PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

FACULTAD DE CIENCIAS E INGENIERÍA

SECCIÓN INGENIERÍA DE LAS TELECOMUNICACIONES

Uso de una arquitectura basada en eventos como capa de comunicación para microservicios

TRABAJO DE INVESTIGACIÓN PARA OPTAR POR EL GRADO DE BACHILLER EN CIENCIAS CON MENCIÓN EN INGENIERÍA DE LAS TELECOMUNICACIONES

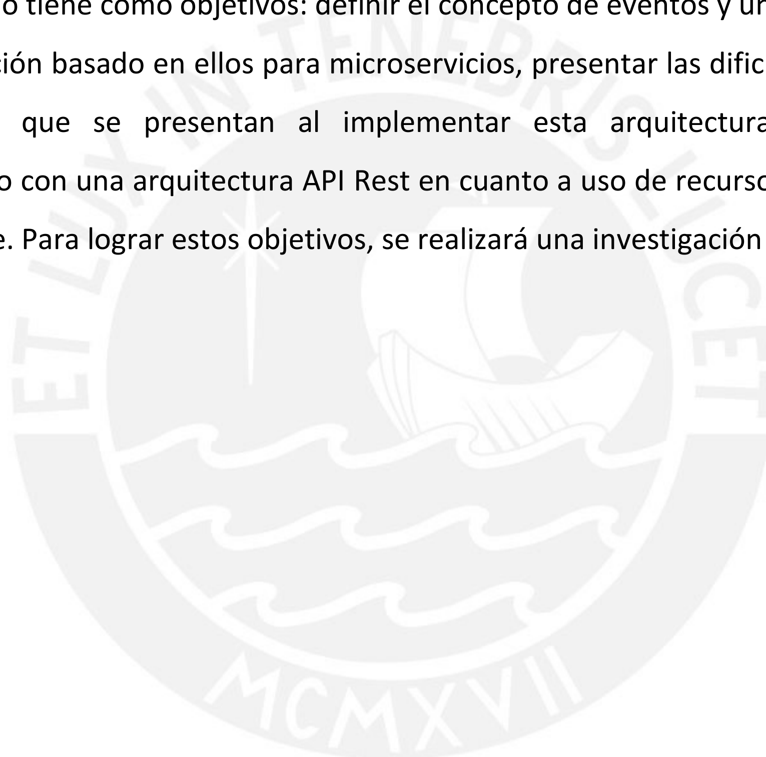
AUTOR
Alejandro Macedo Pereira

ASESOR
Jorge Benavides Aspiazu

Junio, 2020

Resumen

La presente investigación tiene como enfoque presentar el uso de eventos para la comunicación entre servicios/aplicaciones en un sistema distribuido, como es el de microservicios. Aunque exista la posibilidad de usar APIs Rest como arquitectura de comunicación, esta cuenta con la gran desventaja de ser un intercambio de mensajes síncrono y que, en una arquitectura con un gran número de microservicios, estos mensajes consumen un gran ancho de banda por ser comunicaciones 1 a 1. Por lo tanto, este trabajo tiene como objetivos: definir el concepto de eventos y una arquitectura de comunicación basado en ellos para microservicios, presentar las dificultades y posibles soluciones que se presentan al implementar esta arquitectura; y, finalmente, compararlo con una arquitectura API Rest en cuanto a uso de recursos y overhead que se produce. Para lograr estos objetivos, se realizará una investigación bibliográfica.



Índice

Resumen i

Índice.. ii

Lista de figuras iv

Lista de tablas v

Introducción.. 1

Capítulo 1. La problemática existente de la comunicación entre servicios en una arquitectura de microservicios 2

1.1. Antecedentes 2

1.1.1. El uso de aplicaciones y servicios en línea 2

1.1.2. La computación en la nube y su impacto en el software 4

1.1.3. Arquitectura de aplicación monolítica 5

1.1.4. *Application Programming Interface* (API) Rest 6

1.2. Arquitectura de microservicios 8

1.2.1. Problemáticas presentes 11

1.2.1.1. La comunicación entre servicios 11

1.2.1.2. Manejo de la información 12

Capítulo 2. El uso de una arquitectura basada en eventos como alternativa de solución 14

2.1. Eventos en una arquitectura de microservicios 14

2.1.1. Definición 14

2.1.2. Uso de esta arquitectura en microservicios como capa de comunicación .. 16

2.1.3. Uso de eventos para mantener la consistencia de la información 17

2.1.3.1. Sagas 17

2.1.3.2. Transacciones BASE 18

2.1.3.3. Transactional outbox 19

2.1.3.4. Event-sourcing 20

2.1.3.5. CQRS 22

Capítulo 3. Revisión de la literatura 24

3.1. Uso de arquitectura basada en eventos 24

Conclusiones 28
Bibliografía 29

Lista de figuras

Figura 1-1 - Esquema de una aplicación monolítica 6

Figura 1-2 Ejemplo de una especificación API Rest 7

Figura 1-3 Ejemplo de una aplicación en microservicios... 8

Figura 1-4 Diferencias en comunicaciones síncronas y asíncronas 12

Figura 1-5 Dificultades de contar con capas de almacenamiento independientes 13

Figura 2-1 Ejemplo de mensaje que se envía a varios servicios 16

Figura 2-2 Uso de mensajes para comunicaciones de solicitud/respuesta.................... 17

Figura 2-3 Ejemplo del uso del patrón de sagas 18

Figura 2-4 Uso de polling para implementar transactional outbox 20

Figura 2-5 Conversión al patrón event-sourcing 21

Figura 2-6 Ejemplo de CQRS de servicios que actualizan la vista de historial de órdenes 23

Figura 3-1 Tiempo de respuesta y rendimiento entre CRUD y event-sourcing.............. 26

Figura 3-2 Framework de guardado de eventos en V2X 27

Lista de tablas

Tabla 1-1 Ventas en retail de e-Commerce en Latinoamérica entre 2015 y 2019........... 3

Tabla 1-2 Actividades de diversión dentro del hogar peruano 3

Tabla 1-3 Características de una interfaz API Rest 7

Tabla 1-4 Comparación entre varios interfaces de comunicación web 8

Tabla 1-5 Comparación entre la arquitectura monolítica y de microservicios 11

Tabla 2-1 Características de una arquitectura basada en eventos 15

Tabla 2-2 Diferencia entre transacciones ACID y BASE 19

Tabla 2-3 Comparación entre Event Sourcing, Transaction Log y CQRS 23

Introducción

En los últimos años, las aplicaciones en línea se han vuelto una parte fundamental del día a día de las personas y que, año a año, el número de usuarios con los cuentan crece de forma exponencial. Como ejemplo, Netflix pasó de tener 25 millones de usuarios en el tercer trimestre del 2011 a más de 180 millones en el primer trimestre del 2020 [1]. Es en este contexto de gran crecimiento que se han diseñado nuevos paradigmas de construcción de software que se adapten, no solo a la demanda creciente de las aplicaciones que se crean con ella, sino también a las demandas de los desarrollares que requieren diseñar, implementar y desplegar nuevas funcionalidades en el menor tiempo posible y, en muchos casos, a escala global.

A partir de esas ideas es que nació la arquitectura de microservicios, donde una aplicación está separada en pequeños servicios que cumplen una función específica y son desarrolladas por equipos distintos y de forma independiente. Esto permite que se usen las mejores herramientas para cada uno de ellos, ya sea en lenguaje de programación, forma de almacenamiento de datos, etc. también brindando mejoras en cuanto a despliegue y escalabilidad, propiedad muy importante para el aumento considerable del número de usuarios [2].

Sin embargo, el tener una aplicación distribuida genera diversos problemas que no se habían presenciado en las aplicaciones no distribuidas, o también llamadas monolíticas. Uno de estos problemas es la comunicación eficiente entre servicios, ya que en una aplicación compleja pueden existir un gran número de ellos y usar una capa de comunicación API Rest, muy común en aplicaciones cliente-servidor, que se propuso como primera solución genera alto *overhead* por su propiedad de ser una comunicación 1 a 1.

Por lo tanto, en el presente trabajo de investigación, se va a estudiar una nueva arquitectura de comunicación entre servicios que está basada en eventos, sus desventajas y finalmente la revisión de las fuentes consultadas.

Capítulo 1. La problemática existente de la comunicación entre servicios en una arquitectura de microservicios

El presente capítulo describe los antecedentes, la definición de una arquitectura de microservicios y la problemática existente correspondiente a la comunicación entre servicios.

1.1. Antecedentes

1.1.1. El uso de aplicaciones y servicios en línea

Con la aparición de los teléfonos inteligentes y las redes de datos que permiten a las personas conectarse a Internet, el segmento de servicios en línea ha visto incrementado el número de usuarios e ingresos cada año, con ejemplos como el comercio electrónico que año tras año aumentan su cuota del mercado frente a las ventas tradicionales, pasando de representar un 10.4% de las ventas totales en 2017 al 14.1% en el 2020 con ventas mayores a 3.5 billones de dólares, y se espera que para el 2023 este porcentaje aumente al 22% con ventas mayores a 6.5 billones de dólares [3]. Otro sector con un incremento importante en usuarios e ingresos es el del entretenimiento digital con empresas como Netflix (películas bajo demanda) y Tencent (videojuegos), que logran

capitalizaciones bursátiles de 158 y 398 miles de millones de dólares respectivamente para junio del 2019 [4], y con Netflix que ganó más de 160 millones de suscriptores en el periodo 2011-2020 [1]. Este crecimiento también se ve reflejado en nuestro país, donde según la Tabla 1-2 casi el 60% de las actividades de diversión en los hogares peruanos corresponde a entretenimiento digital.

Tabla 1-1 Ventas en retail de e-Commerce en Latinoamérica entre 2015 y 2019

Número de ventas (millones de dólares)	Año
29.8	2015
36.9	2016
45.4	2017
54	2018
64.4	2019

Fuente: [5]

Tabla 1-2 Actividades de diversión dentro del hogar peruano

Actividades	Porcentaje
Ver televisión	54%
Conversar por Whatsapp	25%
Ver videos en Youtube	25%
Chatear en redes sociales	23%
Escuchar música por apps	20%
Leer noticias, blogs, libros por internet	17%

Fuente: [6]

Este acelerado crecimiento obliga a que las aplicaciones y servicios que ofrecen las empresas sean diseñados con capacidades que soporten accesos simultáneos y masivos de los usuarios, disponibilidad continua a lo largo del año, y con mejoras constantes que agreguen funcionalidades pedidas por los usuarios en cada actualización. Es en este contexto donde la computación en la nube se volvió una pieza fundamental para el desarrollo de este tipo de aplicaciones.

1.1.2. La computación en la nube y su impacto en el software

La computación en la nube es un modelo que brinda una plataforma con acceso a recursos compartidos de computación (máquinas, servidores, servicios, etc.) que pueden ser auto aprovisionados [7]. Actualmente existen muchos proveedores de computación en la nube, siendo los más grandes Amazon Web Services (AWS), Microsoft Azure, Alibaba Cloud y Google Cloud teniendo los dos primeros más del 60% de cuota del mercado en el 2018 [8]. Entre ellos, el que más destaca es AWS por el alto número de servicios disponibles, que van desde máquinas virtuales, almacenamiento en la nube, *blockchain*, *Internet of Things*, Inteligencia Artificial, etc. [9].

Otro factor contribuyente a la adopción de la computación en la nube es el menor costo de la infraestructura ya que este se encuentra delegado al proveedor de servicios en la nube, y solo se factura el consumo de acuerdo al tiempo de uso. Y aunque el costo de despliegue en las grandes plataformas como AWS sea más alto que en servidores propios para cierto tipo de aplicaciones empresariales, para casos de uso donde se requiera alta flexibilidad en cuanto a uso de recursos y presencia global, se use extensivamente servicios propios de la plataforma o no se cuente con la capacidad de pago para adquirir infraestructura propia es mucho más rentable usar una plataforma de servicio en la nube [10, pp. 34–35].

La aparición de este modelo influyó notablemente en la forma como se diseña y desarrolla el software. Aunque aún existan desarrollos de software monolítico, como se verá en la sección siguiente, la mayoría de aplicaciones están optando por migrar a una arquitectura que pueda aprovechar de mejor forma todos los recursos que otorga el modelo de computación en la nube, con lo que nacieron arquitecturas como microservicios y metodologías de desarrollo como *DevOps*, que permiten desarrollar cambios y mejoras de forma constante y, por ende, adaptarse más rápidamente a los cambios en el mercado [11].

1.1.3. Arquitectura de aplicación monolítica

Es una arquitectura con aplicaciones donde "la interfaz de usuario, las reglas del negocio y el código de acceso a la data están contenidas en un solo programa ejecutable y desplegados en una sola plataforma" [12].

Como indica [13], este tipo de aplicaciones tienen la ventaja de ser más fáciles de desarrollar porque la mayoría de entornos de desarrollo (*IDE*) son creados con este tipo de aplicaciones en mente; más simples de desplegar porque la aplicación está contenida en un solo paquete y más fáciles de escalar porque solo requiere correr una nueva instancia de esta aplicación [13]. Sin embargo, esto también trae consigo dificultades al momento que la aplicación comienza a crecer, sobre todo si se trata de una aplicación con miras a obtener una gran cantidad de usuarios. El gran tamaño del código fuente genera que sea mucho más difícil de mantener y agregar nuevas funcionalidades, por miedo a romper alguna funcionalidad de forma inesperada. Esto también implica que la seguridad de la aplicación se vea comprometida, porque implementar mejoras de seguridad es mucho más complicado. Además, si un nuevo desarrollador desea unirse al equipo de trabajo, el tiempo que demora en entender los módulos y funcionamiento del programa será completamente lineal con el tamaño del mismo. Y aun cuando sea mucho más sencilla de escalar, este se realiza en bloques de la aplicación completa y no solo de los módulos que están sobrecargados, lo que resulta en un uso ineficiente de los recursos de hardware.

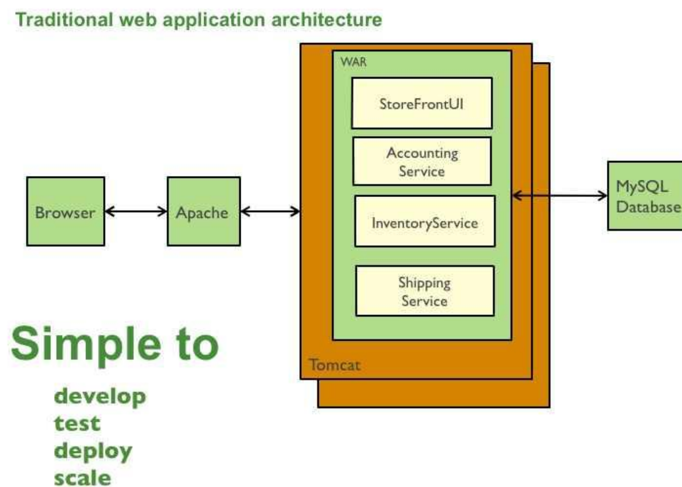


Figura 1-1 - Esquema de una aplicación monolítica

Fuente: [13]

En la Figura 1-1 se observa una aplicación monolítica Java que corre en un servidor Tomcat. Se observa que todos sus módulos se encuentran encapsulados en un solo paquete y se cuenta con una sola base de datos.

1.1.4. *Application Programming Interface* (API) Rest

Una aplicación necesita interfaces para poder comunicarse aplicaciones, usuarios o diferentes módulos dentro de la misma aplicación. Estas definen las operaciones que se invocan para realizar una determinada acción y, también ocultando la implementación que puede contener lógica o código propietario. Esto permite que los usuarios no tengan que preocuparse por cómo funciona la lógica de la interfaz, ni se vean afectados por cambios internos que puedan ocurrir [14, p. 68].

Aunque existan infinidad de interfaces y protocolos que se usan para la comunicación, una de las más usadas a través de Internet son las APIs Rest (*Representational State Transfer)*, el cual es un mecanismo de comunicación entre procesos o IPC (*Inter Process Communication*) que se usa, mayormente, a través del protocolo HTTP. Los elementos principales de un API Rest son los recursos, un objeto que se va a manipular, y los verbos que simbolizan las diversas operaciones que se van a realizar sobre ese recurso [14, p. 73].

Workspaces

POST /workspaces Create a new workspace

GET /workspaces/{id} Find an item by ID

PUT /workspaces/{id} Update an item by ID

DELETE /workspaces/{id} Delete an item by ID

POST /workspaces/all Lists tests by ids

GET /workspaces/member-of List workspaces whose logged user is a member

Figura 1-2 Ejemplo de una especificación API Rest

Fuente: [15]

Por ejemplo, en la Figura 1-2 se observa un ejemplo de la especificación de un API Rest usando HTTP sobre el recurso llamada *Workspaces*. Este documento contiene la información necesaria para que desarrolladores externos o usuarios puedan invocar las operaciones porque contiene las rutas y los parámetros necesarios para su ejecución. Una ventaja de usar el protocolo HTTP es que ya cuenta con verbos que son usados para sus solicitudes, por lo que se reutilizan para las operaciones Rest, además que son fácilmente auto descriptibles. Por ejemplo, el verbo *DELETE* generalmente se usa para una operación de borrado y el verbo *GET* se usa para la obtención de información.

Tabla 1-3 Características de una interfaz API Rest

Característica	**REST**
Estilo	Arquitectónico
Función	Basado en datos: Acceder a un recurso para datos
Formato de data	Incluye muchos formatos, entre ellos texto plano, HTML, XML y JSON
Seguridad	Soporta SSL y HTTPS
Banda ancha	Requiere menos recursos y es ligero
Caché de datos	Puede ser almacenada en caché

Fuente: [16]

Tabla 1-4 Comparación entre varios interfaces de comunicación web

Diferencias	REST	GraphQL	gRPC
Arquitectura	Muy conocida y popular	Ganando popularidad	Poco conocida
Comunicación	Síncrona: solo en protocolo HTTP	Síncrona/asíncrona en múltiples protocolos como HTTP, AMQP, MQTT	Asíncrona en protocolo HTTP/2
Diseño	Totalmente basado en HTTP	Basado en intercambio de mensajes	Basado en mensajes
Estándares de especificación	Usa OpenAPI, RAML, entre otros	Usa Schema	Usa protobuf (IDL)
Estándar de *payload*	JSON	JSON	Es serializado (protobuf)

Fuente: [17]

1.2. Arquitectura de microservicios

Esta arquitectura, a diferencia de la arquitectura de aplicación monolítica, separa cada uno de los componentes de la aplicación en distintos componentes donde cada uno de ellos cumple una función específica, cuenta con una lógica específica del negocio y cuenta con una capa de almacenamiento de información propia independiente de las demás [18].

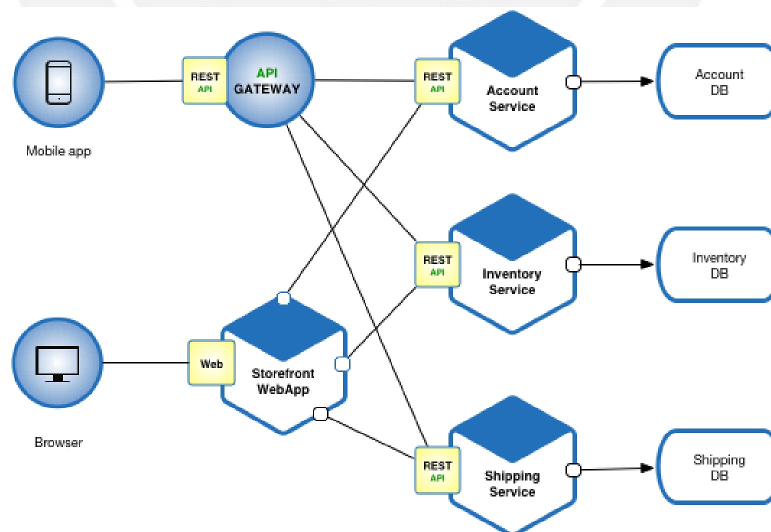


Figura 1-3 Ejemplo de una aplicación en microservicios

Fuente: [18]

En la Figura 1-3 se observa un ejemplo de una aplicación de comercio electrónico simple en una arquitectura de microservicios. Se observa que se cuentan con 4 microservicios y un API Gateway. La aplicación web es la que se encarga de mostrar la interfaz a un usuario que se conecta a la aplicación mediante un navegador, y se encarga de comunicarse con los servicios necesarios mediante un API Rest. Un punto importante a tener en cuenta es que los protocolos e interfaces de comunicación pueden ser completamente diferentes entre servicios, solo se necesita que cada una de estas interfaces se encuentre completamente definida y se cuenten con servicios que se encarguen única y exclusivamente de traducir la información entre los protocolos necesarios. Como se dijo anteriormente, cada microservicio usa las herramientas más eficientes para realizar su lógica interna, de acuerdo a los conocimientos del equipo de desarrollo y la forma cómo se maneja la información [2].

También se observa como cada uno de los servicios con lógica del negocio cuentan con bases de datos completamente independientes y la única forma en la que pueden conversar entre ellos es mediante sus interfaces de comunicación. Esto, a diferencia de una aplicación monolítica, resulta en diseño mucho más complejo para las relaciones de los datos de las tablas y la consistencia que se requiere de la información y que se analizará más adelante.

Otro elemento interesante de esa aplicación es el API Gateway, el cual sirve como punto de ingreso para la aplicación en general para aquellos equipos que no usen el navegador como interfaz de usuario, encargándose de comunicarse con los servicios necesarios y entregar la información en la forma óptima para el dispositivo. Netflix, por ejemplo, al ser una plataforma disponible en más de 800 diferentes tipos de dispositivos cuenta con un API Gateway complejo que se encarga de entregar el contenido de acuerdo al tipo de dispositivo que lo requiera, enviando una respuesta totalmente personalizada y optimizada [19]. Estas llamadas a los servicios muchas veces requieren usar protocolos distintos que el dispositivo no maneja (como puede ser gRPC), por lo que otra función del Gateway es la de traducir la data en protocolos que el dispositivo solicitante entienda [14, p. 262]. Aparte de encargarse de las llamadas a los servicios, el API Gateway también

cumple el rol de autenticador y autorizador de las solicitudes invocando a un servicio de autenticación, con lo que enlaza las solicitudes con los permisos de cada usuario utilizando tokens de acceso [14, p. 354].

Tener esta arquitectura con servicios distribuidos permite que los equipos de trabajo puedan dividirse de forma más concreta la implementación de cada uno de ellos, donde el código es mucho más pequeño y manejable que una aplicación monolítica, se pueden usar herramientas de integración y despliegue continuo para permitir los cambios y mejoras constantes, también permitiendo que nuevos desarrolladores se integran fácilmente al equipo y a las herramientas utilizadas. Sin embargo, esta arquitectura también cuenta con grandes desventajas, entre las que se encuentran la complejidad adicional de tener un sistema distribuido como es el manejo de información entre diversos servicios, la comunicación entre servicios y su eficacia, manejar solicitudes que puedan recorrer un gran número de servicios y lidiar con problemas como servicios no disponibles o caídos, etc. [18]. En la siguiente sección se analizarán algunos de los problemas mencionados.

Tabla 1-5 Comparación entre la arquitectura monolítica y de microservicios

Arquitectura Monolítica		**Arquitectura de Microservicios**	
A favor	En contra	A favor	En contra
Fácil de desarrollar	Tiende a volverse complejo y difícil de entender	Fácil de comprender	Se vuelve complejo al manejar sistemas distribuidos
Despliegue sencillo	Para un nuevo desarrollo, se requiere desplegar otra vez.	Facilidad para realizar cambios gracias a sus servicios independientes	Se complica hacer pruebas *End-to-End*
Testing sencillo	Se tiene que escalar la aplicación completa	Facilidad para escalar servicios	Implementar características con múltiples componentes requiere mayor atención
	Inicio más lento Punto único de fallo	Respuestas más rápidas a los requerimientos	Difícil de desarrollar

Fuente: [20]

1.2.1. Problemáticas presentes

1.2.1.1. La comunicación entre servicios

Un problema presenta al migrar a una arquitectura de microservicios es el de diseñar las comunicaciones e interacciones que se realicen entre los servicios. En una aplicación monolítica esto no es un problema grave porque las interacciones entre los módulos son a través de interfaces del propio lenguaje de programación que se usa; sin embargo, las interfaces que se presentan en los servicios son de red y se requiere usar un modelo que use de forma eficiente recursos como ancho de banda y tenga consideración de problemas como pueden surgir como latencia alta y caídas en los microservicios [21]. Aunque exista el modelo Rest para la comunicación entre procesos, su uso en microservicios cuenta con varias desventajas que lo vuelven ineficiente, siendo el más notorio su estilo de comunicación síncrono. Esto significa que, para realizar una solicitud, tanto el cliente como el servidor deben estar disponibles todo el tiempo que dura la operación y, además, no se puede realizar otras solicitudes que dependan de la respuesta de la primera. Otra desventaja es que solo soporta comunicaciones 1 a 1, por lo que una llamada que requiera de la información de varios servicios va a consumir el

ancho de banda de forma no eficiente [14, pp. 75–76]. En el siguiente capítulo se verá como una comunicación basada en eventos permite solucionar esta problemática usando un sistema de comunicación asíncrono y de 1 a muchos.

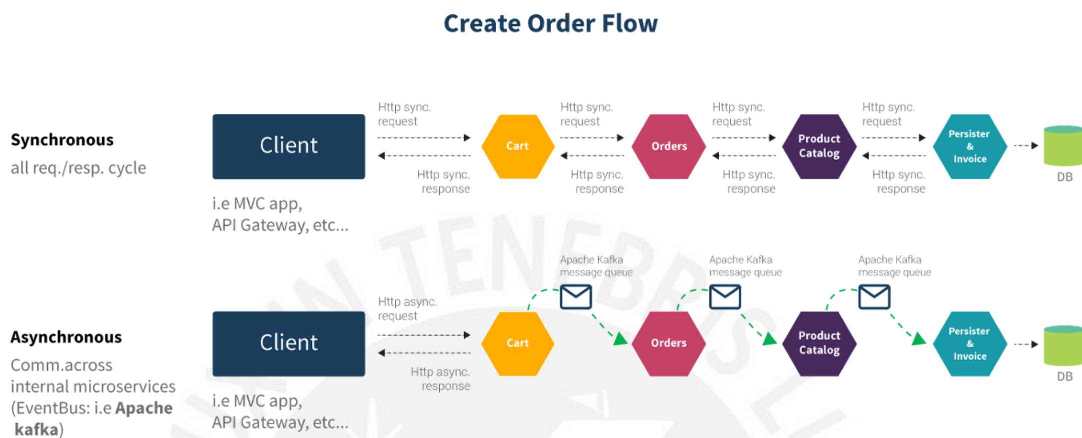


Figura 1-4 Diferencias en comunicaciones síncronas y asíncronas

Fuente: [22]

1.2.1.2. Manejo de la información

Otro de los problemas presentes al tratar de migrar a una arquitectura de microservicios es el del manejo de la información y las transacciones. Una transacción es un conjunto de pasos que se realizan para manejar la información en la capa de almacenamiento, como podría ser guardar o extraer información de una base de datos [23]. Como cada uno de los servicios cuenta con una capa de almacenamiento propia, el realizar transacciones que requiera operar sobre la información disponible entre diversos servicios requiere de lógica y complejidad adicional que no se observa en una aplicación monolítica. Esto sucede porque el tener una sola capa de almacenamiento de datos permite realizar las transacciones de forma atómica (que todos los pasos de la operación se realicen como una única operación), consistente (que los estados de la data se mantengan antes y luego de la transacción), aisladas (que la data usada por una transacción no puede ser usada hasta que acabe la transacción) y durables (que los cambios que se realice la transacción no se van a revertir, aun cuando ocurra un fallo del sistema) [24]. Estas propiedades se les denomina *ACID* (*Atomicity, Consistency,*

Isolation, Durability) y están presentes en las implementaciones de los diferentes *softwares* de bases de datos. Sin embargo, en una arquitectura de microservicios no se puede asegurar que estos principios se cumplan por diversos motivos como latencia de la red de comunicación, fallo en algún microservicio, pérdida de mensajes, etc. En el siguiente capítulo se mostrará cómo el uso de eventos ayuda a solucionar esta problemática.

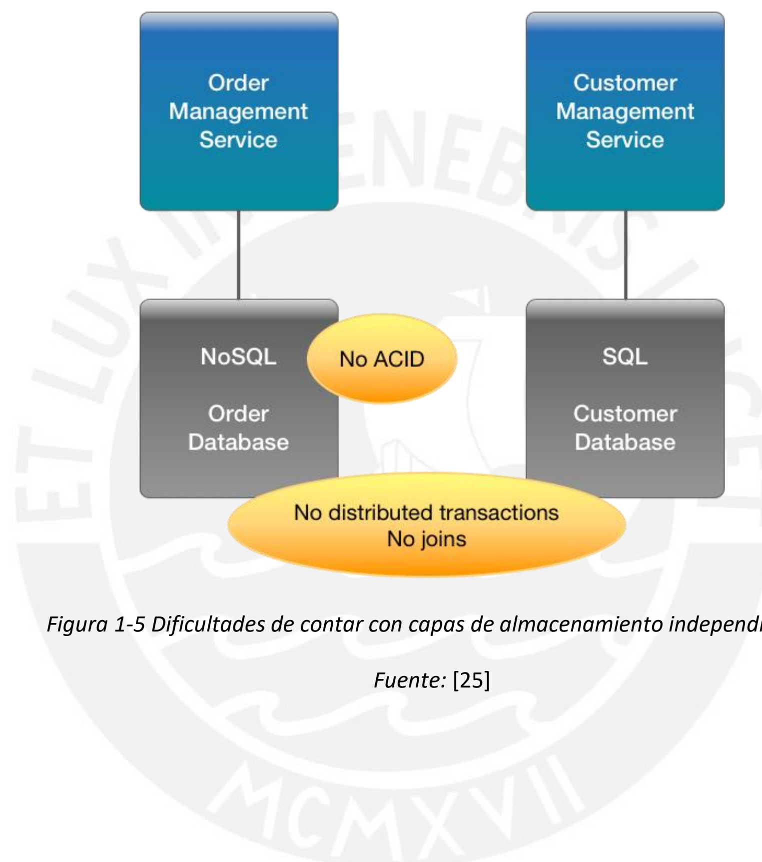


Figura 1-5 Dificultades de contar con capas de almacenamiento independientes

Fuente: [25]

Capítulo 2. El uso de una arquitectura basada en eventos como alternativa de solución

En este capítulo se presentarán a los eventos y una arquitectura basado en ellos para microservicios, las problemáticas que solucionan, así como sus ventajas y desventajas frente al uso de una comunicación por medio de APIs Rest.

2.1. Eventos en una arquitectura de microservicios

2.1.1. Definición

En términos simples, un evento es cualquier cambio de estado que se genera por alguna acción externa y que es reconocida por el software que se está ejecutando. Por ejemplo, la compra de un producto dentro de una aplicación de comercio electrónico genera un evento por la creación de una nueva entidad Compra que tiene que ser procesada o que un teclado registre que se está presionando una tecla [26].

La noción de eventos dio paso a una nueva arquitectura de construcción de aplicaciones, donde la creación de un evento da paso al envío de mensajes que son enviados a consumidores que procesan la información enviada. Estos consumidores no requieren

realizar *polling* constante al productor de eventos; más bien, estos llegan de forma asíncrona reduciendo considerablemente el uso de recursos [27]. Un elemento adicional e importante que aparece en esta arquitectura es el *broker* de mensajes, el cual se encarga de recolectar y distribuir los mensajes. Esto permite que los productores y consumidores se encuentren completamente desacoplados permitiendo una mayor escalabilidad y resistencia a fallos, además de que este sistema de comunicación permite que el mensaje de un evento pueda enviarse a un gran número de consumidores al mismo tiempo, reduciendo notablemente el consumo de ancho de banda y reduciendo significativamente el número de mensajes que viajan a través de la red, reduciendo la latencia total de la aplicación [28].

Con la aparición del *broker* se puede argumentar que se generan varias desventajas al momento de usar esta arquitectura, ya que este elemento se convierte en punto de falla importante o de cuello de botella. Sin embargo, la mayoría de implementaciones de *brokers* son sistemas altamente escalables y altamente eficientes, tales como *RabbitMQ, Apache Kafka, AWS Kinesis*, etc. Pero para lograr tal confiabilidad estos sistemas deben ser correctamente configurados y desplegados, resultando en complejidad adicional frente a una arquitectura de comunicación síncrona [14, p. 94].

Tabla 2-1 Características de una arquitectura basada en eventos

Características	**Arquitectura basada en eventos**
Enfoque	Eventos, enrutamiento de mensajes
Patrón	*Publish-and-Subscribe*
Comunicación	Asíncrono
En la práctica	Provee flexibilidad a la organización. Es adaptable
Trascendencia	Explosivo interés a partir del 2011
Definiciones	Solo se comparte la semántica de cómo se envían los mensajes
Aplicable a	Interacción horizontal, ideal para integrar múltiples sistemas o mantener procesos autónomos
Implementación	Utilizando colas de mensajes

Fuente: [29, p. 27]

2.1.2. Uso de esta arquitectura en microservicios como capa de comunicación

La arquitectura basada en eventos puede ser fácilmente usada dentro de una aplicación desarrollada usando microservicios, donde los servicios se registran como consumidores o productores y se registra un nuevo servicio adicional formado por el clúster de *brokers* en configuración de alta disponibilidad. Estos otorgan canales de comunicación a los que los servicios se suscriben para que puedan enviar y recibir los mensajes a los cuales se encuentran interesados. Por ejemplo, en la Figura 2-1 se ve el ejemplo de dos microservicios suscritos al canal de "Usuarios Actualizados". El contar con este broker permite que el servicio de "Perfil de Usuario" envíe un mensaje a N servicios al mismo tiempo, eliminando también la necesidad de que se espere la respuesta del evento producido. Y si ocurre el caso de que, por ejemplo, el servicio de "Carrito" se encuentra no disponible, el *broker* se encarga de mantener ese mensaje en caché a la espera de que el servicio se encuentre nuevamente disponible para su consumo [28].

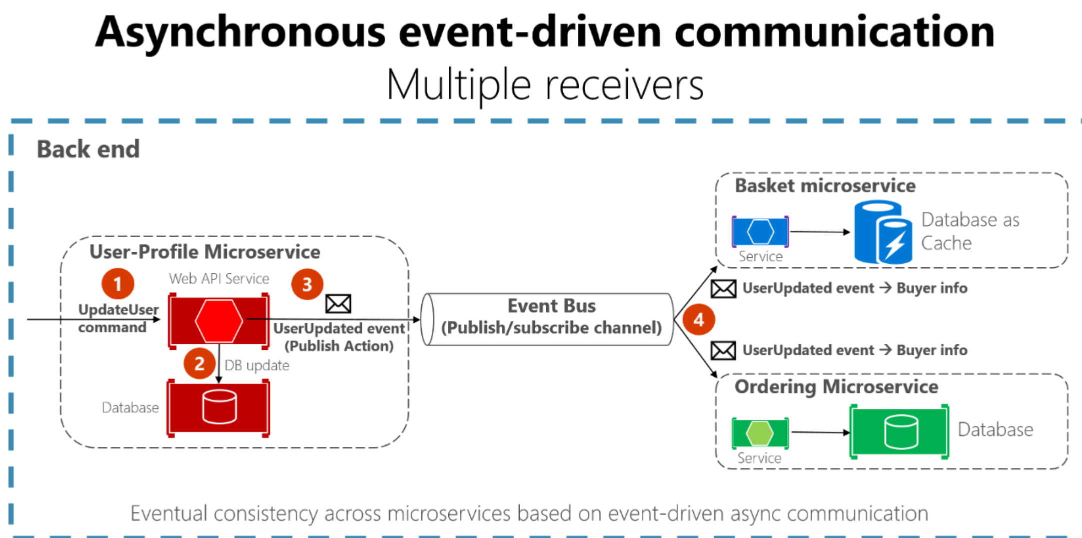


Figura 2-1 Ejemplo de mensaje que se envía a varios servicios

Fuente: [28]

Además, si se quiere un sistema que reemplace completamente a las peticiones síncronas generadas por un modelo como Rest, es necesario que puedan enviar mensajes de forma bidireccional como un sistema de solicitud/respuesta. Esto es posible si se configuran canales de comunicación como en la Figura 2-2, permitiendo así que se

eliminen completamente las llamadas síncronas, eliminando la problemática descrita en el capítulo anterior.

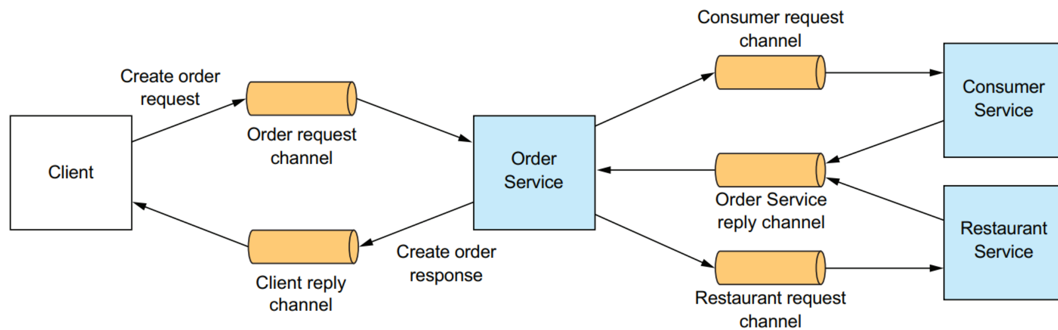


Figura 2-2 Uso de mensajes para comunicaciones de solicitud/respuesta

Fuente: [14, p. 105]

2.1.3. Uso de eventos para mantener la consistencia de la información

Como se explicó en el capítulo anterior, mantener la consistencia de la información se vuelve complicado en un sistema distribuido porque esta se encuentra distribuida en diferentes capas de almacenamiento de cada servicio. Una alternativa de solución es la de usar una transacción distribuida; sin embargo, la necesidad de que todos los servicios involucrados se encuentren disponibles en la duración completa de la transacción lo que ocasiona también que esa data no pueda ser usado para otras solicitudes, generando alta latencia en la respuesta del servicio la convierte en una solución poco viable.

2.1.3.1. Sagas

Con la introducción de una arquitectura de eventos y su capa de comunicación correspondiente se puede llegar a que los servicios logren un estado de consistencia y consenso en el resultado de la transacción usando el patrón de Sagas. Este patrón define un nuevo de transacciones locales que se realizan con la capa de almacenamiento local, lo cual permite que todas tengan las propiedades *ACID*, para luego generar eventos que permiten que los otros servicios comiencen sus propias transacciones con la información enviada por el servicio anterior. Si alguna transacción local falla, entonces se genera un evento que desencadenan operaciones complementarias a las antes realizadas para evitar que se tenga información en estado no consistente [14, p. 114], [30, p. 484].

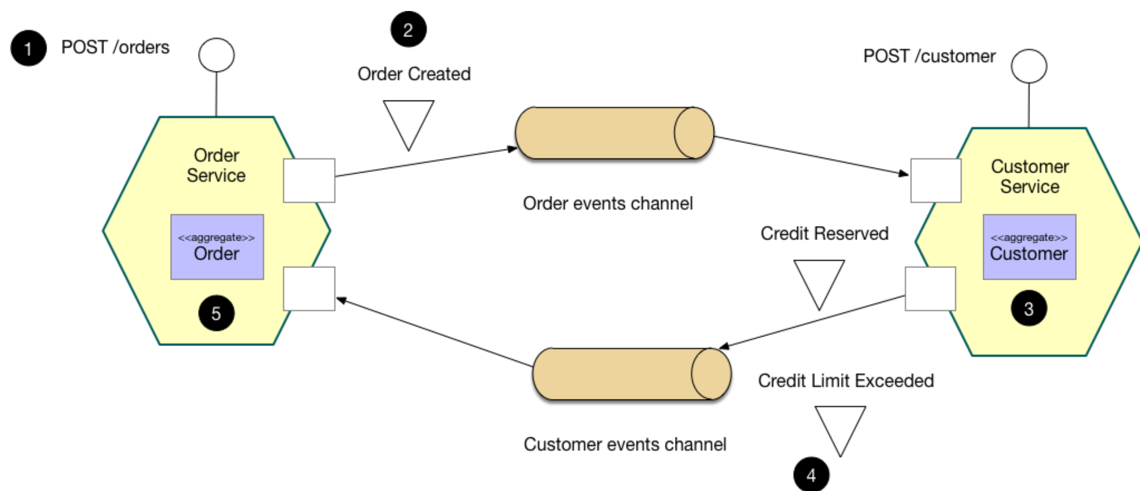


Figura 2-3 Ejemplo del uso del patrón de sagas

Fuente: [31]

En la Figura 2-3 se muestra un ejemplo del uso de sagas para la creación de una orden de venta. Cuando el usuario realiza la acción mediante el API Rest de órdenes, este produce que en la base de datos del servicio de Órdenes se registre una nueva Orden con un estado de “Procesando” y se envíe un evento de orden creada al servicio de Cliente. Este servicio se encarga de verificar que el cliente cuente con el crédito suficiente para realizar la compra y guarda en su base de datos la reserva de crédito exitosa o falla por falta de crédito. Cada una de estas posibilidades generan eventos que son procesados nuevamente por el servicio de Órdenes, ya sea aprobando la orden de venta o cancelando la transacción con su operación complementaria [31].

2.1.3.2. Transacciones BASE

Algo importante a tener en cuenta es que este patrón no convierte a la transacción en de tipo *ACID* porque la propiedad de aislamiento no se cumple. Esto se debe a que cada uno de los servicios actualiza su información de forma independiente y, mientras no se complete la transacción, queda de forma parcialmente incompleta y puede ser usada por otras solicitudes. Además, de acuerdo al teorema de los sistemas distribuidos CAP de *Consistency* (la información que se entrega es la misma en cualquier nodo al que se consulte)*, Availability* (cualquier solicitud que se realice recibirá una respuesta así se cuenten con nodos fallidos) *and Partition tolerance* (los nodos deben ser inmunes a fallos en la red como pérdida de paquetes) [32] el cual indica que no se puede asegurar

estas tres propiedades al mismo tiempo, por lo que cuando ocurre una falla en la red del sistema se tiene que escoger entre contar con disponibilidad o consistencia. A partir de este teorema nació un nuevo tipo de transacciones llamado *BASE* el cual cuenta con las propiedades de ser *Basically Available* (se garantiza la disponibilidad sobre la consistencia), *Soft State* (el estado del sistema podrá cambiar durante la transacción porque eventualmente será consistente) y *Eventual Consistency* (se permite que el sistema se encuentre en un estado parcialmente inconsistente porque se asegura que, cuando dejen de recibirse actualizaciones que modifiquen la información, el estado pasará a ser consistente). Este tipo de transacciones ha permitido que el número de transacciones en los sistemas distribuidos aumente a miles de millones manteniendo latencias de acceso bajas, permitiendo un gran escalamiento y disminuyendo el uso de recursos al no tener que verificar constante la consistencia de la data. Además, nacieron nuevos tipos de capas de almacenamiento que se basan completamente en este modelo, las llamadas base de datos NoSQL tales como MongoDB, Cassandra y muchos otros [33], [34, pp. 4–7].

Tabla 2-2 Diferencia entre transacciones ACID y BASE

ACID	**BASE**
Es acrónimo de atomicidad, consistencia, aislamiento y durabilidad	Es acrónimo de *Basically Available*, *Soft state* y *Eventually consistent*
Son pesimistas: Garantiza que se debe lograr la consistencia al final de cualquier operación	Son optimistas: No brindan garantías de que se debe lograr la consistencia al final de la operación
Requerimiento básico en bases de datos RDBMS	Requerimiento básico en bases de datos NoSQL
Implementación compleja de todas sus propiedades	Implementación simple de todas sus propiedades
Proporciona fiabilidad	No proporciona fiabilidad

Fuente: [35, p. 13]

2.1.3.3. Transactional outbox

Otro problema que se presenta al momento de implementar el patrón de sagas es el de la atomicidad que se requiere cuando se modifica la base de datos local y se envía el mensaje del evento generado. El no cumplirse podría un caso donde el servicio pueda fallar entre ambas operaciones generando inconsistencia en la operación total. La forma

en la que se logra esto es mediante el uso del patrón *transactional outbox*, el cual usa la capa de almacenamiento para guardar los mensajes que deben enviarse y se cuenta con un elemento adicional que se encarga de extraer y propagar los mensajes al *broker*. Esto se debe a que, como las bases de datos locales pueden realizar guardados atómicos, no se corre el riesgo de que el mensaje no se logre enviar. Luego, se cuentan con dos técnicas que permiten extraer los mensajes de la base de datos: *polling* y *log trailing*. El primer caso es sencillo, el elemento encargado de publicar los mensajes (llamado *relay*) se encarga de leer constantemente la base de datos y envía los mensajes nuevos que van apareciendo. El segundo, en cambio, se encarga de leer los registros (*logs*) que genera la base de datos cuando se registra alguna acción y notifica los cambios que se generen como eventos al *broker*. Esta última es mucho más eficiente, pero requiere de lógica y conocimiento adicional de cómo funcionan las bases de datos para poder implementarla correctamente [14, pp. 98–99], [36].

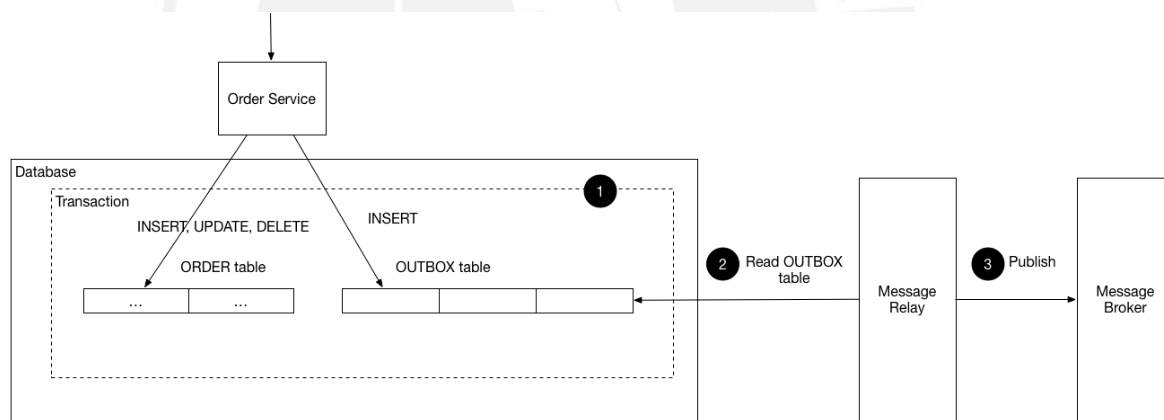


Figura 2-4 Uso de polling para implementar transactional outbox

Fuente: [36]

2.1.3.4. Event-sourcing

Otra forma de mantener la consistencia de la información es usando un patrón completamente diferente llamado *event-sourcing*. Este patrón difiere de la arquitectura en la forma cómo se guarda la información. En una arquitectura como la descrita en la sección 1.2.1.2, cada servicio contaba con su propia capa de almacenamiento donde persistían la información de una entidad en su ese estado. Si existía alguna operación que cambie alguna propiedad de la entidad, entonces el estado anterior se perdía y

sobrescribía perdiendo toda clase de registro histórico que puede ser usado para auditorías o regulaciones de protección de datos. Aunque esta funcionalidad puede ser implementada, es muy costosa y complicada de realizarla correctamente, con la posibilidad de contar con fallos o *bugs* [14, p. 185].

En cambio, *event-sourcing* usa un concepto totalmente diferente. En vez de guardar cada entidad como su último estado, cada entidad se guarda como el conjunto de eventos que generan ese estado. Si se requiere saber su estado, simplemente se ejecutan cada uno de los eventos guardados hasta llegar al punto en el tiempo deseado, por lo que se mantiene un registro histórico completo del ciclo de vida de cada una de las entidades guardadas. Para poder implementar este patrón, es necesario contar con un *event-store*, lugar donde se almacenan todos eventos y se publican o consumen los mensajes generados. Por lo tanto, es muy común que implementaciones de *brokers*, como Apache Kafka, cuenten con un sistema de registro de eventos ya incorporados permitiéndoles funcionar como ambos sin usar software adicional. Además, puede usarse en conjunto con el patrón de Sagas permitiendo que las transacciones que abarcan diversos servicios se completen satisfactoriamente [14, pp. 186–187].

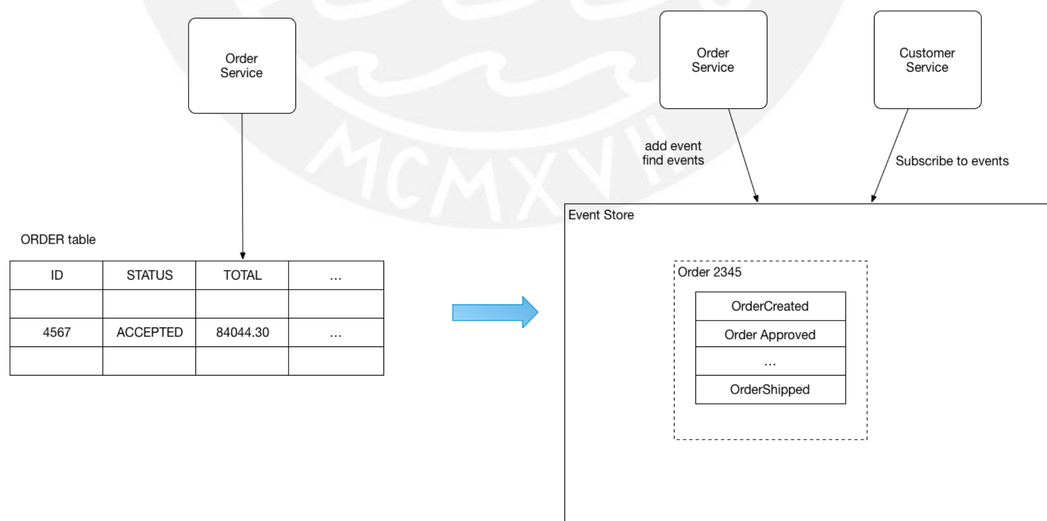


Figura 2-5 Conversión al patrón event-sourcing

Fuente: [37]

2.1.3.5. CQRS

Como todos los patrones, este también cuenta con varias desventajas, siendo la más importante el problema de la visualización de los datos. Esto ocurre porque las consultas para conocer el estado de una entidad requieren repetir todos los eventos que lo componen, resultando en alta latencia y uso innecesario de los recursos. Por ello, se usa el patrón *CQRS* (*Command Query Responsibility Segregation*) el cual define que exista una separación de acuerdo al tipo de operación que se realice en la capa de almacenamiento. Si es una operación de escritura se cuenta con una base de datos especializada en ello y las operaciones de lectura se realizan en otra aparte. Con ello, se puede pueden tener el *event-store* como el lugar donde se almacenan los eventos, y contar una base de datos aparte donde se actualice, mediante eventos, la información de cada entidad en un formato que sea el apropiado para las consultas [14, p. 202].

Lo interesante de este patrón es que puede usarse fuera del contexto de *event-sourcing* y usarlo en la arquitectura anterior, beneficiándose de tener que realizar consultas a cada una de las bases de datos locales disminuyendo el tiempo de respuesta de las consultas. Además, pueden usarse bases de datos especializadas en la lectura de datos o tipos de datos con los que trabaja el servicio, aumentando así también el desempeño final de este [14, pp. 228–230], [38].

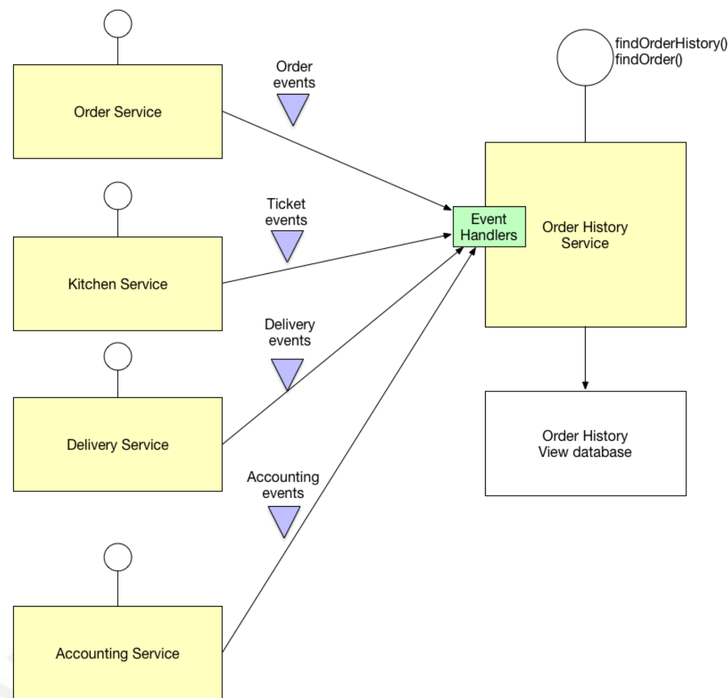


Figura 2-6 Ejemplo de CQRS de servicios que actualizan la vista de historial de órdenes

Fuente: [38]

Tabla 2-3 Comparación entre Event Sourcing, Transaction Log y CQRS

Atributos	***Event Sourcing***	**Transactional Log**	**CQRS**
Propósito	Captura el estado en un *event-store* que contiene eventos de dominio	Exporta eventos de cambio desde el *transaction log*	Usa eventos de dominio para generar proyecciones de datos
Tipo de evento	Evento de dominio	Evento de cambio	Evento de dominio
Fuente de verdad	*Event-store*	*Transaction log*	Depende de la implementación
Límite	Aplicación	Sistema	Aplicación o Sistema
Consistencia	No tiene (Solo escribe en el *event-store*)	Tablas: Fuertemente consistente Evento de cambio: Eventualmente consistente	Eventualmente consistente
¿Re ejecución de eventos para conocer el estado?	Si	Si	Depende de la implementación

Fuente: [39]

Capítulo 3. Revisión de la literatura

En este capítulo se realizará una revisión de trabajos realizados sobre arquitecturas basada en eventos y otros patrones asociados.

3.1. Uso de arquitectura basada en eventos

Como primera publicación, los autores de [40] implementan una aplicación de planificación de vuelos para la empresa Lufthansa usando esta arquitectura y los patrones CQRS para aprovechar de mejor manera los recursos que existen en la nube. Además, se apoyan del Manifiesto de los Sistemas Reactivos para el diseño de su arquitectura, el cual propone que sea: dirigido por mensajes, elástico, resiliente y responsivo.

La implementación del sistema fue posible gracias al uso de Akka, un conjunto de herramientas que provee un *middleware* facilitador de la comunicación dirigida por mensajes entre entidades. La aplicación en sí fue escrita en el lenguaje Scala y se eligió una interfaz Rest con el formato de mensajes JSON para la comunicación. Finalmente,

se eligió a Apache Kafka y Apache Cassandra como los sistemas de persistencia y base de datos de los mensajes enviados, usados en modos de alta escalabilidad [40, p. 4].

El sistema fue puesto a prueba para evaluar el impacto que tiene la escalabilidad en los sistemas de lectura y escritura que se crean por el uso de CQRS, para lo cual se desplegó la aplicación en una nube OpenStack y la carga fue realizada por la herramienta Gatling. Los resultados observados fueron los siguientes: en el sistema de lectura se observó que la aplicación era capaz de escalar linealmente con la carga recibida sin incrementar el tiempo de respuesta mientras se tenga en cuenta la carga que es capaz de procesar cada uno de los nodos y ajustar los parámetros de escalado correctamente; mientras que en el sistema de escritura se observó que la adición de nuevos nodos para el procesamiento de los usuarios permite reducir considerablemente el tiempo de respuesta y el número de mensajes no procesados [40, pp. 5–6].

Un segundo trabajo presentado en [41] muestra un estudio del patrón CQRS y el impacto de su uso en la implementación de un software ERP. Como cada uno de los clientes de estas empresas tienen modelos de negocio distintos, contar con un diseño que permita obtener la máxima flexibilidad es de suma importancia para la empresa desarrolladora. Por tanto, se realizaron charlas informativas y seminarios taller del patrón de diseño y luego se realizaron entrevistas donde se preguntó acerca de la flexibilidad que otorga CQRS para la implementación del software que desarrollan. Finalmente, se discutió con los arquitectos de software de la empresa la implementación que se iba a realizar, donde se observa un diseño usando herramientas patrones como *event-sourcing* y *event-store* descritos anteriormente.

En el trabajo presentado en [42] se muestra la implementación de un sistema de puntos para una aplicación de venta online, donde se explica que el uso de un patrón tradicional como CRUD se vuelve completamente ineficiente cuando se requiere de un sistema escalable y resiliente, por lo que se opta por usar el patrón CQRS con una arquitectura basada en eventos. Luego de presentarse la implementación, se muestra una figura comparativa entre los tiempos de respuesta y número de solicitudes procesadas de los patrones CRUD y *event-sourcing*, donde se observa que el rendimiento del segundo

antes de sufrir degradamiento notable en los tiempos de respuesta es alrededor de 450 solicitudes por segundo, el doble que el primero.

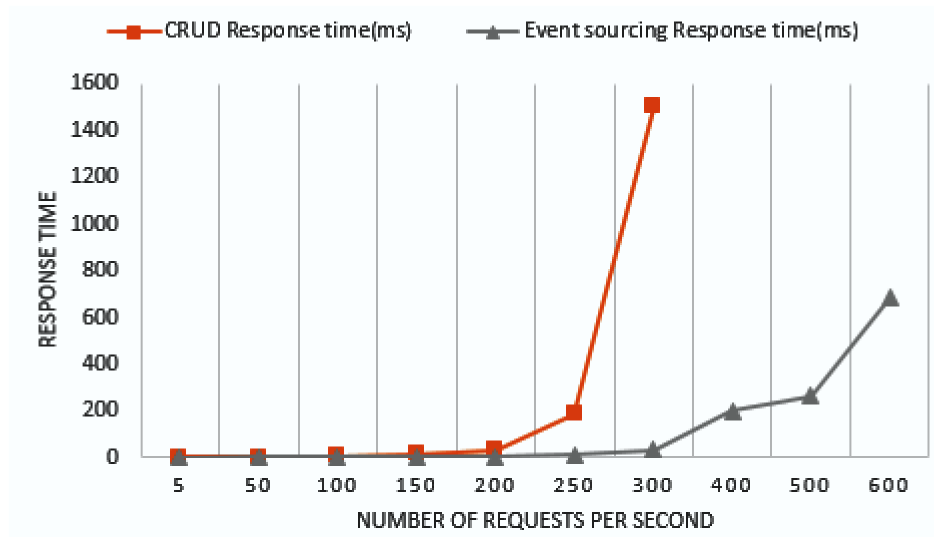


Figura 3-1 Tiempo de respuesta y rendimiento entre CRUD y event-sourcing

Fuente: [42]

En otra publicación presentada por [43] se muestra la implementación de una plataforma para aplicaciones *serverless*, donde la infraestructura y los entornos de ejecución del código son entregados como servicios, donde el eje fundamental es la posibilidad de usar la computación retroactiva. Este modelo permite realizar cambios en la historia pasada de la aplicación, para observar el cambio de los estados que ocasiona. Este tipo de computación puede aplicarse usando patrones como *event-sourcing* ya que se cuenta con la historia completa de los eventos producidos y con comandos que permiten manipular a voluntad el estado en el que se encuentran. Por tanto, en el trabajo se presentan los diferentes desafíos a solucionar, el modelo conceptual a implementar y una descripción de la arquitectura de la plataforma final.

Como publicación final se estudiará el presentado en [44], donde se analiza una propuesta de solución para reproducir y evitar los accidentes en la conducción autónoma mediante el framework llamado V2X. Como los autos autónomos contarán con diversos sensores para permitir su tránsito sin la intervención de un conductor, el trabajo propone usar almacenar los eventos usando *event-sourcing* para que, cuando

ocurra un accidente, estos se analicen gracias a modelos de *machine-learning* que permitan analizar e identificar las causas y predecirlas en un futuro.

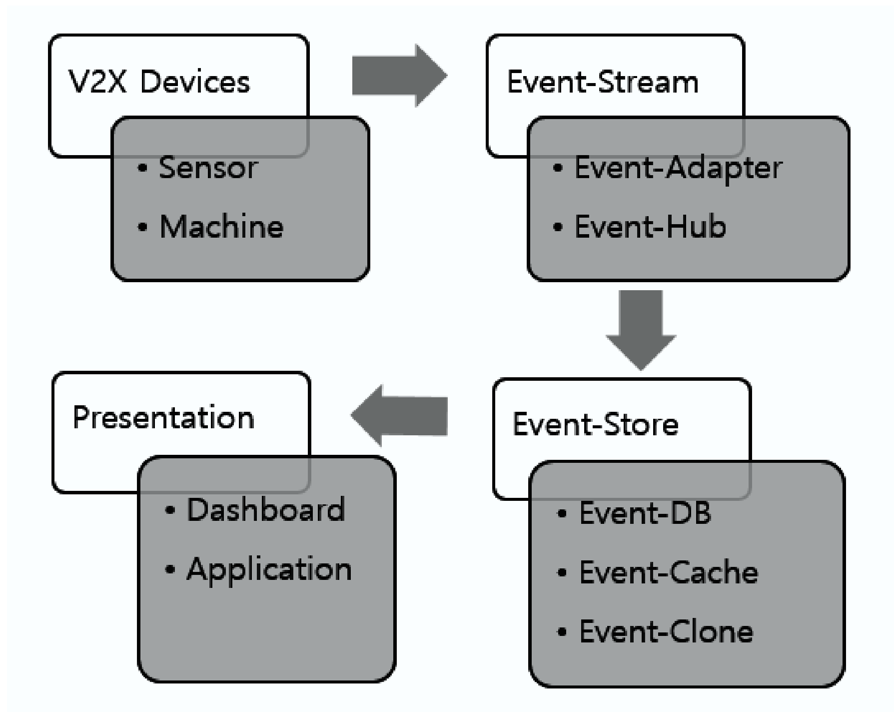


Figura 3-2 Framework de guardado de eventos en V2X

Fuente: [44]

Conclusiones

En este trabajo de investigación se presentó el uso de una arquitectura basada en eventos para solucionar dos problemáticas presentes en una arquitectura de microservicios, el problema de la comunicación entre servicios y la consistencia dela información en un entorno distribuido.

Se estudiaron las diversas ventajas y desventajas de los diversos patrones que presenta una arquitectura basada en eventos y como solucionan las problemáticas descritas anteriormente, para finalmente realizar una revisión de la literatura donde se presentan publicaciones que hacen uso de esta arquitectura para diversos casos.

Por lo tanto, se concluye que el uso de esta arquitectura es recomendable cuando se desarrollen aplicaciones con arquitecturas distribuidas, como es el caso de microservicios, porque permite aprovechar los beneficios de plataformas *cloud* y obtener mejor rendimiento cuando se necesite un sistema con escalabilidad y resiliencia, como se muestra en la Figura 3-1.

Bibliografía

[1] A. Watson, “Number of Netflix paying streaming subscribers worldwide from 3rd quarter 2011 to 1st quarter 2020”, *Statista*, 2020. https://www.statista.com/statistics/250934/quarterly-number-of-netflix-streaming-subscribers-worldwide/ (consultado may 20, 2020).

[2] C. Richardson, “Introduction to microservices”, *NGINX*, 2015. https://www.nginx.com/blog/introduction-to-microservices/ (consultado ene. 20, 2020).

[3] D. Chaffey, “Ecommerce growth statistics - UK, US and Worldwide forecasts”, nov. 20, 2019. https://www.smartinsights.com/digital-marketing-strategy/online-retail-sales-growth/ (consultado abr. 06, 2020).

[4] Statista, “Top internet companies: global market value 2019”, *Statista*, jun. 07, 2019. https://www.statista.com/statistics/277483/market-value-of-the-largest-internet-companies-worldwide/ (consultado mar. 20, 2020).

[5] BlackSip, “Reporte de Industria: El e-Commerce en Perú 2019”, Lima, 2019. Consultado: mar. 07, 2020. [En línea]. Disponible en: https://asep.pe/wp-content/uploads/2019/08/Reporte-de-industria-del-eCommerce-Peru-2019-eBook.pdf.

[6] Ipsos Perú, “El 60% de actividades de diversión preferidas por hogares ya es digital”, *Gestión*, Lima, p. 12, mar. 25, 2020.

[7] T. Boillat y C. Legner, “From On-Premise Software to Cloud Services: The Impact of Cloud Computing on Enterprise Software Vendors’ Business Models”, 2013, doi: 10.4067/S0718-18762013000300004.

[8] K. Costello y L. Goasduff, “Gartner Says Worldwide IaaS Public Cloud Services Market Grew 31.3% in 2018”, *STAMFORD, Conn.*, 2019. https://www.gartner.com/en/newsroom/press-releases/2019-07-29-gartner-

says-worldwide-iaas-public-cloud-services-market-grew-31point3-percent-in-2018 (consultado abr. 06, 2020).

[9] Amazon Web Services, “Overview of Amazon Web Services”, 2020.

[10] Happi, “AWS Instances Versus Physical Servers”, 2018. Consultado: may 20, 2020. [En línea]. Disponible en: https://www.happi.io/wp-content/uploads/2019/01/aws-instances-versus-physical-servers.pdf.

[11] L. Morgan, “DevOps and Microservices”, *Datamation*, ene. 29, 2019. https://www.datamation.com/applications/devops-and-microservices.html (consultado may 08, 2020).

[12] Office of Information Technology Services, “ITS Technical Glossary (Archived)”, *ITS*, 2001. https://web.archive.org/web/20070902151937/http://www.its.state.nc.us/Information/Glossary/Glossm.asp (consultado abr. 15, 2020).

[13] C. Richardson, “Monolithic Architecture pattern”, *Microservices.io*. https://microservices.io/patterns/monolithic.html (consultado mar. 20, 2020).

[14] C. Richardson, *Microservices Patterns with examles in Java*, 2a ed. Shelter Island: Manning Publications Co., 2019.

[15] J. Loisel, “Rest API Testing With JMeter (Step by Step Guide)”, *OctoPerf*, abr. 23, 2018. https://octoperf.com/blog/2018/04/23/jmeter-rest-api-testing/ (consultado mar. 08, 2020).

[16] C. Wodehouse, “SOAP vs. REST: A Look at Two Different API Styles”, *Upwork*, abr. 19, 2017. https://www.upwork.com/hiring/development/soap-vs-rest-comparing-two-apis/ (consultado mar. 08, 2020).

[17] R. Rocha, “APIs REST, GraphQL o gRPC - ¿Quién gana este partido?”, *Sensedia*, may 02, 2019. https://sensedia.com/es/apis/apis-rest-graphql-o-grpc-

comparacion/ (consultado mar. 08, 2020).

[18] C. Richardson, “Microservice Architecture pattern”, *Microservices.io*. https://microservices.io/patterns/microservices.html (consultado mar. 20, 2020).

[19] D. Jacobson, “Embracing the Differences : Inside the Netflix API Redesign”, *Netflix TechBlog*, jul. 09, 2012. https://netflixtechblog.com/embracing-the-differences-inside-the-netflix-api-redesign-15fd8b3dc49d (consultado may 23, 2020).

[20] R. İnanç, “TIL-6: Which is better? Microservice vs. Monolithic Architecture in a Nutshell”, *Noteworthy - The Journal Blog*, dic. 02, 2019. https://blog.usejournal.com/til-6-microservices-vs-monolithic-architecture-b58488b846dd (consultado mar. 08, 2020).

[21] Microsoft, “Communication in a microservice architecture”, ene. 30, 2020. https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture (consultado feb. 20, 2020).

[22] Walking Tree Technologies, “Inter-service communication in Microservices”, *Medium*, ago. 16, 2018. https://medium.com/@walkingtreetech/inter-service-communication-in-microservices-c54f41678998 (consultado may 20, 2020).

[23] Java EE Team, “What Is a Transaction?”, 2010. https://docs.oracle.com/javaee/5/tutorial/doc/bncii.html (consultado may 23, 2020).

[24] MariaDB Team, “ACID: Concurrency Control with Transactions - MariaDB Knowledge Base”. https://mariadb.com/kb/en/acid-concurrency-control-with-transactions/ (consultado mar. 08, 2020).

[25] Eventuate, “Asynchronous microservices”. https://eventuate.io/whyeventdriven.html (consultado may 23, 2020).

[26] Amazon, “Event-Driven Architecture”. https://aws.amazon.com/es/event-driven-architecture/ (consultado may 20, 2020).

[27] E. Mortoray, “What is event programming?”, *Musing Mortoray*. https://mortoray.com/2017/06/26/what-is-event-programming/ (consultado may 08, 2020).

[28] Microsoft Team, “Asynchronous message-based communication”, *.NET microservices - Architecture e-book*, sep. 20, 2018. https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/asynchronous-message-based-communication (consultado may 08, 2020).

[29] N. Estrada, “Arquitectura Orientada a Eventos para Redes de Sensores aplicada a control de tráfico”, Universidad Técnica Federico Santa María, Santiago, 2017.

[30] M. Stefanko, O. Chaloupka, y B. Rossi, “The saga pattern in a reactive microservices environment”, en *ICSOFT 2019 - Proceedings of the 14th International Conference on Software Technologies*, 2019, pp. 483–490.

[31] C. Richardson, “Sagas”, *Microservices.io*. https://microservices.io/patterns/data/saga.html (consultado mar. 20, 2020).

[32] IBM Team, “What is the CAP Theorem?”, *IBM*. https://www.ibm.com/cloud/learn/cap-theorem (consultado may 23, 2020).

[33] C. Roe, “ACID vs. BASE: The Shifting pH of Database Transaction Processing”, *Dataversity*, mar. 01, 2013. https://www.dataversity.net/acid-vs-base-the-shifting-ph-of-database-transaction-processing/ (consultado feb. 20, 2020).

[34] C. Roe, “THE QUESTION OF DATABASE TRANSACTION PROCESSING: AN ACID, BASE, NOSQL PRIMER”, 2013.

[35] B. Garg, “Schema Extraction of Document Database - MongoDB”, Thapar

University, Patiala, 2015.

[36] C. Richardson, “Transactional outbox”, *Microservices.io*. https://microservices.io/patterns/data/transactional-outbox.html (consultado mar. 20, 2020).

[37] C. Richardson, “Event sourcing”, *Microservices.io*. https://microservices.io/patterns/data/event-sourcing.html (consultado mar. 20, 2020).

[38] C. Richardson, “Command Query Responsibility Segregation (CQRS)”, *Microservices.io*. https://microservices.io/patterns/data/cqrs.html (consultado mar. 20, 2020).

[39] E. Murphy, “Distributed Data for Microservices — Event Sourcing vs. Change Data Capture”, *Debezium*, feb. 10, 2020. https://debezium.io/blog/2020/02/10/event-sourcing-vs-cdc/ (consultado may 20, 2020).

[40] A. Debski, B. Szczepanik, M. Malawski, S. Spahr, y D. Muthig, “A scalable, reactive architecture for cloud applications”, *IEEE Softw.*, vol. 35, núm. 2, pp. 62–71, mar. 2018, doi: 10.1109/MS.2017.265095722.

[41] J. Kabbedijk, S. Jansen, y S. Brinkkemper, “A case study of the variability consequences of the CQRS pattern in online business software”, en *ACM International Conference Proceeding Series*, 2012, pp. 1–10, doi: 10.1145/2602928.2603078.

[42] Y. Zhong, W. Li, y J. Wang, “Using event sourcing and CQRS to build a high performance point trading system”, en *ACM International Conference Proceeding Series*, feb. 2019, pp. 16–19, doi: 10.1145/3317614.3317632.

[43] D. Meißner, B. Erb, F. Kargl, y M. Tichy, “Retro-λ: An event-sourced platform for serverless applications with retroactive computing support”, en *DEBS 2018 - Proceedings of the 12th ACM International Conference on Distributed and Event-*

Based Systems, jun. 2018, pp. 76–87, doi: 10.1145/3210284.3210285.

[44] S. Han y J. I. Choi, "V2X-Based event acquisition and reproduction architecture with event-sourcing", en *ACM International Conference Proceeding Series*, ene. 2020, pp. 164–167, doi: 10.1145/3379247.3379290.