UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Event-Manager – Manejador centralizado de notificaciones

Autores: Mariano Wadel (N° Alumno 9685/5)
Director: Dra. Patricia Bazán
Carrera: Licenciatura en Sistemas – Plan 2012

Resumen

Las grandes empresas de tecnología y que son referentes del mercado en sectores como el comercio electrónico (e-commerce), redes sociales, plataformas de contenido online, y sistemas de trabajo colaborativo, son algunas de las organizaciones que hoy impulsan el constante cambio y crecimiento de Internet, en un contexto de ofrecer cada vez más y mejores servicios, lo más barato y rápido posible. Para poder hacer frente a este desafío, estas empresas, también llamadas .com, dividen su arquitectura tecnológica en cientos de componentes, los cuales intercambian mensajes para coordinar procesos de negocio. En este contexto, puede resultar costoso y poco práctico controlar que estas comunicaciones sean recibidas y procesadas adecuadamente. En este trabajo se propone una aplicación que provea una capa de abstracción para que las aplicaciones emisoras deleguen la correcta entrega y procesamiento de sus notificaciones y que a la vez mejore la visibilidad y control de estas notificaciones en toda la organización.

Palabras Claves

Esquema Publisher/Subscriber, Service Oriented Architecture, Entreprise Service Bus, Cluster, Notificaciones, Sistemas Distribuídos, Web Services, Java, Hazelcast, MongoDB, NoSQL, Procesos de negocio

Conclusiones

La implementación de la solución propuesta permitió, por un lado, reducir los tiempos de desarrollo de las distintas aplicaciones que se comunican dentro de la organización, al brindar una capa de abstracción que se encarga del manejo de notificaciones y por otro lado mejorar la visibilidad y eficacia de todo el sistema en general gracias al debido control de los errores causados por notificaciones perdidas y errores de recepción de las mismas.

Trabajos Realizados

- *Diseño e implementación de una aplicación Java centralizada encargada de la recepción de las notificaciones de los publicadores y el reenvío de ellas a los suscriptores con reintentos en caso de errores y persistencia de ellas en MongoDB.*
- *Diseño y desarrollo de una interfaz web que permite monitorear y controlar la actividad de las notificaciones en tiempo real, así como registrar nuevos publicadores y suscriptores y configurarlos*

Trabajos Futuros

- *Desarrollar e integrar en la aplicación existente un módulo que permita configurar alertas según distintas condiciones de error.*
- *Desarrollar e integrar un módulo de estadísticas que permita a los usuarios tener una visión global del flujo de las comunicaciones.*
- *Soportar nuevos protocolos de comunicación*
- *Permitir aislar a los usuarios por grupos para controlar el acceso a la información del sistema.*

Fecha de la presentación: **Septiembre 2016**

Tesina de grado

Event-Manager – Manejador centralizado de notificaciones

Autor: Mariano Wadel

Directora: Dra. Patricia Bazán

Fecha: Septiembre 2016

Facultad de Informática

UNIVERSIDAD NACIONAL DE LA PLATA

Índice general

Introducción ... 1
1.1 Motivación ... 1
1.2 Propuesta de solución ... 2
1.3 Concepto de *notificación*. Soporte de diferentes protocolos ... 3
1.4 Resumen del capítulo ... 4
Conceptos generales ... 5
2.1 Service Oriented Architecture (SOA) ... 5
2.2 Protocolos de red e intercambio de datos ... 6
2.2.1 Extensible Markup Language (XML) ... 6
2.2.2 JavaScript Object Notation (JSON) ... 7
2.2.3 Hypertext Transfer Protocol (HTTP) ... 8
2.2.4 Representational State Transfer (REST) ... 11
2.3 Enterprise Service Bus (ESB) ... 13
2.4 *Clúster* de computadoras ... 15
2.4.1 Intercambio de datos y memoria compartida ... 16
2.5 Bases de datos NoSQL (Not Only SQL) ... 18
2.5.1 Introducción ... 18
2.5.2 Ventajas y desventajas ... 19
2.5.3 Principales diferencias con las bases de datos SQL ... 20
2.5.4 Tipos de BD NoSQL ... 20
2.5.5 MongoDB ... 21
2.6 Frameworks Web ... 23
2.6.1 Introducción ... 23
2.6.2 Spring MVC ... 23
2.7 Resumen del capítulo ... 25
Trabajos previos relacionados ... 26
3.1 *Thialfi* (Adya, Cooper, Myers y Piatek, 2011) ... 26
3.1.1 Introducción ... 26

3.1.2 Solución 26
3.1.3 Conclusión del trabajo 27
3.2 *A Publish / Subscribe Mechanism for Web Services* (Tcherevik, 2003) 28
3.2.1 Introducción 28
3.2.2 Solución 28
3.2.3 Conclusión del trabajo 29
3.3 Resumen del capítulo 29
Event-Manager 30
4.1 Objetivo 30
4.2 Características funcionales 30
4.2.1 Recepción y reenvío de notificaciones 30
4.2.2 Manejo de errores y reintentos 31
4.2.3 Registro de parámetros e información adicional 31
4.2.4 API (Interfaz de Programación de Aplicaciones) 31
4.2.5 Interfaz gráfica 31
4.2.6 Tolerancia a fallos 32
4.2.7 Recuperación ante fallos 32
4.3 Descripción del funcionamiento 33
4.3.1 Descripción del flujo 33
4.4 Comparación entre Event-Manager y ESB 35
4.5 *Benchmarking* (pruebas de banco) 37
4.6 Resumen del capítulo 39
Implementación en Despegar.com 40
5.1 Contexto del problema 40
5.2 Implementación 41
5.3 Ejemplo de funcionamiento 42
5.4 Ventajas de usar Event-Manager en Despegar.com 43
5.5 Conclusiones del capítulo 45
Extensibilidad 47
6.1 Módulo de alertas 47
6.2 Módulo de estadísticas 47

6.3 Grupos de usuarios....47
6.4 Soporte para otros protocolos....48
6.5 Resumen del capítulo....49
Conclusión....50
Referencias bibliográficas....53
Modelo de datos....55
Manejo de concurrencia....58

Índice de Figuras

FIGURA 1.1: Funcionamiento de Event-Manager con varios protocolos. 4
FIGURA 2.1: Estructura de una lista de contactos representada en formato XML. 6
FIGURA 2.2: Estructura de una lista de contactos representada en formato JSON. 7
FIGURA 2.3: Requerimiento y respuesta en una petición HTTP. 8
FIGURA 2.4: Composición de una URL. 8
FIGURA 2.5: Composición de un requerimiento REST. 11
FIGURA 2.6: Respuesta del servidor en formato JSON a la invocación del ejemplo. .. 12
FIGURA 2.7: Ejemplo de la diversidad de sistemas que un ESB puede interconectar. 14
FIGURA 2.8: Clúster de servidores web con balanceador de carga. 17
FIGURA 2.9: Estructura de un documento. 21
FIGURA 2.10: Ejemplo de un replica set de MongoDB. 22
FIGURA 3.1: Arquitectura del sistema Thialfi. 27
FIGURA 3.2: Esquema de un broker con publicador y suscriptores. 29
FIGURA 4.1: Diagrama de actividad de Event-Manager. 34
FIGURA 4.2: Prueba de rendimiento de Event-Manager. 38
FIGURA 5.1: Flujo de notificaciones en la creación de una reserva. 42
FIGURA 6.1: Configuración de un sistema de Event-Manager por equipo. 48
FIGURA 6.2: Configuración de Event-Manager unificado para varios equipos. 49
FIGURA A1.1: Diagrama de modelo de Event-Manager. 55
FIGURA A2.1: Flujo de notificaciones con una cola única de procesamiento 58
FIGURA A2.2: Flujo de notificaciones con una cola de procesamiento por cada suscriptor 59
FIGURA A2.3: Código del servicio encargado de obtener el ejecutor para cada suscripción. 60
FIGURA A2.4: ThreadPoolTaskExecutor. 61
FIGURA A2.5: Diagrama de secuencia del procesamiento de una notificación. 62

Índice de Tablas

TABLA 2.1: Métodos HTTP más utilizados. 9
TABLA 2.2: Comparativa entre configuración tradicional y configuración clúster. 15
TABLA 4.1: Fallos y mecanismos de contingencia. 32
TABLA 4.2: Comparativa entre Mule ESB y Event-Manager. 36
TABLA 5.1: Antes y después de Event-Manager en Afiliadas Despegar.com 45

Capítulo 1

Introducción

Este trabajo presenta "Event-Manager", un sistema de notificaciones basado en Java, desarrollado en Despegar.com para aplicaciones con envío y recepción de múltiples mensajes. Event-Manager entrega de forma casi inmediata las notificaciones, brinda una visión en tiempo real de las comunicaciones tanto a través de una interfaz gráfica como por API, y provee mecanismos de contingencia frente a fallos. Aunque originalmente fue pensado para HTTP, Event-Manager soporta múltiples protocolos de entrada y salida, y otorga estabilidad, redundancia y distribución de carga debido a que está diseñada para trabajar en arquitectura de clúster.
En este capítulo se verán algunos de los inconvenientes frecuentes que motivaron el desarrollo propuesto por este trabajo, junto con algunos casos de uso. Además, se definirá formalmente qué es una notificación en un contexto de publicador/suscriptores (más comúnmente conocido como *Publisher/Subscriber*, en inglés).

1.1 Motivación

Las grandes empresas de tecnología y que son referentes del mercado en sectores como el comercio electrónico (*e-commerce*), redes sociales, plataformas de contenido online, y sistemas de trabajo colaborativo, son algunas de las organizaciones que hoy impulsan el constante cambio y crecimiento de Internet, en un contexto de ofrecer cada vez más y mejores servicios, lo más barato y rápido posible. Para poder hacer frente a este desafío, estas empresas, también llamadas *.com,* dividen su arquitectura tecnológica en cientos de servicios, tanto privados (para uso intra-organización) como públicos, para proporcionar soluciones a las diferentes necesidades. De hecho, cada vez más están optando por *atomizar* estos servicios, con componentes cada vez más chicos [1], que en vez de hacer muchas cosas (a costa de una alta complejidad y un difícil mantenimiento), hagan pocas cosas (pero bien) y se mantengan simples. Esta filosofía crece de la mano del hardware barato, y las redes de enlace cada vez más rápidas y con menor latencia. Ahora bien, dicho concepto trae de la mano un problema serio: la creciente interacción entre distintos componentes intra-empresa. La información ya no está toda en un solo lugar, ni tampoco los procesos que la manipulan. En este contexto surge la necesidad de abstraer lo más posible todo comportamiento repetido y proveer ciertas garantías al desarrollador para evitar que éste malgaste tiempo y recursos para implementar su propia solución ante un problema recurrente. Un ejemplo de lo dicho es el de las notificaciones que se deben enviar de un proceso a otro. Estas notificaciones se envían a través de enlaces de diversa calidad, no todos los receptores están siempre disponibles al momento de recibir la notificación.

Por ejemplo, supongamos que existe un proceso **A** que está interesado en eventos generados por el proceso **B**. De la misma forma, existe otro proceso **C**, interesado en las notificaciones de **A** (y por qué no, también de **B**). Sin una solución centralizada, cada proceso debería implementar su notificador. ¿Qué ocurre si alguno de los procesos no puede recibir la notificación? Cada proceso debería tener su propia implementación de un notificador, que haga el mejor esfuerzo para entregar la notificación, y notifique a un responsable si la misma no se puede entregar.
Usando como base el análisis anterior, resulta adecuado pensar en un sistema centralizado que elimine la necesidad de implementar un notificador por cada aplicación, que unifique la recepción y manejo de notificaciones de un sistema y que provea diferentes mecanismos para la contingencia de errores.

1.2 Propuesta de solución

La solución propuesta en esta tesina al problema del manejo de notificaciones es construir una aplicación central que actúe como intermediaria entre los suscriptores y los notificadores, y que tome toda la responsabilidad de la recepción y el reenvío de las notificaciones oportunamente a los suscriptores que correspondan. Esta aplicación permitirá hacer más sencilla la programación del resto de las aplicaciones y desligar a los desarrolladores de la responsabilidad de programar un notificador para cada una de ellas. Además, brindará un respaldo adicional frente a problemas como interrupción de servicio o fallos de red.
La aplicación a construir para la gestión de notificaciones debe reunir las siguientes características:

- **Centralizada:**
  La solución debe centralizar el flujo de notificaciones entrantes y salientes con el objetivo de mejorar la trazabilidad y simplificar el control de las mismas.

- **Transparente**:
  Las aplicaciones que utilicen Event-Manager deben poder comunicarse sin notar que existe en un manejador de eventos intermedio.

- **Estable:**
  Debe tener una cantidad de fallos baja o nula dado el rol central que tiene dentro de la organización.

- **De alta disponibilidad:**
  Por su criticidad, debe estar diseñada para trabajar en redundancia, con 2 o más instancias, y su base de datos debe estar replicada. No debe tener *downtime*.

- **Tolerante a fallos:**
  En caso de caída o fallas de sistema, se debe recuperar automáticamente a un estado consistente, que permita continuar su normal funcionamiento.

- **Bien documentada:**
  Su alcance, su configuración, y sus especificaciones deben estar adecuadamente detallados para que el cliente tenga pleno conocimiento de las funcionalidades que provee, y de las que no.

- **Eficiente:**
  Debe hacer buen uso de los recursos. No puede haber demoras innecesarias al momento de entregar las notificaciones.

**El título original de este trabajo era "Event Manager - Manejador de notificaciones REST". Sin embargo, se decidió eliminar la palabra "REST", debido a que se está incurriendo en una mezcla de conceptos. "REST" es un estándar de acceso a recursos a través de la web (como veremos más adelante), y si bien los servicios y notificaciones sobre los que este trabajo está basado guardan una estrecha relación con este estándar, se propone una solución que no distingue tipos de notificaciones.*

1.3 Concepto de *notificación*. Soporte de diferentes protocolos

En informática, una notificación es "*un mensaje enviado a través de uno o más canales a un conjunto de receptores conocidos (consumidores) avisando la ocurrencia de un evento determinado*" [5]. Usualmente, los receptores se suscriben previamente para recibir notificaciones acerca de tal evento.
Como se mencionó anteriormente, Event-Manager surge por necesidad en una organización moderna, como tantas otras, que utiliza REST, JSON y HTTP para el intercambio de recursos e información, por lo que no es de sorprender que hoy una *notificación* consista de, por ejemplo, un POST HTTP que luego de arribar a Event Manager se retransmita a los suscriptores adecuados. No obstante, el diseño interno de la aplicación está pensado para que el núcleo se abstraiga tanto del punto de entrada como de los de salida de una notificación. Esto quiere decir que, por ejemplo, se podría recibir una notificación por HTTP POST y retransmitirse por MQTT (Figura 1.1). Basta con implementar el conector respetando la interfaz de los notificadores. De hecho, este ejemplo no es al azar, y surgió a partir de un nuevo requerimiento interno dentro de Despegar.com, que plantea que Event Manager reciba un POST HTTP y a partir de esto genere un POST HTTP y un evento MQTT, demostrando la extensibilidad de la aplicación.

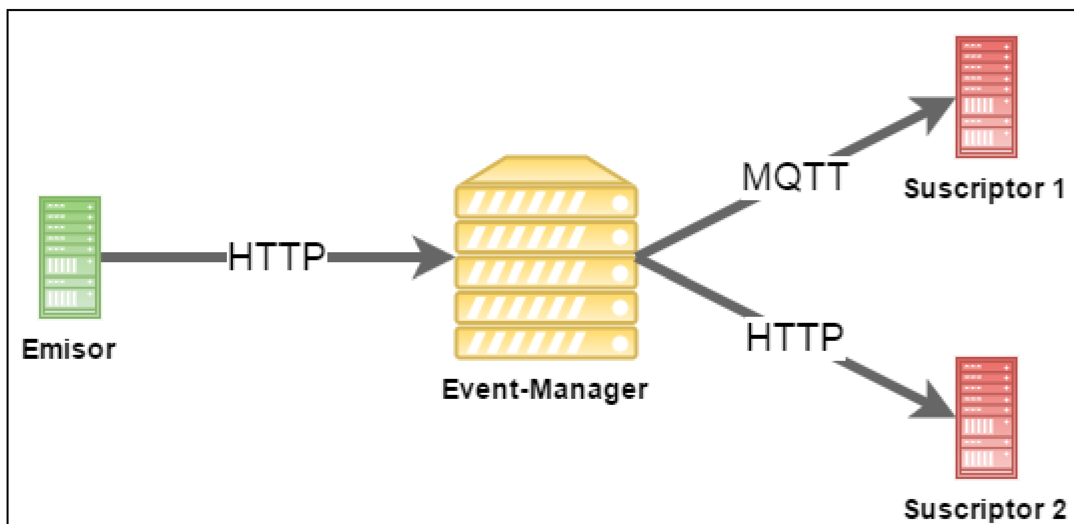

FIGURA 1.1: Funcionamiento de Event-Manager con varios protocolos.

1.4 Resumen del capítulo

En este capítulo se enumeraron algunos de los problemas que motivaron el desarrollo de este trabajo, y el contexto en que estas surgieron. Posteriormente, se presentó la primera definición general de Event-Manager, así como su funcionalidad, las bases que lo sustentan, y su extensibilidad. También se describieron las características deseables con las que debe contar un sistema para poder ser considerado una solución fiable a estos problemas. Finalmente, se presentó una definición formal de notificación y se comenta sobre las posibilidades de extensibilidad de Event-Manager. En los siguientes capítulos se verá más detalladamente cómo Event-Manager resuelve las problemáticas planteadas.

Capítulo 2

Conceptos generales

El objetivo del presente capítulo es ilustrar al lector el contexto en el que se desarrolla este trabajo y presentar algunos conceptos y tecnologías subyacentes para comprender las razones que motivaron el desarrollo de Event-Manager, la aplicación propuesta por esta tesina, y su funcionamiento.

2.1 Service Oriented Architecture (SOA)

El concepto de Arquitectura Orientada a Servicios es el que describe la composición y organización de las aplicaciones dentro de las instituciones y empresas para las que fue diseñado Event-Manager, por lo que resulta útil que sea lo primero que se analice.
La arquitectura SOA establece un marco de diseño para la integración de aplicaciones independientes de manera que desde la red pueda accederse a sus funcionalidades, las cuales se ofrecen como servicios. La forma más habitual de implementarla es mediante Servicios Web, una tecnología basada en estándares e independiente de la plataforma, con la que SOA puede descomponer aplicaciones monolíticas en un conjunto de servicios e implementar esta funcionalidad en forma modular.
¿Qué es un servicio exactamente? Un servicio es una funcionalidad concreta que puede ser descubierta en la red y que describe tanto lo que puede hacer como el modo de interactuar con ella. Desde la perspectiva de la empresa, un servicio realiza una tarea concreta: puede corresponder a un proceso de negocio tan sencillo como introducir o extraer un dato como "Código del Cliente". Pero también los servicios pueden acoplarse dentro de una aplicación completa que proporcione servicios de alto nivel, con un grado de complejidad muy superior –por ejemplo, "introducir datos de un pedido"-, un proceso que, desde que comienza hasta que termina, puede involucrar varias aplicaciones de negocio.
La estrategia de orientación a servicios permite la creación de servicios y aplicaciones compuestas que pueden existir con independencia de las tecnologías subyacentes. En lugar de exigir que todos los datos y lógica de negocio residan en un mismo ordenador, el modelo de servicios facilita el acceso y consumo de los recursos de IT a través de la red. Puesto que los servicios están diseñados para ser independientes, autónomos y para interconectarse adecuadamente, pueden combinarse y recombinarse con suma facilidad en aplicaciones complejas que respondan a las necesidades de cada momento en el seno de una organización.
Las aplicaciones compuestas (también llamadas "dinámicas") son las que permiten a las empresas mejorar y automatizar sus procesos manuales, disponer de una visión consistente de sus clientes y socios comerciales y orquestar sus procesos de negocio para que cumplan

con las regulaciones legales y políticas internas. El resultado final es que las organizaciones que adoptan la orientación a servicios pueden crear y reutilizar servicios y aplicaciones y adaptarlos ante los cambios evolutivos que se producen dentro y fuera de ellas, y con ello adquirir la agilidad necesaria para ganar ventaja competitiva. [5]

2.2 Protocolos de red e intercambio de datos

En este apartado describiremos algunos de los protocolos de red y de intercambio de datos/formateo de texto más populares utilizados para la interacción entre servicios. Estos protocolos actualmente juegan un rol protagónico en las comunicaciones de las SOAs, y es importante describirlos, al menos resumidamente, para poder comprender un poco mejor qué sucede "*under the hood*" en ciertas configuraciones de arquitecturas orientadas a servicios.

2.2.1 Extensible Markup Language (XML)

XML es un lenguaje de marcas desarrollado por el World Wide Web Consortium (W3C), utilizado para almacenar y transmitir datos en forma legible.

```
<ContactsResponse xmlns:i="http://www.w3.org/20
  <Contacts>
    <Contact>
      <Email>demis.bellot@gmail.com</Email>
      <FirstName>Demis</FirstName>
      <LastName>Bellot</LastName>
    </Contact>
    <Contact>
      <Email>steve@apple.com</Email>
      <FirstName>Steve</FirstName>
      <LastName>Jobs</LastName>
    </Contact>
    <Contact>
      <Email>steve@microsoft.com</Email>
      <FirstName>Steve</FirstName>
      <LastName>Ballmer</LastName>
    </Contact>
    <Contact>
      <Email>eric@google.com</Email>
      <FirstName>Eric</FirstName>
      <LastName>Schmidt</LastName>
    </Contact>
    <Contact>
      <Email>larry@oracle.com</Email>
      <FirstName>Larry</FirstName>
      <LastName>Ellison</LastName>
    </Contact>
  </Contacts>
</ContactsResponse>
```

FIGURA 2.1: Estructura de una lista de contactos representada en formato XML.

XML proviene del lenguaje SGML y permite definir la gramática de lenguajes específicos (de la misma manera que HTML es a su vez un lenguaje definido por SGML) para estructurar documentos grandes. XML establece el uso de una etiqueta de apertura (<etiqueta>) y de cierre (</etiqueta>) antes y después de cada elemento, al estilo HTML. Las etiquetas se pueden anidar, para conformar un objeto más complejo. Además, XML permite definir a través de una hoja auxiliar (XSD) el tipo de dato que se recibirá en cada campo, y el formato que deberá respetar el documento. De esta forma, un validador puede tomar esta hoja XSD (cuya ubicación se define al principio del documento) y comprobar que el documento sea válido. Esta es una de las características más importantes de XML, la cual brinda legibilidad y seguridad al desarrollador al momento de interactuar entre aplicaciones, entre otras ventajas. En la Figura 2.1 vemos un ejemplo de un arreglo objetos representados en formato XML, con las etiquetas anidadas mencionadas anteriormente.

2.2.2 JavaScript Object Notation (JSON)

JSON, como XML, es un formato de texto para el intercambio de datos. Adoptado del lenguaje de programación JavaScript, JSON es un formato de texto que es completamente independiente del lenguaje subyacente de la aplicación en la que sea usado, pero utiliza convenciones que son ampliamente conocidas por los programadores de la familia de lenguajes C, incluyendo C, C++, C#, Java, JavaScript, Perl, Python, y muchos otros. Estas propiedades hacen que JSON sea un lenguaje ideal para el intercambio de datos. [17]

```
{
  - Contacts: [
      - {
            FirstName: "Demis",
            LastName: "Bellot",
            Email: "demis.bellot@gmail.com"
        },
      - {
            FirstName: "Steve",
            LastName: "Jobs",
            Email: "steve@apple.com"
        },
      - {
            FirstName: "Steve",
            LastName: "Ballmer",
            Email: "steve@microsoft.com"
        },
      - {
            FirstName: "Eric",
            LastName: "Schmidt",
            Email: "eric@google.com"
        },
      - {
            FirstName: "Larry",
            LastName: "Ellison",
            Email: "larry@oracle.com"
        }
    ]
}
```

FIGURA 2.2: Estructura de una lista de contactos representada en formato JSON.

El lenguaje JSON es muy utilizado actualmente y está reemplazando a los demás lenguajes de intercambio de datos (como XML) por diversas razones, tales como velocidad de interpretación, sencillez, y escalabilidad, entre otros. Como contraparte, JSON no brinda soporte para hojas de validación del tipo XSD como sí brinda XML.
En la Figura 2.4 se puede ver el mismo arreglo de objetos que en el apartado anterior, pero en este caso representado en formato JSON.

2.2.3 Hypertext Transfer Protocol (HTTP)

HTTP es un protocolo de red perteneciente a la capa de aplicación [13], definido por primera vez en la RFC 2616. Es un protocolo sin estado, que funciona bajo el esquema cliente/servidor. Es el más habitual de la red porque es el utilizado para servir las páginas web de la WWW (World Wide Web). Los mensajes HTTP son transaccionales, es decir, cada vez que el cliente envía una petición al servidor, quedará esperando la respuesta.

Como se aprecia en la Figura 2.3, una transacción HTTP consta de un requerimiento, de la forma **URL** + **Verbo** + **Encabezados** + **Cuerpo** (opcional), y una respuesta **Código de estado** + **Encabezados** + **Cuerpo** (opcional). A continuación, se explicará de qué se trata cada componente.

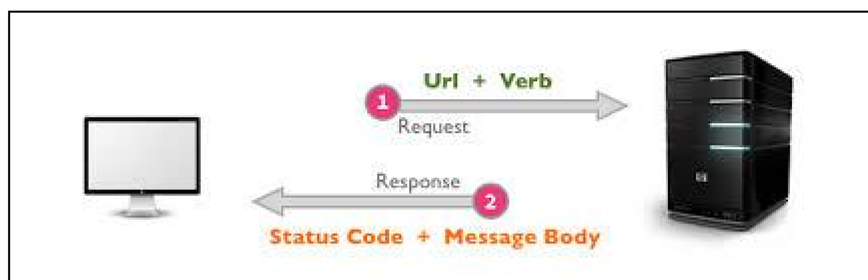

FIGURA 2.3: Requerimiento y respuesta en una petición HTTP.

Uniform Resource Locator (URL)

El URL (Localizador Uniforme de Recursos, más comúnmente conocido como <<*La*>> URL) es una dirección única que sirve para identificar un recurso en la red y que tiene la forma:

esquema://host.dominio:puerto/ruta/recurso

FIGURA 2.4: Composición de una URL.

Esquema: Define el tipo de servicio Internet. El más común es *http*. Otros ejemplos pueden ser *https* o *ftp*.
Host: Por defecto para http es *www*.
Dominio: Define el nombre del dominio como por ej.: *unlp.edu.ar*.

Puerto: Define el número de puerto en el host. El número de puerto por defecto para HTTP es *80*.
Ruta: Define el camino o subdirectorio en el servidor. Si es omitido, el documento debe encontrarse en el directorio principal, de lo contrario no será localizado.
Recurso: Define el nombre del recurso solicitado.

Verbo (o método)

El protocolo especifica distintos verbos que expresan la "intención" del requerimiento. Existen cinco verbos principales, y dos adicionales que se usan menos frecuentemente. Ellos son POST, GET, PUT, PATCH y DELETE, y HEAD y OPTIONS, respectivamente. Los primeros se relacionan con operaciones CRUD (*Create, Retrieve, Update, Delete* o Crear, Recuperar, Actualizar, Eliminar). Su funcionamiento se explica en la siguiente tabla (Tabla 2.1).

TABLA 2.1: Métodos HTTP más utilizados.

Método	**Operación CRUD**	**Descripción**
POST	Creación	Este método se usa para crear un objeto en el servidor remoto. Suele venir acompañado de un cuerpo, aunque esto no es obligatorio.
GET	Recuperación	Este es el método más usado y sirve para solicitar un recurso al servidor. No tiene cuerpo.
PUT	Creación/Actualización	Similar al POST, este método sirve para crear un objeto en el servidor, aunque tiene algunas diferencias en su especificación y no se usa tan frecuentemente como POST. Se usa para actualizaciones completas de un objeto ya existen. Generalmente viene acompañado de un cuerpo, aunque no es obligatorio.
PATCH	Actualización	Este método sirve para actualizar parcialmente un objeto en el servidor, y viene acompañado de un cuerpo.
DELETE	Eliminación	Este método se usa para eliminar un recurso del servidor, y no contiene cuerpo.

Encabezados

Los encabezados o cabeceras HTTP son parámetros adicionales que viajan en el requerimiento HTTP y tienen la forma **Clave:Valor**. Si bien existe una variedad de encabezados utilizados con frecuencia, casi ninguno es obligatorio según el estándar HTTP.
Hay un sinnúmero de ejemplos de encabezados, algunos más frecuentes que otros, y utilizados con diversos fines. A modo de ejemplo, algunos encabezados habitualmente

usados son [14]:

Accept-Language: Este encabezado especifica el *locale* (combinación de país y lenguaje) preferido por el navegador. Esto sirve para que, en caso de estar disponible, el servidor devuelva la versión del recurso en el idioma solicitado por el navegador.
Un ejemplo de este encabezado sería `Accept-Language: “es-AR”` (por favor, sírvame los recursos en idioma español de Argentina).

Content-Length: Este encabezado se usa cuando el requerimiento o la respuesta llevan cuerpo, y especifica el tamaño en octetos del cuerpo del mensaje HTTP.

Content-Type: Al igual que el Content-Length, el Content-Type se usa cuando el requerimiento o la respuesta llevan un cuerpo, y es el encargado de especificar el tipo del contenido del cuerpo, para que el receptor sepa cómo decodificar dicho contenido.
Un ejemplo de este header es `Content-type: application/json`. En este caso, se está informando al receptor del mensaje que el tipo de su contenido es JSON.

Cuerpo

El cuerpo es la carga útil del requerimiento HTTP, y es opcional. Como se dijo anteriormente, el cuerpo puede viajar cuando si el método del requerimiento lo permite y el cliente lo desea, o si la respuesta lo amerita. Cuando hay cuerpo, se deben especificar los encabezados Content-Type y Content-Length.

Código de estado

Los códigos de estado son números de tres cifras utilizados en la respuesta para indicar si el requerimiento fue exitoso o no.
Al igual que en los encabezados, existen muchos códigos, pero los más frecuentemente utilizados son los mencionados a continuación:

2xx (OK): Cualquier código del rango 200-299 indica a grandes rasgos que el requerimiento fue exitoso, aunque hay códigos específicos que indican detalles adicionales.

3xx (Redirección): Cualquier código de estado del rango 300-399 indica que el requerimiento es correcto, pero el recurso solicitado no existe más en esa ruta. El recurso puede ubicarse temporal o definitivamente en otra ubicación, y dependiendo del código puede haber o no una redirección automática.

4xx (Error en el requerimiento): El rango 400-499 indica que hay un error en el requerimiento por parte del cliente. El estado más común de este rango es el 404, que indica que el recurso solicitado no existe.

5xx (Error interno del servidor): El rango de estados 500-599 indica que hubo un error interno del servidor al procesar el requerimiento, siendo el más común de estos el error 500.

2.2.4 Representational State Transfer (REST)

Al igual que con otros conceptos definidos en este capítulo, lo primero que hay que aclarar sobre REST es que no es una arquitectura, ni una tecnología, ni un estándar en sí mismo. Es, más bien, un *estilo de arquitectura* que plantea algunas premisas para el diseño de servicios web y sus interfaces.

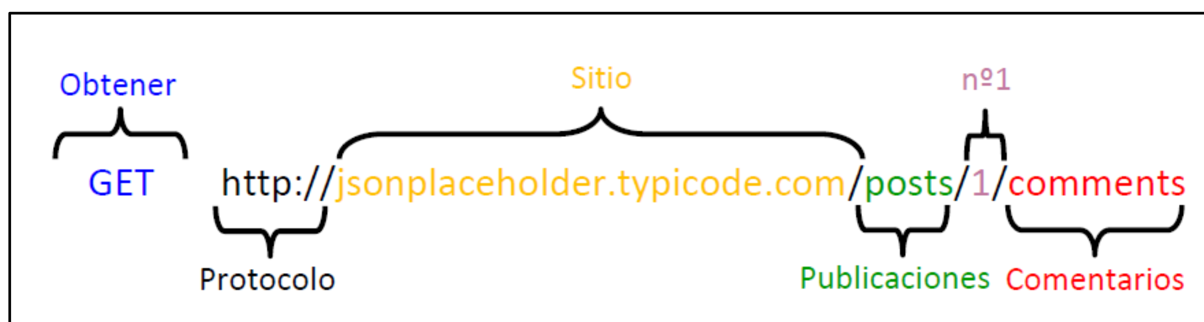

FIGURA 2.5: Composición de un requerimiento REST.

REST teóricamente funciona con cualquier protocolo de comunicación cliente-servidor *stateless*, es decir que el servidor no mantiene información de sesión entre un requerimiento y otro, eso debe ser manejado por el cliente y *cacheable* [15]. Esto último significa que se puede conservar temporalmente una copia de la respuesta de un servicio para disminuir la carga de red y mejorar los tiempos de respuesta. La realidad es que en casi todos los casos se usa HTTP como protocolo subyacente, que cumple con estas premisas. Como se mencionaba anteriormente, REST es un estilo de arquitectura usado para diseñar aplicaciones en red. La idea es que no se provean servicios de elevada complejidad como proponen algunos estándares más antiguos, si no que sencillamente se use HTTP para hacer las llamadas a "micro servicios". Es decir, servicios que por sí solos no proveen gran funcionalidad, pero son de bajo tiempo de respuesta, y de codificación, configuración, y uso simples. Las aplicaciones RESTful (así se llama a las que adoptan este concepto) usan requerimientos HTTP para enviar información (crear y/o actualizar), leer información (por ejemplo, hacer consultas), y eliminar información. Así, REST utiliza HTTP para todas las operaciones CRUD.

Ejemplos de invocaciones REST (en formato JSON)

JSONPlaceholder.typicode.com es un portal que permite crear y probar servicios REST con datos ficticios. Aquí lo utilizaremos para demostrar la funcionalidad de las APIs REST. El ejemplo precargado en esta web simula los datos de una red social o blog, y contiene información sobre publicaciones, comentarios, fotos, entre otros.

El potencial de REST radica en la sencillez de los llamados gracias, entre otras cosas, a que se apoya en los verbos HTTP.

Por ejemplo, supongamos que queremos **obtener** (ver Tabla 2.1) los **comentarios** de la **publicación nº1** del **sitio**. Con dichos datos, podemos determinar a priori la forma que tendrá nuestro requerimiento, como se ve en la Figura 2.5.

En este caso, la respuesta del servidor al requerimiento anterior será una lista de comentarios expresada en JSON, como se ve en la Figura 2.6

```
[
  - {
        postId: 1,
        id: 1,
        name: "id labore ex et quam laborum",
        email: "Eliseo@gardner.biz",
        body: "laudantium enim quasi est
        quidem magnam voluptate ipsam eos
        tempora quo necessitatibus dolor quam
        autem quasi reiciendis et nam
        sapiente accusantium"
    },
  - {
        postId: 1,
        id: 2,
        name: "quo vero reiciendis velit
        similique earum",
        email: "Jayne_Kuhic@sydney.com",
        body: "est natus enim nihil est
        dolore omnis voluptatem numquam et
        omnis occaecati quod ullam at
        voluptatem error expedita pariatur
        nihil sint nostrum voluptatem
        reiciendis et"
    },
  + {…},
  + {…},
  + {…}
]
```

FIGURA 2.6: Respuesta del servidor en formato JSON a la invocación del ejemplo.

2.3 Enterprise Service Bus (ESB)

El bus de servicios empresariales o Enterprise Service Bus es, a grandes rasgos, el encargado de centralizar las interacciones entre los componentes de una organización, y por lo tanto, comparte algunas funcionalidades con Event-Manager (las cuales se detallarán en los próximos capítulos). Sin embargo, los objetivos de uno y otro son bien distintos. Por esta razón es importante que primeramente el lector se familiarice con el concepto de ESB, sus características, casos de uso, y beneficios, de forma que cuando se introduzca Event-Manager, se puedan detectar a simple vista las diferencias entre este y el concepto de ESB.
Como se mencionará más adelante, hay que tener en cuenta que hay muchas implementaciones tanto comerciales como gratuitas y de código abierto de distintos ESBs, algunas con más características de Event-Manager que otras, y, por tanto, es imposible abarcar a todas. En este apartado se hará un punteo por las cualidades más usuales de los ESB. Dicho esto, a continuación, definiremos qué es (y qué no es) un ESB.

El ESB surge como herramienta para proveer un camino homogéneo para las comunicaciones dentro de una organización en la que los diferentes componentes tecnológicos probablemente sean muy distintos entre sí (ver Figura 2.7). Sobre una arquitectura SOA podemos definir un bus de servicios empresariales (Enterprise Service Bus o ESB) como "**una plataforma de software que da soporte a muchas funcionalidades resueltas a nivel de la capa de aplicación en los enfoques tradicionales de construcción de aplicaciones**" [20]. Tales funcionalidades son:

- **La comunicación:** el ESB se ocupa del ruteo de los mensajes entre los servicios.
- **La conectividad:** el ESB resuelve la conectividad entre extremos mediante la conversión de protocolos entre solicitante y servicio.
- **La transformación:** es responsabilidad del ESB resolver la transformación de formatos de mensajes entre solicitante y servicio.
- **La portabilidad:** los servicios serán distribuidos independientemente del lenguaje de programación en el que estén escritos y del sistema operativo subyacente.
- **La seguridad:** el ESB posee la capacidad de incorporar los niveles de seguridad necesarios para garantizar servicios que puedan autenticarse, autorizarse y auditarse.

A pesar de esta definición formal, cuando hablamos de ESB, podemos estar hablando de tres cosas al mismo tiempo [3]

- **ESB como una "forma de hacer las cosas":**

  Es el enfoque incremental de construir una *arquitectura orientada a servicios* (SOA) por medio de conectar todas las aplicaciones a una infraestructura que alcanza a toda la organización.

- **ESB como una arquitectura:**

  Las aplicaciones se conectan a los servicios a través de un bus basado en un MOM o "*message oriented middleware"* (Middleware orientado a mensajes), el cual no solo es capaz de enviar y recibir mensajes, sino también de transformarlos. Este estilo de arquitectura permite construir paso a paso una SOA, para crear, automatizar, e integrar procesos de negocio basándose en los servicios de negocio provistos, y fácilmente manejar y monitorear estos procesos de negocio.

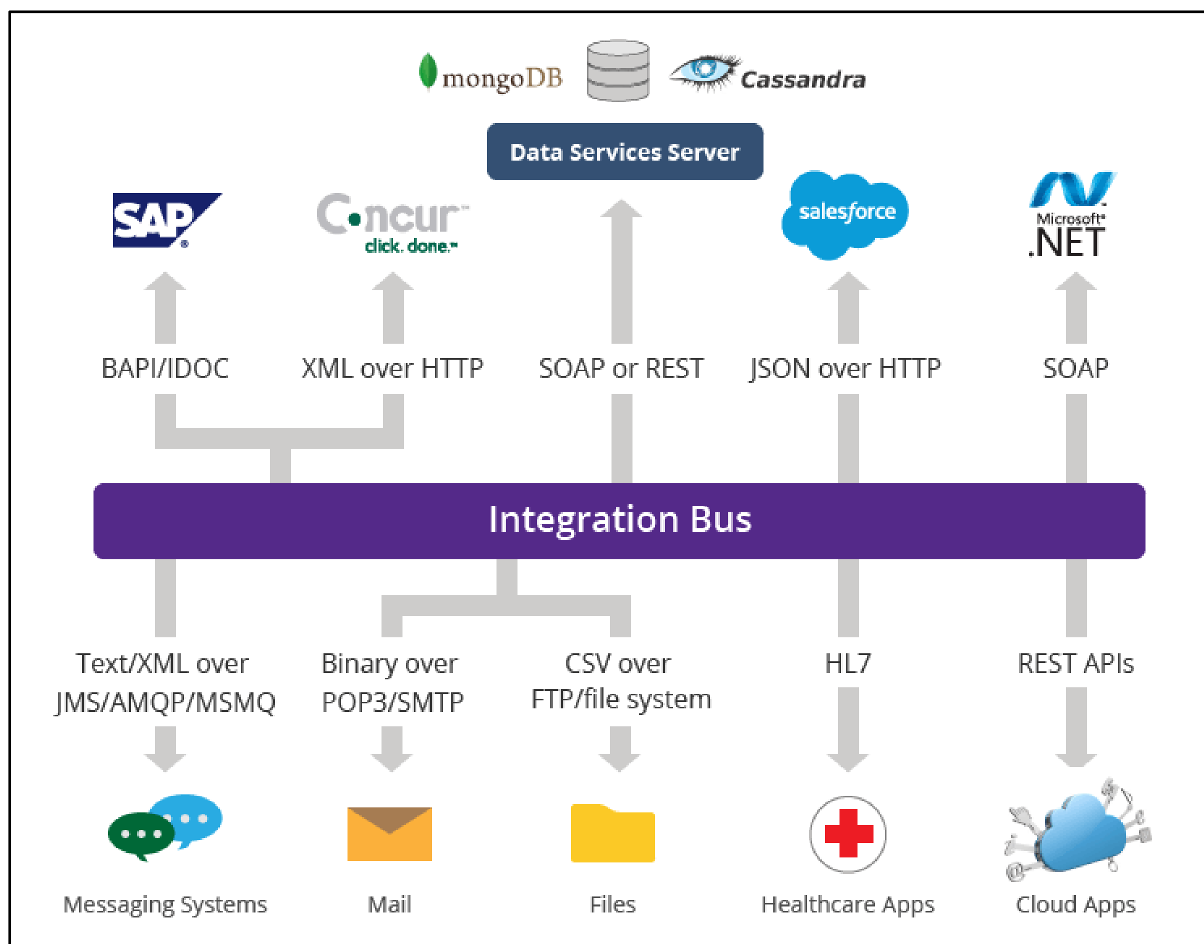

FIGURA 2.7: Ejemplo de la diversidad de sistemas que un ESB puede interconectar.

- **ESB como producto:**

  Hay muchas compañías que venden productos de infraestructura de ESB que les otorgan a las empresas clientes las herramientas para construir un ESB. Tales productos habitualmente están compuestos por componentes tecnológicos ya existentes. Estas compañías comercializan la infraestructura y también la asesoría para que los clientes construyan su ESB. Finalmente, existen productos comerciales y no comerciales que proveen directamente implementaciones de ESB.

2.4 *Clúster* de computadoras

Si bien no es requisito, Event-Manager está diseñado para trabajar de manera más eficiente y confiable si se configura en una arquitectura clúster (ver Capítulo 4, inciso "Tolerancia a fallos"). Para entender esto, es necesario que primero expliquemos qué es un *clúster* de computadoras, y por qué es tan necesaria para que Event-Manager brinde un servicio confiable con *downtimes* (interrupciones de servicio) y pérdidas de información lo más bajos posibles.

Un **cúmulo**, **granja** o ***clúster* de computadoras** se puede definir como un sistema de procesamiento paralelo o distribuido. Consta de un conjunto de computadoras independientes, interconectadas entre sí, de tal manera que funcionan como un solo recurso computacional. A cada uno de los elementos del *clúster* se le conoce como **nodo**. [6]

La principal ventaja de usar una configuración clúster es la **redundancia**. La redundancia consiste en tener dos o más sistemas similares funcionando en paralelo para reducir las interrupciones de servicio por cortes eléctricos, fallas de hardware, y cualquier tipo de falla en general. De esta característica se desprende que el clúster es **tolerante a fallos**; en un clúster típico, pueden fallar hasta N-1 nodos, siendo N la cantidad total de nodos, y la aplicación continuar funcionando, si bien su rendimiento seguramente no sea el mismo.

Para esto existen diversos mecanismos que permiten orquestar el acceso a la información, asegurar que cada nodo tenga la última información disponible, y coordinar esfuerzos para hacer un uso lo más eficiente posible del hardware disponible.

Cabe mencionar que una arquitectura clúster es sensiblemente más costosa en términos de hardware que una arquitectura tradicional, ya que, para un caso ideal, se requieren varios equipos similares y una red de interconexión de alta velocidad entre los nodos para el buen funcionamiento del clúster.

TABLA 2.2: Comparativa entre configuración tradicional y configuración clúster.

	Configuración tradicional	**Configuración en clúster**
Cantidad de nodos	Uno	Dos o más
Tolerancia a fallos	No tolera	Soporta fallos de hasta N-1 nodos.
Sincronización	No requerida	Requerida para coordinar trabajo y acceso a datos entre los miembros del clúster.
Concurrencia	Limitada a los hilos o *threads* disponibles en el sistema	Cantidad de hilos por nodo * cantidad de nodos
Distribución de carga	No provee	Permite distribuir la carga de procesamiento entre las diferentes instancias
Infraestructura	Simple	Compleja

En la Tabla 2.2 se realiza una comparativa con las diferencias a favor y en contra de la arquitectura tradicional de un solo nodo ejecutando una instancia de la aplicación, frente a la arquitectura clúster.

2.4.1 Intercambio de datos y memoria compartida

Dependiendo del fin para el que se diseñe el clúster, puede hacerse necesario tener un mecanismo que permita compartir información, datos, y eventos de sincronización entre los distintos nodos de un clúster. A continuación, se mencionan algunos casos de uso comunes para estos mecanismos.

Ejemplo 1) Un nodo recupera un objeto de la base de datos. Esta operación suele ser costosa, porque requiere solicitar por red el objeto, que el administrador de base de datos lea el medio en donde el objeto está almacenado (típicamente un disco rígido), y devuelva el elemento. En el caso de que varios nodos requieran el dato en el mismo instante, o en dos instantes relativamente cercanos en el tiempo, puede haber un desaprovechamiento de tiempo de procesamiento. Sería mucho más eficiente que el nodo que requiere el objeto lo persista en una memoria compartida entre los integrantes del clúster para su posterior acceso. En este ejemplo, el mecanismo de memoria compartida o (MC) está funcionando como una **caché** para el clúster.

Ejemplo 2) Un nodo está trabajando en una operación con un conjunto de datos que no puede ser alterado por otro nodo. El programador podría bloquear los datos en la base de datos, pero esto solo es posible si todos los datos que la operación utilizará están en la base de datos; además, esto puede generar demoras innecesarias y posibles inconsistencias. La alternativa usando este mecanismo es tener una estructura de datos compartida que sirva de señalización y que permite bloquear y desbloquear elementos. Aquí, la MC hace las veces de **coordinador**.

Ejemplo 3) Como se describía al inicio de este apartado, es común que las aplicaciones se encuentren replicadas en varios nodos. Las aplicaciones web no son la excepción, y suelen estar dispuestas detrás de un balanceador de carga, que es un servidor que recibe todos los requerimientos y los reparte entre los nodos del clúster, como se aprecia en la Figura 2.8.
Pensemos en un usuario **X** que inicia sesión en el servidor web **A**. Esto se representa en la memoria del servidor **A** almacenando una estructura de datos con información del usuario **X**, y luego en cada requerimiento el usuario envía parte de esa información. Al procesar el requerimiento, la aplicación compara la información recibida con la que tiene en memoria, y determina si el usuario está autenticado o no. Este proceso no genera inconvenientes si todos los requerimientos del usuario son dirigidos al mismo servidor, pero es imposible asegurar esto (de hecho, es poco deseable) en un clúster. Cuando un servidor distinto al **A** reciba un requerimiento del usuario **X**, determinará que este no está autenticado. Para resolver este problema, se utiliza **memoria compartida**. Esto consiste en usar un sector

especial de la memoria de cada nodo que es compartido entre todos los miembros del clúster, y almacenar ahí la información, de la sesión. De esta forma todos los servidores tendrán acceso a la estructura de datos con información del usuario autenticado.

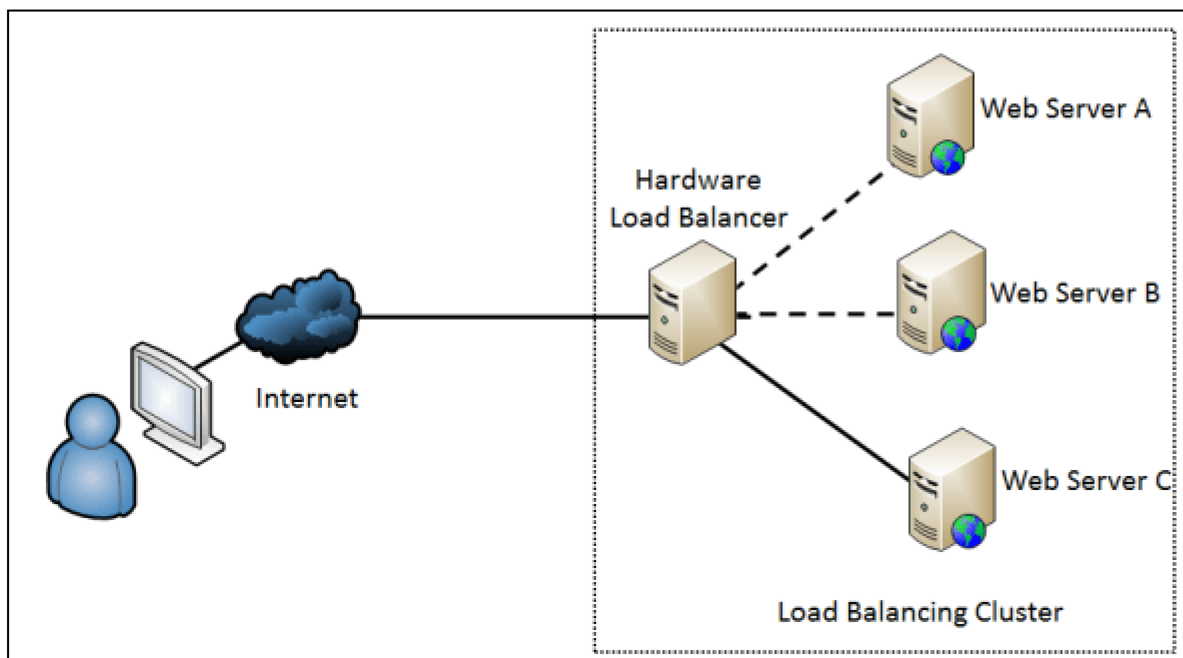

FIGURA 2.8: Clúster de servidores web con balanceador de carga.

A continuación, se describen dos productos comerciales (Hazelcast y Redis) que pueden ser utilizados como mecanismos de memoria compartida.

Hazelcast

Hazelcast es una plataforma de código abierto y basada en Java, para la distribución de datos. [11] Es una grilla de datos en memoria (o *in memory data-grid*) de Hazelcast, los datos son distribuidos equitativamente entre los nodos de un clúster, permitiendo la escalabilidad horizontal tanto en términos de almacenamiento disponible como de poder de procesamiento. Su característica distintiva es que es una librería embebida, por lo que no necesita un servidor aparte para ejecutarse. Esto reduce drásticamente los tiempos de acceso, ya que no hace falta usar red para acceder a los datos mantenidos en memoria.
Hazelcast provee distintos mecanismos de comunicación entre los nodos del clúster, y replica y actualiza constantemente la información de cada uno.

Redis

Redis es un motor de base de datos de código abierto de tipo clave-valor (key-value) que reside en memoria RAM y posteriormente vuelca el conjunto de datos almacenados al disco duro. [12] Redis, a diferencia de Hazelcast, no es embebido; es decir, reside un proceso aparte (o *standalone*). Esto permite consultarlo como a una BD típica, con diferentes clientes, pero genera retardo adicional debido a la red, puesto que típicamente se instala en

un servidor aparte. Redis permite tener un nodo maestro y varios esclavos, para soportar fallos y replicar datos, como se explicará en el siguiente apartado de bases de datos NoSQL.

2.5 Bases de datos NoSQL (Not Only SQL)

Las bases de datos NoSQL brindan muchas ventajas por sobre las SQL, dependiendo el uso que se les dé a las mismas. Para aplicaciones de tareas repetitivas, sin grandes cambios en su modelo, se recomienda usar bases de datos SQL, mientras que para aplicaciones que pretendan ser más dinámicas y flexibles, porque así su objetivo lo requiera, se recomienda utilizar bases de datos NoSQL (en esta sección explicaremos los fundamentos de esta afirmación). Este último es el caso de Event-Manager, que procesa notificaciones con contenido de diverso tamaño, y que permite implementar rápidamente el soporte para nuevos protocolos.
En este apartado haremos un repaso breve por los principales conceptos de las bases de datos NoSQL [7] así como los diferentes tipos que existen. Finalmente, comentaremos varias características y ventajas de MongoDB, la base de datos elegida para Event-Manager.

2.5.1 Introducción

Se puede decir que la aparición del término NoSQL se da con la llegada de la Web 2.0, ya que, hasta ese momento, la mayoría del contenido era generado por los administradores de los portales, y no por sus usuarios. Con la aparición de aplicaciones como Facebook, Twitter o YouTube, cualquier usuario podía subir contenido, provocando así un crecimiento exponencial de los datos. Es en este momento cuando empiezan a aparecer los primeros problemas de la gestión de toda esa información almacenada en bases de datos relacionales. En un principio, para solucionar estos problemas, las organizaciones optaron por utilizar un mayor número de máquinas; pero pronto se dieron cuenta de que esto no solucionaba el problema, además de ser una solución muy cara. La otra solución posible, era la creación de sistemas pensados para ese uso específico, lo que con el paso del tiempo dio lugar a soluciones más y más robustas, apareciendo así el movimiento NoSQL.
Hablar de bases de datos NoSQL es hablar de estructuras que nos permiten almacenar información en aquellas situaciones en las que las bases de datos relacionales generan ciertos problemas, debido principalmente a problemas de escalabilidad y rendimiento de las bases de datos relacionales, en donde se dan cita miles de usuarios concurrentes y con millones de consultas diarias. Además de lo comentado anteriormente, las bases de datos NoSQL son sistemas de almacenamiento de información que no cumplen con el esquema entidad–relación. Tampoco utilizan una estructura de datos en forma de tabla, sino que para el almacenamiento hacen uso de otros formatos como clave–valor, mapeo de columnas o grafos.

2.5.2 Ventajas y desventajas

Esta forma de almacenar la información ofrece ciertas ventajas sobre los modelos relacionales. Si bien NoSQL es un concepto que abarca diferentes bases de datos, se pueden encontrar varias ventajas en común. Estas son:

- **Escalabilidad horizontal**: Mejorar el rendimiento de estos sistemas simplemente se consigue añadiendo más nodos, con la única operación de indicar al sistema cuáles son los nodos que están disponibles.

- **Pueden manejar gran cantidad de datos**: Esto es debido a que utilizan una estructura distribuida, en muchos casos mediante tablas Hash.

- **Baja latencia y alta performance:** Debido a su naturaleza no relacional, las operaciones de escritura y lectura requieren menos computación, por lo cual suelen ser muy performantes.

Siguiendo en línea con las ventajas, las desventajas [8] también deben ser analizadas en el caso específico de cada producto; sin embargo, se pueden encontrar algunas en común, entre otras:

- **Menor madurez:**
  Las primeras bases de datos relacionales surgieron en 1990, mientras que las bases de datos no relacionales recién están saliendo de las etapas de pruebas, y siguen mejorando cada día.

- **Menor soporte:**
  La mayoría de los vendedores de bases de datos relacionales proveen soporte las 24 horas a las organizaciones; incluso algunos ofrecen servicios de administración remota de la base de datos en caso de errores. Por el contrario, las bases NoSQL tienden a ser de código abierto, y solo una o dos empresas ofrecen servicio de soporte.

- **Administración**:
  La idea original de las bases de datos NoSQL era la de ofrecer una solución que no requiriera administración; por el contrario, la realidad es muy distinta, y se necesita de un relativamente alto grado de conocimiento para instalarlas y administrarlas.

2.5.3 Principales diferencias con las bases de datos SQL

Las diferencias entre las bases de datos SQL y NoSQL son grandes. Las bases de datos NoSQL:

- **No utilizan estructuras fijas como tablas para el almacenamiento de los datos:**

  Permiten hacer uso de otros tipos de modelos de almacenamiento de información como sistemas de clave–valor, objetos o grafos.

- **No suelen permitir operaciones JOIN:**

  Al disponer de un volumen de datos tan extremadamente grande suele resultar deseable evitar los JOIN. Esto se debe a que, cuando la operación no es la búsqueda de una clave, la sobrecarga puede llegar a ser muy costosa. Las soluciones más directas consisten en des-normalizar los datos, o bien realizar el JOIN mediante software, en la capa de aplicación.

- **Son de arquitectura distribuida:**

  Las bases de datos relacionales suelen estar centralizadas en una única máquina o bien en una estructura maestro–esclavo. Sin embargo, en los casos NoSQL la información puede estar compartida en varias máquinas mediante mecanismos de tablas Hash distribuidas.

2.5.4 Tipos de BD NoSQL

Existen diversos tipos de bases de datos NoSQL, los cuales se enumeran a continuación.

- **Bases de datos clave – valor:**

  Son el modelo de base de datos NoSQL más popular, además de ser la más sencilla en cuanto a funcionalidad. En este tipo de sistema, cada elemento está identificado por una llave única, lo que permite la recuperación de la información de forma muy rápida). Se caracterizan por ser muy eficientes tanto para las lecturas como para las escrituras. Algunos ejemplos de este tipo son Cassandra, BigTable o HBase.

- **Bases de datos en grafo:**

  En este tipo de bases de datos la información se representa como nodos de un grafo y sus relaciones con las aristas del mismo, de manera que se puede hacer uso de la teoría de grafos para recorrerla. Para sacar el máximo rendimiento a este tipo de

bases de datos, su estructura debe estar totalmente normalizada, de forma que cada tabla tenga una sola columna y cada relación dos. Este tipo de bases de datos ofrece una navegación más eficiente entre relaciones que en un modelo relacional. Algunos ejemplos de este tipo son Neo4j, InfoGrid o Virtuoso.

- **Bases de datos orientadas a objetos:**

  En este tipo, la información se representa mediante objetos, de la misma forma que son representados en los lenguajes de programación orientada a objetos (POO) como ocurre en JAVA, C# o Visual Basic .NET. Algunos ejemplos de este tipo de bases de datos son Zope, Gemstone o Db4o.

- **Bases de datos orientadas a documentos:**

  Este tipo de BD almacena la información como un documento (Figura 2.9), generalmente utilizando para ello una estructura simple como JSON o XML, en donde se utiliza una clave única para cada registro.

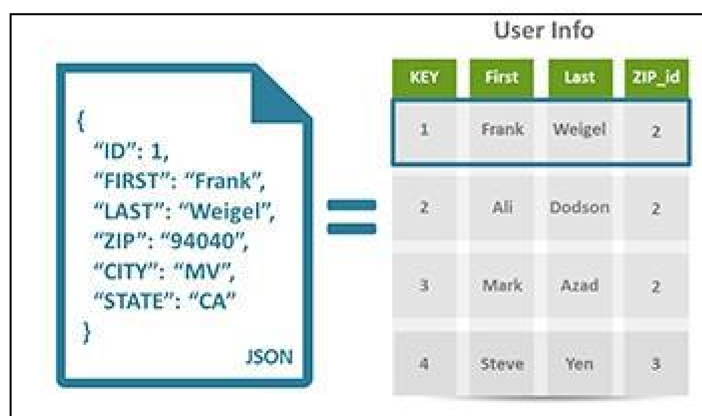

FIGURA 2.9: Estructura de un documento.

Este tipo de implementación permite, además de realizar búsquedas por clave–valor, realizar consultas más avanzadas sobre el contenido del documento. Son las bases de datos NoSQL más versátiles. Se pueden utilizar en gran cantidad de proyectos, incluyendo muchos que tradicionalmente funcionan sobre bases de datos relacionales. Algunos ejemplos de este tipo son MongoDB o CouchDB.

2.5.5 MongoDB

MongoDB es un sistema de bases de datos NoSQL orientada a documentos, desarrollada bajo el concepto de código abierto.
MongoDB brinda gran flexibilidad ante datos de estructura variada, y es especialmente recomendada si se piensa trabajar con JSON, puesto que provee una interfaz sencilla y un

lenguaje de consultas JavaScript (la misma notación de JSON). Además, MongoDB, al igual que la mayoría de las BBDD NoSQL, no requiere un modelo estático de datos para funcionar correctamente, lo que, sumado a su interfaz y lenguaje de consultas benigno, la hacen ideal para la persistencia de datos de Event-Manager. Veremos estas y otras virtudes de este DBMS a continuación.

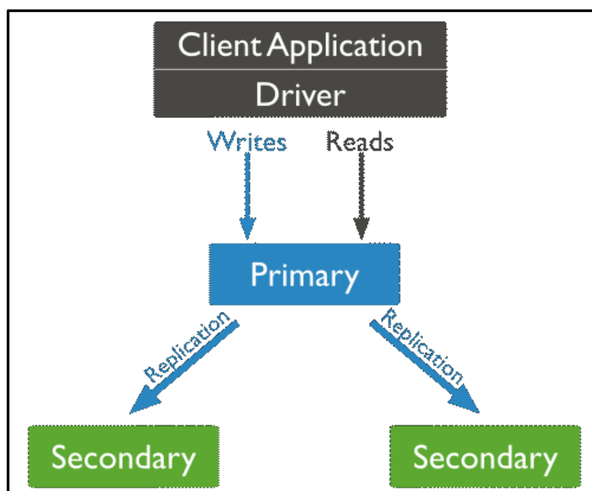

FIGURA 2.10: Ejemplo de un replica set de MongoDB.

Características principales de MongoDB

- **Consultas Ad hoc**:
  MongoDB permite la búsqueda de campos, consulta de rangos y expresiones regulares.

- **Indexación**:
  Cualquier campo en un documento de MongoDB puede ser indexado y es posible realizar índices secundarios, similar a las bases de datos relacionales.

- **Replicación**:
  Tal vez la característica más notoria de MongoDB es el soporte de la replicación maestro-esclavo. Para esto, MongoDB usa los llamados **replica set**, que son grupos de nodos constituidos por un maestro y uno o más esclavos. El cliente de la BD escribe y lee al maestro, y el maestro se encarga de replicar los datos con los esclavos. En caso de falla del maestro, los nodos esclavos restantes acuerdan quién será el maestro, sin interrumpirse el servicio. En la Figura 2.10 podemos ver el funcionamiento del mismo.

- **Consultas en lenguaje JavaScript**:
  Esta es otra de las principales características de MongoDB; las consultas se realizan en lenguaje JavaScript, introduciendo el concepto de “desarrollo JavaScript *fullstack*”. Esto quiere decir que se puede programar una aplicación completa,

incluyendo las consultas a la BD usando un solo lenguaje (JavaScript).

- **Balanceo de carga**:
  Reparte la carga de lectura/escritura en múltiples servidores, brindando una alta performance.

2.6 Frameworks Web

En esta sección comentaremos resumidamente en qué consiste un framework web en general, y en particular describiremos las características principales de Spring MVC, el framework sobre el que está desarrollado Event-Manager.

2.6.1 Introducción

El concepto de *framework* se emplea en muchos ámbitos del desarrollo de sistemas software, no solo en el ámbito de aplicaciones Web. Podemos encontrar frameworks para el desarrollo de aplicaciones médicas, para el desarrollo de juegos, y para cualquier ámbito que se nos pueda ocurrir. En general, con el término framework, nos estamos refiriendo a una estructura software compuesta de componentes personalizables e intercambiables para el desarrollo de una aplicación. En otras palabras, un framework se puede considerar como una aplicación genérica incompleta y configurable a la que podemos añadirle las últimas piezas para construir una aplicación concreta.
Los objetivos principales que persigue un framework son: acelerar el proceso de desarrollo, reutilizar código ya existente y promover buenas prácticas de desarrollo como el uso de patrones. Un framework Web, por tanto, se puede definir como un conjunto de componentes (por ejemplo, clases en java y descriptores y archivos de configuración en XML) que componen un diseño reutilizable que facilita y agiliza el desarrollo de sistemas Web [9].

2.6.2 Spring MVC

Spring es un framework web, basado en el lenguaje Java, que nació en una época en la que los servidores de aplicaciones eran altos consumidores de recursos, inflexibles y era demasiado complejo trabajar con ellos. En ese contexto, Spring popularizó ideas como la inyección de dependencias o el uso de objetos convencionales (POJOs) como objetos de negocio. Estas ideas permitían un desarrollo más sencillo y rápido y unas aplicaciones más ligeras y posibilitaron que de ser un framework inicialmente diseñado para la capa de negocio, pasara a ser un completo stack de tecnologías para todas las capas de la aplicación. Las ideas "innovadoras" que en su día popularizó Spring se han incorporado en la actualidad a las tecnologías y herramientas estándar. Así, ahora mismo no hay una gran diferencia entre el desarrollo con Spring y el desarrollo JavaEE "estándar", o al menos no tanta como

hubo en su día. No obstante, Spring ha logrado aglutinar una importante comunidad de desarrolladores en torno a sus tecnologías y hoy por hoy sigue constituyendo una importante alternativa al estándar que merece la pena conocer. En la actualidad, las aportaciones más novedosas de Spring se centran en los campos de Big Data/NoSQL, HTML5/móviles y aplicaciones sociales [10].

Algunas características distintivas de Spring son:

- **Servicios *enterprise***:

  Se puede hacer de manera sencilla que un servicio sea transaccional, o que el acceso a un objeto esté restringido a ciertos roles de usuario, o que sea accesible de manera remota y transparente para el desarrollador, o acceder a otros muchos servicios más, sin tener que escribir el código de manera manual. En la mayoría de los casos solo es necesario anotar el objeto.

- **Estereotipos configurables para los objetos de la aplicación**:

  Podemos anotar nuestras clases indicando por ejemplo que pertenecen a la capa de negocio o de acceso a datos. Se dice que son configurables porque podemos definir nuestros propios estereotipos "a medida": por ejemplo, podríamos definir un nuevo estereotipo que indicara un objeto de negocio que además sería *cacheable* automáticamente y con acceso restringido a usuarios con determinado rol.

- **Inyección de dependencias:**

  Esta es la característica más distintiva de Spring. La inyección de dependencias permite solucionar de forma sencilla y elegante cómo proporcionar a un objeto cliente acceso a un objeto que da un servicio que este necesita. Por ejemplo, que un objeto de la capa de presentación se pueda comunicar con uno de negocio. En Spring las dependencias se pueden definir con anotaciones o con XML.

En resumen, podemos ver que las ventajas de trabajar con Spring (y con Java) son varias, pero fundamentalmente se destacan su madurez, su fuerte presencia en el mercado (siendo uno de los principales frameworks usados por diversas organizaciones), su reconocida estabilidad, posibilidad de monitoreo, y extensibilidad, entre otras.
También cuenta con una documentación oficial muy completa, y con una comunidad de desarrolladores vasta, lo que hace que reciba actualizaciones y soporte frecuentemente. Debido a esto, existe una gran cantidad de librerías, drivers de bases de datos, y otras extensiones (plug-ins) diseñados para trabajar con Spring.

2.7 Resumen del capítulo

Los contenidos sobre conceptos, tecnologías, y arquitecturas vistos en este capítulo ayudarán al lector a tener una comprensión más profunda y completa no solo de Event-Manager, sino de todo el trabajo y de los sistemas informáticos de las organizaciones en general.

En los próximos capítulos veremos cómo lo visto se utiliza tanto en los trabajos similares, como para desarrollar la solución final de este trabajo.

Capítulo 3

Trabajos previos relacionados

En este capítulo mencionaremos algunos trabajos que comparten objetivos y propuestas con las del presente trabajo. Estos trabajos tienen la misma finalidad que esta tesina, en el sentido de que intentan solucionar de una manera eficaz y eficiente un problema común como es la entrega de notificaciones sin incurrir en grandes desarrollos y con una política de mejor esfuerzo tratan de sobreponerse a fallas en la red y en las aplicaciones receptoras.

3.1 *Thialfi* (Adya, Cooper, Myers y Piatek, 2011)

Thialfi [2] es una solución desarrollada por empleados de Google para resolver el problema del tiempo de respuesta para notificar actualizaciones simultáneas en distintas plataformas a gran escala.

3.1.1 Introducción

Muchas aplicaciones de escala masiva están estructuradas alrededor de información compartida entre múltiples usuarios, sus dispositivos, y la infraestructura de la nube. Las aplicaciones cliente mantienen una *caché* local de su información, la cual debe mantenerse actualizada. Por ejemplo, si un usuario cambia el horario de una reunión en un calendario, ese cambio debería verse rápidamente reflejado en los dispositivos de todos los asistentes. Tales escenarios aparecen con frecuencia en Google. Aunque los servicios de infraestructura proveen almacenamiento fiable, actualmente no existe un mecanismo de propósito general para notificar a los clientes sobre modificaciones en la información compartida. En la práctica, muchas aplicaciones consultan periódicamente para detectar cambios (polling), lo que resulta en demoras significativas o alta carga en los servidores. Otras aplicaciones desarrollaron sistemas de notificaciones personalizados, pero estos han demostrado ser difíciles de generalizar y complicados de mantener.

3.1.2 Solución

Thialfi es un sistema de notificaciones altamente escalable desarrollado en Google para aplicaciones orientadas al usuario final con cientos de millones de usuarios y miles de millones de objetos. *Thialfi* entrega, en promedio, notificaciones en menos de un segundo y visibilidad clara a pesar de fallas, incluso de centros de datos enteros.

Thialfi está compuesto por una librería a ser incluida en las aplicaciones cliente, la cual soporta distintos lenguajes de programación tales como JavaScript y C++ (entre otros) y una aplicación de lado servidor. La aplicación de lado servidor de *Thialfi*, a su vez, se divide en dos clases:

- ***Matchers***: Existe una instancia de *Matcher* por cada objeto, y son los encargados de recibir y reenviar las notificaciones.
- ***Registrars***: Existe una instancia de *Registrar* por cada cliente, y son los encargados de manejar la registración del cliente interesado en un objeto y su estado de presencia.

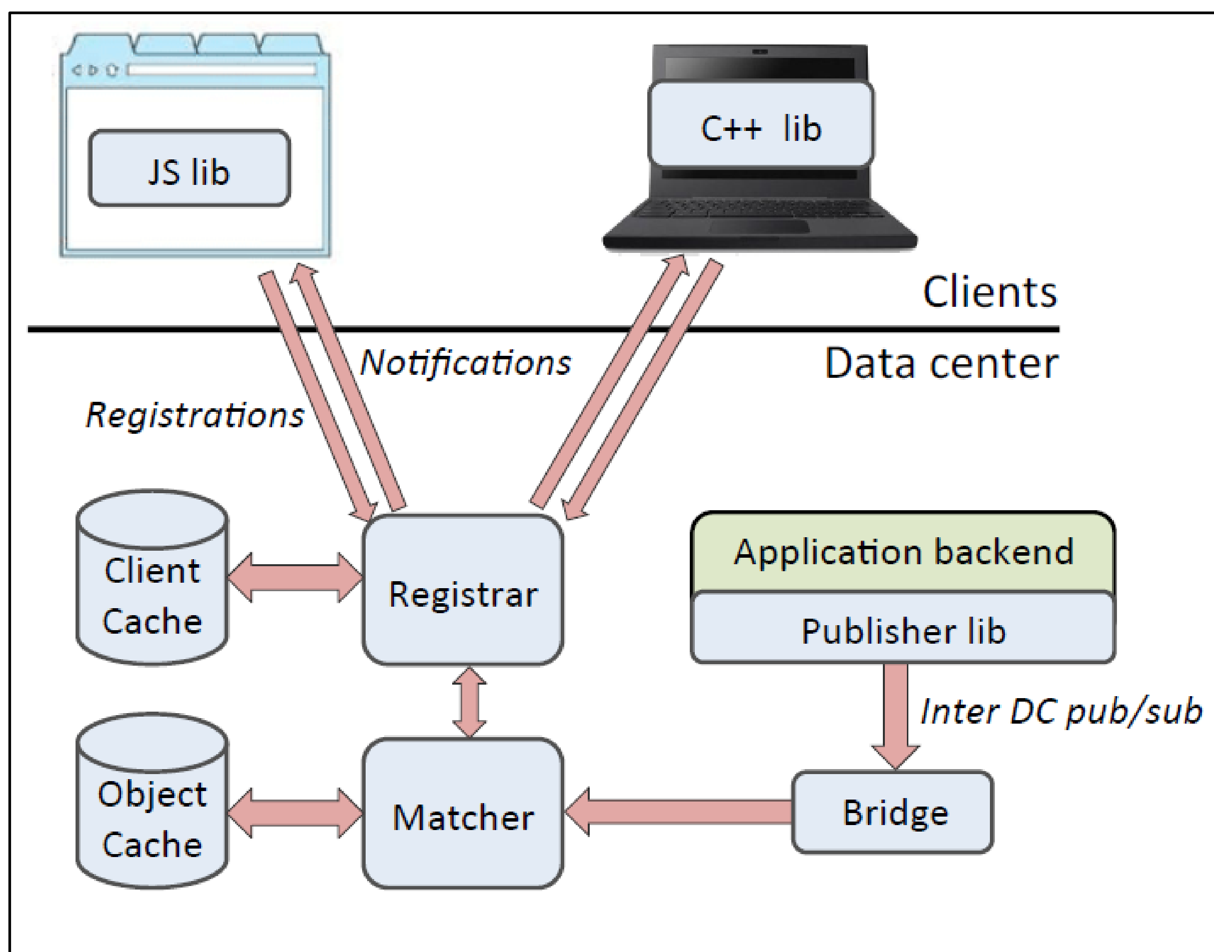

FIGURA 3.1: Arquitectura del sistema Thialfi.

3.1.3 Conclusión del trabajo

Thialfi es una solución inteligente, efectiva, porque resuelve el problema, y eficiente, porque lo hace de manera práctica y sin desperdiciar recursos, a un problema cada vez más habitual, que es la mejora de tiempos de respuesta para el usuario. Estos (los usuarios) son cada vez más exigentes, la demanda de información es cada vez más alta, y no siempre basta con invertir en infraestructura para poder con esta demanda; a veces, se necesitan soluciones simples y escalables, que hagan un mejor aprovechamiento de la infraestructura.

Son esta misma premisa, y otras necesidades similares, las que motivaron el desarrollo de esta tesina.

3.2 *A Publish / Subscribe Mechanism for Web Services* (Tcherevik, 2003)

Este es un trabajo[16] desarrollado por Dmitri Tcherevik, Jefe de Tecnología de Computer Associates, una de las empresas de desarrollo de software más grandes del mundo, fundada en 1976.
A diferencia de Thialfi, que es un producto real diseñado para mejorar los tiempos de procesamiento de las notificaciones generadas por la modificación de archivos compartidos, este trabajo es más bien de tenor teórico, y presenta una solución con algunos puntos en común con la ofrecida por Event-Manager, que será descripta en detalle en los próximos capítulos.

3.2.1 Introducción

En los últimos años las organizaciones SOA han sido la norma. Pero en 2003, cuando Tcherevik escribió este paper, esta clase de organizaciones orientadas a servicios, con aplicaciones distribuidas interactuando entre ellas a través de mensajes máquina a máquina, recién estaban convirtiéndose en protagonistas.
Tcherevik vio que los protocolos de Internet (como HTTP) utilizados para esta interacción carecían de una representación nativa del esquema *Publisher/Subscriber*, y en base a eso definió los lineamientos de una aplicación que tome ese lugar, y que además permita su configuración también a través de servicios web.

3.2.2 Solución

Como se adelantaba en la introducción, el autor propone un *broker* de eventos basado en Java que permita que una o varias aplicaciones se suscriban, y que cuando reciba un evento de un publicador, éste lo reenvíe asincrónicamente a todos los suscriptores interesados en el evento. La suscripción no se realiza utilizando una interfaz visual, ni manualmente; en cambio la aplicación provee un servicio web en XML para que las aplicaciones interesadas puedan suscribirse de manera programática. En la solicitud de inscripción, los suscriptores deben informar, entre otras cosas, el *endpoint* donde la aplicación suscriptora espera recibir las notificaciones reenviadas, por ejemplo http://myapp.org: 8080/listener, y una expresión para filtrar los eventos que le serán remitidos. Una vez realizada la suscripción, el *broker* reenviará los eventos que reciba a los *endpoints* definidos en cada configuración, según los criterios de filtrado definidos por cada uno de los suscriptores.

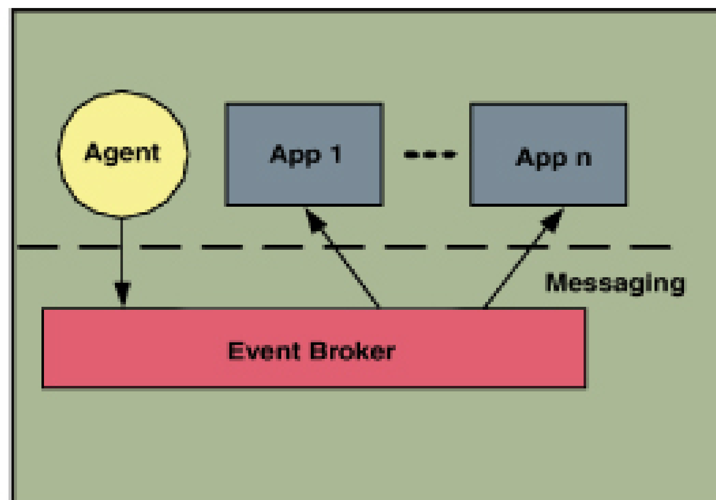

FIGURA 3.2: Esquema de un broker con publicador y suscriptores.

3.2.3 Conclusión del trabajo

A pesar de que Tcherevik provee a modo orientativo algunas clases Java que se encargan del comportamiento que él describió, en realidad su trabajo es más bien conceptual, y si bien él eligió Java para explicarlo, se deja en claro que queda a criterio del implementador qué tecnología utilizar. Se puede decir que el autor fue de alguna forma un adelantado en el sentido de ver un problema actual hace trece años.
El núcleo de la solución propuesta en esta tesina toma como base teórica los lineamientos que Tcherevik estableció en su trabajo, y a esto le suma una interfaz tanto gráfica como de máquina (API) para simplificar el trabajo diario, configuración de reintentos, manejo de errores, persistencia, usuarios, y configuración de suscripciones, entre otras.

3.3 Resumen del capítulo

En este capítulo hemos comentado dos soluciones que de alguna forma vinieron a solucionar problemas similares que los que Event-Manager intenta resolver.
Asimismo, estos trabajos dejan en evidencia que estos problemas son recurrentes, y que, de alguna manera u otra, y en mayor o menor medida, aparecen en todas las organizaciones.

En el próximo capítulo se explicará en detalle cómo hizo Event-Manager para agilizar el procesamiento de notificaciones sin sacrificar el control ni aumentar el riesgo de pérdida de ellas.

Capítulo 4

Event-Manager

En este capítulo se describe el objetivo y las características principales del sistema propuesto por este trabajo, como así también su funcionamiento y comportamiento de acuerdo a diferentes circunstancias.

4.1 Objetivo

El propósito de Event-Manager es el de funcionar como un intermediario entre las aplicaciones que publican eventos y las interesadas en recibirlos.
El valor agregado que ofrece se puede resumir en los siguientes ítems, a saber:

- Evitar a los programadores la tarea de implementar un notificador para cada proceso generador de notificaciones.
- Proveer una interfaz de usuario web centralizada que permita la visualización de los eventos recibidos, las notificaciones generadas a partir de estos eventos, y los suscriptores de esos eventos.
- Brindar funcionalidad adicional como manejo de errores y reintentos, trazabilidad, soporte a diferentes protocolos.
- Desacoplar la capa de comunicaciones de los procesos, de esta manera mejorando la tolerancia a caídas temporales de los mismos, y permitiendo reenviar las notificaciones una vez que el proceso vuelve a estar en línea.

4.2 Características funcionales

Estas funcionalidades surgen de analizar el comportamiento de las aplicaciones receptoras y emisoras de notificaciones y las necesidades de los equipos a cargo de éstas.

4.2.1 Recepción y reenvío de notificaciones

La característica principal del sistema planteado es la de recibir notificaciones en un punto de entrada previamente configurado y propagarlas a los suscriptores interesados en los eventos recibidos en ese punto de entrada.

4.2.2 Manejo de errores y reintentos

El sistema permite configurar una política de reintentos en caso de no poder notificar al suscriptor en el primer intento. La misma consiste en una cantidad de reintentos y un retardo diferente para cada reintento. Finalmente, si el envío supera la cantidad de reintentos establecidos, se marca como fallada y solo se puede volver a enviar manualmente.

4.2.3 Registro de parámetros e información adicional

Además de los datos de la notificación, como el cuerpo de la misma, se registran varios parámetros más, como encabezados recibidos, parámetros adicionales, fecha, respuesta recibida luego de enviar la notificación, cantidad de reintentos, entre otros, para facilitar el análisis posterior en caso de que la notificación falle.

4.2.4 API (Interfaz de Programación de Aplicaciones)

Event Manager cuenta con una API que permite a los desarrolladores construir otras aplicaciones que consuman información de la aplicación. Esta información puede ser estado de notificaciones, estado del sistema, cantidad de eventos recibidos, notificaciones con error, etc.

4.2.5 Interfaz gráfica

Además de la API para la comunicación máquina-a-máquina, Event Manager dispone de una interfaz gráfica web adaptable a distintos dispositivos o *responsive* que permite:

- Consultar las comunicaciones entrantes y salientes minuto a minuto.
- Administrar usuarios con diferentes permisos (administrador, usuario, o solo lectura).
- Administrar suscripciones, es decir, las aplicaciones a las que se les reenviará determinados eventos.
- Administración de endpoints, o sea, los puntos de entrada donde llegarán los eventos externos.
- Filtrar las notificaciones por su contenido, por su estado, fecha, entre otras opciones.

4.2.6 Tolerancia a fallos

Event-Manager fue diseñada para funcionar en una configuración de clúster.
Esto significa que hay varias instancias de la aplicación funcionando en paralelo.
Dicha configuración favorece la distribución de carga y la alta disponibilidad; basta con que un nodo o instancia esté disponible para que la aplicación funcione correctamente. Pero esto no es suficiente para garantizar alta disponibilidad, y es que por más que haya una o varias instancias funcionando correctamente, si la base de datos por algún motivo queda inoperativa, la aplicación dejará de funcionar. Teniendo esto en cuenta, se eligió usar MongoDB, que permite la configuración en *shards* y, por tanto, de manera similar a Event-Manager, soporta la caída de uno o varios nodos sin dejar de funcionar.

4.2.7 Recuperación ante fallos

A pesar de los recaudos tomados que se detallan en el inciso anterior, los fallos aún siguen siendo una posibilidad latente y que hay que controlar.
Existen distintos tipos de fallos, algunos más graves que otros. Entre ellos podemos nombrar:

TABLA 4.1: Fallos y mecanismos de contingencia.

Tipo de fallo	Mecanismo de contingencia
Fallo de red	Sistema de reintentos
Fallo en la aplicación receptora	Sistema de reintentos
Fallo del sistema	Tarea programada

- **Fallos de red**:
  Event-Manager no puede enviar una notificación por problemas en la red.
  Este caso es contemplado directamente por el sistema de reintentos, que en caso de fallo volverá a intentar enviar la notificación tantas veces como haya sido configurado. Si llegado a la cantidad máxima de reintentos, no se obtuvo una respuesta satisfactoria, la notificación quedará marcada como fallida para que posteriormente pueda ser revisada por un administrador.

- **Fallos en la aplicación receptora**:
  De la misma manera que un fallo de red, por una falla en la aplicación receptora, no es posible entregar la notificación.

El mecanismo de contingencia es el mismo sistema de reintentos comentado.

- **Fallos del sistema**:

  Event-Manager deja de funcionar mientras está procesando una notificación. Para estos casos existe un *job* o tarea programada para ejecutarse al inicio la aplicación y cada una hora a partir de ese momento.
  Esta tarea comprueba que las notificaciones estén en un estado consistente; es decir, que hayan llegado a un estado final (exitoso o fallido), que hayan llegado a su máxima cantidad de reintentos configurada en caso de que el estado no sea exitoso, y que no estén en un estado intermedio (o sea, distinto a exitoso o fallido) habiendo superado su tiempo máximo de ejecución (que es la suma del retardo de todos los reintentos configurados para esa notificación).
  De esta manera se asegura que **en el peor de los casos** una notificación demorará una hora en ser enviada al cliente receptor.

4.3 Descripción del funcionamiento

El flujo de Event-Manager comienza cuando recibe una notificación en alguno de sus puntos de entrada (o *endpoints*) configurados, y puede continuar de distintas maneras dependiendo del resultado de algunas operaciones clave, como son la persistencia exitosa del evento recibido, y la entrega correcta o no de las notificaciones a los suscriptores.
Para detallar mejor este flujo, a continuación, podemos ver un Diagrama de Actividad UML (Figura 4.1) que representa el funcionamiento paso a paso de la aplicación, con todas sus variantes, e incluye a los tres actores (Emisor o Productor, Event-Manager propiamente dicho, y Suscriptor) involucrados en la operación del sistema completo.

4.3.1 Descripción del flujo

En cuanto llega un evento de un publicador, se intenta persistir al mismo junto con algunos metadatos (como fecha, por ejemplo), y se devuelve el control a la aplicación que notificó el evento. Esto se representa con una respuesta HTTP 200 (OK) suponiendo que el evento se recibió por esa vía. Si se hubiera recibido por otro protocolo, queda en la implementación del mismo la manera correcta de responder. En caso de no haber podido persistir este evento (por algún problema con la base de datos, por ejemplo), se retorna un código de error para que la aplicación que notificó el evento “sepa” que no se pudo procesar su mensaje.
Una vez persistido correctamente el evento recibido, se procede a buscar a los clientes suscriptos a las notificaciones que se reciban en el *endpoint* mencionado, para reenviarles el evento recibido. A continuación, se notifica a los clientes el evento recibido. Si la respuesta es negativa, o no se obtuvo respuesta, se vuelve a notificar al cliente tantas veces como esté

configurado para esa suscripción. Si se supera la máxima cantidad de reintentos, la notificación se marca como fallida. Si no, termina el procesamiento.
En cualquier caso, se almacena la respuesta recibida para posterior revisión.

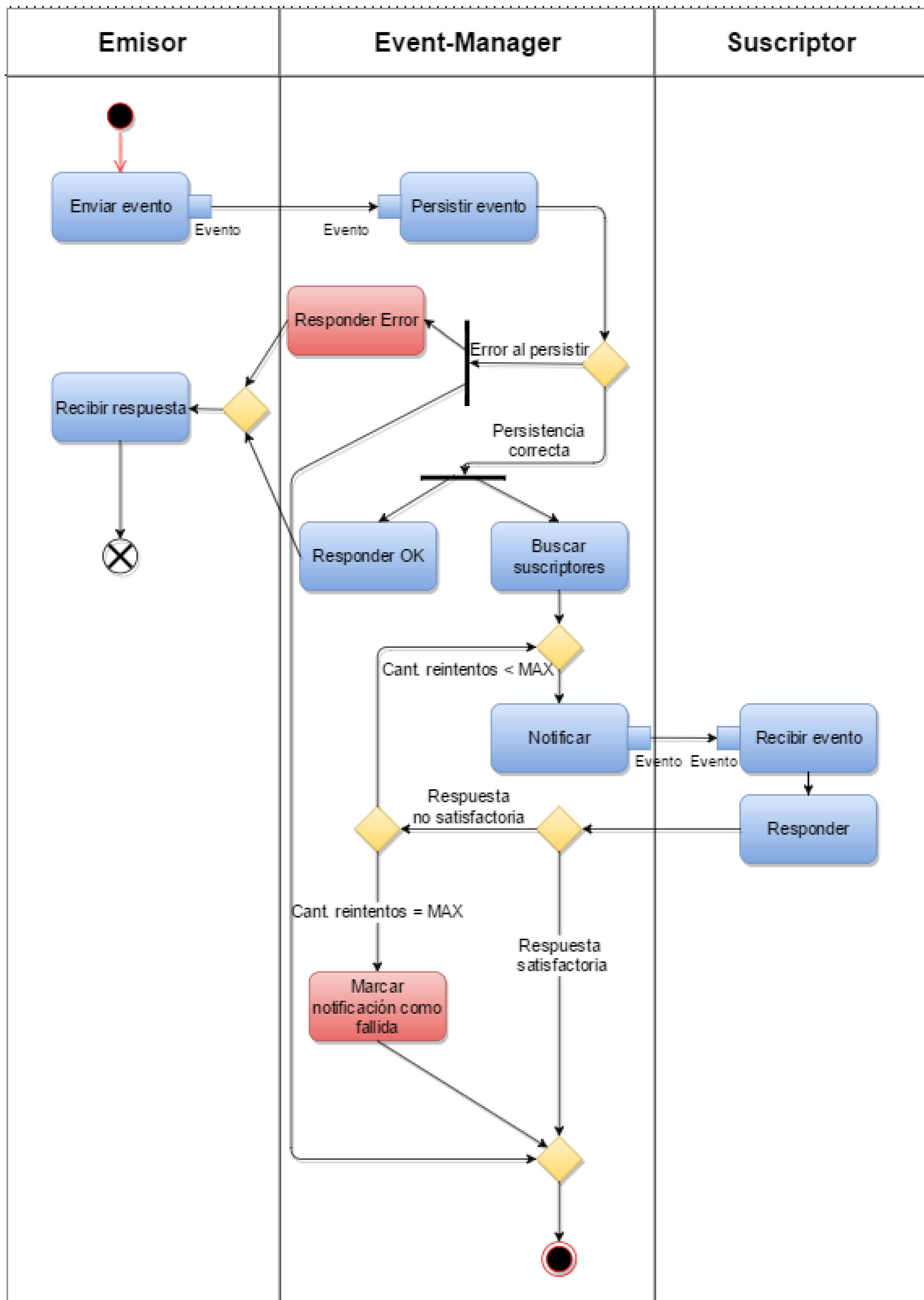

FIGURA 4.1: Diagrama de actividad de Event-Manager. *Los elementos de color azul representan actividades normales, y los de color rojo, actividades anormales o de error.*

4.4 Comparación entre Event-Manager y ESB

Luego de analizar los conceptos de ESB (Enterprise Service Bus) y Event-Manager, es posible que el lector encuentre varias similitudes entre ellos. En esta sección se analizarán algunos puntos similares y disímiles que presentan Event-Manager y un ESB. Para mayor detalle, en esta comparación a modo de referencia utilizaremos Mule ESB [18].
Mule ESB es un ESB liviano basado en Java. Su característica más distintiva es que se trata de uno de los pocos ESBs de código abierto exitosos [19]. Al ser de código abierto, existe una gran variedad de *plugins* y librerías diseñadas para usar con Mule ESB. Es por esto, entre otras cosas, que resulta un buen caso de análisis.

- **Objetivo del sistema:**

  Debemos empezar por definir claramente el objetivo de cada sistema.
  El objetivo de cualquier ESB en general es el de ser el punto central de intercomunicación entre todos los procesos de una organización, soportando distintos protocolos y lenguajes, distintos formatos de transmisión de datos, entre muchas otras funcionalidades. Por otra parte, el objetivo de Event-Manager es mucho más simple y concreto: proveer visibilidad y una capa de prevención y recuperación frente a problemas de entrega o de procesamiento de notificaciones críticas entre aplicaciones, por lo que no provee ningún valor agregado para comunicaciones o servicios que no estén basados en notificaciones.

- **Esquema de funcionamiento:**

  Para los servicios basados en notificaciones, podemos decir que tanto Event-Manager como Mule ESB (y cualquier ESB) comparten el mismo esquema base de funcionamiento, denominado *Publisher/Subscriber*. Ya hemos repasado este concepto, que consiste en que una aplicación (*Publisher*) publica eventos en un canal (puede ser un ESB, Event-Manager, u otro), y luego el o los clientes interesados (*Subscribers*) en recibir esos eventos, se suscriben a este canal.

- **Reintento de entrega de notificaciones:**

  Esta probablemente sea la funcionalidad más importante de Event-Manager, que también ha sido explicada anteriormente. Resumidamente, la misma consiste en volver a enviar una notificación a un cliente si el mismo no pudo recibirla correctamente. Este mecanismo solo es aplicable a protocolos de entrega que contemplen un reconocimiento de recepción (como HTTP). Por su parte, Mule ESB no provee esta funcionalidad.

- **Control de fallos:**

  Tanto Mule ESB como Event-Manager soportan (y alientan) el uso de nodos replicados (clusters), memoria compartida entre las distintas instancias, y la persistencia de datos (opcional en Mule ESB, predeterminada en Event-Manager) para garantizar la alta disponibilidad y resguardar los datos de posibles pérdidas en caso de falla de los servidores. Mule ESB también provee un sistema de reconexión ante fallos de red (ver Tabla 4.1), pero, como se explicó, no cuenta con un sistema de reintentos que contemple posibles fallos en la aplicación receptora.

- **Visualización:**

  Otro de los puntos fuertes de Event-Manager es la posibilidad de ver el procesamiento de las notificaciones recibidas y enviadas, y sus estados, en vivo. En cambio, Mule ESB solo permite monitorear el estado de las aplicaciones involucradas, pero no permite un control tan atómico notificación por notificación.

TABLA 4.2: Comparativa entre Mule ESB y Event-Manager.

Característica	**Event-Manager**	**Mule ESB**
Esquema Publisher/Subscriber	✓	✓
Reintento de entrega de notificaciones	✓	X
Control de fallos	✓	✓
Visualización de comunicaciones en tiempo real	✓	X
Persistencia	✓	•
Configuración	Simple	Avanzada
Punto único de interconexión	X	✓

✓ Soportado X No soportado • Opcional

- **Persistencia:**

  Como se aprecia en el diagrama de actividad, y como se ha mencionado anteriormente, Event-Manager persiste los mensajes recibidos antes de procesarlos, y los rechaza si no puede hacerlo. Mule ESB puede ser configurado para utilizar persistencia, aunque se debe programar manualmente.

- **Configuración e implementación:**

  Al ser uno de uso general (Mule ESB) y otro de uso específico (Event-Manager) es de esperar que la configuración del más genérico sea mucho más compleja que la del más específico. El primero brinda gran cantidad de posibilidades de integración, y de configuración, mientras que el último no requiere configuración especial y se puede poner en funcionamiento rápidamente sin tener que escribir código en aplicaciones suscriptoras ni publicadoras.

Finalmente, es importante destacar que, si bien Mule ESB es un excelente producto, muy versátil y con un sinfín de funcionalidades y extensiones, la cantidad de ESBs existentes es muy grande, por lo que probablemente haya otros ESBs que provean ciertas funcionalidades y no otras, y viceversa.

4.5 *Benchmarking* (pruebas de banco)

Con el fin de poder visualizar la velocidad de respuesta de Event-Manager, se elaboró un gráfico de los tiempos de requerimiento, procesamiento, y notificación de un evento procesado por Event-Manager (Figura 4.2).

El banco de pruebas utilizado se compone de la siguiente manera:

- Un cliente (Publicador) enviando por HTTP POST cincuenta (50) notificaciones con un cuerpo JSON de 83 bytes como cuerpo una tras otra. Para esto se utilizó el cliente de línea de comandos cURL y un script bash.
- Una instancia de Event-Manager recibiendo y procesando las notificaciones.
- Una instancia de MongoDB, sin réplicas.
- Una aplicación receptora de prueba (Suscriptor), conectada a un servidor MySQL a través de la red local (en otro host diferente) que recibe la notificación, la almacena, y confirma su recepción.

Este banco de pruebas se configuró de manera de simular lo más posible las características de los sistemas usados en Despegar.com, los cuales veremos en más detalle en el próximo capítulo.
Cada aplicación y base de datos corre en su propio servidor virtual dedicado Ubuntu compuesto por un procesador de doble núcleo y 4GB de RAM. Estos servidores están interconectados por una red LAN gigabit.

A partir del gráfico podemos concluir que el procesamiento de Event-Manager representa una demora despreciable en el camino de una notificación desde el Publicador hasta el Suscriptor. También es importante destacar cómo el hecho de que Event-Manager sea intermediario entre Publicador y Suscriptor desliga al Publicador del retardo de procesamiento de este último. En promedio el Publicador tuvo un tiempo completo de

requerimiento de 70ms, mientras que, si hubiera tenido que notificar al Suscriptor y esperar a que éste procese su requerimiento, habría tenido una demora aproximada del doble de tiempo (tener en cuenta que en la realidad hay varios suscriptores por cada Publicador, lo que agravaría este problema).

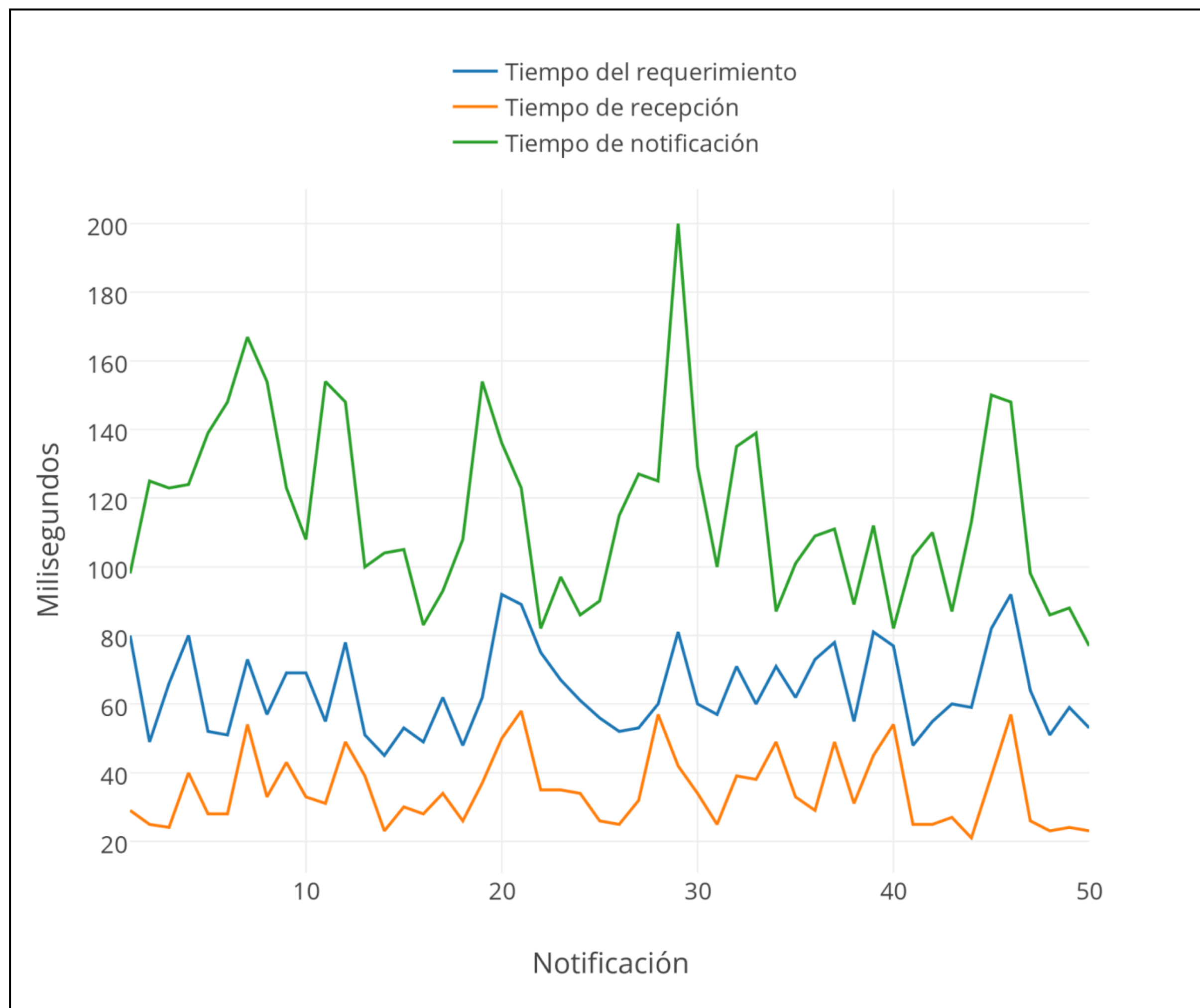

FIGURA 4.2: Prueba de rendimiento de Event-Manager. La congestión de red juega un rol clave en los tiempos y los afecta a todos por igual.

En el gráfico observamos tres líneas diferentes:

- La línea inferior (naranja) representa el tiempo que demora Event-Manager en recibir, almacenar la notificación, determinar qué suscriptores deben ser notificados, y lanzar el proceso de notificar a cada uno de ellos asincrónicamente. Esto quiere decir que no se retiene al publicador mientras se notifica a cada uno de los suscriptores (ver Figura 4.1).

- La línea intermedia (azul) representa el tiempo del requerimiento. Esto es tiempo de ida del requerimiento + el tiempo de procesamiento del requerimiento (inciso anterior) + el tiempo de llegada de la respuesta. Es el tiempo que demora el publicador en enviar la notificación a Event-Manager y recibir la respuesta de que

fue recibida correctamente.

- La línea superior (verde) representa el tiempo que demoró enviar la notificación a la aplicación receptora desde Event-Manager, que ésta la procese, responda, y que la respuesta regrese.

En este ejemplo, ninguna notificación demoró más de 250 milisegundos desde que partió del Publicador, aun siendo conservadores con los cálculos y teniendo en cuenta la congestión de red y el tiempo de procesamiento del Suscriptor.

4.6 Resumen del capítulo

A lo largo de este capítulo se presentó Event-Manager, un sistema que brinda un mecanismo de contingencia frente a problemas que se dan habitualmente en una organización como son los errores de comunicación, además de mejorar la visibilidad y control de las comunicaciones. Event-Manager hace esto sin necesidad de que los desarrolladores de las aplicaciones que lo utilizan tengan que hacer ningún desarrollo adicional, puesto que usa las mismas comunicaciones HTTP que se utilizan entre las aplicaciones habitualmente. Aquí se describió su modelo de datos, con los detalles de cada entidad y su utilidad. Posteriormente, se presentaron las herramientas con las que la aplicación cuenta para mitigar fallos de sistema, cortes de energía, y cualquier otro problema que interrumpa su normal funcionamiento, y finalmente realizamos una descripción paso a paso y un diagrama de actividad que detallan la secuencia de actividades que se pone en funcionamiento a partir de la recepción de un evento.

Capítulo 5

Implementación en Despegar.com

Luego de haber descrito las bases del funcionamiento de Event-Manager, así como su estructura de control, modelo, y comportamiento ante fallos, veremos cómo su implementación ayudó a simplificar el día a día en Despegar.com

5.1 Contexto del problema

Despegar.com es la agencia de viajes más grande de Latinoamérica, con más de 3000 empleados en distintas sedes ubicadas en La Plata, Buenos Aires, Montevideo, Santiago de Chile, Quito, y San Pablo, entre otros. Pero en realidad, la gerencia define a Despegar como "*una empresa de tecnología que comercializa productos turísticos*". Esto quiere decir que siempre se está buscando la innovación, el rápido cambio, el avance tecnológico, y la eficiencia; todo esto en pos de mejorar la oferta y la experiencia de usuario.
La arquitectura de sistemas de esta empresa comparte varios puntos en común con la de los grandes referentes del sector tecnológico, y se compone de cientos de aplicaciones que a su vez ofrecen decenas de *micro servicios* cada una. Estas aplicaciones o componentes en los comienzos se comunicaban entre sí a través de diversos protocolos, y fueron paulatinamente migrando todas hacia un estándar más común y sencillo en su mayoría. Por eso, hoy casi todo el universo de aplicaciones de la organización se comunica entre sí por HTTP utilizando servicios REST.
Estas aplicaciones están implementadas en diversos lenguajes y lo único que tienen en común entre sí es que se pueden comunicar utilizando el protocolo HTTP y mensajes JSON.
A medida que la cantidad de aplicaciones crece, obviamente aumenta el intercambio de mensajes entre ellas. Esto genera tres problemas:

1) Se dificulta la visibilidad de las comunicaciones, por su cantidad y frecuencia.
2) La política de mejor esfuerzo (o *best-effort*) en la entrega de los mensajes recae en el emisor de la notificación, lo que le genera carga de procesamiento y de desarrollo adicional.
3) El control del flujo de notificaciones lo tiene el emisor, por lo que se hace difícil pausar, replicar, o desviar notificaciones para hacer pruebas, por ejemplo.

Para resolver parcialmente el **primer problema**, los desarrolladores hacen uso de *logs* o bitácoras de texto plano y alternativas menos ortodoxas como persistir todos los mensajes recibidos. Esto, además de engorroso, suele no ser suficiente, puesto que, ni todas las comunicaciones se copian a la bitácora (el formato de las bitácoras suele ser incompleto), ni

todas las aplicaciones implementan tal persistencia; además, para poder persistir la notificación recibida, la aplicación receptora debe estar funcionando correctamente.

El **segundo problema** habitualmente se resuelve implementando una capa especial de notificaciones en la aplicación emisora, pero esto trae problemas como código repetido, ya que el responsable de cada aplicación implementa su propia solución, necesidad de código específico para manejar el problema, y carga adicional en la aplicación emisora. Además, no todas las aplicaciones notificadoras necesariamente cuentan con dicha capa adicional.

El **último problema** es el más complicado de manejar, porque depende de la implementación de la capa notificadora de cada aplicación productora. En algunos casos, fue posible por configuración suspender o desactivar temporalmente las notificaciones. En otros casos, donde el notificador usado es más rudimentario, es necesario esperar a que el equipo responsable de la aplicación notificadora haga el desarrollo correspondiente para desactivar las notificaciones. Incluso a veces esto es imposible, dados los tiempos que maneja el equipo de desarrollo.

5.2 Implementación

En Despegar.com actualmente se utiliza Event-Manager en cuatro equipos distintos, cada uno con su clúster de Event-Manager propio.
Además, hay dos instancias de prueba (Desarrollo y Beta) que son compartidas entre todos los equipos. Sabiendo esto, se diseñó la aplicación para que su instalación sea configurable dependiendo de las necesidades de sus usuarios.
Entre otros aspectos modificables al momento de la instalación, se encuentran:

- **Cantidad de instancias**:
  La aplicación es *clusterizable*, por lo que se puede distribuir entre uno o más nodos.

- **Base de datos y replica-set**:
  Es necesario definir al menos un nodo de MongoDB (aunque se pueden usar varios para tener redundancia) y el nombre de la colección que se usará para almacenar los datos de Event-Manager.

- **Configuración de log**:
  Se permite configurar un servidor para registro de bitácora.

Cada equipo tiene una configuración distinta, de entre uno y tres nodos de redundancia, dependiendo del tráfico de cada uno.
Una vez configurados los servidores, y configuradas las suscripciones internas correctamente, los equipos fueron solicitando a los publicadores que les empezaran a notificar los eventos a sus Event-Manager, y que dejaran de notificar los eventos

directamente a las aplicaciones cliente. Esto otorgó una gran flexibilidad a los desarrolladores, ya que ahora pueden administrar las suscripciones de sus aplicaciones sin depender de ningún otro equipo. Además, se aseguran que aún ante un fallo terminal de alguna aplicación cliente, las notificaciones no se van a perder, y en caso de alguna falla temporal de los clientes, Event-Manager volverá a intentar notificarlos.

5.3 Ejemplo de funcionamiento

Describiremos un caso testigo de funcionamiento de Event-Manager, publicadores, y suscriptores, que es el de la **creación de una venta** para el área de Agencias Afiliadas. En este proyecto, Despegar.com es un proveedor para agencias de viaje y les permite a éstas ofrecer los productos de Despegar.com. La agencia recibe una comisión por las ventas realizadas, y a cambio Despegar.com logra llegar a clientes que no están acostumbrados a comprar *online* o que prefieren el trato personalizado con su agencia de confianza.

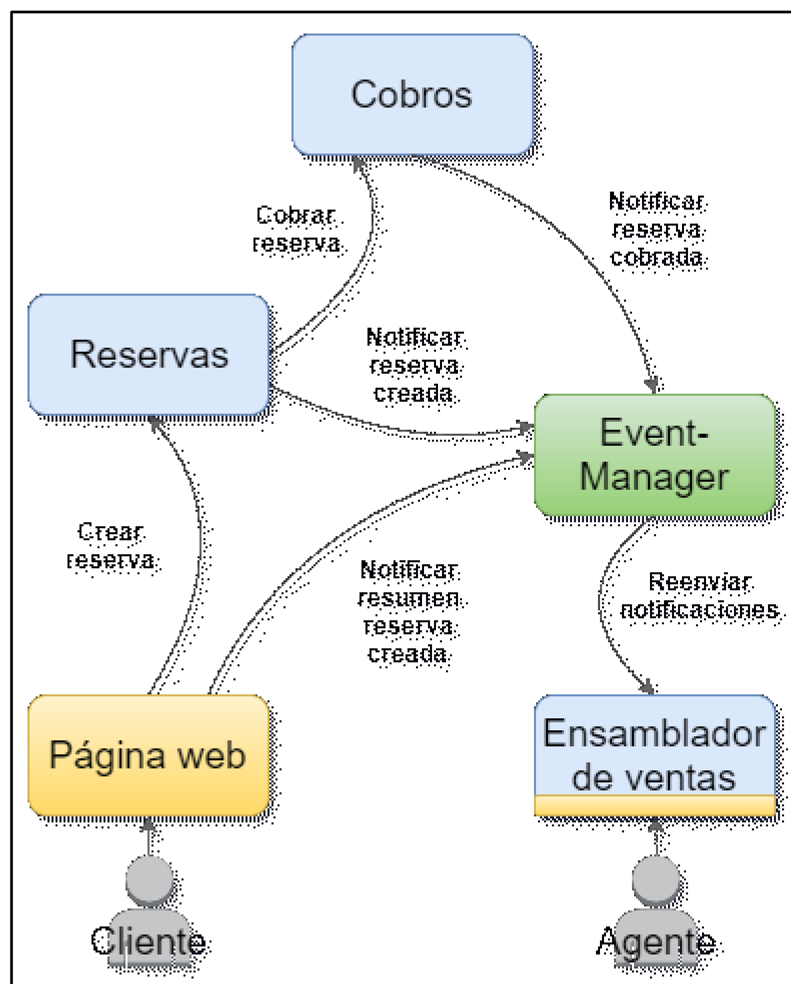

FIGURA 5.1: Flujo de notificaciones en la creación de una reserva.

El proyecto de Agencias Afiliadas brinda a la agencia una página web genérica configurable de acuerdo a los estilos de la agencia (colores, logos, información de contacto) en donde los pasajeros pueden realizar sus reservas de viajes, y luego un tablero de administración para que el o los agentes de la agencia las controlen.

En la Figura 5.1 se presenta un diagrama simplificado de las aplicaciones y notificaciones involucradas en el procesamiento de una venta.

El procesamiento de una reserva comienza cuando un cliente realiza una compra a través de la Página Web de la agencia. En ese momento se solicita la creación de la reserva al Sistema de Reservas de Despegar, y al mismo tiempo se envía a Event-Manager un resumen sintético de la misma, que posee información crítica (identificador de la reserva, email del cliente, comisión calculada para la agencia, entre otras cosas) y que luego será re-enviado al Ensamblador de Ventas. Adicionalmente, este resumen permite mejorar el control y la visibilidad de las reservas de todo el sistema, y si por algún motivo ocurre una falla en el Sistema de Reservas, ayuda a que la falla se detecte rápidamente, y, a veces, a que la reserva pueda recuperarse.
Una vez confirmada la reserva, el Sistema de Reservas notifica a Event-Manager de la creación efectiva de la misma. Event-Manager reenviará este evento al Ensamblador de Ventas, el cual consultará al Sistema de Reservas para obtener la información completa de la reserva, y combinará esa información con la recibida previamente en el resumen para ensamblar la venta y almacenarla. A partir de ahora la reserva ya está disponible en el panel del Agente para ser consultada y/o modificada, aunque se encuentra pendiente de cobro.
El Sistema de Reservas también encarga al Sistema de Cobros el cobro de la reserva. Una vez que ésta se pudo cobrar correctamente, el Sistema de Cobros notifica a Event-Manager que el cobro fue exitoso, y Event-Manager enviará esta notificación al Ensamblador de Ventas, que actualizará la información de la venta y modificará su estado a cobro realizado.

5.4 Ventajas de usar Event-Manager en Despegar.com

Desde su implementación hasta el día de hoy, varias veces quedaron demostradas las ventajas de haber establecido a Event-Manager como centralizador de las notificaciones de venta. Veremos los distintos casos en los que Event-Manager provee ventajas frente a un sistema en el que las notificaciones viajen punto a punto, sin intermediarios.

- **Fallos recuperables:**

  Debido a la gran cantidad de sistemas involucrados en la creación de una venta dentro del modelo de Despegar (tener en cuenta que el diagrama de la Figura 5.1 es un caso muy simplificado), y a la naturaleza del protocolo de red subyacente (IP) en todas las comunicaciones intra y extra organización, a veces ocurre que el Ensamblador de Ventas recibe una notificación de venta, acude al Sistema de

Reservas para solicitar más información, pero esa información aún no está disponible. En este caso, el procesamiento del Ensamblador falla. Para mitigar esto, el **sistema de reintentos** de Event-Manager está configurado para, en caso de recibir una respuesta errónea del Ensamblador de Ventas, aguardar un tiempo prudencial y volver a enviarle la notificación de creación. La gran mayoría de estos casos se resuelve automáticamente sin necesidad de intervención humana.

- **Fallos no recuperables:**

  Existen situaciones en las que, a pesar de tener definida una política de reintentos con esperas cada vez más largas, las notificaciones simplemente no se pueden procesar y quedan en estado fallidas. Esto puede suceder por diversas razones, como una aplicación inalcanzable, un fallo de red, o de base de datos. En este caso, Event-Manager marca la notificación como fallida (en la interfaz visual se ve de color rojo) para que el responsable la vea fácilmente y realice las acciones pertinentes para permitir su recuperación. De otra manera, el responsable podría no saber que una notificación no pudo ser procesada, corriendo la notificación serio riesgo de perderse definitivamente.
  Actualmente Event-Manager no cuenta con un módulo de alertas propio, aunque es una característica que se planea agregar en el futuro, como se verá en el Capítulo 6.

- **Prueba de nuevas funcionalidades**:

  Como se explicaba al principio del capítulo, Despegar.com siempre está buscando la novedad y mejorar la oferta de servicios. Esto se traduce en desarrollo y testeo constante en ambientes de prueba para activar nuevas funcionalidades. Al otorgar el control de las notificaciones al desarrollador, Event-Manager permite activar y desactivar suscripciones temporales, para que además de reenviar las notificaciones al Ensamblador de Ventas real, se reenvíen esas mismas notificaciones, por ejemplo, a un Ensamblador de Ventas *beta* o de prueba. De esta manera se puede monitorear el funcionamiento de una nueva característica en un caso lo más real posible sin peligrar la estabilidad del sistema de producción.

- **Simplificar la migración**:

  A veces no basta con agregar una nueva funcionalidad o refaccionar una aplicación en funcionamiento, y se necesita migrar el procesamiento de las ventas a otro sistema. Hace un año aproximadamente fue necesario migrar de un Ensamblador de Ventas antiguo a uno completamente nuevo, sin posibilidad de desactivar el sistema, puesto que eso significaba perder ventas y generar incomodidad en los usuarios. En un esquema sin Event-Manager esto hubiera implicado avisar a cada una de las aplicaciones publicadoras sobre la nueva aplicación, y esperar a que todos hagan los desarrollos correspondientes para notificar a la misma. En vez de eso, simplemente

se crearon sendas suscripciones adicionales para que la nueva aplicación ensambladora de ventas reciba las notificaciones en paralelo con la antigua, y finalmente se desactivaron las suscripciones de la aplicación antigua. De esta manera, el cambio fue transparente para los publicadores y para los usuarios.

- **Trazabilidad de notificaciones:**

  El constante crecimiento de Despegar genera que cada vez haya más tráfico de notificaciones; en caso de haber un problema, esto puede traerle un dolor de cabeza al desarrollador al momento de investigar las causas del mismo: ¿Qué notificaciones se recibieron? ¿Con qué atributos? ¿Fueron procesadas por el receptor? ¿En qué momento y en qué orden se recibieron? El hecho de tener un organizador como punto de entrada, con una interfaz visual sencilla pero potente, simplifica en gran medida este trabajo, permitiendo por ejemplo buscar rápidamente notificaciones por identificador de la reserva o por hora de llegada.

TABLA 5.1: Antes y después de Event-Manager en Afiliadas Despegar.com

Característica	**Antes de Event-Manager**	**Después de Event-Manager**
Pérdida de notificaciones	Frecuente, no había punto central de entrada de notificaciones, ni se almacenaban.	Casi nula, las notificaciones se persisten tan pronto como se reciben para evitar pérdidas incluso ante una falla de Event-Manager.
Control de notificaciones	No era posible. Si se necesitaba de nuevo una notificación había que solicitarle al responsable de la aplicación notificadora que la reenvíe.	Casi total, se vuelven a procesar notificaciones manualmente (en caso de necesidad) directamente desde la interfaz.
Reintento de envío de notificaciones	Dependiendo del notificador, algunas notificaciones se volvían a enviar en caso de falla, y otras no.	El desarrollador configura los reintentos de cada suscripción de acuerdo a su criticidad.
Visualización de notificaciones	Limitado a que el desarrollador copie las notificaciones recibidas en el registro de la aplicación.	El desarrollador puede buscar notificaciones de hasta tres meses atrás (configurable) por diversos criterios.

5.5 Conclusiones del capítulo

Con la implementación de Event-Manager en el equipo de Agencias de Despegar.com se obtuvieron varios beneficios muy importantes que facilitan el trabajo día a día y mejoran la

estabilidad y confiabilidad del conjunto de aplicaciones que integran a este proyecto. Estas características, además, limitan la responsabilidad del desarrollador al momento de implementar nuevas funcionalidades, ya que no debe encargarse de la capa de notificaciones, y permiten que el mismo se pueda concentrar en la funcionalidad propiamente dicha.
A modo de resumen, en la Tabla 5.1 comparamos la situación de Agencias antes y después de esta implementación.

Capítulo 6

Extensibilidad

En este capítulo trataremos las futuras mejoras y características propuestas para extender la funcionalidad de Event-Manager.

6.1 Módulo de alertas

A partir de lo explicado en el "Ventajas de usar Event-Manager Despegar.com" – "Fallos no recuperables" (Capítulo 5, inciso 4), sería útil adicionar un módulo de alertas que permita configurar destinatarios de correo (por ejemplo) para que sean alertados en caso de que haya notificaciones que no se pudieron entregar. Actualmente, para subsanar este problema, se utiliza una tarea programada en una aplicación externa que consulta regularmente la API de Event-Manager por notificaciones falladas y si las encuentra, emite una alerta por correo electrónico. Con este módulo se evitaría la necesidad de una aplicación adicional para este fin.

6.2 Módulo de estadísticas

Usando la información que entrega la base de datos utilizada para persistir las notificaciones recibidas y enviadas, y los suscriptores y emisores, se podrían calcular diversas métricas que permitirían a los desarrolladores tener una imagen a gran escala del comportamiento de sus aplicaciones. Se podrían calcular métricas como Cantidad de notificaciones diarias en promedio, Promedio de fallas mensual, Cantidad de notificaciones concurrentes (o sea, recibidas cuando aún no se terminaron de procesar otras), entre otras. El potencial de esta característica es muy grande. Por ejemplo, a partir de las estadísticas y métricas informadas por este, se podrían tomar decisiones de arquitectura y diseño de nuevas aplicaciones, y mejora de las ya existentes.

6.3 Grupos de usuarios

Como mencionábamos en "Implementación en Despegar.com" (Capítulo 5, inciso 2), actualmente hay cuatro instalaciones de Event-Manager en el equipo de Afiliados (ver Fig. 6.1). Estas instalaciones consisten de al menos dos nodos (servidores) para las instancias de Event-Manager, y al menos dos nodos para la base de datos MongoDB.

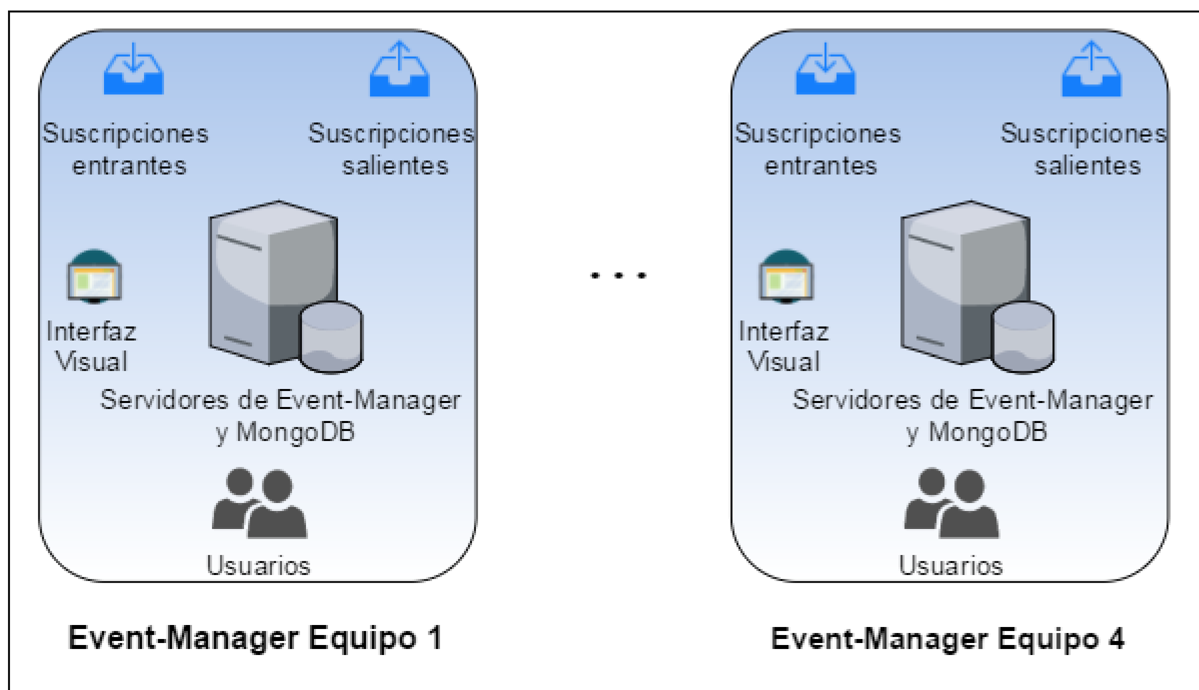

FIGURA 6.1: Configuración de un sistema de Event-Manager por equipo.

Es decir que, en total, este esquema implica 16 nodos (como mínimo) para cuatro instalaciones de Event-Manager. Si bien estos nodos son relativamente *baratos* (Despegar.com cuenta con una plataforma de servidores *virtualizados* muy grande), este un uso poco eficiente de los recursos. El fundamento detrás de contar con una instalación por cada equipo es hacer más fácil la tarea de controlar las suscripciones y las notificaciones al desarrollador, permitiendo a un administrador configurar usuarios con distintos roles. La interfaz solamente filtra las opciones disponibles de acuerdo a los roles del usuario que inició sesión. Con la característica "Grupos de usuarios" (ver Fig. 6.2), se podría con una sola instalación de Event-Manager contemplar grupos de usuarios que emulen el funcionamiento de un Event-Manager exclusivo. Cada grupo contaría con sus propios usuarios con los distintos roles, con sus suscripciones, y, lo más importante, su propia interfaz visual, que no es más que la misma interfaz visual que actualmente posee Event-Manager, pero configurada para mostrar a todos los usuarios de un grupo las configuraciones pertenecientes al mismo. De esa forma, se logra aislar la visualización y control de notificaciones y suscripciones a cada equipo y se le puede dar la *sensación* al usuario de estar usando un Event-Manager exclusivo, cuando en realidad están utilizando una instalación compartida.

6.4 Soporte para otros protocolos

Hemos comentado varias veces a lo largo de este trabajo que una de las líneas con las que se trabajó desde la concepción de Event-Manager fue permitir que la integración de nuevos

protocolos de envío y recepción de notificaciones fuera un proceso sencillo. Teniendo esto en cuenta, sería ventajoso soportar protocolos como MQTT en Event-Manager.

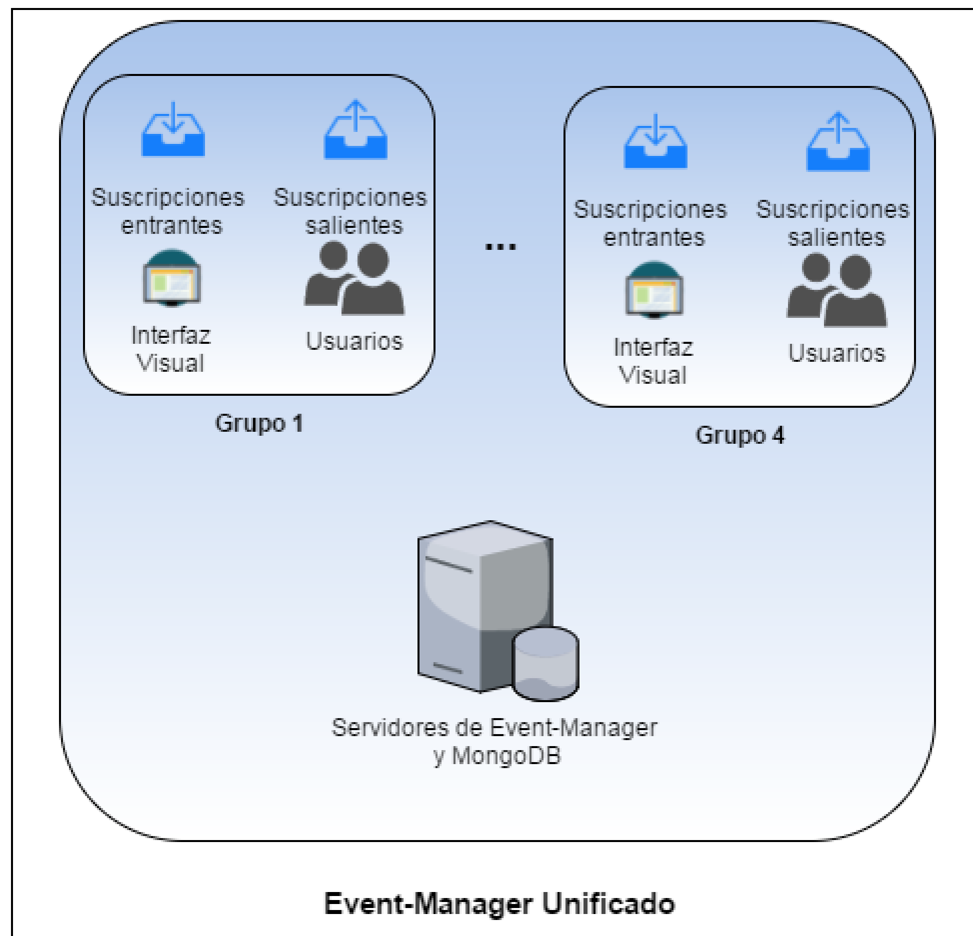

FIGURA 6.2: Configuración de Event-Manager unificado para varios equipos.

6.5 Resumen del capítulo

En este capítulo hemos descripto varias de las mejoras más destacadas que sería importante que Event-Manager implemente en un futuro cercano. Estas mejoras le sumarían valor a su rol en los sistemas en los que actualmente funciona y proporcionaría más razones para que se implemente en sistemas en los que aún no se utiliza.

Capítulo 7

Conclusión

A lo largo de este trabajo hemos desarrollado y presentado el marco teórico con los lineamientos necesarios para implementar una solución que permita delegar el control de notificaciones en una organización establecida dentro de las definiciones de la Arquitectura Orientada a Servicios. La aplicación propuesta permite:

- Delegar el control y reenvío de las notificaciones de un sistema de aplicaciones, relegándoles esa responsabilidad.

- Facilitar la visualización y administración de las notificaciones entrantes y salientes del sistema.

- Permitir con facilidad la recuperación manual en caso de que las aplicaciones receptoras hayan fallado en realizar las tareas que debían ejecutar cuando recibieran las notificaciones.

- Salvar la mayoría de las fallas comunes de manera automática y sin intervención humana mediante una política de reintentos configurable.

- Reducir significativamente el tiempo que toma probar una nueva funcionalidad y dar de alta o de baja una aplicación.

- Mejorar la estabilidad, confiabilidad y los tiempos de respuesta del sistema en su totalidad gracias al uso de mecanismos sencillos, documentados, y correctamente probados, en detrimento de soluciones diferentes desarrolladas en cada una de las aplicaciones que comprenden a éste.

A partir de este marco teórico se implementó Event-Manager, una aplicación que permite disminuir la tasa de errores por fallas en la recepción y procesamiento de notificaciones, y que provee una interfaz práctica para poder verificar en tiempo real el correcto funcionamiento del sistema.

Esta aplicación luego fue utilizada en cuatro equipos en Despegar.com, una organización orientada a SOA con miles de notificaciones generadas a diario. La utilización de esta solución superó las expectativas y generó beneficios tales como:

- Una disminución sensible de los errores recuperables en el procesamiento de las notificaciones.
- Una reducción importante de los tiempos de desarrollo de las aplicaciones que funcionan con notificaciones.
- Mejoras en los tiempos de procesamiento de dichas notificaciones.
- Permitir un control ágil de las suscripciones minimizando la dependencia con otros equipos.
- Permitir a todo el equipo en general, tanto desarrolladores como mesa de soporte y *testers* la visualización del estado y el contenido de las notificaciones, lo que facilita el diagnóstico de problemas y también disminuye la carga de los desarrolladores.

Finalmente, se puede decir que la implementación de Event-Manager en Despegar.com fue la solución definitiva a un problema histórico que ya había intentado ser resuelto de distintas maneras, sin éxito.

Glosario

API: Interfaz de programación de aplicaciones (del inglés: *Application Programming Interface*).
ESB: Bus de servicios empresarial (del inglés: *Enterprise Service Bus*).
FIFO: Algoritmo de procesamiento de colas. Primero en llegar, primero en salir (del inglés: *First in, First Out*).
IP: Protocolo de Internet (del inglés: *Internet Protocol*).
JSON: JavaScript Object Notation.
MQTT: Message Queue Telemetry Transport.
REST: Transferencia de Estado Representacional (del inglés: *Representational State Transfer*).
SOA: Arquitectura Orientada a Servicios (del inglés: *Service Oriented Architecture*).
SQL: Lenguaje de consulta estructurado (del inglés: *Structured Query Language*).
URI: Identificador de recursos uniforme (del inglés: *Uniform resource identifier*).
URL: Localizador de recursos uniforme (del inglés: *Uniform resource locator*).
XML: Lenguaje de marcas extensible (del inglés: *eXtensible Markup Language*).

Referencias bibliográficas

[1] T. Sobh, K. Elleithy. (2013). *"Emerging Trends in Computing, Informatics, Systems Sciences, and Engineering".*

[2] A. Adya, G. Cooper, D. Myers y M. Piatek. Google Inc. (2011). "*Thialfi: A Client Notification Service for Internet-Scale Applications".* [En línea]. Disponible: http://research.google.com/pubs/pub37474.html

[3] M. Breest. (2006). "*An Introduction to the Enterprise Service Bus".*

[4] Microsoft Corporation. (2006). *"La Arquitectura Orientada a Servicios (SOA) de Microsoft aplicada al mundo real".* Disponible: http://download.microsoft.com/download/c/2/c/c2ce8a3a-b4df-4a12-ba18-7e050aef3364/070717-Real_World_SOA.pdf

[5] N. M. Josuttis. *"SOA In Practice. The Art of Distributed System Design"*, 2007.

[6] L. Hernández Cervantes, A. J. Santillán González, y E. Caballero Cruz Reyna (2003). *"Maestros y Esclavos. Una aproximación de los Cúmulos de Computadoras"* [En línea]. Revista Digital Universitaria. 30 de junio 2003, vol. 4, nro. 2. Disponible: http://www.revista.unam.mx/vol.4/num2/art3/jun_art3.pdf

[7] Ascens (Telefónica). (2014). *"Whitepaper: BBDD NoSQL"* [En línea]. Disponible: http://www.acens.com/wp-content/images/2014/02/bbdd-nosql-wp-acens.pdf

[8] J. Richards. (2015). Hadoop 360 Blog. *"Advantages and Disadvantages of NoSQL databases – what you should know"* [En línea]. Disponible: *http://www.hadoop360.com/blog/advantages-and-disadvantages-of-nosql-databases-what-you-should-k*

[9] J. J. Gutiérrez, *"¿Qué es un framework web?"*, 2005.

[10] O. Colomina Pardo (Universidad de Alicante). (2012). *"Spring MVC"* [En línea]. Disponible: http://expertojava.ua.es/j2ee/publico/spring-2012-13/sesion03-apuntes.html

[11] R. Huges. (2013). *"Getting Started with Hazelcast".* [En línea]. Disponible: https://www.javacodegeeks.com/2013/11/getting-started-with-hazelcast.html

[12] L. Hernández. (2013). *"Redis – Tu base de datos libre en memoria"* [En línea]. Disponible: https://leninmhs.wordpress.com/2013/08/18/redis-tu-base-de-datos-libre-en-memoria/

[13] Ing. A. C. Cortés. (2008). *"Aspectos básicos de redes"*. [En línea]. Disponible: http://www.ie.itcr.ac.cr/acotoc/CISCO/R&S%20CCNA1/R&S_CCNA1_ITN_Chapter10_Capa%20de%20aplicacion.pdf

[14] IETF. (1999). "*Hypertext Transfer Protocol -- HTTP/1.1 - RFC 2616"*. [En línea]. Disponible: https://tools.ietf.org/html/rfc2616

[15] Dr. M. Elkstein. (2008). *"Learn REST: A tutorial".* [En línea]. Disponible: http://rest.elkstein.org/2008/02/what-is-rest.html

[16] D. Tcherevik. (2003). *"A Publish/Subscribe Mechanism for Web Services"*. [En línea]. Revista Web Services Journal, vol. 3, nro. 2. Disponible: http://www2.sys-con.com/itsg/virtualcd/webservices/archives/0302/tcherevik/index.htm

[17] T. Bray. *"ECMA-404 The JSON Data Interchange Standard".* [En línea]. Disponible: http://www.json.org/json-es.html

[18] MuleSoft Inc. *"What is Mule ESB?"*. [En línea] Disponible: https://www.mulesoft.com/resources/esb/what-mule-esb

[19] K. Wähner. (2013). *"Choosing the Right ESB for Your Integration Needs".* [En línea]. Disponible: https://www.infoq.com/articles/ESB-Integration

[20] Dra. P. Bazán, *"Un modelo de integrabilidad con SOA y BPM"*, Tesis de maestría en Redes de Datos. FI, UNLP, 2009.

[21] P. Krzyzanowski (18 de Febrero de 2015). *"Process Scheduling: Who gets to run next?".* [En línea] Disponible: https://www.cs.rutgers.edu/~pxk/416/notes/07-scheduling.html

[22] Anónimo. *"Spring Framework Reference: Task Execution and Scheduling".* [En línea] Disponible: http://docs.spring.io/spring/docs/current/spring-framework-reference/html/scheduling.html#scheduling-task-executor

Anexo 1

Modelo de datos

El modelo de datos está compuesto por varias entidades cada una con distinto fin. En este anexo explicaremos en detalle la estructura y función de cada una de ellas.
En la Figura A1.1 a continuación se presenta el diagrama de modelo con las entidades y sus campos.

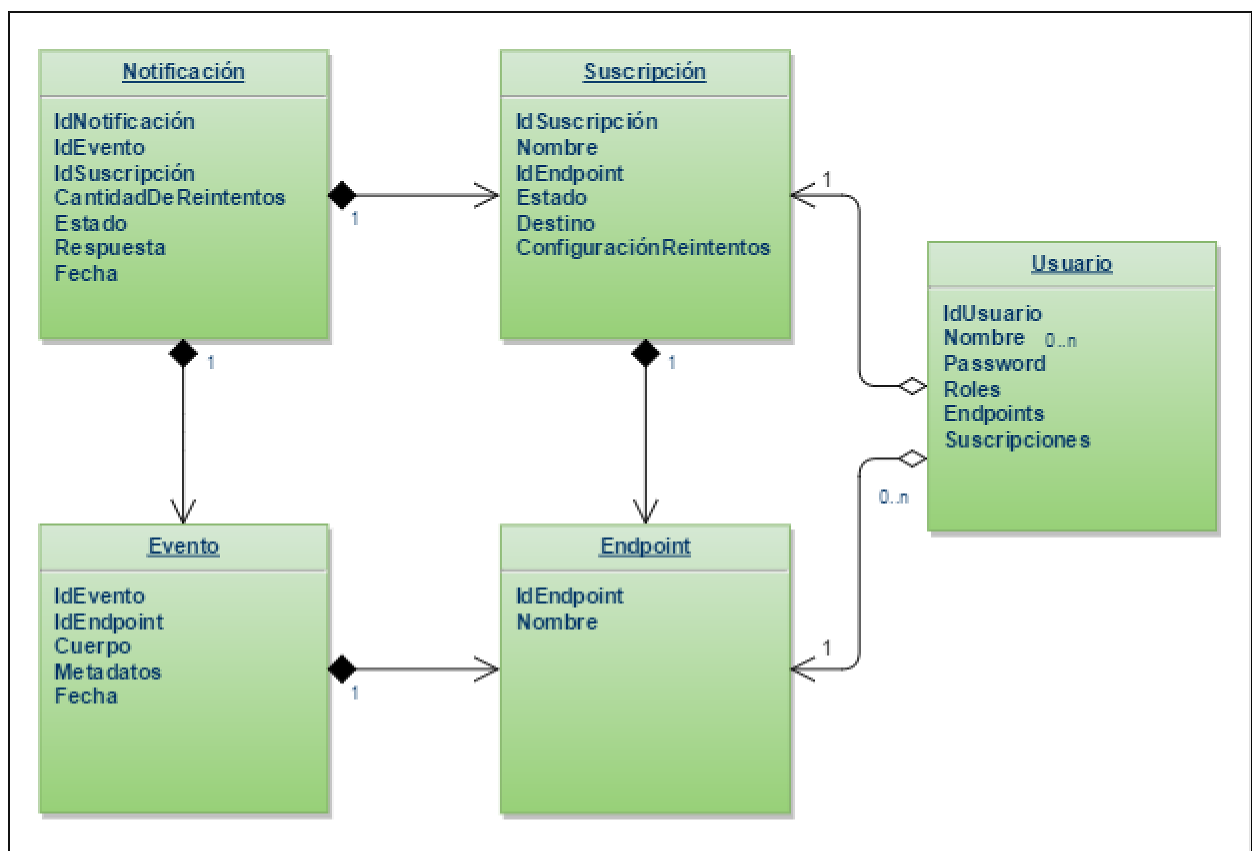

FIGURA A1.1: Diagrama de modelo de Event-Manager.

- **Suscripción**: Es la entidad que representa al cliente interesado en ser notificado cuando un evento arriba a Event-Manager.
  Consta de:
  - **Nombre**: Es el nombre con el que será identificada la suscripción. Este nombre es único en toda la aplicación, no se puede repetir.
  - **ID de Endpoint**: Es el identificador del endpoint al que se suscribe. Todos los eventos recibidos en ese endpoint serán remitidos a este suscriptor.

- **Estado**: Es el estado de la suscripción. Puede estar **activa** (le serán remitidos todos los eventos recibidos), **suspendida** (los eventos no le serán remitidos, pero se almacenarán para su posterior envío cuando la suscripción se reactive), o **desactivada** (los eventos le serán reenviados en ningún momento).
- **Destino**: Es la dirección a la que los eventos serán reenviados. En el caso de una suscripción HTTP, esta dirección es una URL.
- **Cantidad y retardo de reintentos (opcional):** Si se desea, se puede configurar a una suscripción para que, en caso de fallo, se vuelva a intentar notificar al suscriptor. Es posible configurar la cantidad de reintentos y el tiempo que debe esperarse antes de cada reintento.

- **Evento**: Es el mensaje que inicia el flujo de trabajo, como se puede ver en la Figura 4.1. Es disparado por un agente externo, recibido por Event-Manager, persistido, y luego remitido a los suscriptores. El modelo del Evento cuenta con los siguientes datos:
  - **Endpoint**: Es el punto de entrada por el que la notificación fue recibida. Este punto de entrada es al que los suscriptores se asocian, y permitirá determinar a quiénes hay que reenviarle el evento.
    Por ejemplo, para un Evento HTTP, el punto de entrada será una URI.
  - **Cuerpo**: Contiene el cuerpo del mensaje recibido, si está disponible.
  - **Metadatos**: Los datos adicionales como encabezados (si el evento recibido es HTTP), parámetros en la URL, entre otros.
  - **Fecha**: La fecha y hora en la que se recibió el mensaje.

- **Endpoint**: Es el punto de entrada de los Eventos. Se compone por:
  - **Nombre**: Identificador del Endpoint. En el caso de un Endpoint HTTP, el nombre será la URI relativa a Event-Manager al que los publicadores deberán enviar las notificaciones.

- **Notificación:** Esta entidad representa el reenvío del Evento a una Suscripción. Se genera una por Evento recibido y Suscripción. Es decir, si hay tres suscripciones asociadas a un Endpoint, y este recibe un Evento, se crearán tres notificaciones. Una Notificación contiene los siguientes campos:
  - **ID de Evento:** El identificador del Evento que será notificado al Suscriptor.
  - **ID de Suscripción:** El identificador de la Suscripción a la que pertenece.
  - **Cantidad de reintentos:** Cantidad de veces que se reintentó enviar la notificación.
  - **Estado:** El estado actual de la notificación, y el histórico de estados por el que la misma pasó. Estos estados pueden ser:
    - *Exitosa*: La notificación se creó y entregó exitosamente.
    - Encolada: La notificación se creó, pero no puede ser procesada aún. Esto pasa cuando se están procesando muchas notificaciones en un instante.

        - *Corriendo*: La notificación está creada, y en proceso de entrega o reintentando.
        - *Pausada*: La notificación está en espera a que la Suscripción a la que pertenece pase de estado Suspendida a estado Activa.
        - *Notificada*: Es un caso especial en que la notificación está entregada y aguardando a que Event-Manager reciba la confirmación de que ha sido procesada por el Suscriptor.
        - *Fallida*: La notificación alcanzó la cantidad de reintentos configurada sin éxito.
        - *Descartada*: Es un estado en el que la notificación no es más tomada en cuenta para ser procesada. Se puede descartar una notificación manualmente, o se descartan las notificaciones en estado Corriendo cuando una Suscripción pasa a estado Desactivada.
    - **Respuesta**: En caso de que la respuesta del Suscriptor al recibir la Notificación tenga cuerpo, se persiste para posterior control. Útil cuando ocurre un error en un Suscriptor.
    - **Fecha:** Fecha de creación de la Notificación.

- **Usuario**: Se utiliza para el autorizar acceso a la interfaz web y a diferentes funcionalidades de la misma. El Usuario cuenta con:
    - **Nombre**: El nombre con el que se identificará en el sistema.
    - **Password**: La contraseña del mismo.
    - **Rol**: El rol del usuario. Este rol puede ser **administrador** (tiene poder completo sobre toda la aplicación), **usuario** (tiene poder sobre sus Suscripciones), y **solo lectura** (puede ver todo, pero no puede modificar nada).

Anexo 2

Manejo de concurrencia

Una de las más importantes características requeridas para el diseño de Event-Manager en la sección "Propuesta de solución" (Capítulo 1, Inciso 2) es la **eficiencia**. Si bien por sí mismo este es un concepto muy general, como lineamiento se refiere a que el sistema debe completar la ejecución de las tareas lo más rápido posible, sin demoras innecesarias, y haciendo el mejor aprovechamiento de los recursos disponibles (tiempo, hardware, etc.). Sin embargo, este uso de los recursos no puede ir en detrimento de la estabilidad y confiabilidad del sistema. Por ejemplo, si se usara toda la memoria disponible para procesar las notificaciones de una suscripción, y en el intervalo se recibieran notificaciones de otra suscripción, éstas no podrían ser procesadas hasta que las notificaciones de la primera suscripción liberen memoria (Figura A2.1).

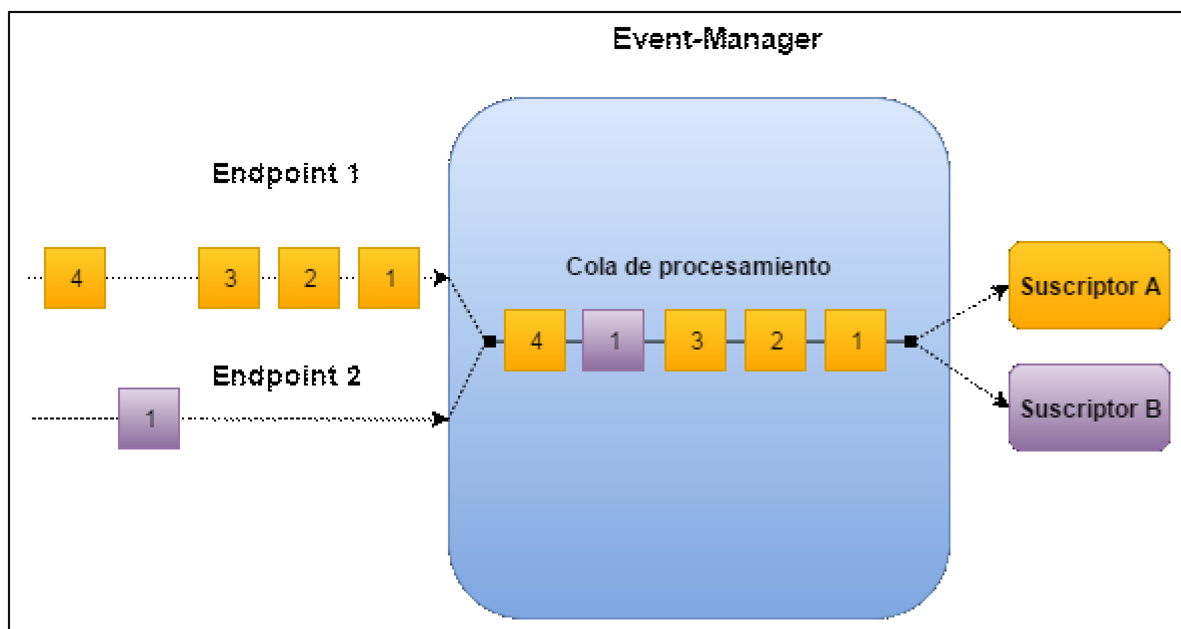

FIGURA A2.1: Flujo de notificaciones con una cola única de procesamiento

Podemos observar en la Figura A2.1 el funcionamiento de la cola de procesamiento interna *FIFO* (Primero en llegar, primero en salir) de Event-Manager, la cual utiliza toda la memoria disponible, y en teoría hace el mejor uso de los recursos. Sin embargo, este esquema trae consigo un problema serio, como se explicaba anteriormente, que es que las notificaciones que se reenvían a **Suscriptor B** se verán demoradas por el procesamiento de las notificaciones de **Suscriptor A**. Este problema se puede agravar considerablemente si la cantidad de notificaciones en la cola de procesamiento es muy grande.

Para contrarrestar el inconveniente de la demora innecesaria en el procesamiento de notificaciones, existen varios mecanismos posibles. Por ejemplo, se podría utilizar una cola con prioridad (y no FIFO o "primero en llegar, primero en salir") junto con una política de *aging* [21], lo que permitiría que la prioridad de una notificación de una suscripción que lleve un cierto tiempo (configurado arbitrariamente) sin ser procesada, aumente a medida que pasa el tiempo. El problema de este algoritmo es que es de implementación relativamente compleja y, si bien garantiza que la notificación se va a procesar, y lo va a hacer más rápido que en una cola sin prioridad, no garantiza la eliminación de demoras innecesarias entre distintas suscripciones, ya que la política de *aging* es gradual y requiere de un cierto tiempo para entrar en funcionamiento. En cambio, se aplicó una solución de colas separadas por suscripción (ver Figura A2.2).

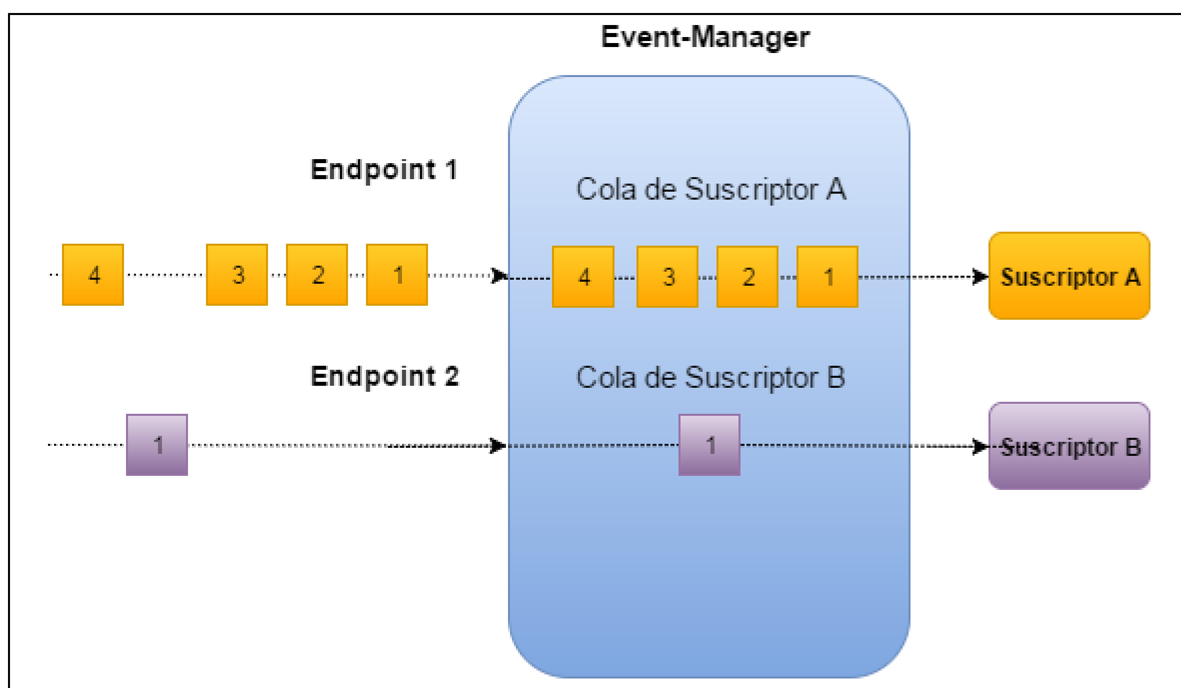

FIGURA A2.2: Flujo de notificaciones con una cola de procesamiento por cada suscriptor

La implementación de colas por suscripción consiste en tener una cola de procesamiento por cada una de las suscripciones. De esta manera se garantiza que no haya demoras innecesarias entre las distintas suscripciones. Esto permite que el comportamiento de Event-Manager sea el esperado independientemente de si una suscripción se ve saturada por un exceso de notificaciones. La demora quedará exclusivamente limitada a la cola de esa suscripción, sin afectar los tiempos de respuesta de los demás.
A continuación, analizaremos algunas secciones del código utilizado para la implementación de las colas de notificaciones por suscriptor (Figura A2.3).

Funcionamiento

El fragmento de código de la Figura A2.3 es la implementación de la cola de notificaciones

```
@Service
public class CustomExecutorService {

    @Value("${threadpool.size}")
    private int poolSize;
    private HashMap<String, ThreadPoolTaskExecutor> executors;
    executors = new HashMap<String, ThreadPoolTaskExecutor>();
    private static final Logger LOGGER;
    LOGGER = Logger.getLogger(CustomExecutorService.class);

    /**
    * Returns a {@link ThreadPoolTaskExecutor} from the executors map
    * for a given subscription id. If it doesn't exist in the map,
    * it creates a new one and places it in the map.
    * @Param String id
    * @Returns ThreadPoolTaskExecutor
    **/
    public ThreadPoolTaskExecutor getExecutor(String id) {
        LOGGER.info("Getting executor for subscription {}.", id);
        if (this.executors.get(id) != null) {
            LOGGER.info("Returning executor for {} from map.", id);
            return this.executors.get(id);
        } else {
            LOGGER.info("Creating executor for {}.", id);
            ThreadPoolTaskExecutor ex = new ThreadPoolTaskExecutor();
            ex.setCorePoolSize(this.poolSize);
            ex.setMaxPoolSize(this.poolSize);
            ex.setThreadNamePrefix(id);
            ex.initialize();
            this.executors.put(id, ex);
            return ex;
        }
    }
}
```

FIGURA A2.3: Código del servicio encargado de obtener el ejecutor para cada suscripción.

por suscripción. Este funcionamiento se lleva a cabo otorgándole a cada suscripción un ejecutor (representado por la clase ThreadPoolTaskExecutor de Spring), único mientras la suscripción tenga notificaciones por procesar. Este ejecutor puede procesar un número

máximo de notificaciones a la vez definido por el parámetro poolSize. El mismo es configurable y su valor es de 20 de manera predeterminada. Si hay poolSize notificaciones procesándose en un instante, y llega una notificación poolSize+1, la misma se encolará y será procesada cuando se libere alguno de los hilos del ejecutor. Esta configuración permite que las notificaciones se encolen en su propia cola, y que el procesamiento de las notificaciones de una suscripción no interfiera con el procesamiento de las notificaciones de las demás suscripciones.

ThreadPoolTaskExecutor

La clase ThreadPoolTaskExecutor [22] (Figura A2.4) es una implementación de la abstracción TaskExecutor de Spring, que permite la ejecución asincrónica de código. Esta ejecución asincrónica significa que, ante un nuevo requerimiento, se crea un nuevo hilo (o se reutiliza uno existente) para procesar la tarea, pero el hilo principal de ejecución no se detiene a esperar que la misma finalice. Se podrán crear tantos hilos para ese ejecutor como se hayan configurado por el parámetro poolSize (por defecto 1). Si el parámetro poolSize es alcanzado, es decir, no hay más hilos de ejecución disponibles, se encolan los nuevos requerimientos hasta que se alcance el parámetro queueCapacity o se libere un hilo. El parámetro queueCapacity es por defecto ilimitado, aunque en la realidad está obviamente limitado a la cantidad de memoria disponible en el sistema. Una vez que se alcanza queueCapacity, el ejecutor creará nuevos hilos hasta alcanzar el parámetro maxPoolSize (por defecto del mismo tamaño que poolSize). Una vez que la cantidad de hilos alcanza maxPoolSize, el ejecutor comienza a rechazar nuevos requerimientos y lanzará una excepción de tarea rechazada.

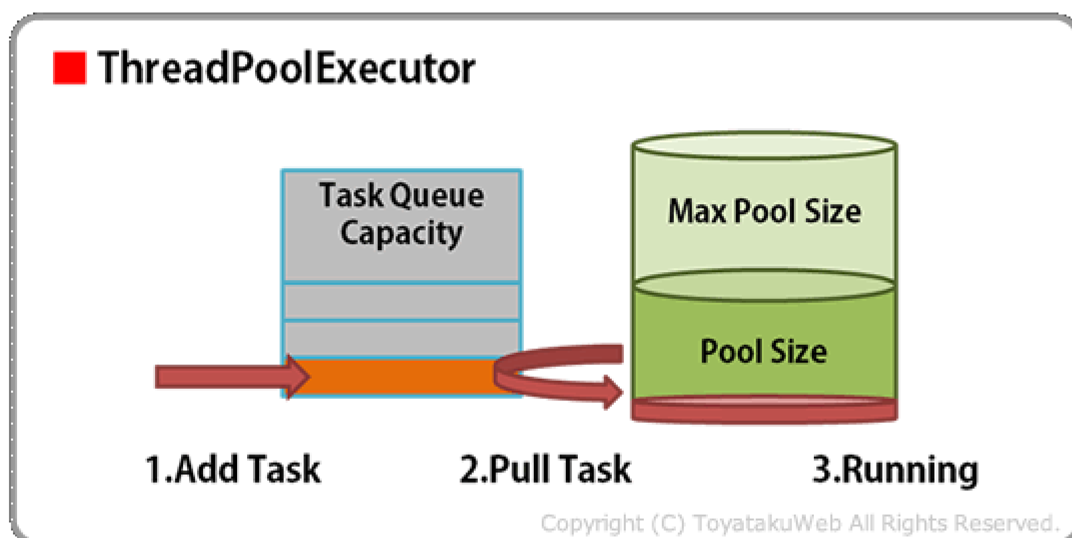

FIGURA A2.4: ThreadPoolTaskExecutor.

Gracias a la implementación de los mecanismos comentados anteriormente, se han conseguido dos grandes ventajas:

- La primera consiste en lograr que el procesamiento de las notificaciones funcione en un ambiente controlado, es decir que **el mal funcionamiento de una suscripción no afecte el procesamiento de las notificaciones de otra suscripción.**
- La segunda, **desacoplar al publicador del procesamiento de las notificaciones de los suscriptores**.

Para concluir, se detalla este comportamiento a través de un diagrama de secuencia (Figura A2.5), el cual describe paso a paso los procesos disparados por la recepción de una notificación con el mecanismo de colas de notificaciones por suscripción descripto en este anexo.

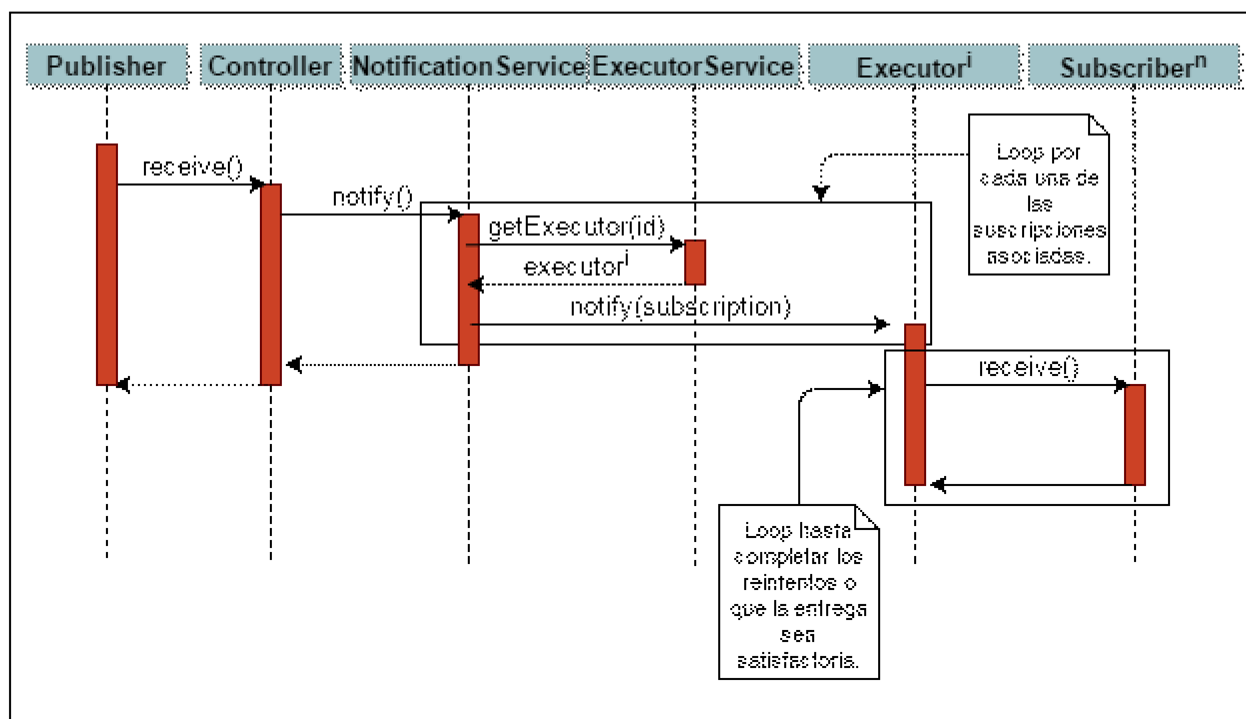

FIGURA A2.5: Diagrama de secuencia del procesamiento de una notificación.