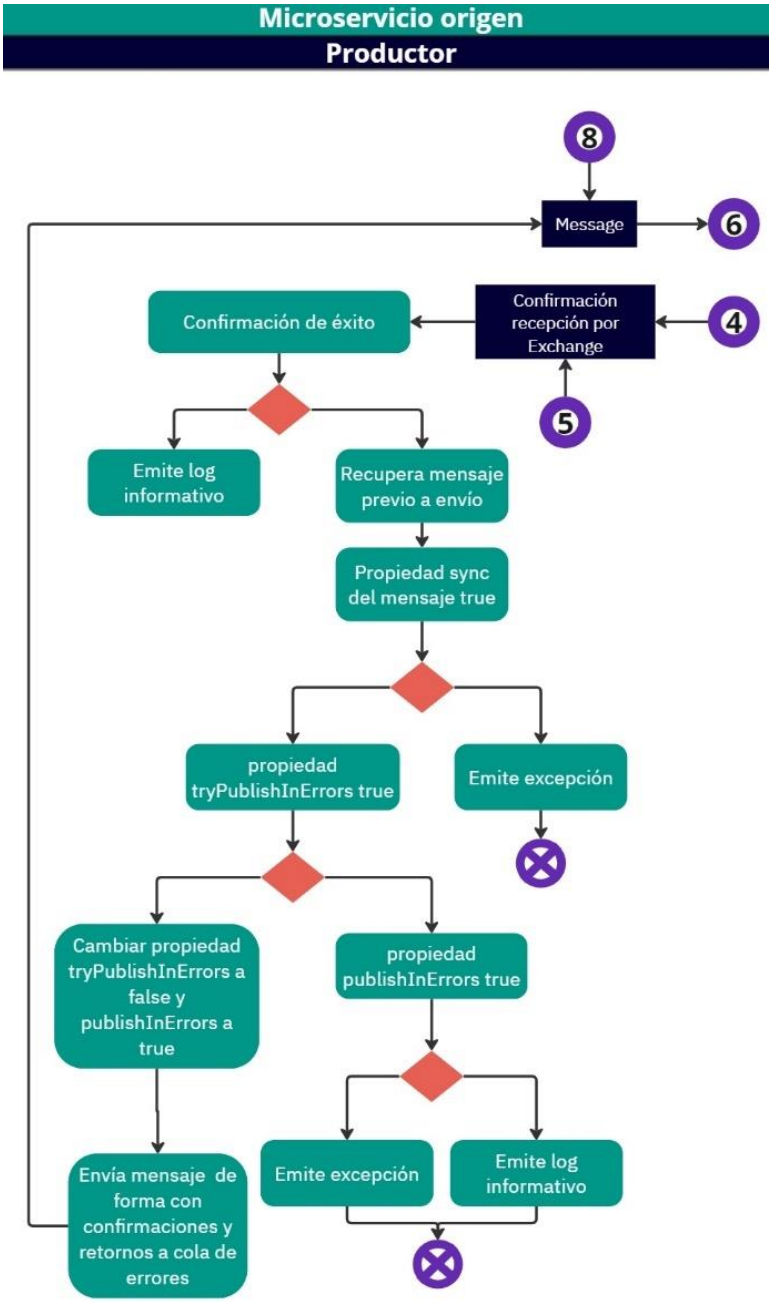
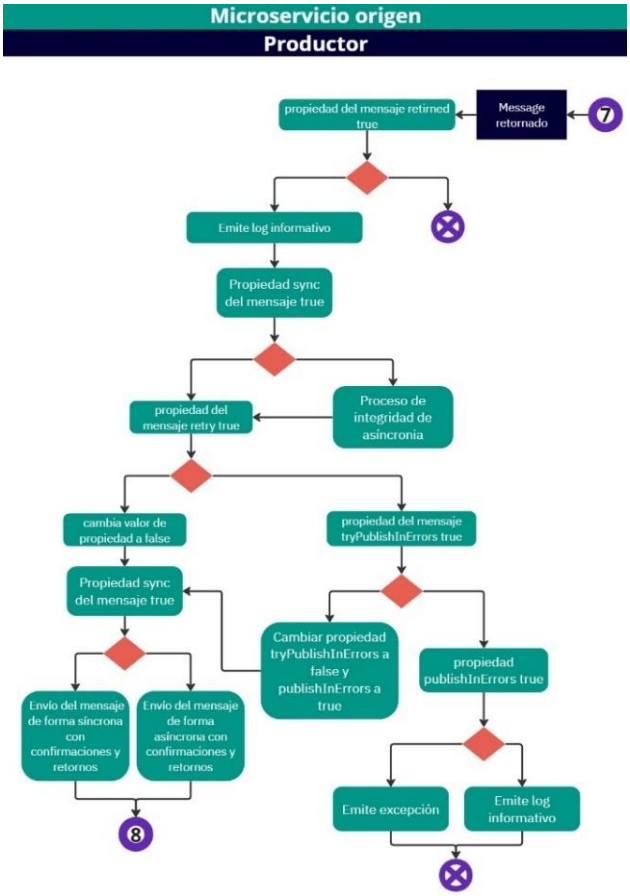
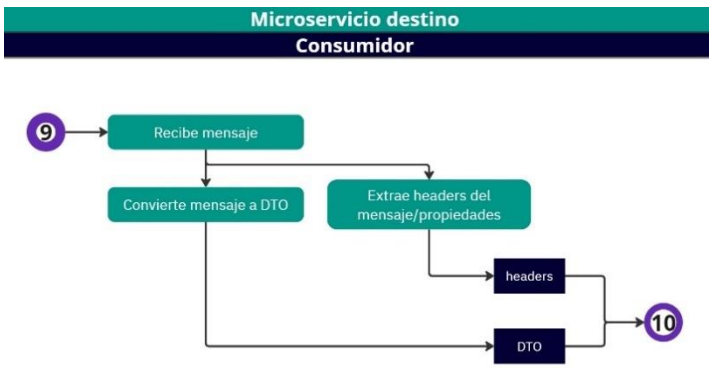


Figura 25. Cuarto diagrama de flujo de proceso del productor



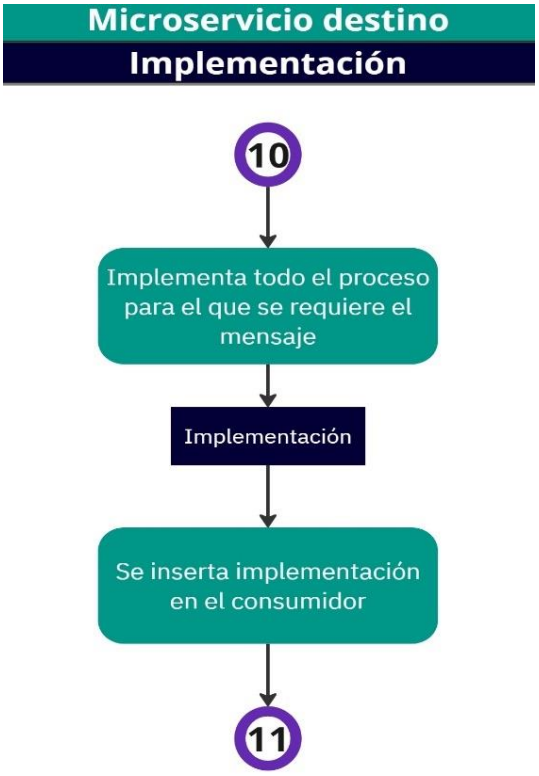
Continuando con el proceso, como se observa en la figura 26, en el microservicio destino tenemos la parte del consumidor. Este recibe el mensaje de una cola específica, extrae los headers y convierte los datos del mensaje al DTO requerido.

Figura 26. Primer diagrama de flujo de proceso del consumidor



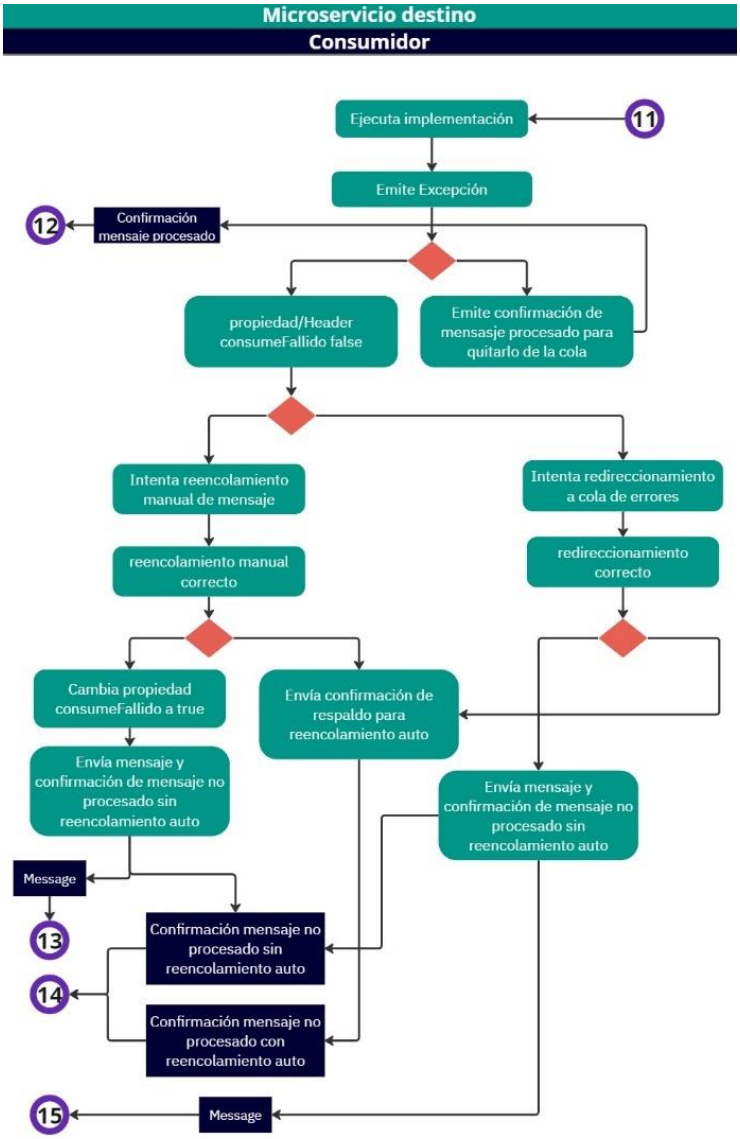
Una vez se obtiene lo necesario del mensaje consumido, como se ve en la figura 27, se introduce la implementación necesaria en la plantilla propuesta del consumidor para su respectivo manejo.

Figura 27. Diagrama de flujo de proceso del microservicio destino



En la siguiente parte, que se encuentra en la figura 28, se puede observar el manejo que se le da al mensaje consumido, ya sea que el proceso se ejecute correctamente o que ocurra algún error que requiera reencolamiento o redireccionamiento a la cola de errores.

Figura 28. Cuarto diagrama de flujo de proceso del consumidor



Para concluir, se agrega la figura 29 en la que se observa la parte del proceso, donde se elimina el mensaje de la cola emisora o se envía a la cola de errores, según sea el caso.

Figura 29. Segundo diagrama de flujo de proceso del Broker

En el anexo E se puede encontrar el código debidamente documentado del proceso anteriormente explicado, el cual consta de tres clases principales que se explicarán de forma general a continuación.

RabbitMQConfig: Esta clase se encarga de configurar las plantillas de Spring AMQP, inyectándoles la lógica para integrar sus funcionalidades ofrecidas con las modificaciones necesarias para hacerlas parte de las plantillas desarrolladas para RSI, proporcionando así la flexibilidad necesaria para manejar los diversos escenarios posibles de envío de mensajes y la persistencia de estos.

RabbitMQProducer: Junto con las plantillas ya modificadas de Spring AMQP, esta clase constituye la primera de las plantillas generalizadas para RSI. Se encarga de recibir los datos y la configuración con la cual serán enviados. Aquí se crea el mensaje y inicia el envío del mensaje.

RabbitMQConsume: Junto con las plantillas ya modificadas de Spring AMQP, esta clase constituye la segunda de las plantillas generalizadas para RSI. Se encarga de ejecutar el código inyectado por el desarrollador, el cual requiere el mensaje recibido. Además,

implementa la lógica necesaria para asegurar la persistencia del mensaje en caso de que el consumidor no lo procese correctamente.

6.3 Implementación en RSI

Una vez realizadas las plantillas y el proceso, se implementó la solución en los microservicios críticos de RSI identificados en las reuniones con el líder de Backend y especialista en gestión de proyectos. Estos microservicios incluyen el backend de Hoja de Vida, específicamente en el proceso que requiere el envío de correos de subsidios familiares, y el backend de Notificaciones, en el proceso que envía correos electrónicos. En ambos casos, se integró el código de las plantillas desarrolladas y se utilizó conforme a lo necesario.

En el anexo F, se encuentra una explicación más detallada de la implementación específica en cada uno de estos microservicios, sin profundizar demasiado, por razones de confidencialidad. Además, para complementar se incluye todo el código de las plantillas, explicado línea por línea en el anexo E.

7. EVALUACIÓN Y RESULTADOS

Para concluir este proyecto se presentan las actividades correspondientes a la evaluación de la tecnología implementada dentro en el ecosistema de RSI, así como los resultados obtenidos durante la realización de estas pruebas, de igual manera, se añade la información asociada a los procesos de documentación y difusión asociados al trabajo.

7.1 Pruebas funcionales

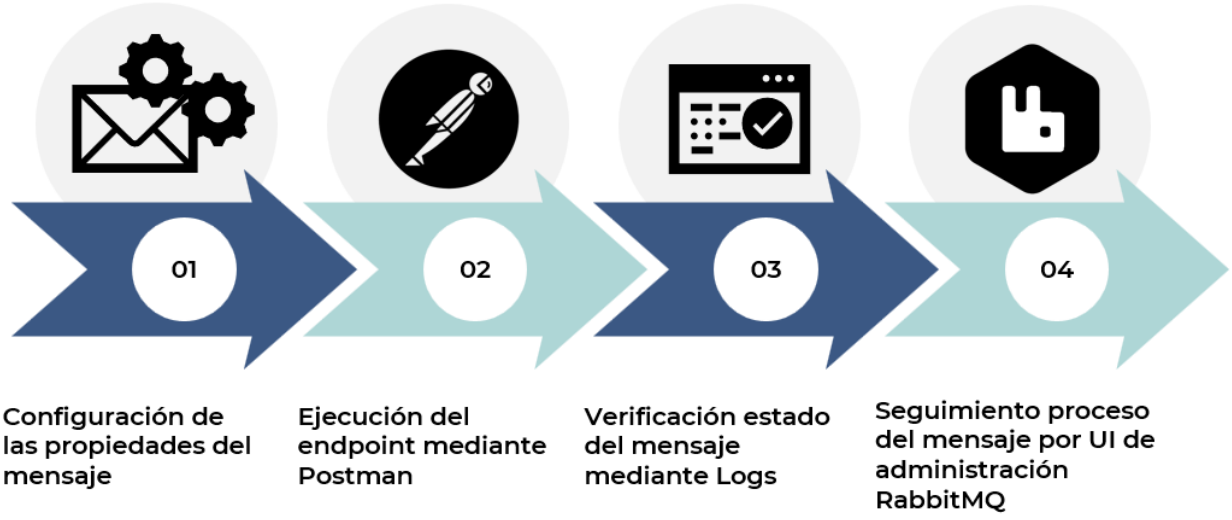
Un componente esencial para la elaboración de este proyecto es la ejecución de pruebas sobre la implementación, a través de pruebas funcionales del entorno configurado, con el fin de verificar que su funcionamiento sea coherente con el diseño previamente establecido. Este proceso de prueba es esencial para validar la integración de RabbitMQ con otros microservicios, garantizando que la comunicación entre ellos sea eficiente y fiable.

Las pruebas comienzan al ejecutar los entornos que utilizan la plantilla del consumidor implementada en el microservicio de Notificaciones y la plantilla del productor desarrollada en el microservicio de HojaDeVida, los cuales ya anteriormente fueron configurados para hacer uso del broker RabbitMQ como se explicó en la sección de implementación. Para llevar a cabo estas pruebas, se llevaron a cabo la siguiente serie de pasos, presentados también en la figura 30:

- Se ejecuta el código previamente desarrollado desde la perspectiva de un desarrollador dándole al mensaje correspondiente las propiedades acordes al escenario a revisar.
- Para simular el consumo desde un entorno real, se ejecutaron los métodos se asociados a la implementación realizada mediante la herramienta Postman.
- Mediante Logs en consola se puede revisar el estado del mensaje enviado, para confirmar el proceso de fondo realizado.

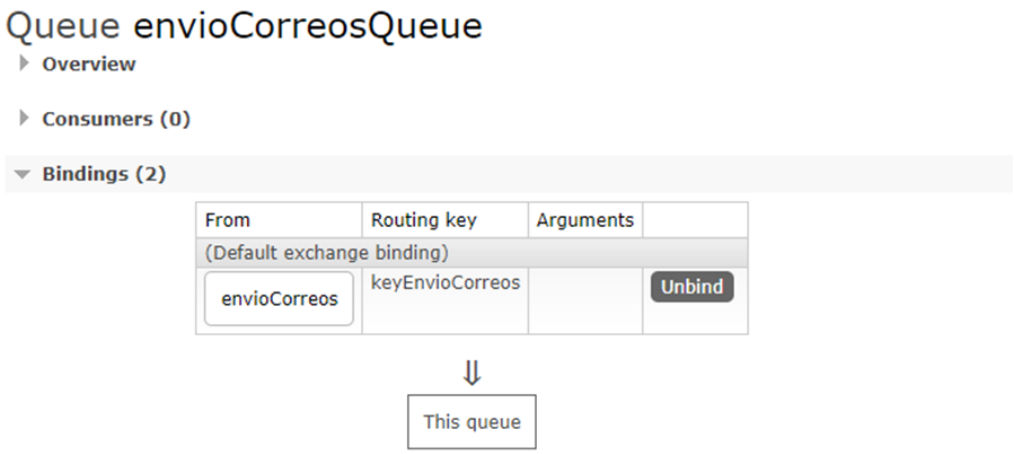
- Se realizó un seguimiento detallado de los mensajes al entrar a las colas definidas utilizando el UI de administración proporcionado por el plugin Management de RabbitMQ.

Figura 30. Proceso para pruebas funcionales



Para estas pruebas, se configuró una cola para conectar eficazmente los mensajes entre ambos microservicios, esta se encuentra configurada como se detalla en la figura 31.

Figura 31. Configuración de cola de envío de correos



Asimismo, se configuró una cola dedicada a almacenar mensajes que presentan errores en su envío, con las configuraciones especificadas en la figura 32.

Figura 32. Configuración de cola de errores

7.1.1 Definición de escenarios

Estas pruebas se realizaron en consonancia con los escenarios descritos en la sección 5.4 al plantear los casos de uso del sistema, con el objetivo de asegurar que el entorno y la gestión de errores funcionen conforme a lo planificado. Además, estas se registraron en formato de vídeo, disponible en el anexo G para su verificación. Los escenarios probados son los siguientes:

- **Escenario 1: Envío de mensajes de manera correcta:** Este se realiza cuando el mensaje enviado no presenta inconsistencias. En este escenario el mensaje debe configurarse con las propiedades que permitan el direccionamiento a la cola envioCorreosQueue como se muestra en la tabla 2.

Tabla 2. Propiedades del mensaje en envío correcto

	Valor
Exchange	envioCorreos
Rountingkey	KeyEnvioCorreos

- **Escenario 2: Envío de mensajes con consumidor desconectado:** Este representa las posibles ocasiones donde el consumidor del mensaje se encuentre caído o no esté funcionando de manera adecuada, tras esto, el mensaje se mantendrá en la cola a la que fue enviada hasta que esta pueda hacer contacto con el microservicio destino.
- **Escenario 3: Envío de mensajes con error del Exchange:** Este implica que no se encuentran colas disponibles con el Exchange digitado en las propiedades del mensaje, este error puede producirse cuando el desarrollador digita de manera incorrecta esta propiedad, cuando esto sucede, el mensaje se redirige a la cola de almacenamiento de errores. Para simular este fallo se modificaron las propiedades del mensaje como se muestra en la tabla 2.

Tabla 3. Modificación para simular fallo del Exchange

	Valor esperado	**Valor usado**
Exchange	envioCorreos	envioCorr
Rountingkey	KeyEnvioCorreos	KeyEnvioCorreos

- **Escenario 4: Envío de mensajes con error del RotingKey:** Este escenario se presenta cuando el RoutingKey digitado no corresponde a ninguno de los asociados al Exchange de la cola a enviar, por consiguiente, este mensaje se dirige a la cola de almacenamiento de errores. Para simular este fallo se modificaron las propiedades del mensaje como se muestra en la tabla 3.

Tabla 4. Modificación para generar error en el Routing Key

	Valor esperado	**Valor usado**
Exchange	envioCorreos	envioCorreos
Rountingkey	KeyEnvioCorreos	KeyEnvioCorr

- **Escenario 5: Envío de mensajes con error del consumo:** Cuando se digita un mensaje con las características adecuadas, pero con información errónea en la

petición que recibe el endpoint del microservicio consumidor, este comienza el protocolo de enrutamiento hacia la cola de almacenamiento errores.

- **Escenario 6: Envío de mensajes con error del consumo y del reencolamiento:** Como escenario final se propone que el microservicio que recibe el mensaje, además de fallar el consumo, falla el redireccionamiento a la cola de errores, en esta ocasión, este se mantendrá en la cola desde la que se recibió. Para generar este error se modificó la ruta de la cola de errores como se muestra en la tabla 4.

Tabla 5. Modificación para fallos de reenvío a errores

	Valor esperado	**Valor usado**
Exchange	defaultError	defaultErr
Rountingkey	KeyDefaultErrors	KeyDefaultErr

7.1.2 Análisis de resultados

Los resultados de las pruebas realizadas se han documentado exhaustivamente en la tabla 5.

Tabla 6. Resultado de pruebas funcionales

Escenario evaluado	**Resultado**
Envío de mensajes de manera correcta	✓
Envío de mensajes con consumidor desconectado	✓
Envío de mensajes con error del Exchange	✓
Envío de mensajes con error del RotingKey	✓
Envío de mensajes con error del consumo	✓
Envío de mensajes con error del consumo y del reencolamiento	✓

Tal como se evidencia en la tabla 5, las pruebas realizadas confirman un funcionamiento exitoso y coherente con la propuesta realizada. Con respecto a los escenarios individuales anteriormente planteados es posible realizar las siguientes conclusiones.

El primer escenario demuestra el funcionamiento correcto del sistema al enviar mensajes que serán recibidos de manera adecuada por el consumidor, la probación de esta prueba demuestra que el sistema establecido es capaz de cumplir con el objetivo inicial de transferir información de peticiones entre servicios de manera segura y capaz.

Además, gracias al segundo caso es posible identificar la permanencia del mensaje demostrando que el sistema es capaz de resistir de manera exitosa a fallos relacionados a caídas del servidor, cumpliendo con los requerimientos de tolerancia a fallos propuestos de manera satisfactoria.

Con respecto a los casos tercero y cuarto, basados en el manejo de errores del productor, el sistema mostró un adecuado manejo de manera óptima y mediante el seguimiento de los mensajes enviados se verificó el cumplimiento del trayecto planeado desde el diseño de la propuesta.

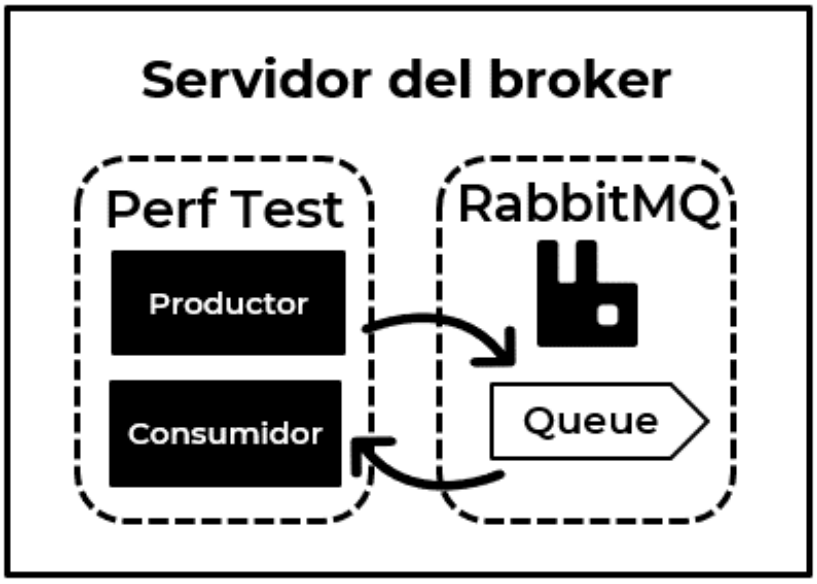
Finalmente, de acuerdo con los casos cinco y seis, los cuales están basados en tratar problemas relacionados al consumidor, se puede afirmar el correcto manejo por parte de los dos microservicios asociados a la comunicación, permitiendo el almacenamiento de errores exitosamente en casos de fallas desde el consumidor y confirmando una vez más el correcto funcionamiento del sistema implementado.

7.2 Pruebas de rendimiento

Como parte del desarrollo del proyecto se plantea la realización de una serie de pruebas de rendimiento sobre el broker elegido para la implementación de la propuesta. El uso de un broker de mensajería eficiente es crucial para asegurar la operación fluida y confiable de los servicios de microservicios en el sistema RSI, donde la coordinación y el intercambio de información entre diferentes componentes es fundamental. Esta evaluación tiene como objetivo demostrar la capacidad de RabbitMQ para manejar diferentes cargas de trabajo y escenarios de concurrencia, y así validar su idoneidad para su integración en el entorno universitario.

Estas pruebas fueron realizadas mediante el uso de una herramienta proporcionada por una extensión de RabbitMQ conocida como “Perf Test”, una herramienta diseñada para evaluar y medir diversas métricas de rendimiento del sistema que simula cargas de trabajo personalizables, este mecanismo permite evaluar la escalabilidad, estabilidad y eficiencia del sistema bajo condiciones cercanas a las de producción. Además, brinda información sobre la latencia, este término alude al intervalo necesario para completar el ciclo de un mensaje desde su envío hasta su entrega. Esta herramienta se ejecuta desde el mismo servidor que contiene al broker de mensajería y crea en la misma máquina entidades lógicas encargadas de enviar o recibir mensajes de manera automática, este mecanismo se representa mediante a continuación la figura 33.

Figura 33. Funcionamiento interno Perf Test



7.2.1 Definición de escenarios

Las pruebas realizadas representan el comportamiento que tiene el broker con las capacidades asociadas al servidor que está usando para su implementación y estos

valores varían según se aumenten o disminuyan, para la implementación de RabbitMQ. El servidor usado cuenta con las características especificadas en la tabla 6.

Tabla 7. Características del servidor

Descripción	Métrica
IP	10.0.44.24
CPU	2 Core
RAM	4 GB
DISK	64 GB

Para la evaluación adecuada de las métricas se revisaron en total tres escenarios, en los cuales se varía el número de productores, consumidores y el número de colas. Estos factores son críticos para simular situaciones del mundo real, donde se espera que el sistema RSI maneje grandes volúmenes de datos y múltiples usuarios simultáneos.

Finalmente, para mantener las mismas condiciones para cada una de las pruebas, se realizó un total de 30 réplicas para cada escenario a evaluar y se tomó el promedio de todos los intentos realizados, de igual manera, se configuró la herramienta para que los mensajes fueran enviados sin límite de durante 30 segundos para cada uno de los escenarios propuestos. En la tabla 8 se presenta cada uno de los casos evaluados.

Tabla 8. Escenarios pruebas de rendimiento

Escenario	Descripción	Tiempo	Réplicas	Tamaño mensajes
1	Variación de productores y consumidores en única cola	30 S	30	1MB
2	Variación de colas para un productor y un consumidor	30 S	30	1MB
3	Variación productores y consumidores en múltiples colas	30 S	30	1MB

Escenario 1: Variación de productores y consumidores en una cola:

Para este primer escenario se realizó una serie de simulaciones de envío de mensajes a una única cola, esta configuración pretende definir el desempeño de una única cola cuando esta se somete a requerimientos extremos. En estas simulaciones, se varió el número de productores y consumidores desde 1 hasta 50 para revisar la latencia en este tipo de pruebas. Los datos recopilados son los siguientes:

Tabla 9. Variación de productores y consumidores en única cola

Número de Colas: 1

Número productores	Número consumidores	Sending Rate [msg/s]	Receiving Rate [msg/s]	Consumer Latency				
				1th	50th	75th	95th	99th
1	**1**	678,36	677,1	7754,67	99420,27	143766,7	243476,9	305509,2
10	**10**	727,83	729,23	15039,4	164889,5	240849,7	404750,9	597545,4
50	**50**	446,9	453,26	72571,57	508047,5	714555,6	1153285	1581871
Promedio		**617,7**	**619,86**	**31788,54**	**257452,42**	**366390,66**	**600504,26**	**828308,53**

Escenario 2: Variación de colas para un productor y un consumidor:

Como siguiente escenario múltiples colas operan simultáneamente, cada una con un número reducido de productores y consumidores. Los estudios se llevaron a cabo bajo condiciones idénticas a las del escenario anterior. Los datos obtenidos se presentan en la siguiente tabla:

Tabla 10. Variación de colas con un productor y un consumidor

Número de productores: 1			**Número de Consumidores: 1**				
Número de colas	**Sending Rate [msg/s]**	**Receiving Rate [msg/s]**	Consumer Latency				
			1th	**50th**	**75th**	**95th**	**99th**
1	678,36	677,1	7754,66	99420,26	143766,7	243476,86	305509,2
10	689,066	690,66	36620	234765,9	316546,07	564840,33	882733,66
50	337,4	352,26	42079,26	853351,23	1353916,5	2897982,77	4204868,7
Promedio	**568,27**	**573,34**	**28817,97**	**395845,8**	**604743,08**	**1235433,32**	**1797703,86**

Escenario 3: Variación de productores y consumidores en múltiples colas:

Finalmente, se analiza el escenario final del estudio de rendimiento, el cual corresponde al escenario más probable en entornos de producción. En este escenario, se mantienenen siempre un número multiple de colas dentro del sistema de estudio, siendo 10 para esta prueba, las cuales presentan variaciones en la cantidad de productores que le envían mensajes y consumidores que se conectan a ellas para recibirlos. En esta ocasión, la cantidad de productores o consumidores varió únicamente desde 1 hasta 20 por restricciones del servidor.

Figura 34. Variación de productores y consumidores en múltiples colas

Número de Colas: 10

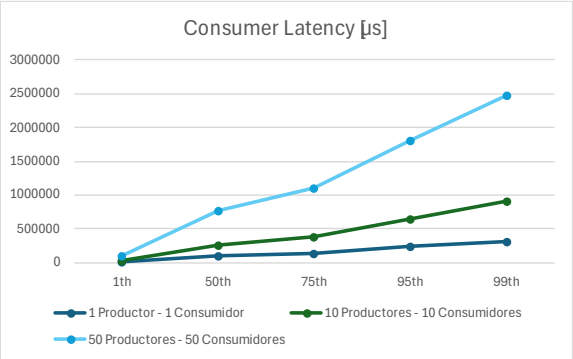
Productores	Consumidore	Sending Rate [msg/s]	Receiving Rate [msg/s]	Consumer Latency				
				1th	50th	75th	95th	99th
1	**1**	689,06	690,66	36620	234765,9	316546,06	564840,33	882733,67
10	**10**	291,8	298,67	216849,33	1452986,9	2101794,1	3292368,27	4256208,97
20	**20**	188,6	194,53	294984,36	2805522,2	3746458,97	5229726,33	7034538,23
Promedio		**389,82**	**394,62**	**182817,89**	**1497758,34**	**2054933,04**	**3028978,31**	**4057826,96**

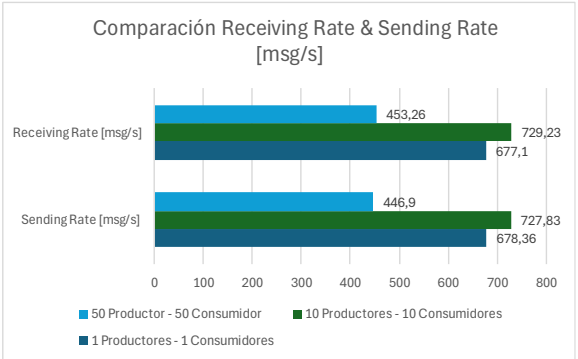
7.2.1 Análisis de resultados

Tras la realización de estas pruebas y la tabulación de resultados, se graficaron estos valores y se realizó un análisis con el fin de identificar el comportamiento del broker implementado en situaciones de extrema carga, esto a su vez permite encontrar límites adecuados de capacidad de uso con las condiciones actuales del servidor.

Escenario 1: Mediante la figura 35 se recompilan los datos obtenidos para el primer caso.

Figura 35. Comparativa variación de productores y consumidores en una cola





Para el primer caso asociado se encontró una variación de la latencia menor en los casos donde se emplean uno y diez productores y consumidores, teniendo estos un valor de latencia relativamente bajo y demostrando que las colas con pocas cargas funcionan de manera idónea, esto sin embargo cambia cuando llega a tener 50 productores y consumidores, estos hallazgos indican que el sistema puede volverse cada vez más ineficiente a medida que se intensifica la demanda sobre él haciendo de este caso uno con una alta tasa de demora, la saturación de información durante el proceso de consumo inevitablemente genera demoras en el sistema, haciendo de este sin duda un caso poco práctico y nada recomendable para el sistema.

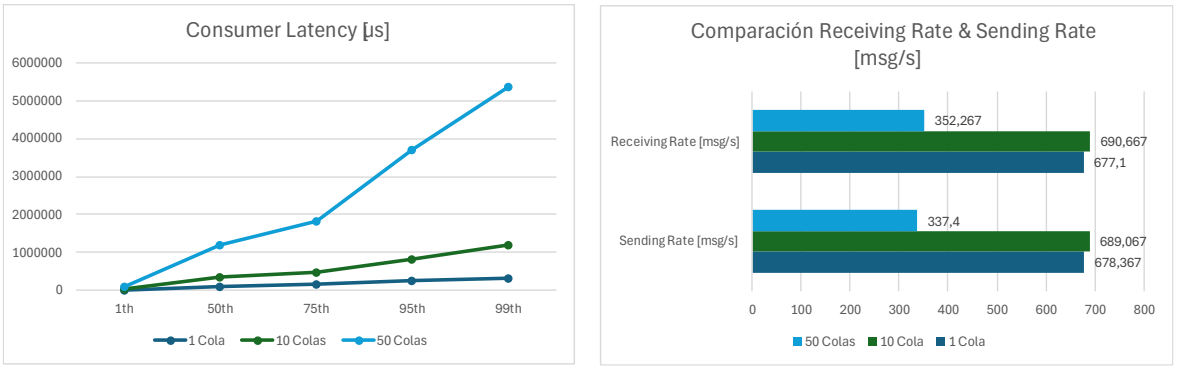
Como se observa en la figura 35, la tasa de envío y consumo del broker se aumentó cuando el escenario pasó de llevar uno a diez productores y consumidores, esto indica un rendimiento sobresaliente del broker en la gestión de mensajes, asegurando un manejo adecuado de las capacidades de envío en cada cola permitiendo que sea capaz de recibir y procesar de manera adecuada las peticiones recibidas cuando la cola cuenta con un número aceptable de entidades conectadas a ella. Sin embargo, al hacer que la cola esté escuchando a un número elevado de consumidores y productores este rendimiento decrece llegando a mandar y procesar un número de mensajes muy bajo a comparación de las dos primeras ocasiones.
Como se ha mencionado anteriormente, aunque una sola cola puede manejar volúmenes elevados de carga, la centralización de todas las conexiones en una única cola limita el rendimiento del sistema a medida que aumenta el número de usuarios asociadas a ella. En el escenario analizado, se demuestra que una sola cola, al procesar diversas solicitudes, tiende a experimentar una disminución progresiva en su rendimiento, lo que con el tiempo hace al sistema insostenible.

Al encontrar el equilibrio adecuado de usuarios en cola, se concluye que es fundamental no superar la cantidad óptima de productores y consumidores. Aunque esta situación es improbable debido a la naturaleza del sistema implementado, se afirma la importancia de

mantener un balance adecuado de la carga y la necesidad de una gestión adecuada de las solicitudes.

Escenario 2: Los datos obtenidos para este escenario fueron representados en la figura 36, encontrada a continuación.

Figura 36. Comparativa variación de colas para un productor y un consumidor



Se observa nuevamente un comportamiento similar al mostrado anteriormente en cuanto a la tasa de envío y recepción de mensajes en los dos primeros escenarios, donde el sistema está configurado con una y diez colas respectivamente, donde a pesar de tener un aumento notable sobre los valores de los percentiles en comparación con al escenario anterior, estos cuentan con una latencia relativamente baja. Contrario a cuando se alcanza el escenario con 50 colas operando simultáneamente en el sistema donde se registra una caída considerable en el rendimiento incluso mayor a la presentada en el escenario anterior.

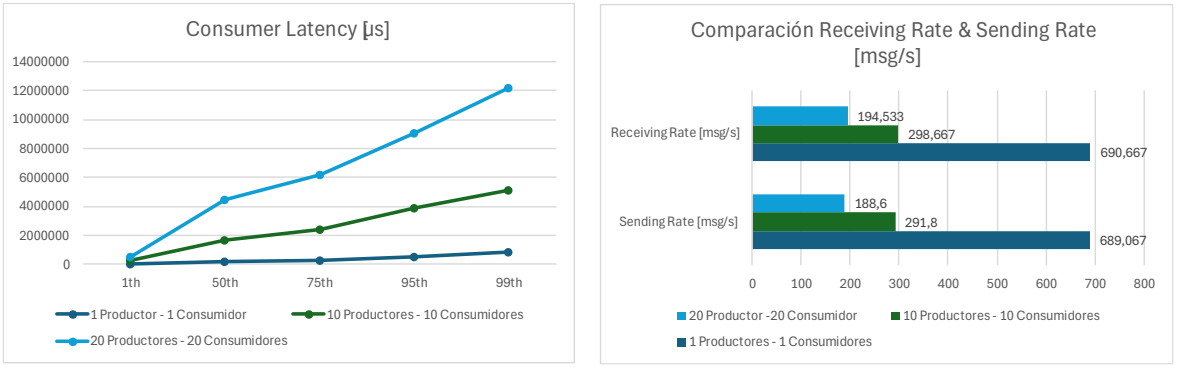
De igual manera, se puede apreciar un decrecimiento de cantidad de envío y procesamiento de mensajes en 50 colas, atribuible principalmente a limitaciones del servidor que hospeda el broker de RabbitMQ. La implementación de RabbitMQ presenta como una de sus principales restricciones los recursos asignados, y se anticipa que aumentar estos recursos mejorará el rendimiento del sistema.

En este segundo escenario, el sistema presenta ciertas diferencias significativas en términos de rendimiento al anteriormente estudiado. Lo primero que se observa es el notable incremento en la tasa de envío de solicitudes que el broker debe manejar, debido al aumento en los canales de comunicación entre productores y sus respectivos consumidores. Inicialmente, esto mejora el rendimiento del sistema; sin embargo, dicho rendimiento disminuye a medida que aumenta el número de usuarios de las colas. Esto provoca un mayor deterioro del rendimiento en comparación con el escenario anterior, evidenciando cómo el número excesivo de colas puede saturar la memoria del sistema, volviéndolo ineficiente y no sostenible a largo plazo. Asimismo, la latencia aumenta exponencialmente con el incremento en el número de mensajes en situaciones con múltiples colas, lo que reduce la eficacia del sistema.

Como se menciona en el análisis, estos problemas no son atribuibles al broker, sino que son consecuencia del servidor utilizado para ejecutarlo. Este escenario ilustra cómo los recursos del servidor pueden afectar significativamente el rendimiento del sistema, generando eventualmente su insostenibilidad.

Escenario 3: Finalmente, se analiza el escenario final del estudio de rendimiento, el cual corresponde al escenario más probable en entornos de producción, donde se encuentran un número de colas adecuado con variaciones de productores y consumidores, la información de este escenario fue representado en la figura 37.

Figura 37. Comparativa variación de productores y consumidores en múltiples colas



Tal como se aprecia en la figura 37, la latencia obtenida tiende a mantenerse más constante, sin presentar aumentos significativos en comparación con los obtenidos en estudios previos. Sin embargo, gracias a los valores obtenido en esta, se observa como el servidor tuvo cierta dificultad al procesar tal cantidad de entidades. De esta manera, se puede demostrar que, tras definir un margen de circunstancias adecuadas, el broker implementado en el sistema presenta un rendimiento mejor distribuido para escenarios de alta demanda, pero con demoras significativas de rendimiento.

Mientras tanto la tasa de envío y recibimiento de mensajes se mantiene decreciendo entre mayor sea la cantidad de productores y consumidores, esto es obvio debido a la cantidad de entidades que tiene que procesar cada una de las colas.

En el último escenario analizado, se observa como a pesar de tener un mejor balanceo de carga el principal problema del entorno implementado se encuentra en las características del servidor, el cual con mejoras puede volverse mucho más ágil. Sin embargo, la latencia del sistema presenta las menores variaciones entre todos los escenarios, permitiendo demostrar un comportamiento más predecible y controlable, incluso cuando el broker fue sometido a la mayor cantidad de consumidores adecuada al sistema.

7.3 Documentación y difusión del proyecto

Como parte del planteamiento inicialmente establecido en el desarrollo de la metodología y para asegurar el cumplimiento satisfactorio del proyecto, se realiza finalmente la etapa de entrega de resultados, constituida por la documentación y difusión de la propuesta desarrollada e implementada en el entorno de RSI. Esta incluye la elaboración de documentación detallada de la implementación, disponible en la wiki de RSI, donde se encuentra definida la guía de uso e implementación de la tecnología al alcance de los desarrolladores del proyecto interesados en informarse a fondo de esta tecnología. La documentación de la tecnología en la wiki puede ser accedida mediante el link “https://wikirsi.uis.edu.co/en/Arquitectura/Backend/RabbitMQ” y haciendo uso de claves proporcionadas por los directores del proyecto RSI.

Además, se llevó a cabo una actividad de divulgación, comenzando con una presentación a los directivos, diseñada para proporcionar una visión general estratégica de las mejoras y beneficios que el proyecto aporta a la organización. Posteriormente, se realizó una capacitación para los desarrolladores, centrada en aspectos técnicos y prácticos, que no solo abarcó lo realizado, sino que también ofreció información sobre el uso y funcionamiento del broker de mensajería RabbitMQ.

El propósito de esta fase es asegurar la comprensión y adopción de las herramientas y conocimientos adquiridos durante el proyecto. Al fomentar un entorno de aprendizaje continuo y compartido, se garantiza que todos los miembros de la comunidad RSI puedan beneficiarse de las mejoras implementadas, aplicarlas eficazmente en sus propios proyectos y contribuir al desarrollo continuo de la organización.

8. CONCLUSIONES

En primer lugar, se evidencia que RSI, al basarse en microservicios, enfrenta las problemáticas inherentes a mantener esta arquitectura distribuida. Estas dificultades fueron abordadas eficazmente mediante el uso de tecnologías de comunicación asíncrona, como RabbitMQ. La elección de este broker de mensajería se fundamentó en el análisis de los requisitos específicos del proyecto, revelando que la selección de tecnologías en este campo puede variar dependiendo de los objetivos particulares a alcanzar.

Además, se concluye que gracias al uso de frameworks modernos como Spring Boot y sus librerías asociadas, como Spring AMQP, la implementación y comprensión de tecnologías complejas como RabbitMQ se facilita considerablemente. Estas herramientas no solo representan un aporte técnico significativo, sino que también sirven como fuente de inspiración para adaptar soluciones existentes a las necesidades específicas de RSI.

Por otro lado, las pruebas realizadas sobre las capacidades y la implementación de RabbitMQ han demostrado la fiabilidad del broker en la resolución de los problemas identificados en RSI. Esto asegura que, no solo cumple con los requisitos funcionales, sino que también con la asignación de los recursos adecuados proporciona una base sólida para la escalabilidad y el mantenimiento del sistema de mensajería en la infraestructura de RSI.

Para finalizar, la integración de RabbitMQ en el ecosistema de microservicios de RSI ha resultado en mejoras significativas en la gestión de comunicaciones asíncronas y en la resolución efectiva de desafíos técnicos, validando su papel como una herramienta fundamental para el desarrollo y la operación de sistemas distribuidos complejos.

9. TRABAJO FUTURO

Si bien el propósito del proyecto limita un poco el alcance abordado y desarrollado, se han tenido en cuenta múltiples conceptos y funcionalidades al plantear la propuesta y realizar la implementación. Esto se hizo con el objetivo de facilitar y asegurar no solo los objetivos planteados, sino también de entregar una base sostenible y mejorable para implementar y aprovechar varias de las virtudes del broker de mensajería RabbitMQ.

Entre estas funcionalidades se encuentra el plugin Shovel, que permite mover mensajes entre colas utilizando patrones. Esto puede ser un gran aporte no solo para mover mensajes de colas de errores a su cola original una vez revisado y resuelto el fallo que generó su redireccionamiento, sino también para moverlos según lo requiera el administrador. Además, se utilizó la plantilla AsyncRabbitTemplate de la librería Spring AMQP, considerando la posibilidad de implementar en el futuro comunicaciones RPC (Remote Procedure Call), que ofrece un comportamiento similar al de solicitud-respuesta, pero con ventajas adicionales como la utilización eficiente de las colas de RabbitMQ.

Adicionalmente, es relevante destacar que los microservicios del proyecto RSI están utilizando Java 11 al momento de la realización de este proyecto. Por lo tanto, la implementación se desarrolló en esta versión. No obstante, al diseñar la plantilla general se consideró la posibilidad de en un futuro actualizar versiones, así como también el encapsulamiento en una biblioteca, con el fin de facilitar tanto el proceso de mantenimiento y mejora de versiones, como la facilidad y rapidez de consumo del código a través de dependencias según sea necesario.

BIBLIOGRAFÍA

AL-MASRI, Eyhab, *et al*. Investigating Messaging Protocols for the Internet of Things (IoT). *IEEE Access* [en línea]. 2020 junio, 8(2020). p. 94911.

AMAZON WEB SERVICES. [¿Qué son los mensajes de publicación y suscripción?] España: AWS. [Consulta: 20 de diciembre 2023]. Disponible en: https://aws.amazon.com/es/what-is/pub-sub-messaging/.

AMAZON WEB SERVICES. [Benefits of Message Queues] United States of America: AWS. [Consulta: 20 de diciembre 2023]. Disponible en: https://aws.amazon.com/message-queue/benefits/

ARAUJO. Estrategias didácticas para la enseñanza para la enseñanza en entornos Virtuales, Citado por LAY, Nelson, *et al*. Uso de las herramientas de comunicación asincrónicas y sincrónicas en la banca privada del municipio Maracaibo. En: *Revista ESPACIOS*. 2019, nro 4, vol 40, 11 p. ISSN N 0798 1015

ARBOLEDA COLA, Carlos Augusto. Propuesta metodológica para migración de sistemas web con arquitectura monolítica hacia una arquitectura basada en microservicios. Trabajo de grado Ingeniero en sistemas informáticos y de computación. Quito: Escuela Politécnica Nacional. Facultad de Ingeniería de sistemas, 2017. 127 p.

BONHOMME, Carlos Augusto y CAMEJO, Enrique. Plataforma de Integración basada en Microservicios. Trabajo de grado Ingeniero en Computación. Montevideo: Universidad de la República. Facultad de Ingeniería, 2019. 10 p.

DIJKSTRA, Edsger. The Structure of the The Multiprogramming System. Citado por ESPINAL, Yanet. *Arquitectura de software. Arquitectura orientada a servicios*. 2012, nro. 5, p.1. ISSN-e 2306-2495

ERL, Thomas. Service-Oriented Architecture: Concepts, Technology and Design. United States of America: Prentice Hall, 2005. 792 p. ISBN: 0131858580

INTERNATIONAL BUSINESS MACHINES CORPORATION. [What is a message broker?] United States of America: IBM. [Consulta: 20 de diciembre 2023]. Disponible en: https://www.ibm.com/topics/message-brokers

INTERNATIONAL CONFERENCE ON TELECOMMUNICATION, POWER ANALYSIS AND COMPUTING TECHNIQUES. (173, 2017 abril, 8: Chennai, India). A survey on MQTT: A protocolo of Internet of Things (IoT). Bharath Institute of Higher Education and Research, 2017, 6 p.

MICROSOFT DEVELOPER DIVISION. [.NET Microservices: Architecture for Containerized .NET Applications] United States of America: Microsoft Corporation. [Consulta: 22 de diciembre 2023]. Disponible en: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/

MICROSOFT DEVELOPER DIVISION. [Comunicación en una arquitectura de microservicio] United States of America: Microsoft Corporation. [Consulta: 22 de diciembre 2023]. Disponible en: https://learn.microsoft.com/es-es/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture

NAIK, Nitin. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. *IEEE International Systems Engineering Symposium (ISSE)* [en línea]. 2017 octubre. 7 p.

NEWMAN, Sam. Monolith to Microservices. Evolutionary Patterns to Transform Your Monolith. 2 ed. United States of America: O'Reilly, 2019. 270 p. ISBN 9781492047841

PERRY, and WOLF. Worldwide Institute of Software Architects, ESPINAL. Citado por Yanet. *Arquitectura de software. Arquitectura orientada a servicios*. 2012, nro. 5, 10 p. ISSN-e 2306-2495

ROY, Gabin. RabbitMQ in Deep. Nueva York: Manning Publications, 2017. 264 p. ISBN 9781638353225

SHAW, Mary. and GARLAN, David. An Introduction to Software Architecture, Citado por VIGIL, Yamila. *La arquitectura de software como disciplina científica*. 2010, nro 3, 4 p. ISSN-e 2306-2495

SÖYLEMEZ, Mehmwt; TEKINERDOGAN, Bedir and KOLUKISA, Tarhan. Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review. *applied sciences* [en línea]. 2022, mayo, 12(11). 40 p.

VILAJOSANA, Xavier and NAVARRO, Leandro. Arquitectura de aplicaciones web. Barcelona, España: Universitat Oberta de Catalunya, 2001. 46 p. PID_00184783

VIVEIROS. Desarrollo de tecnologías y tecnologías para el desarrollo, Citado por LAY, Nelson, *et al*. Uso de las herramientas de comunicación asincrónicas y sincrónicas en la banca privada del municipio Maracaibo. En: *Revista ESPACIOS*. 2019, nro 4, vol 40, 11 p. ISSN N 0798 1015

VMWARE INC. [What is a message broker?] United States of America: VMware. [Consulta: 20 de diciembre 2023]. Disponible en: https://www.vmware.com/topics/glossary/content/message-brokers.html