ANÁLISIS DE RENDIMIENTO DE LOS PATRONES DE INTEGRACIÓN EVENT MESSAGE Y COMMAND MESSAGE EN SISTEMAS DISTRIBUIDOS

PERFORMANCE ANALYSIS OF INTEGRATION PATTERNS EVENT MESSAGE AND COMMAND MESSAGE IN DISTRIBUTED SYSTEMS

JUAN FELIPE GOMEZ VELEZ

Proyecto de grado

Asesor, docente

Paola Andrea Vallejo Correa

Daniel Correa Botero

UNIVERSIDAD EAFIT

ESCUELA DE INGENIERÍAS

MAESTRÍA EN INGENIERÍA

MEDELLÍN

2023

CONTENIDO

RESUMEN

Un sistema distribuido se compone de un conjunto de computadores autónomos interconectados. En estos sistemas, es común utilizar patrones de integración para facilitar la comunicación entre los computadores de forma asíncrona. Entre los patrones de integración más utilizados se encuentran *Event Message y Command Message*. En este estudio analizamos el rendimiento de ambos patrones, considerando el rendimiento como el tiempo que tarda llevar a cabo un proceso de transmisión de mensajes, el cual puede variar en función de la cantidad y el tamaño de los mensajes. Para realizar el análisis, se implementaron los patrones de integración anteriormente mencionados en dos sistemas distintos. Además, se realiza una comparación por cada patrón teniendo en cuenta los siguientes parámetros: tamaño del mensaje, configuraciones del bróker de mensajería, tiempo entre el envío y la recepción de un mensaje y nombre del tipo de prueba. De esta manera, se pueden comprender mejor las configuraciones y el comportamiento de los patrones de integración que intervienen en el flujo de intercambio de mensajes, así como detectar y prevenir problemas relacionados al rendimiento.

Palabras clave: Patrón de integración, rendimiento, arquitectura cliente-servidor, sistema distribuido, bróker de mensajería.

INTRODUCCIÓN

Desarrollar sistemas de software es un proceso complejo (Casas Lizcano & López, 2015) que puede involucrar integraciones entre sistemas, las cuales pueden incrementar la complejidad del sistema global. Aunque un sistema de software puede ser tan simple como un archivo que contenga toda la información necesaria para ejecutarse, en la actualidad, muchos sistemas complejos se diseñan como sistemas distribuidos compuestos por múltiples subsistemas interconectados (Autom et al., 2020). Un sistema distribuido puede definirse como un conjunto de computadores autónomos interconectados (Jiménez García et al., 2017), representados por un estilo arquitectónico como lo pueden ser cliente-servidor, *peer-to-peer*, entre otros.

El desarrollo de software en la actualidad implica la integración de diferentes sistemas, dichos sistemas por lo general se encuentran desarrollados como sistemas distribuidos, entre los cuales, por ejemplo, encontramos el sistema de base de datos, un sistema front-end y un sistema back-end. En algunos casos el back-end se divide en microservicios, lo cual implica que estos sistemas deban comunicarse entre ellos. Para facilitar la integración de estos sistemas distribuidos, existen patrones de integración que pueden ser seleccionados y configurados según la necesidad e intención de comunicación de los mensajes.

Aunque diferentes autores han trabajado en la definición de patrones que facilitan la integración de sistemas distribuidos, nos hemos basado en el libro *Enterprise Integration Patterns* (Hohpe & Woolf, 2003), el cual define seis grandes grupos de patrones, que dependiendo de sus responsabilidades se agrupan en *Message Construction*, *Messaging Endpoints*, *Message Routing*, *Messaging Channels*, *Message Transformation* y *System Management*. Todos estos tienen su participación en puntos concretos en el intercambio de mensajes entre dos sistemas. La literatura nos propone múltiples patrones como por ejemplo *Event Message*, *Command Message*, *Request-Reply*, *Message Sequence*, y *Document Message*. La selección y configuración de un patrón de integración en un

proyecto de software es crucial, ya que impacta en el rendimiento del sistema y en los factores que puedan intervenir en el proceso de intercambio de mensaje.

En este estudio implementaremos y analizaremos el rendimiento de los patrones de integración *Event Message* y *Command Message*, a través de la construcción de dos sistemas y la realización de pruebas de envío de mensajes concurrentes para observar su comportamiento.

La sección 2 del documento presenta los conceptos clave relevantes para el desarrollo de este estudio. La sección 3 describe las etapas de la metodología utilizada en este estudio. La sección 4 detalla el estudio realizado. Finalmente, la sección 5 presenta las conclusiones.

CONCEPTOS CLAVE

A continuación, definiremos varios conceptos clave que son importantes para la comprensión de este estudio.

PROTOCOLO

Un protocolo es “un conjunto de reglas que gobiernan la interacción de procesos concurrentes en sistemas distribuidos, este es utilizado en un gran número de campos como sistemas operativos, redes de computadoras o comunicación de datos” (Casas Lizcano & López, 2015). Algunos ejemplos de protocolos son: TCP/IP, HTTP, MQTT, AMQP.

BRÓKER DE MENSAJERÍA

Un bróker de mensajería es un sistema de intercambio de mensajes, que “presenta formas de enrutar mensajes al destino adecuado sin que la aplicación de origen sepa cuál es el destino final del mensaje” (Hohpe & Woolf, 2003). Los brókers de mensajería proporcionan clasificación de datos, ruteo, traducción de mensajes, persistencia y entrega.

RENDIMIENTO

Es un indicador relacionado “con el tiempo que tarda un sistema en responder a un estímulo o evento”. El rendimiento “busca determinar si el tiempo de respuesta de un sistema es adecuado según los requerimientos” (Blancarte, 2020).

PATRÓN DE INTEGRACIÓN

“Los patrones de integración son soluciones aceptadas para problemas recurrentes dentro de un contexto dado”, definiendo diseños comunes en el desarrollo de funcionalidades relacionadas con la integración entre sistemas (Hohpe & Woolf, 2003).

La Figura 1 ilustra los diferentes componentes que intervienen en la comunicación de dos aplicaciones que intercambian mensajes, donde una aplicación publica un mensaje en un canal y la otra aplicación lee el mensaje de ese canal (Dervaux et al., 2019). Dichos componentes son:

- *Application* (Aplicación): Representa una aplicación que participa en el intercambio de mensajes, ya sea en el rol de remitente o de receptor.
- *Message* (Mensaje): Representa la información que se desea intercambiar, entre aplicaciones, teniendo en cuenta que un mensaje es un registro de datos que un bróker de mensajería puede transmitir. El mensaje se transmite a través de un canal.
- *Channel* (Canal): También llamado cola de mensajería, se encarga de trasladar el mensaje de una aplicación a otra.
- *Routing* (Enrutador): Es el componente que enruta uno o muchos mensajes a uno o muchos receptores.
- *Translation* (Traducción): Representa el componente que transforma el formato de un mensaje de una aplicación (remitente) en el formato que otra aplicación (receptor) requiere.
- *Systems Management* (Administración del sistema): Representa el componente para monitorear el flujo de mensajes de un sistema basado en mensajería, asegura el registro de las actividades del sistema y registra las condiciones de error (en caso de presentarse).

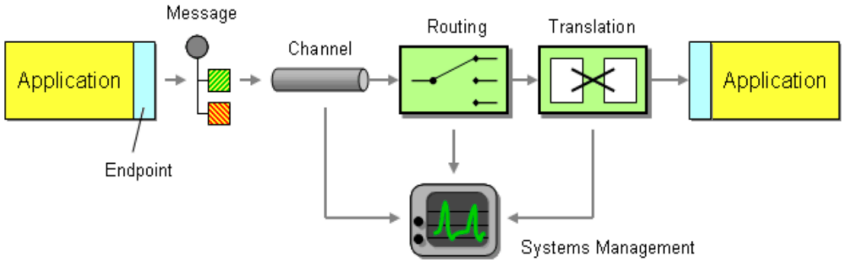

Figura 1 Representación de intercambio de mensajes entre dos aplicaciones. Tomada de (Hohpe & Woolf, 2003)

A continuación, se describen los dos patrones de integración en los cuales nos interesamos en este estudio.

Event Message

Descripción: Este patrón es utilizado para transmitir un evento de mensaje desde un *subject* (sujeto) a uno o más *observer* (observador) interesados, enviándolos a un canal (ver Figura 2). Cuando hablamos de evento de mensaje nos referimos a una estructura que contiene información sobre alguna acción del sistema. Cada *observer* debe ser notificado de un *event message* (evento de mensaje) en particular una vez, pero no debe ser notificado repetidamente del mismo evento de mensaje. El evento de mensaje no se puede considerar consumido hasta que todos los *observer* hayan sido notificados, pero una vez que lo hayan hecho, el evento de mensaje puede considerarse consumido y debe desaparecer del canal.

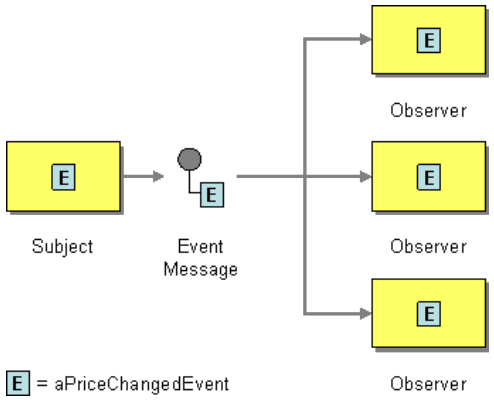

Figura 2 Diagrama del patrón Event Message. Tomado de (Hohpe & Woolf, 2003)

Ventajas: Transmitir un evento de mensaje a todos los *observer* interesados, haciendo que este evento sea duplicado a cuantos *observer* estén inscritos, así el evento de mensaje, que es enviado únicamente una vez por el *subject*, puede ser transmitido de manera paralela.

Desventajas: Vulnerabilidad ante sistemas que puedan suscribirse como *observer* y espiar los mensajes, esto se puede lograr creando un *observer* adicional y suscribiéndolo a un canal, ya que también le llegaría una copia del mensaje. En sistemas críticos y en aquellos en los que la información sea sensible, como lo es un sistema de nómina y uno de contabilidad, se debe tener precaución al usar este patrón, y recordar que existen otros patrones como *Point-to-Point Channel* que pueden ayudar al propósito de entregar el mensaje a un solo *observer*.

Command Message

Descripción: *Command Message* es un patrón que se enfoca en la creación de un mensaje, este mensaje contiene una estructura con la información necesaria para ejecutar un procedimiento o comportamiento de otra aplicación.

La Figura 3 representa el patrón *Command Message*, el cual es usado para invocar de manera fiable un procedimiento en otro sistema, donde un *sender* (emisor) envía un comando de mensaje a un *receiver* (receptor) por un canal punto a punto (*Point to Point Channel*) para que cada comando se consuma e invoque solamente una vez.

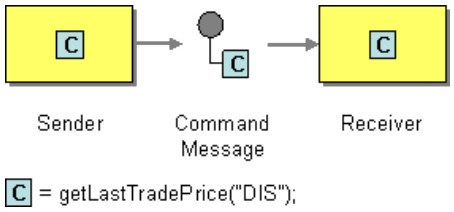

Figura 3 Diagrama de patrón Command Message. Tomada de (Hohpe & Woolf, 2003)

Ventajas: La ejecución de una funcionalidad de manera asincrónica, lo cual implica que el *sender* que envíe el comando no genere espera (bloqueo) por su ejecución.

Desventajas: El patrón no propone una manera de verificar si la ejecución de la funcionalidad se realizó de manera exitosa, dicha implementación o estrategia debe ser desarrollada como un complemento.

SISTEMA DISTRIBUIDO

Un sistema distribuido es un conjunto de computadores ligeramente acoplados que se interconectan en red para colaborar entre sí, y realizar una tarea conjunta como si la realizara un único computador (Casas Lizcano & López, 2015).

ARQUITECTURA CLIENTE-SERVIDOR

Cliente-servidor es la arquitectura más citada cuando se discuten los sistemas distribuidos, en los que los procesos que participan en el sistema desempeñan un rol concreto y bien definido, donde las tareas se reparten entre los proveedores de recursos o servicios llamados servidores y los demandantes llamados clientes. La Figura 4 ilustra la estructura simple de esta arquitectura que se compone de dos componentes: *cliente* y *servidor*, donde un cliente realiza peticiones a otro sistema, y el servidor es quien le da respuesta. En particular, los procesos de cliente interactúan con los procesos de servidor individuales en equipos anfitriones (*server*) potencialmente separados, con el

fin de acceder a los recursos compartidos que administran (Casas Lizcano & López, 2015).

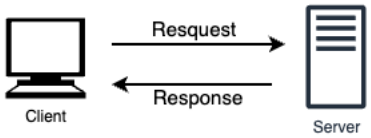

Figura 4 Arquitectura cliente-servidor

RABBITMQ

Es un software de negociación de mensajes de código abierto que funciona como un middleware (Bróker de mensajería) de mensajería implementando el protocolo AMQP (John & Liu, 2017), este protocolo es orientado a mensajes, encolamiento y enrutamiento(Hegde & S, 2020).

Los siguientes son los componentes que podemos encontrar en RabbitMQ (Ćatović et al., 2022):

- **Queue**: Es el componente que guarda el mensaje que es entregado por el *Exchange* y a su vez entrega el mensaje si tiene algún consumidor registrado.
- **Exchange**: Este componente es el encargado de recibir los mensajes que son enviados al bróker por un emisor y según el tipo entregarlo a una *Queue (Cola)* adecuada. Los tipos pueden ser:
  1. *Direct (Directo)*, entrega el mensaje solo a una *Queue*.
  2. *Topic (Tópico)*, puede entregar el mensaje a una o muchas *Queues* según coincidencia del *RoutingKey (Identificador clave)*.
  3. *Fanout (intercambio múltiple)*, siempre entrega el mensaje a las *Queue* que están suscritas al *Exchange.*

METODOLOGÍA

En esta sección describiremos detalladamente las etapas que conforman la metodología que usamos para llevar a cabo este estudio (ver Figura 5). Comenzamos con la formulación de la pregunta de investigación. Luego, en la etapa de selección de los patrones de

integración, se eligieron aquellos que se compararán en el estudio. Posteriormente, definimos los criterios necesarios para la comparación de los patrones de integración seleccionados. En la siguiente etapa, definimos la aplicación base, que nos ayudará en la etapa de desarrollo de la misma. Finalmente, se realizan pruebas y recolección de datos para su análisis y posterior presentación de resultados. Cabe destacar que la metodología utilizada se basó en la investigación de métodos cualitativos en ingeniería de sistemas de SZAJNFARBER y GRALLA (Szajnfarber & Gralla, 2017).

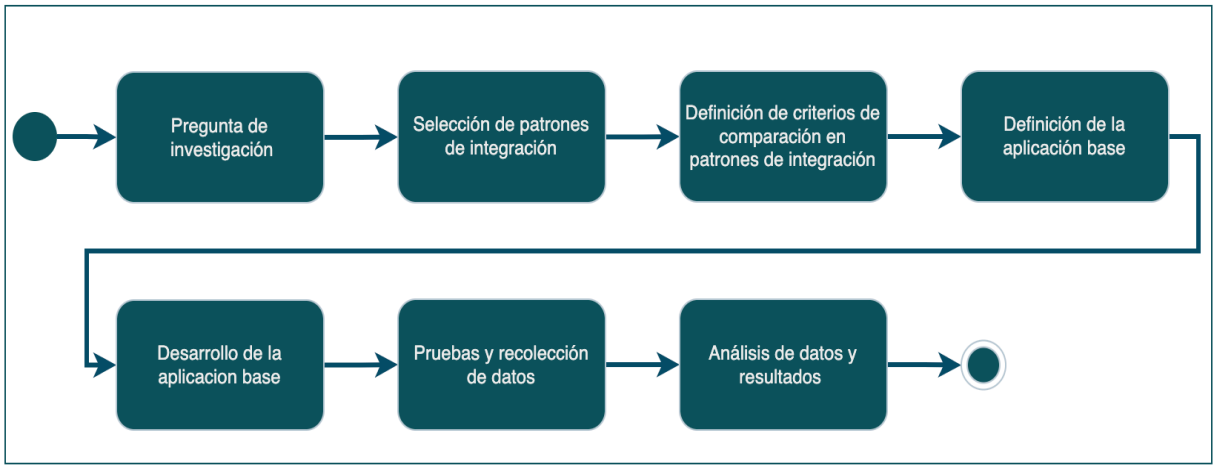

Figura 5 Etapas de la metodología

DESARROLLO DE LA METODOLOGÍA

PREGUNTA DE INVESTIGACIÓN

En este estudio, abordaremos la siguiente pregunta de investigación: ¿Cuál es el comportamiento de los patrones *Event Message* y *Command Message* sometido a pruebas de rendimiento para arquitecturas orientadas a *cliente-servidor* en sistemas distribuidos?

SELECCIÓN DE PATRONES DE INTEGRACIÓN

La Figura 6 ilustra todos los patrones involucrados en la comunicación basada en mensajes, que se divide en seis grupos según su función en la integración: *Message Endpoints* (puntos finales de mensajes), *Message Construction* (construcción de mensajes), *Messaging Channels* (canales de mensajes), *Message Routing* (enrutamiento de mensajes), *Message Transformations* (transformación de mensajes) y *System Management* (gestión del sistema). De este modo, se hace énfasis en Message Construction debido a que es un grupo de patrones que permite ser controlado por el desarrollador para la toma de decisiones acertadas en el momento de elegir la configuración precisa en el channel. En este sentido, el desconocimiento, por parte del desarrollador, para la selección de decisiones reflejaría incoherencia en un adecuado funcionamiento. Así mismo, se eligieron los patrones Event Message y Command Message (que hacen parte de este grupo) ya que son ampliamente utilizados con fines industriales, empresariales, académicos entre otros. Al usar el patrón Command Message, de forma implícita usaremos el patrón Point-to-Point Channel, dado que los mensajes en "el Command Message generalmente se envían a través de Point-to-Point Channel para que cada comando se consuma e invoque solo una vez" (Hohpe & Woolf, 2003). Por otro lado, al usar el patrón Event Message, de forma implícita usaremos el patrón Publisher-Subscribe Channel, ya que "por lo general, no hay motivo para limitar un Event Message a un solo receptor a través de un Point-to-Point Channel; el mensaje generalmente se transmite a través de un Publisher-Subscribe Channel para que todos los procesos interesados reciban una notificación" (Hohpe & Woolf, 2003)

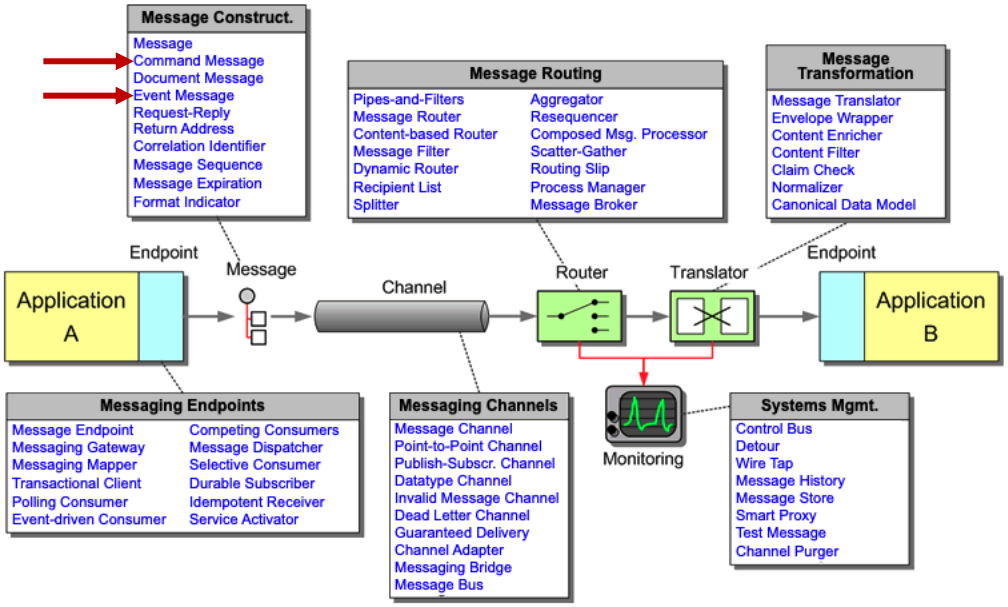

Figura 6 Patrones de integración. Tomada de (Hohpe & Woolf, 2003)

Según Hohpe y colaboradores, "Los patrones de integración son independientes del proveedor y se aplican a la mayoría de las soluciones de mensajería" (Hohpe & Woolf, 2003). Sin embargo, es importante destacar que cada proveedor puede utilizar su propia terminología para referirse a los patrones. La Figura 7 presenta una comparación de los nombres que usan algunos de los proveedores de soluciones en la industria de software para referirse a algunos de los productos de mensajería más utilizados. La primera columna de la tabla presenta los nombres genéricos de los patrones de integración que aparecen en la literatura. Las demás columnas representan soluciones desarrolladas por diferentes empresas tecnológicas que usan patrones de integración. En la tabla encontraremos el nombre sinónimo que cada solución utiliza para cada patrón. Por ejemplo, JMS es una solución desarrollada por Sun Microsystems para el uso de colas de mensajería, mientras que MSQM es una solución de cola de mensajería desarrollada por Microsoft.

La Figura 7 está estructurada de la siguiente manera: La primera columna (*Enterprise Integration Patterns*) representa los nombres con los cuales son reconocidos los patrones en la academia, por parte de algunos autores. A partir de la segunda columna, se muestra la relación entre el nombre del patrón y la tecnología que lo usa. Por ejemplo, en la tecnología *Java Message Service* (JMS), el *Message Channel* es llamado *Destination*. En la figura, podemos observar que los patrones de integración *Point-to-Point Channel* y *Publish-Subscribe Channel* son usados por diversas tecnologías reconocidas.

Enterprise Integration Patterns	Java Message Service (JSM)	Microsoft MSMQ	WebSphere MQ	TIBCO	WebMethods	SeeBeyond	Vitria
Message Channel	Destination	Message Queue	Queue	Subject	Queue	Intelligent Queue	Channel
Point-to-Point Channel	Queue	Message Queue	Queue	Distributed Queue	Deliver Action	Intelligent Queue	Channel
Publish-Subscribe Channel	Topic	-	-	Subject	Publish-Subscribe Action	Intelligent Queue	Publis-Subscribe Channel
Message	Message	Message	Message	Message	Document	Event	Event
Message Endpoint	Message Producer, Message Consumer			Publisher, Subscriber	Publisher, Subscriber	Publisher, Subscriber	Publisher, Subscriber

Figura 7 Nombres de patrones más comunes vs Nombres en entidad de producto correspondiente. Tomada de (Hohpe & Woolf, 2003)

DEFINICIÓN DE CRITERIOS DE COMPARACIÓN EN PATRONES DE INTEGRACIÓN

En esta sección se definen los criterios que se utilizarán para comparar los patrones de integración previamente seleccionados (ver Tabla 1).

Criterio de comparación	Descripción	Medida
Tamaño del mensaje	Longitud del mensaje a enviar.	Tamaño de mensaje: Pequeño, Mediano y Grande
Configuración del bróker de mensajería	Posibles configuraciones que acepta el bróker de mensajería que influyen en el rendimiento.	No Aplica
Latencia en envío y recepción de mensajes	Medida del promedio de tiempos transcurridos entre el envío y la recepción de mensajes. La medición se realiza por cada patrón (*Event Message* y *Command Message*).	No Aplica
Nombre	Nos representa el tipo de prueba realizada, por patrón, tamaño de mensaje y configuración del bróker de mensajería.	No Aplica

Tabla 1 Criterios de comparación de los patrones de integración

DEFINICIÓN DE LA APLICACIÓN DE CONSULTA DEL CLIMA

Para este estudio, desarrollamos una aplicación que permite consultar información sobre el clima, incluyendo variables como *queryCost* (Costo de consulta), *latitude* (Latitud), *longitude* (Longitud), *resolvedAddress* (Dirección resuelta), *address* (Dirección), *timezone* (Zona horaria) y *description* (Descripción). La aplicación consta de dos componentes: el servidor *weather-service Server* y el servidor *weather-api Server*. *Weather-service Server* solicita los datos del clima a *weather-api Server*. Para permitir la comunicación entre estos componentes y obtener la información del clima, implementaremos patrones de integración y llamadas con arquitectura cliente-servidor, la cual describiremos en detalle en los numerales 4.4.1 y 4.4.2. Los datos relacionados con el clima fueron predefinidos en el componente *weather-api Server*, para simular la disponibilidad de la información y con el objetivo de no afectar la latencia a causa de la incorporación de factores por fuera del enfoque de este estudio.

Descripción de componentes de la aplicación de consulta del clima

La Figura 8 ilustra los componentes de la aplicación y la dirección de comunicación entre ellos, para obtener y suministrar información relacionada al clima.

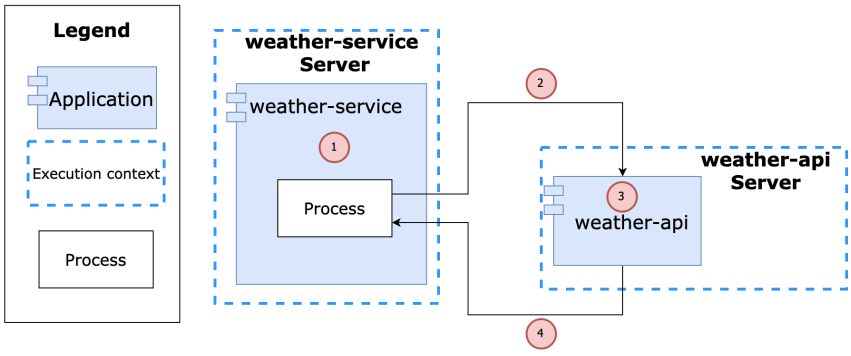

Figura 8 Diagrama de componentes de consulta del clima

1. *weather-service Server*: este componente representa una aplicación que realizará consultas sobre la información del clima.
2. El componente *weather-service Server* se comunica con el componente *weather-api Server* para obtener la información relacionada con el clima.
3. *weather-api Server*: este componente representa una aplicación que suministrará información relacionada con el clima.
4. El componente *weather-api Server* se comunica con el componente *weather-service Server* para suministrar la información del clima.

Descripción de componentes de la aplicación de consulta del clima con el patrón de integración Command Message

Tomando como base una aplicación utilizada para estimar el clima se procede a desarrollar una herramienta similar en la que la integración entre componentes se hace a través del patrón *Command Message*.

La comunicación entre los componentes se realiza a través del componente *RabbitMQ Server*, el cual actúa como intermediario en la recepción y entrega de mensajes. Los mensajes se generan gracias al patrón *Command Message*, al usar este patrón lo típico es que *RabbitMQ Server* use en su *Channel* el patrón *Point-to-Point Channel*, para que el mensaje sea entregado únicamente a un destinatario. Es importante destacar que toda la comunicación con *RabbitMQ Server* debe realizarse mediante el protocolo AMQP, ya que es un protocolo usado por defecto en este componente y además es orientado a mensajes. Así mismo, el flujo de comunicación entre los diferentes componentes tiene como punto de partida el cliente *Apache JMeter Client*, que permite realizar consultas sobre la información del clima al componente *weather-service Server*.

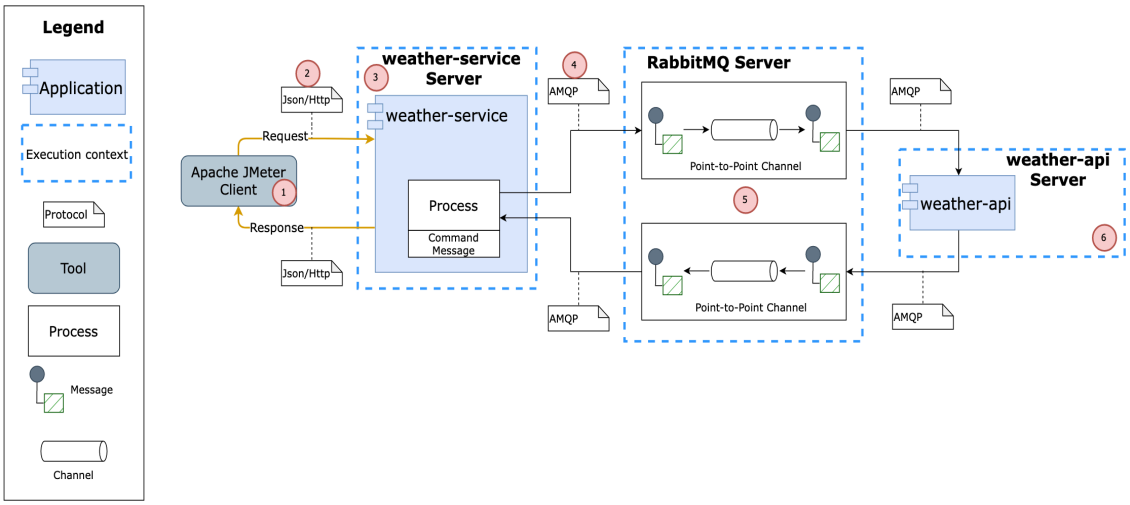

Figura 9 Diagrama de componentes usando patrón Command Message

La Figura 9 ilustra los componentes y el intercambio de mensajes entre ellos, dos de los componentes son *weather-service Server* y *weather-api Server*. A continuación, se describen los elementos:

1. **Apache JMeter Client:** Esta herramienta actúa como cliente y la utilizaremos para simular múltiples solicitudes concurrentes al componente llamado *weather-service Server*.
2. **Json/Http:** Todas las solicitudes se realizarán bajo el protocolo HTTP y enviando un mensaje en formato Json.
3. **weather-service Server:** Este componente representa una aplicación que procesa consultas realizadas a *weather-api Server*, todo este proceso se realiza de manera bloqueante, en un flujo síncrono.
4. **AMQP:** Con RabbitMQ, todas las solicitudes se realizarán bajo el protocolo AMQP y enviando un mensaje en formato Json.
5. **RabbitMQ Server:** Este componente representa el bróker de mensajería, la cual nos ayudará al intercambio de mensajes entre aplicaciones. A continuación, se detalle el patrón usado por RabbitMQ Server en la Figura 9: *Point to Point Channel*, el cual es un patrón de integración que garantiza que solo un receptor consuma un mensaje determinado, si un canal tiene múltiples receptores, solamente uno de ellos puede consumir con éxito un mensaje en particular. Este patrón es usado por el componente *channel*.
6. **weather-api Server:** Este componente representa una aplicación la cual suministra datos relacionados con el clima, estos datos los clasificaremos en tres grupos según su tamaño, los cuales serán pequeño (308 Bytes), mediano

(14.38 KB) y grande (55.77 KB), con el objetivo de tener datos de medición en todo el flujo de los componentes y poder analizar el rendimiento.

Descripción de los componentes de la aplicación de consulta del clima con el patrón de integración Event Message

Tomando como base una aplicación utilizada para estimar el clima se procede a desarrollar una herramienta similar en la que la integración entre componentes se hace a través del patrón *Event Message*.

La Figura 10 muestra los componentes y el intercambio de mensajes entre ellos, en donde se hace referencia a weather-service Server y weather-api Server, estos pueden intercambiar mensajes para lo cual es necesario contar con el componente RabbitMQ Server. Esta última se encarga de recibir y entregar mensajes. Así mismo, se utiliza el patrón *Event Message*, lo que implica que RabbitMQ Server use en su *Channel* (canal) el patrón *Publish-Subscribe Channel*, garantizando que el mensaje sea entregado a todos los destinatarios que estén suscritos al *channel*. La comunicación con RabbitMQ Server debe ser a través del protocolo AMQP, ya que es un protocolo usado por defecto en este componente y además es orientado a mensajes. Al igual que en el caso *Command Message* el flujo de comunicación inicia con el cliente Apache JMeter Client.

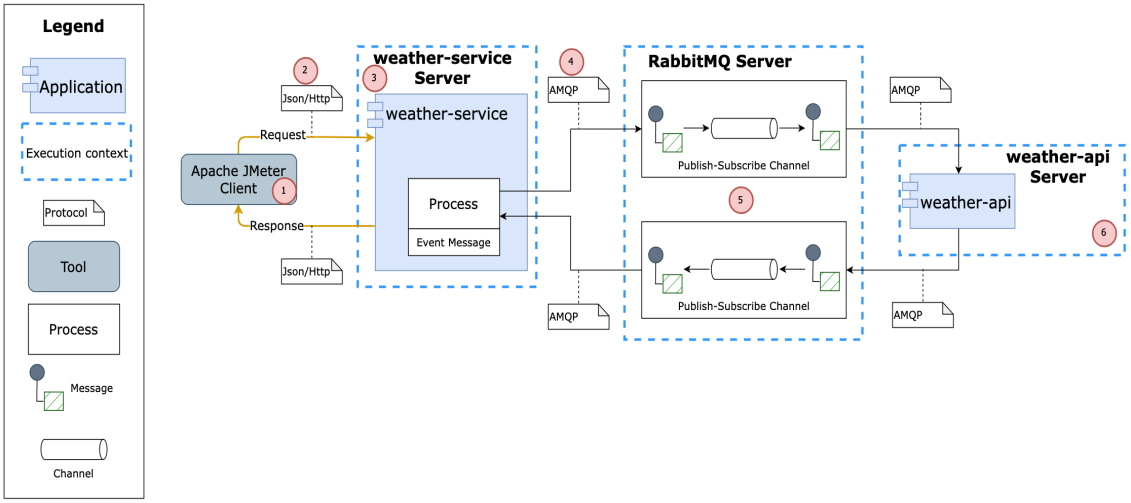

Figura 10 Diagrama de componentes usando patrón Event Message

7. **Apache JMeter Client:** Esta herramienta actúa como cliente y la utilizaremos para simular múltiples solicitudes concurrentes al componente llamado *weather-service Server*.
8. **Json/Http:** Todas las solicitudes se realizarán bajo el protocolo HTTP y enviando un mensaje en formato Json.
9. **weather-service Server:** Este componente representa una aplicación que procesa consultas realizadas a *weather-api Server*, todo este proceso se realiza de manera bloqueante, en un flujo síncrono.
10. **AMQP:** Con RabbitMQ, todas las solicitudes se realizarán bajo el protocolo AMQP y enviando un mensaje en formato Json.
11. **RabbitMQ Server:** Este componente representa el bróker de mensajería, la cual nos ayudará al intercambio de mensajes entre aplicaciones. A continuación, se detalle el patrón usado por RabbitMQ Server en la Figura 10: *Publish-Subscribe Channel*, hace referencia a un patrón de integración el cual entrega una copia del mensaje a todos los canales de salida. Cada canal tiene solamente un suscriptor, al que se le permite consumir un mensaje a la vez. Este patrón es usado en el componente *channel* (Lazidis et al., 2022).
12. **weather-api Server:** Este componente representa una aplicación la cual suministra datos relacionados con el clima, estos datos los clasificaremos en tres grupos según su tamaño, los cuales serán pequeño (308 Bytes), mediano (14.38 KB) y grande (55.77 KB), con el objetivo de tener datos de medición en todo el flujo de los componentes y poder analizar el rendimiento.

Diagrama de secuencia

La Figura 11 presenta el flujo de intercambio de mensajes entre los componentes *Apache JMeter*, *weather-service Server* y *weather-api Server*.

Este flujo empieza con el componente de Apache JMeter, que se comunica con el componente weather-service Server enviando un conjunto de mensajes de manera concurrente. Luego, weather-service Server se comunica con weather-api Server reenviando los mismos mensajes.

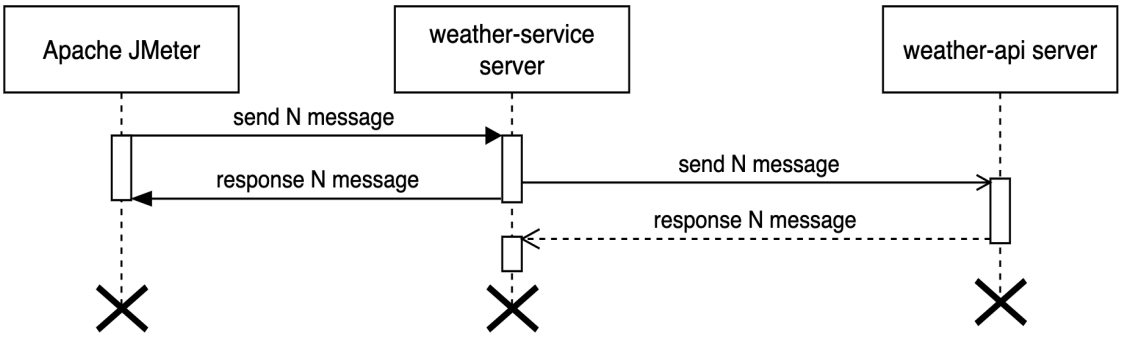

Figura 11 Diagrama de secuencia asíncrono

DESARROLLO DE LA APLICACIÓN DE CONSULTA DEL CLIMA

En esta sección se detallan las tecnologías utilizadas y las configuraciones realizadas a los proyectos desarrollados, así como la configuración adaptada en el bróker de mensajería mediante la implementación de un código fuente de los sistemas (https://github.com/jfgomezvelez/weather-api https://github.com/jfgomezvelez/weather-service). En la propuesta de desarrollo se ha creado un *Branch* (rama) en GitHub, para cada patrón, con el objetivo de garantizar la

misma implementación y clases en cada una de ellas. Sin embargo, es posible que existan pequeñas variaciones en la implementación de cada patrón. En este sentido, se excluyó el tratamiento preliminar atribuido al diseño y arquitectura de software, debido a que no es considerado como objetivo de estudio. De esta manera es posible obtener una aplicación más plana, sencilla y entendible.

Tecnologías

RabbitMQ fue elegido para las pruebas porque es uno de los agentes de mensajes de código abierto más populares respaldados comercialmente por Pivotal y Confluent y se usa con frecuencia en las empresas. Este intermediario de mensajes también puede usar el modelo de comunicación de publish-subscribe, que es adecuado para los sistemas donde existen muchos dispositivos conectados (Bagaskara et al., 2020).

En relación con el desarrollo de las aplicaciones *weather-service Server* y *weather-api Server* se utilizaron tecnologías como:

- Java 1.8
- Spring Boot
- Gradle
- RabbitMQ (Contenerizado con Docker)
- MongoDB
- GitHub
- Docker

Las siguientes son las herramientas usadas:

- Intellij Idea
- Apache JMeter
- MongoDB Compass

Event Message

- **Configuración de proyectos:**

Implementamos dos aplicaciones: *weather-service Server* y *weather-api Server*, ambos sistemas son consumidores y receptores según el flujo del mensaje. Para la creación de los proyectos, utilizamos la estructura por defecto que propone *Spring Boot*.

Cada proyecto contiene un archivo de configuración llamado *application.yaml* que se encuentra en la carpeta *resources.* En este archivo adicionamos las siguientes configuraciones: nombre de la aplicación, conexión con RabbitMQ, conexión con la base de datos (MongoDB) y puerto de ejecución.

Para ejecutar los proyectos, nos ubicamos en el archivo MainApplication, le damos *clic* derecho y seleccionamos la opción de Run MainApplication

- **Configuración de bróker de mensajería:**

  Al iniciar los sistemas de weather-service Server y weather-api Server, se generará de manera automática la topología que cada uno necesita.

  En RabbitMQ, los mensajes son recibidos por el *Exchange* y, dependiendo de la topología establecida (Tipo de Exchange), pueden ser entregados a uno o varios suscriptores, o ambas estrategias. En nuestro caso, será configurado para que los mensajes sean entregados a varios destinatarios, aunque solo haya un sistema suscrito a ese *Exchange*.

  Para el sistema *weather-service Server* se crea un *Exchange* llamado *weather.service.em.exchange* de tipo *fonout*, el cual utiliza el patrón *Event Message* y es apoyado por el patrón *Publish-Subscribe Channel* para la entrega del mensaje. Además, se crea una cola llamada *weather.service.em.queue* para recibir el mensaje.

  En el caso del sistema *weather-api,* se crea un *Exchange* llamado *weather.api.exchange* de tipo *fonout* para poder usar el patrón *Event*

Message y ser apoyado por el patrón *Publish-Subscribe Channel* para la entrega del mensaje, adicional se crea una cola llamada *weather.api.queue* para recibir el mensaje.

Command Message

- **Configuración de proyectos:**

  Implementamos dos sistemas *weather-service Server* y *weather-api Server*, donde ambos sistemas son consumidores y receptores según el flujo del mensaje. Estos proyectos fueron concebidos con la estructura por defecto que propone Spring Boot para la creación de proyecto.

  Los proyectos tienen un archivo de configuración llamado *application.yaml* que está contenido en la carpeta resources de cada proyecto y en este archivo adicionamos las siguientes configuraciones: nombre de la aplicación, conexión con RabbitMQ, conexión con la base de datos (MongoDB) y puerto de ejecución.

  Para ejecutar los proyectos, nos ubicamos en el archivo MainApplication, le damos clic derecho y seleccionamos la opción de Run MainApplication.

- **Configuración de bróker de mensajería:**

  Cuando los dos sistemas inician, crearán de manera automática la topología que cada sistema necesita.

  En RabbitMQ, los mensajes llegan al *Exchange* y dependiendo de la topología (Tipo de Exchange) el mensaje puede ser entregado a varios suscriptores, a uno concreto o tener ambas estrategias, para nuestro caso será configurado para ser entregado a un solo destinatario.

  Para el sistema *weather-service Server* se crea un *Exchange* llamado *weather.service.cm.exchange* de tipo *direct* para poder usar el patrón *Event Message* y ser apoyado por el patrón *Point-to-Point Channel* para la entrega del mensaje, adicional se crea una cola llamada *weather.service.cm.queue* para recibir el mensaje.

Para el sistema *weather-api Server* se crea un *Exchange* llamado *weather.api.cm.exchange* de tipo *direct* para poder usar el patrón *Event Message* y ser apoyado por el patrón *Point-to-Point Channel* para la entrega del mensaje, adicional se crea una cola llamada *weather.api.queue* para recibir el mensaje.

PRUEBAS Y RECOLECCIÓN DE DATOS

A continuación, se describe el procedimiento realizado para recolectar datos y llevar a cabo las diferentes pruebas objeto de presente estudio, con el objetivo de evaluar el comportamiento de los patrones descritos en la sección 4.2 y que servirán de medida para la detección y prevención de problemas relacionados con rendimiento.

- **Objetivo de las pruebas:**

  El objetivo de las pruebas es medir el tiempo transcurrido desde el envío hasta la recepción de un mensaje entre las aplicaciones *weather service Server* y *weather api Server* usando tanto el patrón *Event Message* como el patrón *Command Message*. Además, se busca determinar si la configuración del bróker de mensajería y el tamaño de los mensajes que se transmiten influyen en el rendimiento de ambas aplicaciones.

- **Procedimiento para probar:**

  Comenzamos iniciando las aplicaciones, para asegurarnos que están funcionando y que sus topologías en RabbitMQ se hayan creado correctamente. Luego ejecutamos la herramienta JMeter de Apache para simular un conjunto de solicitudes concurrentes y obtener varios puntos de comparación en función de la cantidad de mensajes enviados.

  Cada solicitud realizada por JMeter consumirá un servicio HTTP de la aplicación *weather service Server*. A su vez, esta aplicación publicará un

mensaje al bróker de mensajería RabbitMQ Server, donde la aplicación *weather api Server* estará suscrita. Este sistema descargará el mensaje y retornará un mensaje publicándolo en el bróker de mensajería RabbitMQ para que la aplicación *weather service Server* lo descargue y al final guarde el tiempo que demoró el mensaje en ser enviado y recibido.

De esta manera, podremos validar si el tamaño de los mensajes que se transmiten afecta el rendimiento de las aplicaciones.

- **Recolección de datos:**

  La aplicación *weather service Server* guarda en una base de datos el tiempo que tarda un mensaje entre su publicación y su recepción. Utilizamos la información almacenada en MongoDB para generar un reporte que incluye los datos generados por cada patrón y las diferentes configuraciones realizadas en el bróker de mensajería RabbitMQ.

Métodos o experimentación

Creamos un archivo de configuración por cada patrón y cada configuración de rabbitMQ a probar (ACK, AutoACK, Concurrency, prefetch). Generaremos un compilado (Jar) por cada patrón a probar para evitar hacer las pruebas con un IDE, ya que puede consumir más de los recursos necesarios y afectar el rendimiento.

Para probar, utilizamos una máquina MacBook Pro (15-inch, 2018) procesador 2,6 GHz 6-Core Intel Core i7, memoria de 32 GB 2400 MHz DDR4 y tarjeta de video Radeon Pro 560X 4 GB Intel UHD Graphics 630 1536, la cual por efectos de pruebas no tuvo ningún tipo de aplicación innecesaria abierta. La aplicación la iniciamos por consola y le indicamos el archivo de configuración a usar.

Paso a Paso event-message

A continuación, vamos a listar los pasos que se deben seguir para iniciar las pruebas al patrón event-message.

1. Nos ubicamos en la rama feature/event-message en ambos proyectos, weather-service Server y weather-api Server. (GitHub - jfgomezvelez/weather-service at feature/event-message , GitHub - jfgomezvelez/weather-api at feature/event-message)
2. Descargamos el código fuente.
3. Generamos el compilado (Jar) para el patrón *Event Message*.
4. Ubicamos solo el compilado en una ruta específica y adicionamos los archivos de configuración del sistema.
5. Iniciamos la aplicación por consola indicando el archivo de configuración a iniciar.
6. Descargamos el archivo de prueba de JMeter.
7. Iniciamos la herramienta de JMeter y seleccionamos el archivo de prueba.
8. Consumimos el servicio http de weather-service Server con Jmeter y la configuración de consumo será de 100 solicitudes.
9. Tenemos 3 tamaños de mensajes, así que repetimos el paso 2 por cada tamaño de mensaje.
10. Validamos en la base de datos de MongoDB que los 100 registros se hayan guardado.
11. Verificamos en la gráfica de los charts de MongoDB que la información insertada se vea reflejada según la prueba que se esté realizando.

Paso a Paso command-message

A continuación, vamos a listar los pasos que se deben seguir para iniciar las pruebas al patrón command-message.

1. Nos ubicamos en la rama feature/command-message en ambos proyectos, *weather-service Server* y *weather-api Server*. (GitHub - jfgomezvelez/weather-service at feature/command-message , GitHub - jfgomezvelez/weather-api at feature/command-message)
2. Descargamos el código fuente.
3. Generamos el compilado (Jar) para el patrón *Command Message*.
4. Ubicamos solo el compilado en una ruta específica y adicionamos los archivos de configuración del sistema.

5. Iniciamos la aplicación por consola indicando el archivo de configuración a iniciar.
6. Descargamos el archivo de prueba de JMeter.
7. Iniciamos la herramienta de JMeter y seleccionamos el archivo de prueba.
8. Consumimos el servicio http de weather-service Server con Jmeter y la configuración de consumo será de 100 solicitudes.
9. Tenemos 3 tamaños de mensajes, así que repetimos el paso 2 por cada tamaño de mensaje.
10. Validamos en la base de datos de MongoDB que los 100 registros se hayan guardado.
11. Verificamos en la gráfica de los charts de mongo que la información insertada se vea reflejada según la prueba que se esté realizando.

ANÁLISIS DE DATOS Y RESULTADOS

En esta sección se presentan los resultados más relevantes de las pruebas realizadas y se analizan en detalle. Se ejecutaron las pruebas correspondientes al patrón *Event Message* tal como se describe en el numeral 4.6, obteniendo un comportamiento para la configuración ACK-Manual (Figura 12) y ACK-Automático (Figura 13). Los resultados muestran una combinación de concurrencia basada en el prefetch. Para ello, se determinó la latencia promedio de los mensajes enviados por segundo, evaluando cada tamaño de mensaje con la misma cantidad de pruebas. Se observó que al configurar la concurrencia se procesan más mensajes por segundo, con una diferencia de aproximadamente 60 segundos en comparación con la configuración sin concurrencia. Además, se evidenció que el tiempo promedio de Procesamiento de los mensajes es proporcional al tamaño del mensaje, y se observó un comportamiento similar tanto para el caso ACK-Manual como para el ACK-Automático.

Las Figuras 14 y 15 presentan las gráficas correspondientes al patrón *Command Message*. De manera similar al patrón *Event Message*, se observa un comportamiento en la latencia promedio de los mensajes enviados (por segundo) para ambas configuraciones de ACK (Manual y Automático). A pesar de que estos dos patrones son diferentes, los resultados muestran una tendencia similar en los cuatro casos, lo que

sugiere que las configuraciones en el bróker de mensajería afectan el comportamiento del patrón de integración. No obstante, se observa que el rendimiento drásticamente afectado, aproximadamente 600% de incremento, si no se utiliza una configuración del bróker adecuada.

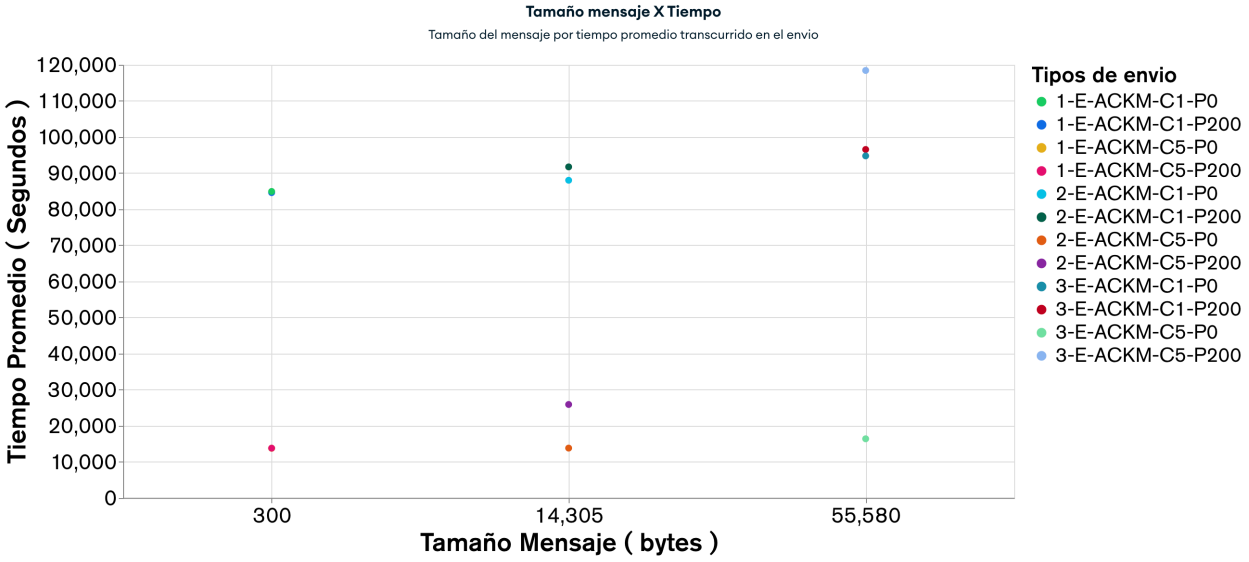

Figura 12 Gráfica que representa el tiempo promedio por tipo de mensaje - ACK manual

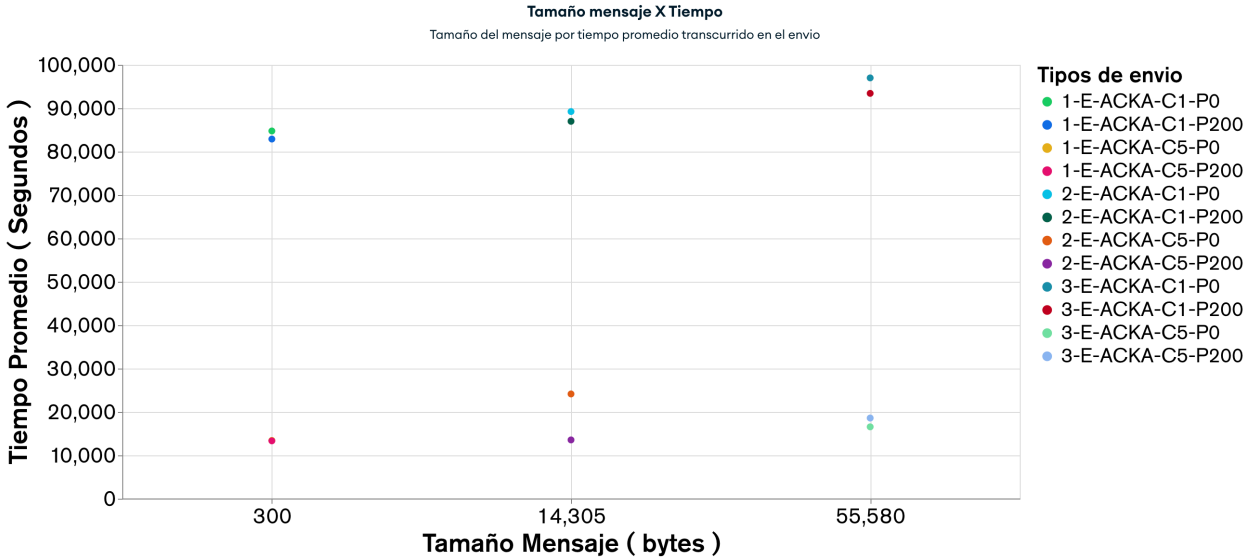

Figura 13 Gráfica que representa el tiempo promedio por tipo de mensaje - ACK Automático

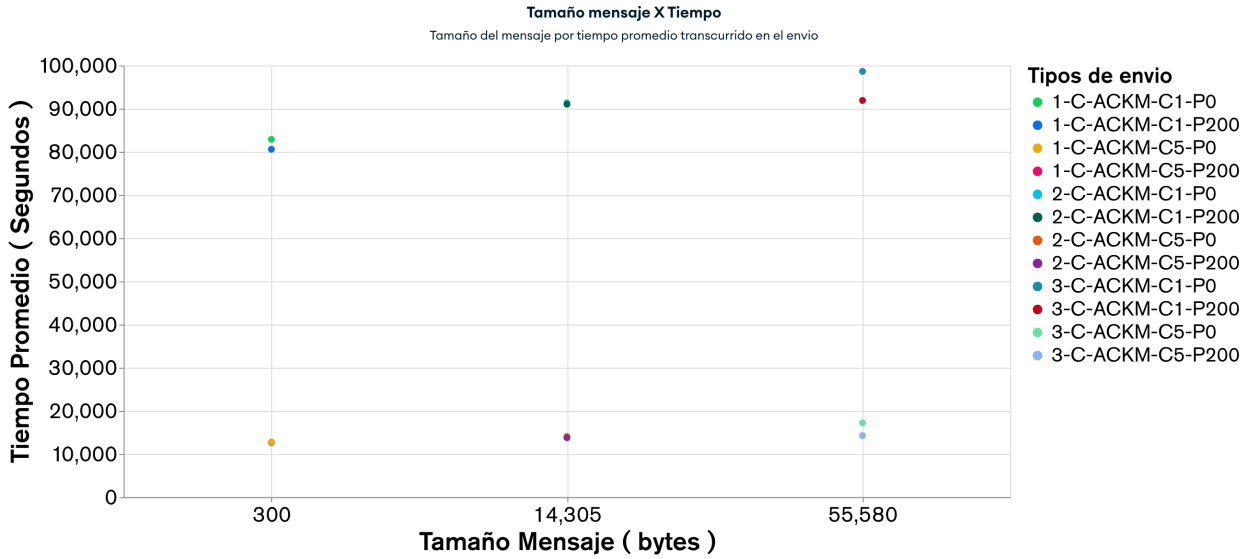

Figura 14 Gráfica que representa el tiempo promedio por tipo de mensaje - ACK manual

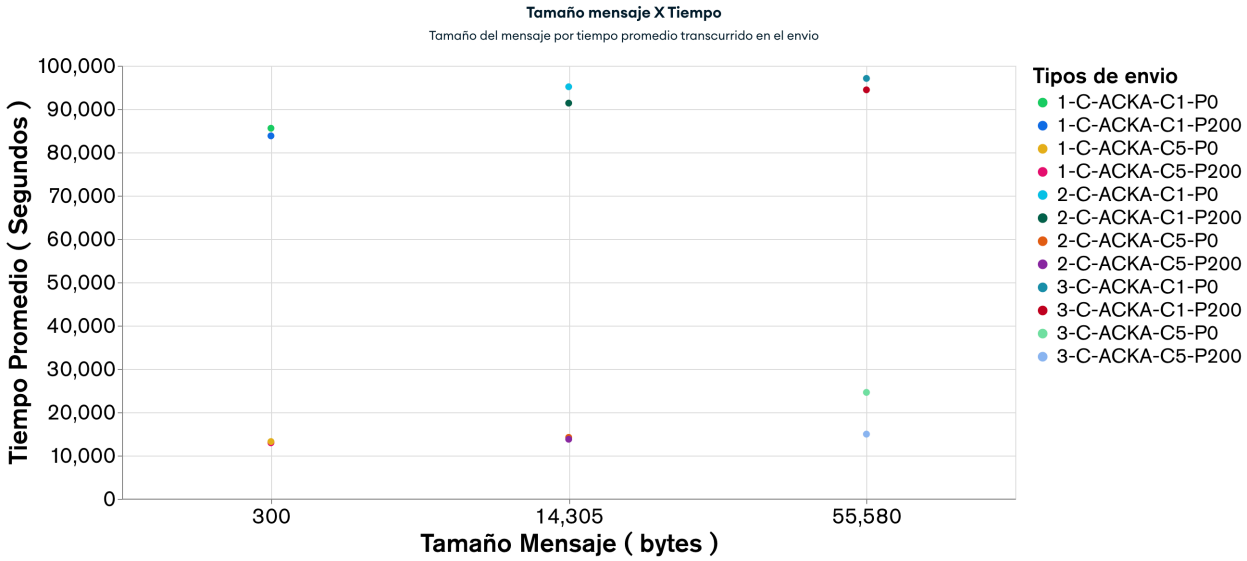

Figura 15 Gráfica que representa el tiempo promedio por tipo de mensaje - ACK Automático

Se realizaron 24 pruebas tanto para el patrón *Event Message* (Tabla 2) como para el patrón *Command Message* (Tabla 3). En estas tablas se presenta información sobre el tamaño del mensaje, la configuración del bróker de mensajería, la categorización de la prueba según la longitud del mensaje, el tiempo promedio de la prueba (es decir, el tiempo de envío de 1000 mensajes de manera concurrente) y el nombre del tipo de prueba realizada. Estos datos tabulados corresponden a los resultados observados en las Figuras 12-15, las cuales contienen los criterios de comparación tal como se definieron en el numeral 4.3. Además, en la segunda columna de las tablas se detalla la configuración del bróker de mensajería utilizada en las pruebas realizadas.

Tamaño mensaje	**Configuración bróker de mensajería**	**Medida**	**Tiempo (Seg)**	**Nombre**
300 bytes	ACK = Manual, Concurrencia (Concurrency) = 1, Captura previa (Prefetch) = 0	Pequeño	84.913	1-E-ACKM-C1-P0
14,305 bytes	ACK = Manual, Concurrencia (Concurrency) = 1, Captura previa (Prefetch) = 0	Mediano	88.010	2-E-ACKM-C1-P0
55,580 bytes	ACK = Manual, Concurrencia (Concurrency) = 1, Captura previa (Prefetch) = 0	Grande	94.768	3-E-ACKM-C1-P0
300 bytes	ACK = Manual, Concurrencia (Concurrency) = 5, Captura previa (Prefetch) = 0	Pequeño	13.739	1-E-ACKM-C5-P0
14,305 bytes	ACK = Manual, Concurrencia (Concurrency) = 5, Captura previa (Prefetch) = 0	Mediano	13.790	2-E-ACKM-C5-P0

55,580 bytes	ACK = Manual, Concurrencia (Concurrency) = 5, Captura previa (Prefetch) = 0	Grande	16.368	3-E-ACKM-C5-P0
300 bytes	ACK = Manual, Concurrencia (Concurrency) = 1, Captura previa (Prefetch) = 200	Pequeño	84.591	1-E-ACKM-C1-P200
14,305 bytes	ACK = Manual, Concurrencia (Concurrency) = 1, Captura previa (Prefetch) = 200	Mediano	91.723	2-E-ACKM-C1-P200
55,580 bytes	ACK = Manual, Concurrencia (Concurrency) = 1, Captura previa (Prefetch) = 200	Grande	96.545	3-E-ACKM-C1-P200
300 bytes	ACK = Manual, Concurrencia (Concurrency) = 5, Captura previa (Prefetch) = 200	Pequeño	13.776	1-E-ACKM-C5-P200
14,305 bytes	ACK = Manual, Concurrencia (Concurrency) = 5, Captura previa (Prefetch) = 200	Mediano	25.879	2-E-ACKM-C5-P200
55,580 bytes	ACK = Manual, Concurrencia (Concurrency) = 5, Captura previa (Prefetch) = 200	Grande	118.439	3-E-ACKM-C5-P200
300 bytes	ACK = Auto, Concurrencia (Concurrency) = 1, Captura previa (Prefetch) = 0	Pequeño	84.756	1-E-ACKA-C1-P0
14,305 bytes	ACK = Auto, Concurrencia (Concurrency) = 1, Captura previa (Prefetch) = 0	Mediano	89.234	2-E-ACKA-C1-P0
55,580 bytes	ACK = Auto, Concurrencia (Concurrency) = 1, Captura previa (Prefetch) = 0	Grande	96.994	3-E-ACKA-C1-P0
300 bytes	ACK = Auto, Concurrencia (Concurrency) = 5, Captura previa (Prefetch) = 0	Pequeño	13.400	1-E-ACKA-C5-P0

14,305 bytes	ACK = Auto, Concurrencia (Concurrency) = 5, Captura previa (Prefetch) = 0	Mediano	24.112	2-E-ACKA-C5-P0
55,580 bytes	ACK = Auto, Concurrencia (Concurrency) = 5, Captura previa (Prefetch) = 0	Grande	16.527	3-E-ACKA-C5-P0
300 bytes	ACK = Auto, Concurrencia (Concurrency) = 1, Captura previa (Prefetch) = 200	Pequeño	82.891	1-E-ACKA-C1-P200
14,305 bytes	ACK = Auto, Concurrencia (Concurrency) = 1, Captura previa (Prefetch) = 200	Mediano	86.982	2-E-ACKA-C1-P200
55,580 bytes	ACK = Auto, Concurrencia (Concurrency) = 1, Captura previa (Prefetch) = 200	Grande	93.429	3-E-ACKA-C1-P200
300 bytes	ACK = Auto, Concurrencia (Concurrency) = 5, Captura previa (Prefetch) = 200	Pequeño	13.294	1-E-ACKA-C5-P200
14,305 bytes	ACK = Auto, Concurrencia (Concurrency) = 5, Captura previa (Prefetch) = 200	Mediano	13.513	2-E-ACKA-C5-P200
55,580 bytes	ACK = Auto, Concurrencia (Concurrency) = 5, Captura previa (Prefetch) = 200	Grande	18.559	3-E-ACKA-C5-P200

Tabla 2 Resultados de pruebas del patrón event-message. Las filas representan el promedio de las pruebas con 1000 mensajes enviados.

Tamaño mensaje	Configuración bróker de mensajería	Medida	Tiempo (Seg)	Nombre
300 bytes	ACK = Manual, Concurrencia (Concurrency) = 1, Captura previa (Prefetch) = 0	Pequeño	82.881	1-C-ACKM-C1-P0
14,305 bytes	ACK = Manual, Concurrencia (Concurrency) = 1, Captura previa (Prefetch) = 0	Mediano	91.364	2-C-ACKM-C1-P0
55,580 bytes	ACK = Manual, Concurrencia (Concurrency) = 1, Captura previa (Prefetch) = 0	Grande	98.635	3-C-ACKM-C1-P0
300 bytes	ACK = Manual, Concurrencia (Concurrency) = 5, Captura previa (Prefetch) = 0	Pequeño	12.778	1-C-ACKM-C5-P0
14,305 bytes	ACK = Manual, Concurrencia (Concurrency) = 5, Captura previa (Prefetch) = 0	Mediano	14.026	2-C-ACKM-C5-P0
55,580 bytes	ACK = Manual, Concurrencia (Concurrency) = 5, Captura previa (Prefetch) = 0	Grande	17.198	3-C-ACKM-C5-P0
300 bytes	ACK = Manual, Concurrencia (Concurrency) = 1, Captura previa (Prefetch) = 200	Pequeño	80.595	1-C-ACKM-C1-P200
14,305 bytes	ACK = Manual, Concurrencia (Concurrency) = 1, Captura previa (Prefetch) = 200	Mediano	91.035	2-C-ACKM-C1-P200

55,580 bytes	ACK = Manual, Concurrencia (Concurrency) = 1, Captura previa (Prefetch) = 200	Grande	91.915	3-C-ACKM-C1-P200
300 bytes	ACK = Manual, Concurrencia (Concurrency) = 5, Captura previa (Prefetch) = 200	Pequeño	12.645	1-C-ACKM-C5-P200
14,305 bytes	ACK = Manual, Concurrencia (Concurrency) = 5, Captura previa (Prefetch) = 200	Mediano	13.756	2-C-ACKM-C5-P200
55,580 bytes	ACK = Manual, Concurrencia (Concurrency) = 5, Captura previa (Prefetch) = 200	Grande	14.241	3-C-ACKM-C5-P200
300 bytes	ACK = Auto, Concurrencia (Concurrency) = 1, Captura previa (Prefetch) = 0	Pequeño	85.572	1-C-ACKA-C1-P0
14,305 bytes	ACK = Auto, Concurrencia (Concurrency) = 1, Captura previa (Prefetch) = 0	Mediano	95.155	2-C-ACKA-C1-P0
55,580 bytes	ACK = Auto, Concurrencia (Concurrency) = 1, Captura previa (Prefetch) = 0	Grande	97.083	3-C-ACKA-C1-P0
300 bytes	ACK = Auto, Concurrencia (Concurrency) = 5, Captura previa (Prefetch) = 0	Pequeño	13.193	1-C-ACKA-C5-P0
14,305 bytes	ACK = Auto, Concurrencia (Concurrency) = 5, Captura previa (Prefetch) = 0	Mediano	14.166	2-C-ACKA-C5-P0
55,580 bytes	ACK = Auto, Concurrencia (Concurrency) = 5, Captura previa (Prefetch) = 0	Grande	24.554	3-C-ACKA-C5-P0
300 bytes	ACK = Auto, Concurrencia (Concurrency) = 1, Captura previa (Prefetch) = 200	Pequeño	83.813	1-C-ACKA-C1-P200

14,305 bytes	ACK = Auto, Concurrencia (Concurrency) = 1, Captura previa (Prefetch) = 200	Mediano	91.375	2-C-ACKA-C1-P200
55,580 bytes	ACK = Auto, Concurrencia (Concurrency) = 1, Captura previa (Prefetch) = 200	Grande	94.443	3-C-ACKA-C1-P200
300 bytes	ACK = Auto, Concurrencia (Concurrency) = 5, Captura previa (Prefetch) = 200	Pequeño	12.904	1-C-ACKA-C5-P200
14,305 bytes	ACK = Auto, Concurrencia (Concurrency) = 5, Captura previa (Prefetch) = 200	Mediano	13.773	2-C-ACKA-C5-P200
55,580 bytes	ACK = Auto, Concurrencia (Concurrency) = 5, Captura previa (Prefetch) = 200	Grande	14.916	3-C-ACKA-C5-P200

Tabla 3 Resultados de pruebas del patrón command-message. Las filas representan el promedio de las pruebas con 1000 mensajes enviados

CONCLUSIONES

En este estudio se realizaron 24 pruebas a cada patrón del foco de estudio, E*vent Message* y *Command Message* con el objetivo de evaluar el comportamiento sometido a pruebas de rendimiento, teniendo como parámetro de salida el tiempo promedio y como parámetro de entra la configuración del bróker de mensajería y tamaño del mensaje. De este modo, según los datos obtenidos y representados en las Gráficas 12-14, podemos evidenciar que la configuración con alta concurrencia tiene un rendimiento significativo en comparación con una configuración sin concurrencia, observando una diferencia en tiempo aproximado de 60 segundos en los resultados de ambas configuraciones. Adicional a esto, podemos observar en el comportamiento que la latencia es afectada de manera proporcional al tamaño de los mensajes enviados y que es inversamente proporcional a la concurrencia. En

cuanto a la configuración de la Captura previa (Prefetch), se determinó que no tiene un impacto significativo en la latencia, lo cual se atribuye a un comportamiento similar en las pruebas que no tuvieron dicha configuración.

En general, se puede concluir que las configuraciones influyen en el comportamiento de los patrones E*vent Message* y *Command Message* afectando la latencia y el rendimiento reduciendo la cantidad de mensajes procesados e incrementando el tiempo de respuesta.

En resumen, los resultados de este estudio proporcionan información valiosa para aquellos que utilizan los patrones *Event Message* y *Command Message* en entornos de concurrencia, permitiendo una mejor comprensión de su comportamiento y desempeño basado en sus configuraciones.

REFERENCIAS

Autom, D., Publish, A., Orientada, S., Ejemplo, U., Digital, M., Fin, P., Autor, S., Garc, J., & Mu, I. (2020). *Máster Universitario en Software de Sistemas Distribuidos y Empotrados Diseño y Despliegue Automático de una Arquitectura Publish / Subscribe Orientada a Microservicios : Un Ejemplo en el Dominio del Marketing Digital Autor : Javier García Menéndez Direct*.

Bagaskara, A. E., Setyorini, S., & Wardana, A. A. (2020). Performance Analysis of Message Broker for Communication in Fog Computing. *ICITEE 2020 - Proceedings of the 12th International Conference on Information Technology and Electrical Engineering*, 98–103. https://doi.org/10.1109/ICITEE49829.2020.9271733

Blancarte, J. (2020). *Introducción a la arquitectura de software – Un enfoque práctico*. 1–653.

Casas Lizcano, D., & López, F. de A. (2015). Sistemas distribuidos. In UDIMA (Ed.), *Universidad Autónoma Metropolitana* (1er edició, Issue 1). http://www1.frm.utn.edu.ar/soperativos/Archivos/Sistemas_Distribuidos.pdf

Ćatović, A., Buzađija, N., & Lemes, S. (2022). Microservice development using RabbitMQ message broker. *Science, Engineering and Technology*, *2*(1), 30–37. https://doi.org/10.54327/set2022/v2.i1.19

Dervaux, J., Cormier, P., Moskovkin, P., Douheret, O., Konstantinidis, S., Lazzaroni, R., Lucas, S., & Snyders, R. (2019). Institutional Repository - Research Portal Dépôt Institutionnel - Portail de la Recherche. *Thin Solid Films*, *636*, 644–657. http://dx.doi.org/10.1016/j.tsf.2017.06.006

Hegde, R. G., & S, N. G. (2020). Low Latency Message Brokers. *International Research Journal of Engineering and Technology*, *May*, 2731–2738. www.irjet.net

Hohpe, G., & Woolf, B. (2003). *Enterprise Integration Patterns*. https://www.enterpriseintegrationpatterns.com/

Jiménez García, L. M., Puerto Manchón, R., & Payá Castellón, L. (2017). *Sistemas distribuidos: Arquitectura y aplicaciones (Spanish Edition)* (Universidad Miguel Hernández de Elche, Ed.; Vol. 0).

John, V., & Liu, X. (2017). *A Survey of Distributed Message Broker Queues*. http://arxiv.org/abs/1704.00411

Lazidis, A., Tsakos, K., & Petrakis, E. G. M. (2022). Publish–Subscribe approaches for the IoT and the cloud: Functional and performance evaluation of open-source systems. *Internet of Things (Netherlands)*, *19*(May). https://doi.org/10.1016/j.iot.2022.100538

Szajnfarber, Z., & Gralla, E. (2017). Qualitative methods for engineering systems: Why we need them and how to use them. *Systems Engineering*, *20*(6), 497–511. https://doi.org/10.1002/sys.21412