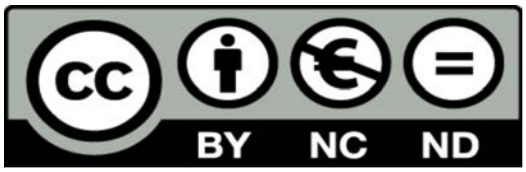
UNIVERSIDAD DE LA RIOJA

TRABAJO FIN DE ESTUDIOS

Título
Comunicación HTTP de Microservicios
Autor/es
Fernando Arconada Orostegui
Director/es
Eloy Javier Mata Sotés y Jesús María Aransay Azofra
Facultad
Titulación
Máster universitario en Tecnologías Informáticas
Departamento
Curso Académico
2015-2016



Trabajo de Fin de Máster

Comunicación HTTP de Microservicios

Autor:

Fernando Arconada Oróstegui

Tutor/es: Eloy Javier Mata, Jesús María Aransay

Fdo.*: Fernando Arconada*

MÁSTER:
Máster en Tecnologías Informáticas (853M)

Escuela de Máster y Doctorado

AÑO ACADÉMICO: 2015/2016

1. Resumen

El trabajo consiste en el estudio de sistemas de integración de microservicios. Se pone el foco en aquellos en los que la comunicación se realiza con HTTP. En especial dentro de HTTP se profundiza más en cómo realizar peticiones asíncronas y subscripciones a eventos. Se destaca GraphQL como lenguaje que permite evolucionar mejor las APIs que REST.

Durante el desarrollo del trabajo, y como anexos, se realizan dos observaciones que han surgido relativas al tamaño y seguridad de los containers de microservicios y la posible estandarización de una cabecera X-Response-To para facilitar la comunicación a tres bandas.

2. Abstract

This paper is about integration of microservices. It's specially focused in those one whose comunication it's based on HTTP. Especially within HTTP it deepens more on how to make asynchronous requests and event subscriptions. GraphQL excels as a language that allows for better evolution of APIs than REST ones.

During the development work and annexes, I made two comments that have arisen concerning the size and safety of containers of microservices and possible standardization of a header X-Response-To to facilitate communication trialogue.

Comunicación HTTP de Microservicios

3. Plan de trabajo

Para desarrollar el trabajo se emplearon lenguajes de programación de tipo interpretado como JavaScript y PHP. La ventaja de emplear estos dos lenguajes que es que se trabaja de forma más rápida que con lenguajes compilados y tenemos dos paradigmas diferentes: EventLoop con Javascript + NodeJs y PHP que se ejecuta como módulo o CGI. Además muchas de las librerías a estudiar soportan estos lenguajes.

El 80% del tiempo se invirtió investigando sobre publicaciones y artículos relacionados con microservicios así como estudiando cómo desarrollan las arquitecturas de microservicios varias de las startups que publican información al respecto. Esta parte de estudio es fundamental puesto que las arquitecturas de microservicios son algo evolutivo y no estandarizado.

Por otro lado para probar diversas funcionalidades se desarrollaron pequeñas implementaciones para poner en práctica las ideas. Por ejemplo se desarrolló un pequeño programa de prueba con NodeJS para probar las subscripciones de GraphQL. El código de este programa está en el texto del documento.

El tiempo invertido en la parte de estudio superó las 400h y se comenzó en la segunda parte del Máster.

4. Contexto

Hoy en día cada vez se habla más de la comunicación interna de las aplicaciones, cosa que hace unos años no se planteaba puesto que casi todas las aplicaciones eran monolíticas. Con el tiempo surgió el concepto de SOA (Service Oriented Architecture), pero es algo que no llegó a cuajar en el desarrollo de pequeñas y medianas aplicaciones por la complejidad que acarreaba. Sólo las grandes empresas se planteaban arquitecturas SOA e indiscutiblemente iban ligadas a SOAP (Simple Object Access Protocol). SOAP ofrece un planteamiento serio y completo para las arquitecturas orientadas a servicios, pero acarrea una complejidad más alta que otras soluciones. El ritmo frenético de creación de startups y la presión por obtener resultados de forma rápida impone cada día más las soluciones tipo “bazar” frente a las soluciones “catedral” (La catedral y el bazar, Eric s. Raymond.).

A estos pequeños bazares de software se les exige casi desde el principio de su creación criterios más propios de catedrales. Aunque no estén implementadas, tienen que tener prevista la escalabilidad y los planes de planes de evolución que den una respuesta casi instantánea a crecimientos exponenciales y cambios en la funcionalidad del software.

A raíz de todo esto surgen entre otros los movimientos de DevOps, entrega continua, automatización y los microservicios

5. Qué son los microservicios

Los microservicios son pequeños servicios autónomos que trabajan juntos. Las principales características de los micro servicios son:

- Pequeños y enfocados en hacer una cosa y hacerla bien. Se crean al estilo de los servicios de Unix. Refuerzan el principio de Robert C. Martin "Single Responsability Principle", que viene a decir algo así como "Mantén juntas las cosas que pueden cambiar por la misma razón, y separar las cosas que pueden cambiar por diferentes razones". La idea de hacer las cosas más pequeñas nos lleva a la pregunta de ¿cómo de pequeñas?
- Autónomos. Nuestro microservicio es una entidad separada. Debe de poder ser desplegada como un servicio aislado o en una nube PAAS, hoy incluso puede tener su propio sistema operativo. Se tratará de evitar empaquetar múltiples servicios en la misma máquina, aunque el concepto de máquina hoy en día sea un poco difuso.

El estilo de arquitectura de microservicios lo está invadiendo todo. Los microservicios son el primer estilo de arquitectura que aparece tras la revolución de DevOps, es el primero en abrazar completamente las prácticas de ingeniería de entrega continua. Es también un ejemplo de arquitectura evolutiva que soporta los cambios incrementales no disruptivos como primer principio en los diferentes niveles estructurales de la aplicación.

Los escépticos dicen que los microservicios no son más que las prácticas de SOA del año 2000 recalentadas. La realidad es que los microservicios son un ejemplo de evolución convergente que emergen del desarrollo modernos, testing, y prácticas de despliegue.

Los usuarios pioneros de las arquitecturas de microservicios dicen que cuando usan Agile de forma autónoma para hacer ingeniería de infraestructuras y una aproximación de DevOps para la entrega, los microservicios permiten una cadencia mucho más rápida de desarrollo de aplicaciones.

La fuerza de los microservicios proviene de su naturaleza no prescriptiva. No hay una especificación de la industria de avance lento y formal. En cambio los

microservicios surgen como un patrón de desarrollo que es practicado, refinado y redefinidos por sus pioneros.

Uno de los conceptos fundamentales a recordar es que la arquitectura de microservicios es de tipo “comparte lo menos posible”, que pone un fuerte énfasis en los contextos delimitados. En cambio SOA tiende a ser de tipo “comparte todo lo que puedas” y pone el foco en la abstracción y la reutilización de la funcionalidad de negocio.

Arquitecturas evolutivas

Una creencia común en el software es que una vez que se establecen algunos elementos de arquitectura son difíciles de cambiar posteriormente. Una arquitectura evolutiva se diseña para hacer cambios incrementales en la arquitectura como primer principio. Las arquitecturas evolutivas están apareciendo porque el cambio ha sido históricamente difícil de anticipar y costoso de modernizar. Si la evolución del cambio está metido dentro de la propia arquitectura, entonces el cambio se vuelve más sencillo y barato, permitiendo cambios en las prácticas desarrollo, prácticas de versionado y, en general, agilidad.

Los microservicios cogen su definición del principio de contexto delimitado, haciendo la división lógica descrita en el libro de Eric Evans, *“Domain Driven Design”* un concepto físico. Los micro servicios consiguen esta separación vía prácticas avanzadas de DevOps como la provisión de máquinas, testing y despliegue automático. Gracias a que cada servicio esta desacoplado de los otros servicios (a nivel estructural) el reemplazar un servicio con otro recuerda a cambiar una pieza de lego por otra.

Los microservicios aportan una ventaja de gestión evidente que ha hecho que las startups las adopten rápidamente. Al ser contextos delimitados cada microservicio puede tener su pila de desarrollo completamente independiente de los otros y emplear la tecnología que más convenga. Así se pueden contratar a los ingenieros más apropiados para cada servicio que se responsabilizarán completamente de su desarrollo incluyendo la parte de sistemas. También se pueden emplear las tecnologías más apropiadas a cada problema.

Características de las arquitecturas evolutivas

Las arquitecturas evolutivas tienen varias características en común.

- Modularidad y acoplamiento: La habilidad de separar componentes con fronteras bien definidas tiene el beneficio obvio para un desarrollador que quiere hacer un cambio no disruptivo. La ausencia de cualquier elemento de arquitectura, la arquitectura conocida como gran bola de barro, no soporta la evolución porque carece de compartimentación.
- Organizadas alrededor de las características del negocio: De forma creciente, las arquitecturas modernas exitosas exhiben modularidad a nivel de arquitectura de dominio, inspiradas por el "*Domain Driven Design*". Las arquitecturas orientadas a servicios se diferencian de las tradicionales SOA principalmente por su estrategia de partición: SOA tradicionalmente se ha particionado en capas técnicas, mientras casa arquitecturas orientadas a servicios se inspiran en las particiones del dominio para determinar los micro servicios.

6. Objetivo

Queremos construir una aplicación web y móvil moderna. Nuestra aplicación tiene partes que se ejecutan en tiempo real y algunas partes que son más pesadas no deben afectar a la usabilidad. Por otro lado nos plantemos un escenario modesto, de una startup que acaba de empezar, queremos evitar desplegar una infraestructura excesiva con una complejidad y coste de mantenimiento elevado para una aplicación que acaba de empezar. Pero no tenemos que perder de vista que la escalibilidad es importante y hay que poder reaccionar a un futuro crecimiento exponencial de los usuarios.

Las aplicaciones grandes parece que tienen justificado cualquier coste debido a su tamaño pero a las startups que nacen como algo muy pequeño se les exige que puedan crecer de forma inmediata sin tener recursos para ello.

Por estos motivos elegimos una descomposición de la aplicación basándonos en microservicios, de forma que podamos tratar y escalar cada uno individualmente. También evitaremos reducir todo el midleware a la mínima expresión, de forma que nos ahorramos gestionar sistemas como RabbitMq (RabbitMQ: https://www.rabbitmq.com/) que pueden ser complejos.

Los microservicios se comunican entre si entre si normalmente o mediante de un middleware de mensajería o o de forma directa. Cuando se comunican de forma directa la hacen con un protocolo y lenguaje propio o usando HTTP o Thrif (Apache Thrift: https://thrift.apache.org/) o similar. HTTP es la opción más común y además ofrecer ventajas. Por ejemplo, HTTP es ampliamente conocido y además tiene la ventaja de que pasa fácilmente los firewalls y proxys. Por otro lado los microservicios puedes ser consumidos directamente desde los clientes finales. Los microservicios no tienen por qué quedarse en el backend de las aplicaciones y no ser expuestos al exterior, aunque lo común es que esto sea así y el patrón API gateway se impone en este sentido. HTTP tiene la ventaja de que casi todos clientes pueden manejarlo sin necesidad de instalar librerías adicionales.

El protocolo HTTP es un protocolo de tipo peticion/respuesta (request/response). La comunicación se inicia en el cliente que espera una respuesta del servidor. Además la comunicación es uno a uno.

Las aplicaciones web y móviles modernas exigen otros modelos de comunicación que no se limiten a request/response. Por ejemplo las aplicaciones cada vez son más en tiempo real y eso implica que cuando el servidor tiene una información la haga llegar a los clientes

sin esperar a que estos hagan un solicitud. También hay peticiones pesadas que se tardan tiempo en procesar y que el cliente no puede permanecer a la espera bloqueado.

Problema: Peticiones pesadas

El primer paso que debemos dar en una aplicación para hacer que pueda escalar fácilmente es identificar las tareas más pesadas y extraerlas del flujo de procesamiento normal. En general esas tareas se deben poder procesar de forma asíncrona para no retrasar al cliente y ser notificado éste cuando el procesamiento termine. De esta forma aumenta la sensación de interactividad.

Imaginemos un microservicio que se dedica a la conversión de vídeos. Este proceso puede durar varios minutos, pero el cliente no puede quedarse a la espera de que finalice.

Empleando HTTP hay dos soluciones a este respecto:

1. Mantener una conexión abierta
2. Cerrar la conexión según se envía la petición

A la hora de procesar peticiones pesadas es común usar un modelo de productor/consumidor. El productor (de mensajes) suele ser el cliente y el consumidor el microservicio. En una arquitectura basada totalmente en HTTP e intentando eliminar la mayor parte del midleware posible. Es factible que haya varios productores de mensajes pero cada uno de estos debe saber a qué consumidor dirigirlo puesto que la comunicación es directa y no a través del midleware. En las arquitecturas de microservicios se ha desarrollado un clase de midleware que son los “service locators”. Es una clase de midleware que se encarga de ayudar al cliente a descubrir dónde está el servicio que debe utilizar de forma que se siguen comunicando de forma directa pero el cliente no tiene por qué conocer la dirección del servidor. De esta forma es mucho más flexible la arquitectura y se pueden implementar balanceos de carga. Ejemplos de “service locators” pueden ser los registros SRV del DNS o software mas específico como etcd, Consul o Zookeeper.

Problema: recibir notificaciones desde el servidor

Es interesante poder recibir notificaciones desde el servidor en el cliente. Lo métodos de polling continuo no son eficaces y además se puede producir un desfase entre el tiempo en el que se produce el evento y el intervalo de polling que lo detecta.

Al igual que con el problema anterior hay dos formar de poder recibir eventos:

- Escribiendo en un canal establecido
- Haciendo una petición desde el microservicio al cliente en una nueva conexión

Con HTTP 1.1 y especialmente con HTTP 2.0 ya se pueden enviar notificaciones desde el servidor web a los clientes, bien empleando websockets o SSE (Server sent events). Por

otro lado desde la perspectiva de comunicación entre dos microservicios (descartamos la comunicación entre un navegador y un microservicio) ambos podrían tener un servidor web escuchando de forma que puedan iniciar la comunicación desde cualquier extremos de la conexión.

En el ámbito de las notificaciones es común ver el patrón “publish/subscribe” (pub/sub o publicador/subscriptor) en el que puede haber N publicadores y M subscriptores. El mejor ejemplo de la aplicación de estos patrones se da con los brokers de mensajería puesto que es su habitat natural. Hay ejemplos de implementación de publish/subscribe sobre HTTP, como puede ser Cometd, lo que pasa es que al final una buena implementación de pub/sub acaba empleando un broker que sustituye los protocolos tradicionales como JMS o AMQP por HTTP, pero no se evita tener un midleware adicional.

Problema: evolucionar el API

Otro problema a solucionar, no menos importante que los anteriores, y muy común en la aplicaciones actuales es la interfaz de comunicación entre las diferentes APIs. Entendiendo que un API es el contrato entre el cliente y el servidor y que normalmente se gestiona unilateralmente por parte del servidor, obligando de esta forma a actualizar todos los clientes o mantener contratos individualizados para cada uno.

7. Intercomunicación de procesos en un arquitectura de microservicios

En una aplicación monolítica los componentes se invocan uno a otro a nivel de lenguaje o con llamadas de funciones. Una aplicación basada en microservicios es un sistema distribuido en múltiples máquinas. Cada instancia del servicio es típicamente un proceso o una máquina, luego hay que aplicar mecanismo de comunicación entre procesos (IPC).

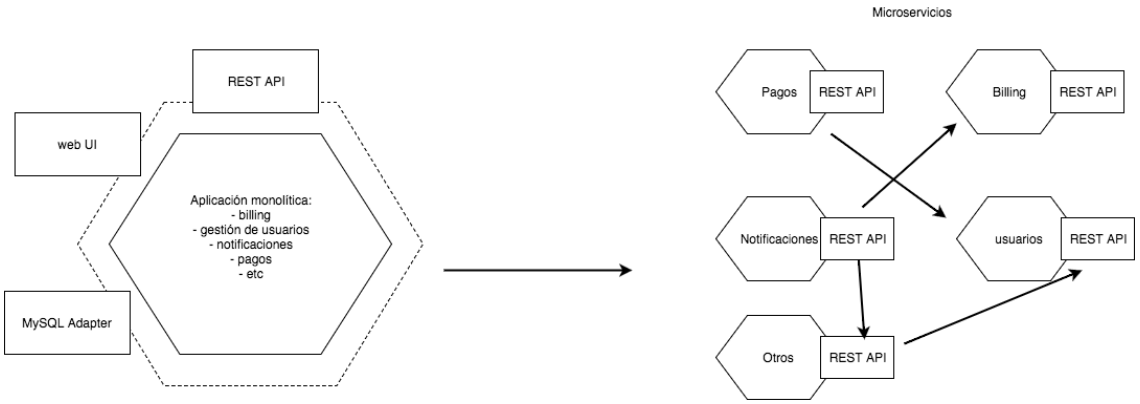


Ilustración 1 Aplicación monolítica Vs Microservicios

Estilos de interacción

Cuando se selecciona un mecanismo de IPC hay que pensar primero cómo van a interactuar los servicios. Hay una variedad de estilos de interacción cliente <-> servicio. Se pueden categorizar siguiendo dos dimensiones. La primera dimensión es si la interacción es uno a uno, o uno a varios:

- Uno a uno: cada petición del cliente es procesada exactamente por una instancia del servicio.
- Uno a varios: cada petición del cliente es procesada por múltiples instancias.

La segunda dimensión es relativa a si la interacción es síncrona o asíncrona:

- Síncrona: El cliente espera una respuesta puntual del servicio y puede incluso bloquearse mientras espera.
- Asíncrona: El cliente no bloquea mientras espera la respuesta, y la respuesta, si la hay, puede que no se envíe inmediatamente.

Tenemos las siguientes clases de interacción uno a uno:

- Request/response: un cliente hace una petición a un servicio y espera la respuesta. El cliente espera que la respuesta llegue de forma puntual (en tiempo). En una aplicación basada en threads, el thread hace la petición y puede quedar bloqueado mientras espera.
- Notificación (interacción en un sentido, sólo request): un cliente envía la request al servicio pero no se espera una respuesta.
- Request/async response: Un cliente envía una petición a un servicio, que responde asíncronamente. El cliente no bloquea mientras espera y se diseña con la presunción de que la respuesta puede tardar en llegar.

Existen los siguientes tipos de interacción uno a varios:

- Publish/subscribe: Un cliente publica un mensaje de notificación, que es consumido por cero o más servicios interesados
- Publish/async responses: Un cliente publica una solicitud (request) y espera un determinado tiempo para obtener respuestas de los servicios interesados

Cada servicio típicamente usa una combinación de estos estilos de interacción. Para algunos servicios, un único mecanismo sencillo de interacción es suficiente. Otros servicios pueden necesitar una combinación de estos mecanismos.

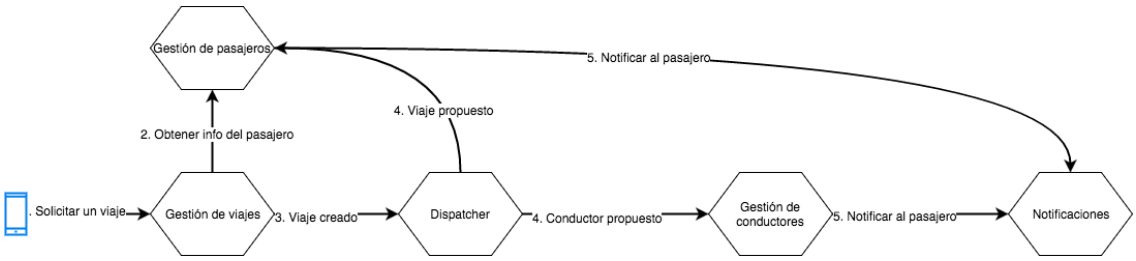


Ilustración 2, Ejemplo de uso de microservicios en una aplicación de gestión de taxis

Los servicios usan una combinación de notificaciones, request/reponse, y publish/subscribe. Por ejemplo un pasajero envía una notificación a la aplicación de gestión de viajes para solicitar una recogida. El servicios de gestión de viajes verifica que la cuenta del usuario está activa usando request/response contra el servicio de pasajeros. El servicio de viajes crea un viaje y usa publish/susbscribe para notificar a otros servicios, incluido el dispatcher, que localiza un conductor disponible.

Definiendo APIs

El API de un servicio es un contrato entre el servicio y sus clientes. Con independencia de la elección de mecanismo de IPC, es importante definir el API de un servicio usando alguna clase de lenguaje de definición de interfaces (IDL). Hay buenos argumentos para usar una aproximación API first (desarrollo en el que se pone el foco en crear primero un API alrededor del cual organizar el resto de la aplicación) para definir servicios. Comienzas el desarrollo de un servicio escribiendo la definición de interfaz y revisándola con los clientes desarrolladores. Haciendo esto desde el principio aumentan las posibilidades de construir un servicio que cumpla con las necesidades del cliente.

La naturaleza de la definición de la API depende del mecanismo de IPC seleccionado. Si se usa mensajería, el API consistirá en los canales para los mensajes y en los tipos de mensaje. Si se emplea http, el API consistirá en las URL y en los formatos de request/response.

Evolución de APIs

El API de un servicio inevitablemente cambia con el tiempo. En una aplicación monolítica el cambio suele ser directo, cambiar el API y actualizar todos los llamantes de ese API. En una aplicación orientada a microservicios suele ser mucho más difícil, incluso si todos los consumidores del API son servicios de la misma aplicación. Normalmente no puedes forzar a todos los clientes a actualizar con el servicio. También es probable que se hagan despliegues incrementales de los servicios, de forma que convivan versiones viejas y nuevas. Hay que tener una estrategia para enfrentarse a estos casos.

Cómo se gestione un cambio en el API dependerá del tamaño del cambio. Algunos cambios son menores y tienen retrocompatibilidad. Puedes por ejemplo añadir atributos a las request o las responses. Se debería seguir el principio de robustez. Los clientes deberían poder hablar con la nueva versión del servicio. El servicio debe proveer valores por defecto para los atributos que falten en las request y los servicios ignorar los atributos extra en las respuestas. Es importante emplear un mecanismo de IPC y formateo de mensajes que permitan evolucionar en API fácilmente.

Algunas veces, sin embargo, hay que hacer cambios mayores, cambios incompatibles en el API. Como se puede forzar a los clientes a actualizar

inmediatamente, el servicio debe soportar versiones viejas del API por algún periodo de tiempo. Si se emplean mecanismos como HTTP REST, una aproximación puede ser embeber el número de versión en la URL. Cada instancia del servicio puede gestionar múltiples versiones simultáneamente. Alternativamente, se pueden desplegar diferentes instancias del servicio y que cada una gestione una versión concreta del API.

Gestión de fallos parciales

En un sistema distribuido hay que tener siempre presente la posibilidad de un fallo parcial. Ya que los clientes y los servicios son procesos separados, un servicio puede no responder de manera puntual a una petición de un cliente. Un servicio puede estar caído por fallo o por mantenimiento. O el servicio puede estar sobrecargado y responder extremadamente lento.

Consideremos una aplicación de venta de productos en la que el servicio de recomendación no responde. Una aproximación simplista podría bloquearse indefinidamente esperando una respuesta. No solamente llevaría a una mala experiencia de usuario, sino que en muchas aplicaciones consumiría un preciosos recurso como es un thread. Eventualmente el runtime podría quedarse sin threads y no responder.

Hay que diseñar los servicios pensando en gestionar posibles fallos parciales.

Una buena aproximación en este sentido es la adoptada por Netflix. Las estrategias para gestionar fallos parciales incluyen:

- Network timeouts: nunca bloquear indefinidamente y siempre usar timeouts cuando se espera una respuesta. Usar timeouts garantiza que los recursos nunca van a estar comprometidos.
- Limitar el número de respuesta extraordinarias. Imponer un número tope de respuesta extraordinarias (extraordinariamente malas) que un cliente puede obtener de un servicio en particular. Si se alcanza el límite es probable que no se deben hacer peticiones adicionales y estos deben fallar inmediatamente.
- Patrón de ruptura de circuito. Consiste en hacer un seguimiento de respuesta las respuestas correctas y los fallos. Si la tasa de error excede un umbral determinado por configuración hay que disparar la apertura del circuito de forma que las siguientes peticiones fallen inmediatamente. Si

un gran número de peticiones están fallando sugiere que el servicio no está disponible. Después de un periodo de timeout los clientes deberían volver a probar y si vuelve a funcionar se vuelve a cerrar el circuito.

- Proveer de plan b. Tener una lógica de plan b implementada por si una petición falla. Por ejemplo, devolver datos cacheados o un valor por defecto como pudiera ser un conjunto vación de recomendaciones.

Netflix Histrix es una librería Open Source para la JVM que implementa estos y otros patrones.

Tecnologías de IPC

Hay muchas tecnologías de IPC para elegir. Los servicios pueden usar mecanismos de comunicación request/response como pueden ser HTTP REST o Thrift. Alternativamente pueden usar mecanismos asíncronos basados en mensajes como AMQP o STOMP. También hay una gran variedad de formatos de mensajes. Los servicios pueden usar formatos legibles, basados en texto como JSON o XML. De forma alternativa pueden usar formatos binarios (que son más eficientes) como Avro (Apache Avro: https://avro.apache.org/) o Protocol Buffers (Protocol Buffers: https://developers.google.com/protocol-buffers/).

Comunicaciones asíncronas basadas en mensajes

Cuando se emplean mensajes, los procesos se comunican con el intercambio asíncrono de mensajes. Un cliente hace una petición a un servicio enviando un mensaje. Si el servicio debe responder lo hace enviando un mensaje de vuelta separado al cliente. Como la comunicación es asíncrona, el cliente no bloquea esperando una respuesta. En lugar de esto el cliente está programado directamente asumiendo que las peticiones no llegarán inmediatamente.

Un mensaje consiste en cabeceras (metadatos como el remitente) y un cuerpo del mensaje. Los mensajes se intercambian en canales. Un número de productores pueden enviar mensajes a un canal. De forma análoga, un número de consumidores pueden recibir mensajes de un canal. Hay dos clases de canales, punto a punto y publish/subscribe. Un canal punto a punto entrega un mensaje solamente a un consumidor de los que está leyendo el canal. Los servicios emplean canales punto a punto para la interacción uno a uno descrita anteriormente. En un canal publish/subscribe cada mensaje se entrega a todos

los consumidores. Los servicios emplean los canales publish/subscribe para la interacción uno a muchos descrita anteriormente.

Hay muchos sistemas de mensajes para elegir. Se debería elegir uno que soporte una amplia variedad de lenguajes. Algunos sistemas de mensajería emplean protocolos estándar como AMQP o STOMP. Otros sistemas tienen protocolos propietarios aunque documentados. Hay un gran número de sistemas de mensajería Open Source para elegir: RabbitMQ, Apache Kafka (http://kafka.apache.org/)Apache ActiveMQ (http://activemq.apache.org/), y NSQ (http://nsq.io/). Desde un punto de vista de alto nivel todos soportan alguna forma de mensajes y canales. Todo tienden a ser confiables, escalables y de alto rendimiento. Sin embargo, hay diferencias significativas en detalles del modelo de broker de mensajes.

Hay muchas ventajas al usar mensajería:

- Desacopla el cliente del servicio. El cliente simplemente hace una petición escribiendo un mensaje en el canal. El cliente desconoce totalmente las instancias de los servicios. No necesita emplear un mecanismo de descubrimiento para localizar la instancia de un servicio.
- Buffering de mensajes. Con mecanismos síncronos de request/response, como HTTP, ambos, el cliente y el servicio, deben estar disponibles durante el intercambio de mensajes. Por contra, un broker de mensajería encola los mensajes en un canal hasta que pueden ser procesados por el consumidor. Esto significa, por ejemplo, que una tienda online puede aceptar pedidos de los clientes incluso cuando el sistema de completado de órdenes esté saturado o no disponible. Los mensajes de pedidos simplemente se encolan.
- Interacciones flexibles cliente/servicio. La mensajería soporta todos los estilos de interacción descritos anteriormente.
- Comunicación IPC explícita. Los mecanismo de RPC tienden a hacer que la llamada a una servicio remoto se parezca a llamar a un método local. Sin embargo, realmente hay muchas diferencias. La mensajería hace estas diferencias explícitas de forma que los desarrolladores no caigan en una falsa sensación de seguridad.

También hay inconvenientes al usar mensajería:

- Hay una complejidad operacional adicional. El sistema de mensajería es otro componente de sistemas que hay que instalar, configurar y operar. Es esencial que el broker de mensajes tenga alta disponibilidad, de otro forma el sistema en conjunto no sería confiable.
- Complejidad para implementar interacciones request/response. Para implementar interacciones request/response se requiere trabajo adicional. Cada petición debe contener un identificador de canal de respuesta y un identificador de correlación. El servicio escribe una respuesta que contiene el ID de correlación en el canal. El cliente usa este ID de correlación para asociar la respuesta con la petición. Normalmente es más sencillo emplear un mecanismo de correlación que directamente soporte request/response.

Comunicación síncrona, IPC Request/response

Cuando se emplea comunicación síncrona, un mecanismo de IPC basado en request/response, un cliente envía una petición a un servicio. El servicio procesa la petición y envía de vuelta una respuesta. Otros clientes pueden usar comunicación asíncrona, código orientado a eventos que quizás esté encapsulado en Futures o Rx Observables. Sin embargo, a diferencia de cuando se emplea mensajería, el cliente asume que la respuesta va a llegar de forma puntual. Hay numerosos protocolos para elegir. Los dos protocolos más populares son REST y Thrift.

REST

Hoy está de moda desarrollar APIs de estilo REST. REST es un mecanismo de IPC que casi siempre usa HTTP. Un concepto clave en REST es el recurso, que típicamente representa un objeto de negocio como un Cliente o un Producto, o una colección de objetos de negocio. REST usa verbos http para manipular recursos, que son referenciados usando una URL. Por ejemplo, un GET devuelve una representación de un recurso, que puede ser en la forma de un documento XML o un objeto JSON. Un POST crea un nuevo recurso y un PUT lo actualiza.

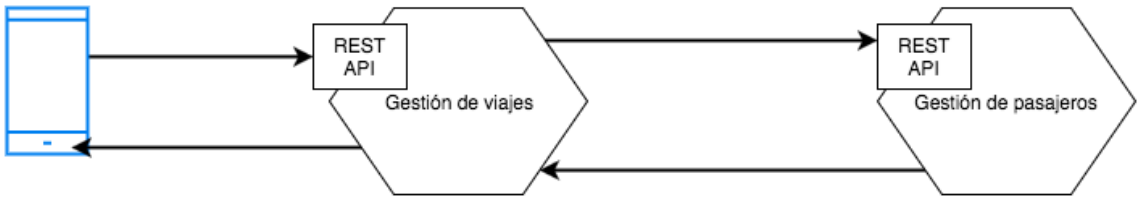


Ilustración 3, Interacción REST entre el usuario y los microservicios

El Smartphone de pasajero solicita un viaje haciendo una petición POST al recurso /trips del servicio de Gestión de Viajes. El servicio gestiona la petición enviando otra petición GET para solicitar información al servicio de gestión de pasajeros. Después de verificar que el pasajero está autorizado para crear un viaje, el servicio de gestión de viajes crea el viaje y le devuelve una respuesta 201 al Smartphone del cliente.

Muchos desarrolladores dicen que sus APIs HTTP son RESTful. Sin embargo, no todas realmente lo son. Leonard Richardson define un modelo de madurez de REST que consiste en los siguientes niveles.

- Nivel 0. Los clientes en el nivel 0 invocan al API del servicio haciendo llamadas POST al único URL endpoint. Cada petición especifica la acción a ejecutar, el objetivo de la acción (el objeto de negocio) y algunos parámetros.
- Nivel 1. En el nivel 1 se soporta la idea de recursos. Para ejecutar una acción contra un recurso, el cliente hace una petición POST que especifica la acción a desarrollar y algunos parámetros.
- Nivel 2. En el nivel 2 del API se emplean verbos http para realizar las acciones: GET para recuperar, POST para crear, PUT para actualizar. Los parámetros de las queries o del body, si los hay, especifican los parámetros de las acciones. Esto permite a los servicios aprovecharse de las ventajas de la infraestructra web como puedan ser los cacheos de las peticiones GET.
- Nivel 3. El diseño de una API de nivel 3 esta basado en el llamado principio HATEOAS (Hypertext As The Engine Of Application State). La idea fundamental es que al representación de un recurso devuelto por un GET contiene links para realizar las acciones asociadas a ese recurso. Por ejemplo, un cliente puede cancelar una Orden empleando un link contenido en la representación de la Orden devuelta en la respuesta GET. Los beneficios de HATEOAS incluyen no tener que volverse a preocupar de tener escrito en el código del cliente las URL de las acciones. Otro beneficio es que como la representación del recurso contiene links con las

acciones que se pueden realizar, el cliente no tiene que adivinar qué acciones se pueden realizar y cuáles no.

Hay numerosos beneficios al emplear un protocolo basado en HTTP:

- HTTP es simple y familiar
- Se puede testear un API HTTP desde un navegador empleando alguna extensión como Postman o desde la línea de comando empleando curl (asumiendo que se emplea JSON o algún otro formato de texto)
- Directamente soporta la comunicación tipo request/response
- HTTP es Firewall friendly
- No requiere brokers intermediarios, lo que simplifica la arquitectura

Hay algunos inconvenientes al emplear HTTP:

- Solamente soporta la interacción del tipo request/response. Se puede emplear http para notificaciones pero el servidor debe devolver siempre una respuesta HTTP.
- Como el cliente y el servicio se comunican directamente (sin ningún intermediario que haga de buffer de mensajes), ambos deben estar ejecutándose mientras dura el intercambio
- El cliente debe conocer la ubicación del servicio (su URL, por ejemplo). Esto no es un problema trivial en una aplicación moderna. El cliente debe emplear un servicio de descubrimiento para localizar el servicio.

La comunidad de desarrolladores ha descubierto recientemente el valor de los lenguajes de definición de interfaces (IDL) para las API RESTful. Hay unas pocas opciones como RAML y Swagger. Algunos IDL como Swagger permiten definir el formato de los mensajes de petición y respuesta. Otros como RAML requieren emplear una especificación separada como un esquema JSON. Los IDL además de describir APIs suelen tener herramientas para generar los stubs de cliente y servidor a partir de la definición de la interfaz.

Thrift

Apache Thrift es una interesante alternativa a REST. Es un framework para hacer llamadas RPC cliente/servidor entre lenguajes. Thrift provee de un IDL estilo C para las APIs. Se usa el compilador de Thrift para generar stubs para el cliente y esqueletos para el servidor. El compilador genera código para una

amplia variedad de lenguajes, incluyendo C++, Java, Python, PHP, Ruby, Erlang y Node.js.

Una interfaz Thrift consiste en uno o más servicios. Una definición de un servicio es análogo a una interfaz Java. Es una colección de métodos fuertemente tipados. Los métodos Thrift pueden devolver un valor (incluyendo void) o pueden ser definidos como de un solo sentido. Los métodos que devuelven un valor implementan un estilo de interacción de tipo request/response. El cliente espera una respuesta y puede que se dispare una excepción. Los métodos de un solo sentido corresponden con el estilo de interacción de notificaciones. El servidor no envía una respuesta.

Thrift soporta varios formatos de mensajes: JSON, binario y binario compacto. El binario es más eficiente que JSON porque es más rápido de decodificar. Y como el nombre sugiere, el binario compacto es el más eficiente en espacio. JSON, sin embargo es el más amigable para humanos y navegadores. Thrift también da la opción de elegir el protocolo de transporte, incluyendo TCP y HTTP. TCP es el más eficiente. Sin embargo, HTTP es el más amigable con personas, navegadores y firewalls.

Formatos de mensajes

Ahora que se ha visto http y Thrift examinemos el tema del los formatos de mensajes. Si se emplea un sistema de mensajería o REST hay que seleccionar un formato de mensajes. Otros mecanismos como Thrift soportan sólo un reducido número de formatos de mensajes, algunos quizás sólo uno. En cualquier caso, es importante emplear un formato de mensaje independiente del lenguaje. Incluso si se están escribiendo todos los microservicios en un único lenguaje, es posible que se use otro lenguaje en el futuro.

Hay dos clases principales de formatos de mensajes: texto y binario. Ejemplos de formatos de mensajes basados en texto son JSON y XML. Una ventaja de estos formatos es que no solamente son legibles por humanos, sino que además son autodescriptivos. En JSON, los atributos de un objeto están representados como una colección de nombres y valores. Esto permite a los consumidores de un mensaje coger los valores en los que está interesados e ignorar el resto. Consecuentemente, los cambios menores puede ser retrocompatibles de forma más sencilla.

La estructura de los documentos XML se especifican por un esquema XML. Con el paso del tiempo la comunidad de desarrolladores se han dado cuenta que en JSON se necesita un mecanismo similar. Una opción es emplear JSON Schema, de forma independiente o formando parte de un IDL como Swagger.

Un inconveniente de emplear un formato basado en texto es que el mensaje tiende a ser extenso, especialmente en XML. Esto es a causa de que los mensajes son autodescriptivos, cada mensaje contiene el nombre de los atributos junto a sus valores. Otro inconveniente es el sobrecoste del parseo del texto. Se puede considerar el formato binario.

Hay varios formatos binarios para elegir. Si se emplea Thrift, se puede elegir Thrift binario. Si se quiere elegir el formato del mensaje, opciones populares son Protocol Buffers y Apache Avro. Ambos de estos formatos proveen un IDL tipado para definir la estructura de los mensajes. Una diferencia, sin embargo, es que Protocol Buffers usa campos etiquetados, sin embargo el consumidor Apache Avro necesita conocer el esquema con motivo de interpretar el mensaje. Como resultado, la evolución de Protocol Buffers es más sencilla que con Avro.

	RPC	Pub/Sub	Transporte	Codificación	1-N	Proxyficable	Contrato	Complejidad	Notas
Apache Thrift	Sí	No	TCP/HTTP y otros	JSON, XML, binary	No	Sí con HTTP	Server	Media	Soporta llamadas Async. Tiene muchos lenguajes
RabbitMQ	Sí	Sí	AMQP	SSL, Texto	Sí	No	Server	Alta	Middleware muy maduro y completo pero complejo
REST	Sí	Sí, básico	HTTP	Texto	No	Sí	Server	Baja	Universal aunque no estandarizado. Complejidad de versionado
GraphQL	Sí	Subscripciones	HTTP y otros	Texto	No	Sí	Cliente	Baja	Las subscripciones todavía no están desarrolladas. Soluciona el problema del versionado
Protocol Buffers	Sí con gRPC	No	Con gRPC usa HTTP 2.0	Binary	No	Sí con gRPC	Server	Alta	Es sólo un sistema de serialización entre lenguajes.
RMI	Sí	No	TCP	Binary	No	No	Server	Media/Alta	Específico de Java

Tabla comparativa de diferentes tecnologías y formas de comunicación

Apache Thrift es una opción bastante compleja y madura. Es muy flexible en cuanto a que soporta muchos lenguajes, codificaciones y protocolos de transporte. Thrift soporta llamadas asíncronas que no devuelven un dato. Thrift es un protocolo de tipo Request/Response, no soporta Pub/Sub. Pub/Sub es nativo para los sistemas de mensajería. El problema de Thrift es que se depende de la librería y el número de lenguajes no es tan amplio como con REST o GraphQL. Thrift no es esta pensado para ser consumido directamente en el navegador del cliente final. REST y GraphQL pueden ser consumidos en un navegador y en la comunicación entre servicios.

GraphQL es una especificación independiente del transporte aunque su principal implementación está sobre HTTP. GraphQL soluciona el problema del versionado permitiendo que el contrato de las API sea especificado desde los clientes. En GraphQL se contempla implementar las subscripciones como sistema para que un cliente pueda ser notificado desde el servidor. Actualmente las subscripciones no están desarrolladas en la especificación.

RabbitMQ (como ejemplo de sistema de mensajería) es un middleware maduro y complejo y más difícil de integrar en un patrón API Gateway.

Protocol Buffer es una librería de serialización de mensajes desarrollada por Google. Soporta pocos lenguajes y no está muy extendida. gRPC (http://www.grpc.io/) implementa un sistema de llamadas remotas Request/Response sobre HTTP 2.0 que emplea Protocol Buffer como mecanismo de serialización.

8. Soluciones

Problema de ejecutar una llamada de forma asíncrona

Supongamos un tarea de conversión de un vídeo que tarda más de 10 minutos. El cliente pasa la URL del vídeo y el servidor devuelve la URL del vídeo convertido. En una llamada REST normal el cliente lanza la petición y se queda a la espera de que termine, es decir, al menos 10 minutos.

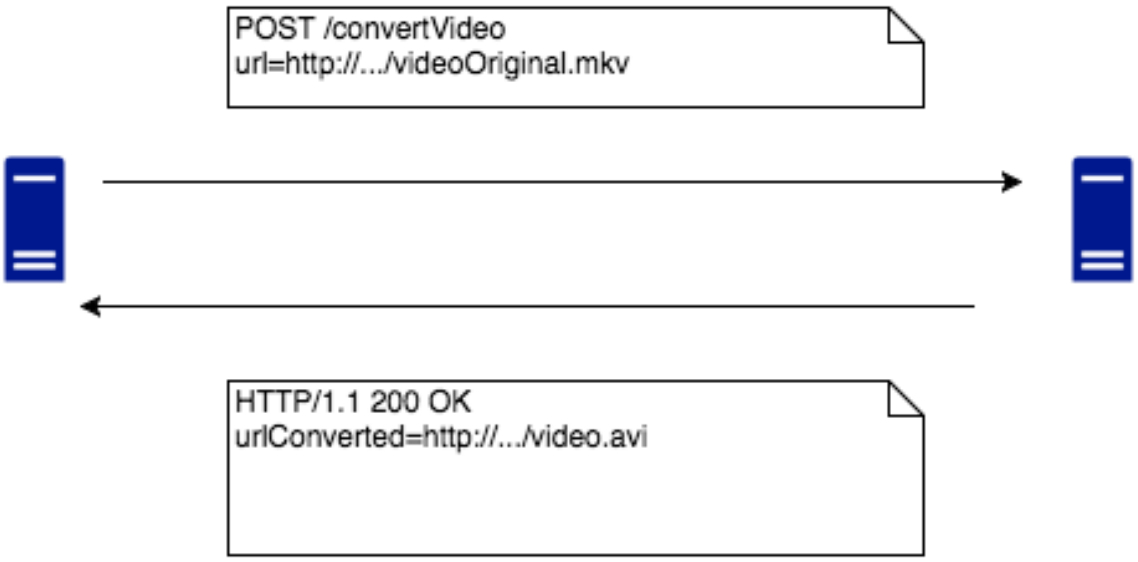


Ilustración 4, Flujo de Request Response

```php
<?php

// web/index.php
require_once __DIR__.'/../vendor/autoload.php';

$app = new Silex\Application();

$app->get('/convertVideo/{url}', function ($url) use ($app) {
    sleep(10);
    return '{urlConverted: "http://..../algo.avi"}';
});

$app['debug'] = true;
$app->run();
```

Ilustración 5, Servidor PHP bloqueante

```
MacBook-Pro:~ fernando$ time curl http://127.0.0.1:8000/convertVideo/laurl
{urlConverted: "http://..../algo.avi"}
real    0m10.044s
user    0m0.005s
sys     0m0.004s
MacBook-Pro:~ fernando$
```

Ilustración 6, Respuesta bloqueada

El protocolo HTTP se queda esperando hasta que termine. Una solución para gestionar llamadas asíncronas pasa por que el servidor devuelva una respuesta una vez que reciba los datos necesarios para procesar la respuesta.

Ilustración 7, Flujo RequestResponse con respuesta inmediata

```
var express = require('express');
var app = express();
var sleep = require('sleep');
var winston = require('winston');
winston.remove(winston.transports.Console);
winston.add(winston.transports.Console, {
  timestamp:true,
  prettyPrint: true,
  colorize: true});

app.get('/convertVideo', function (req, res) {
  winston.info('converting, blocking');
  sleep.sleep(10);
  res.send('{urlConverted: "http://...."}');
  winston.warn('converted');
});

app.get('/convertVideo2', function (req, res) {
  winston.info('converting, non blocking');
  res.send('{msg: "ok"}');
  winston.info('respuesta enviada');
  sleep.sleep(10);
  winston.warn('converted');
});

app.listen(8000, function () {
  winston.debug('Example app listening on port 8000!');
});
```

Ilustración 8, Request Response no bloqueante

```
MacBook-Pro:~ fernando$ time curl http://127.0.0.1:8000/convertVideo
{urlConverted: "http://...."}
real    0m10.038s
user    0m0.005s
sys     0m0.004s
MacBook-Pro:~ fernando$ time curl http://127.0.0.1:8000/convertVideo2
{msg: "ok"}
real    0m0.019s
user    0m0.005s
sys     0m0.004s
```

Profile: Default
Command: /Applications/iTerm.app/Contents/MacOS

Ilustración 9, cliente con llamada a servicio bloqueante vs no bloqueante

```
MacBook-Pro:problema1b fernando$ node index.js
2016-06-06T08:07:56.206Z - info: converting, blocking
2016-06-06T08:08:06.221Z - warn: converted



2016-06-06T08:08:11.240Z - info: converting, non blocking
2016-06-06T08:08:11.242Z - info: respuesta enviada
2016-06-06T08:08:21.247Z - warn: converted
```

Ilustración 10, Tiempos en el servidor de llamada bloqueantes vs no bloqueante

Para devolver inmediatamente una respuesta en HTTP el servidor debe procesar el trabajo en otro hilo. No todos los lenguajes soportan la gestión de hilos, por ejemplo PHP sólo soporta la gestión de hilos en la línea de comandos y no cuando se ejecuta en un servidor HTTP. Para garantizar que el cliente no se queda bloqueado a la espera de la respuesta, puede implementar un mecanismo de timeout. Estableciendo un timeout mínimo se asegura que los datos llegan al servidor y el cliente no se queda a la espera. Con el timeout es muy posible que el cliente no obtenga una respuesta desde el servidor como por ejemplo un número de identificación.

Ilustración 11, El cliente hace la petición pero no espera la respuesta

```php
<?php
// web/index.php
require_once __DIR__.'/../vendor/autoload.php';

$app = new Silex\Application();

$app->register(new Silex\Provider\MonologServiceProvider(), array(
    'monolog.logfile' => __DIR__.'/development.log',
));

$app->get('/convertVideo/{url}', function ($url) use ($app) {
    $app['monolog']->addDebug('Convirtiendo');
    sleep(10);
    $app['monolog']->addDebug('Convertido');
    return '{urlConverted: "http://..../algo.avi"}';
});

$app['debug'] = true;
$app->run();
```

Ilustración 12, Servidor PHP con log en Request Response

```
MacBook-Pro:~ fernando$ time curl http://127.0.0.1:8000/convertVideo/ydfhg --connect-timeout 5 -m 1
curl: (28) Operation timed out after 1003 milliseconds with 0 bytes received

real    0m1.014s
user    0m0.004s
sys     0m0.003s
```

Ilustración 13, Cliente CURL con timeout

```
[2016-06-06 10:43:35] app.INFO: Matched route "{route}". {"route":"GET_convertVideo_url","route_parameters":{"_controller":"[object] (Closure: {})","url":"ydfhg","_route":"GET_conve
rtVideo_url"},"request_uri":"http://127.0.0.1:8000/convertVideo/ydfhg","method":"GET"} □
[2016-06-06 10:43:35] app.DEBUG: > GET /convertVideo/ydfhg □ □
[2016-06-06 10:43:35] app.DEBUG: Convirtiendo □ □
[2016-06-06 10:43:45] app.DEBUG: Convertido □ □
[2016-06-06 10:43:45] app.DEBUG: < 200 □ □
```

Ilustración 14, Log del procesamiento en el servidor de un cliente con timeout

Estableciendo un timeout en el cliente la petición se sigue procesando como se puede ver en los logs, a pesar de que el cliente ya ha desconectado hace tiempo. Establecer un timeout en el cliente tiene varios inconvenientes:

- El cliente no recibe ningún tipo de respuesta, incluida la confirmación de que se ha recibido correctamente
- Hay que gestionar dos tipos de timeout, el de conexión que nos puede interesar un tiempo alto por si el servicio está saturado y el timeout de procesamiento que nos interesa que sea lo más corto posible para poder desconectar en cuanto se envíen los datos
- El timeout de procesamiento puede ser elevado, por ejemplo en curl el mínimo es un segundo. Todas las peticiones van a tardar al menos un segundo y no menos y eso puede ser una limitación.

Respuestas desde el servidor

Cuando se desconecta antes de terminar el procesamiento de la petición hay 2 tipos de respuestas http. Las que se generan en cuanto se envían los datos y la respuesta con los datos procesados

Respuesta intermedias, respuestas a la recepción de la request

El cliente no recibe ningún tipo de respuesta, por ejemplo si se ha ejecutado un timeout. Ya hemos visto que los timeouts son una solución pero que tiene limitaciones claras.

El cliente recibe una confirmación de que la petición ha sido recibida y es correcta o procesable.

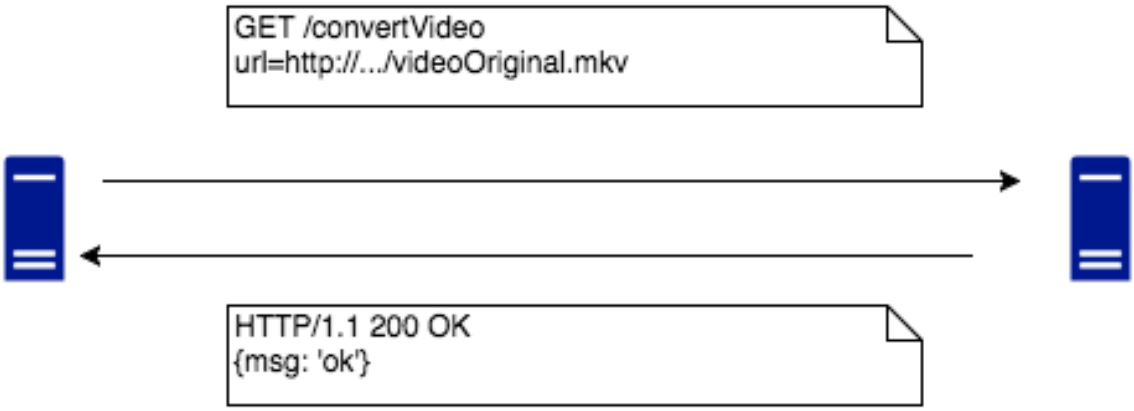


Ilustración 15, Conversacion de recepción (ACK)

El cliente recibe un número de correlación de su petición por parte del servidor que le permite identificar su respuesta.

Ilustración 16, El servidor envía una confirmación y un número de correlación

Como alternativa a que el servidor genere un identificador de respuesta se puede plantear que sea el cliente el que aporte el identificador, esto tiene la ventaja de descargar al servidor de esa tarea y hacer que sea sin estado.

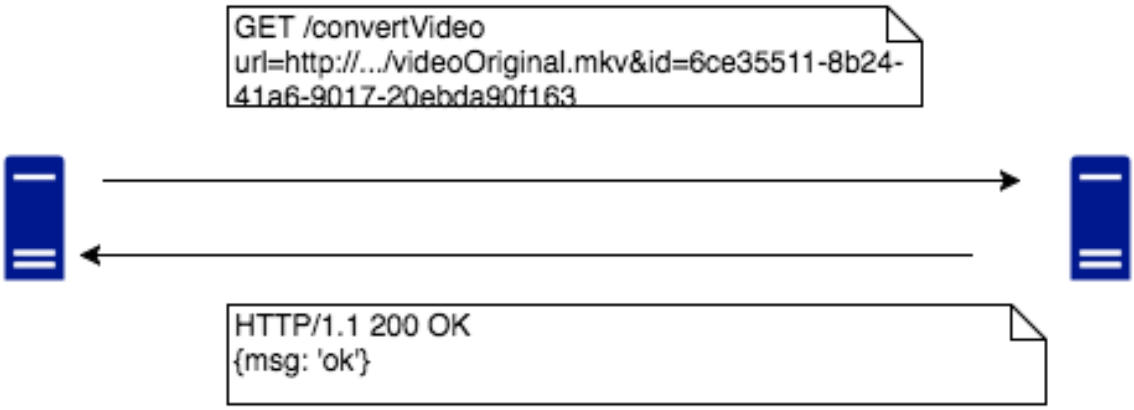
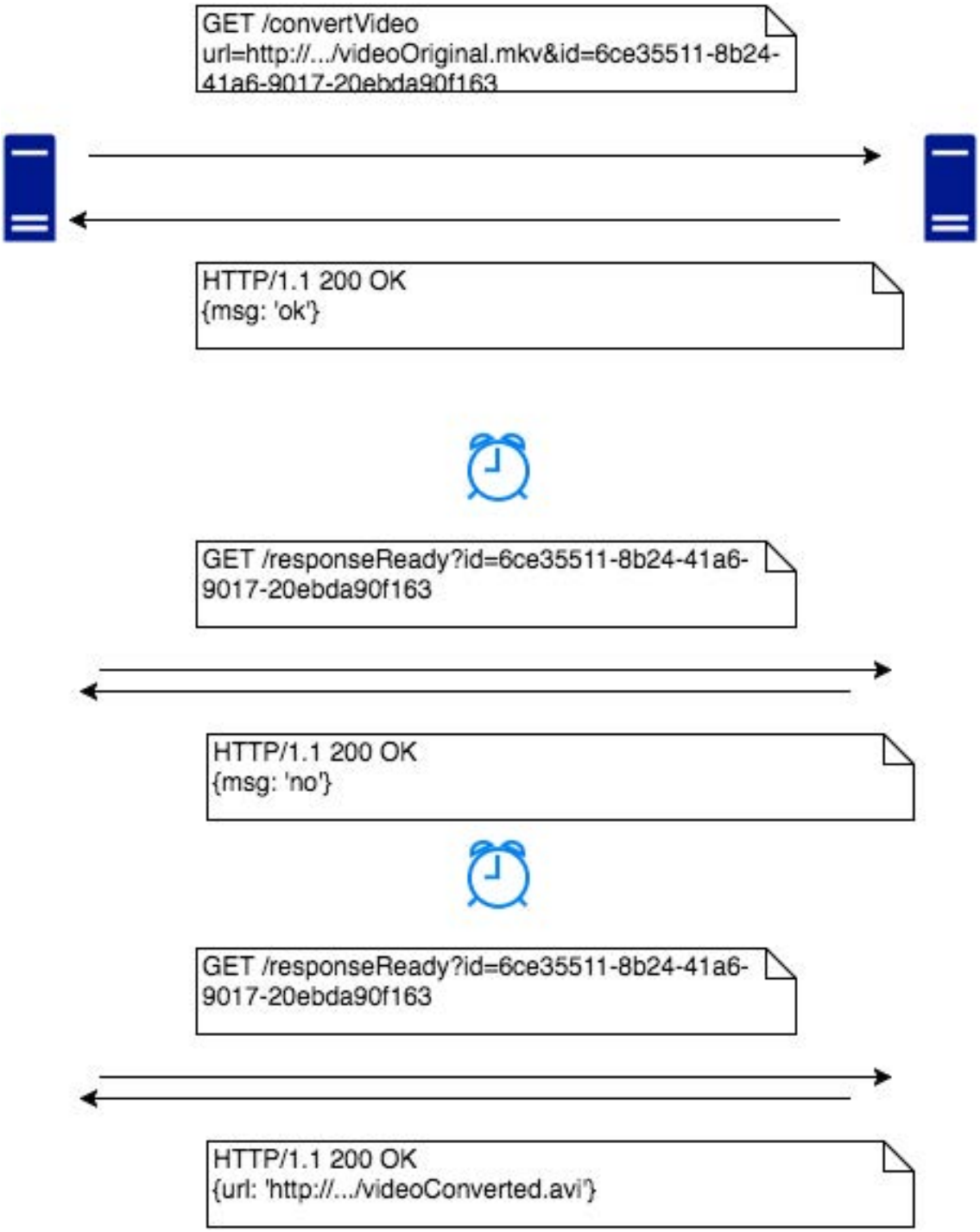


Ilustración 17, El cliente ofrece un número de correlación al servidor

El inconveniente es que estos identificadores deben ser únicos en todo el conjunto de clientes y deben poderse generar de manera descoordinada. Un ejemplo de esta clase de número son los UUIDs. Precisamente los UUId se crearon para que los sistemas distribuidos puedan identificar los elementos de forma universal sin depender de un sistema de información central. Estos UUIDs deben generarse de forma segura de modo que no se puedan suplantar.

Respuestas a finales a la petición

El cliente pregunta cada X tiempo al servidor con su número de correlación de petición.



Este modelo es muy sencillo de implementar, pero también tiene varios inconvenientes:

- El cliente no recibe la petición justo cuando se termina de procesar, sino cuando ha terminado de procesarse y pregunta por ella. El intervalo de preguntar por la respuesta puede ser lento

- Es más susceptible de tener problemas de seguridad puesto que cualquiera podría preguntar por un número de correlación que no es suyo
- Consume recursos en el cliente y en el servidor
- La respuesta debe estar almacenada en el servidor y el servidor debe comprobar el almacén para ver si el proceso ha terminado

La comunicación se hace en un canal paralelo (side channel) que permite una conexión permanente

HTTP permite varios sistemas que soportan la comunicación desde el servidor a los clientes mediante conexiones permanentes de los clientes. Los principales mecanismos son SSE (Server Sent Events) y Websockets.

El mecanismo más sencillo son los SSE. El cliente realiza la petición de forma normal, con una request/response de HTTP. El servidor ofrece un número de correlación (también sirve si el número lo genera el cliente). A su vez el cliente está conectado al canal de SSE escuchando todos los eventos que ofrece el servidor. Cuando ve un evento identificado con su número de correlación procesa la respuesta.

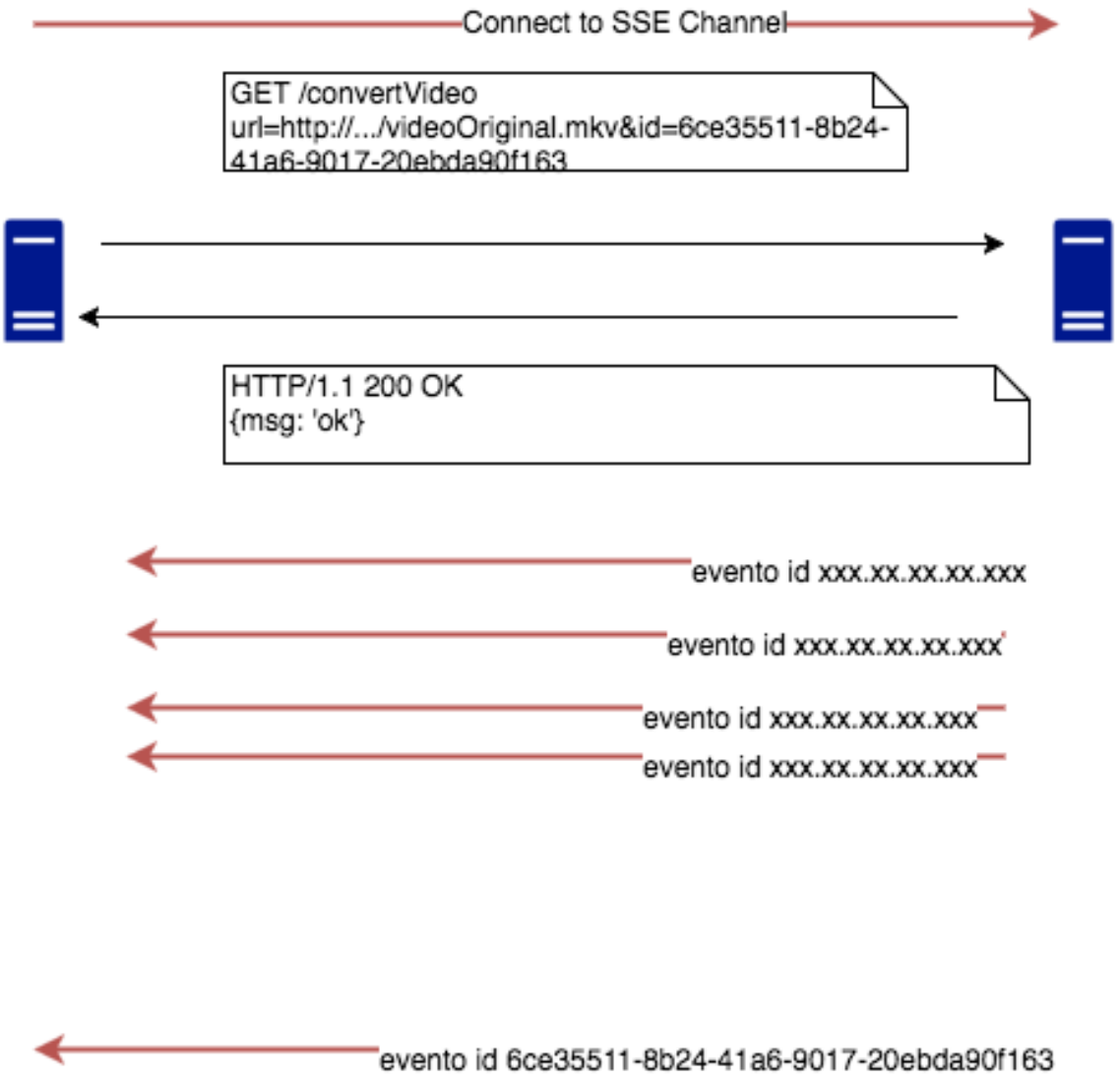


Ilustración 18, Respuesta sobre un side-channel conectado

Este mecanismo requiere 2 terminaciones (endpoints) en el servidor, uno que recibe la petición y otro que es el que van a utilizar los clientes para recibir la respuesta.

Como principales inconvenientes tiene:

- El cliente tiene que estar conectado de forma permanente y esto consume recursos que hace que sea difícil de escalar
- El cliente tiene que gestionar las reconexiones si por lo que sea se desconecta del canal de notificación.
- El cliente recibe todos los mensajes, incluidos los que no van destinados a él.
- Hay que gestionar algún mecanismo de comunicación IPC pero sólo dentro del servidor para que la parte que recibe la petición pueda

notificar a la parte que gestiona el canal SSE que la respuesta está lista. La comunicación IPC dentro del servidor puede ser tan sencilla como una variable global.

```
var express = require('express')
  , app = express()
  , sse = require('./sse')
var uuid = require('node-uuid');
var winston = require('winston');
var sleep = require('sleep');

winston.remove(winston.transports.Console);
winston.add(winston.transports.Console, {
  timestamp:true,
  prettyPrint: true,
  colorize: true});

var connections = [];

app.use(sse)

app.use(function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*");
  res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept");
  next();
});


app.get('/convertVideo', function(req, res) {
  winston.info('converting, non blocking');
  var myUuid = uuid.v4();
  res.send(JSON.stringify({msg: 'ok', id: myUuid}));
  winston.info('respuesta enviada');
  sleep.sleep(10);
  winston.warn('converted');
  for(var i = 0; i < connections.length; i++) {
    winston.warn('inviando datos por SSE a cada cliente');
    connections[i].sseSend({id: myUuid, url: 'http://.../videoconverted.avi'})
  }
  winston.warn('fin');


})

app.get('/canalSSE', function(req, res) {
  res.sseSetup()
  connections.push(res)
})

app.listen(8000, function() {
  winston.debug('Example app listening on port 8000!');
})
```

Ilustración 19, Ejemplo de servidor con notificación SSE

```
MacBook-Pro:~ fernando$ time curl http://127.0.0.1:8000/convertVideo
{"msg":"ok","id":"dd299e8b-0d78-472b-a8c1-9d48f91a26d8"}
real    0m0.015s
user    0m0.005s
sys     0m0.004s
```

Ilustración 20, Llamada al endpoint HTTP para solicitar la conversión de un video

```
> var evtSource = new EventSource("http://127.0.0.1:8000/canalSSE");
< undefined
> evtSource.onmessage = function(e) {
    console.log(e);
  }
< function (e) {
    console.log(e);
  }
  ▶ MessageEvent {isTrusted: true, data: "{"id":"ed35755d-b096-4b13-a16f-112fff84caad","url":"http://.../videoconverted.avi"}", origin: "http://127.0.0.1:8000", lastEventId: "", source: null…}
  ▶ MessageEvent {isTrusted: true, data: "{"id":"dd299e8b-0d78-472b-a8c1-9d48f91a26d8","url":"http://.../videoconverted.avi"}", origin: "http://127.0.0.1:8000", lastEventId: "", source: null…}
```

Ilustración 21, Cliente conectado al endpoint que recibe los eventos de SSE

Otro problema importante a la hora de escalar esta solución es que ya que las conexiones deben mantenerse abiertas hay que mantener un estado global en la parte servidor que en el caso del ejemplo se gestiona con la variable “connections”. Existen ejemplos para distribuir estas conexiones en varios

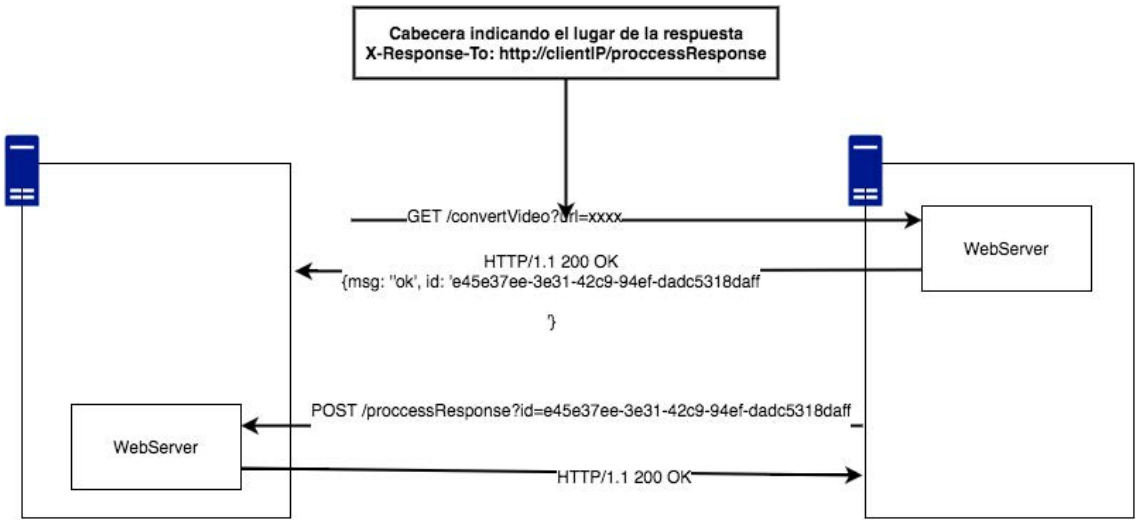
procesos, por ejemplo con socket.io o con SocketCluster, pero ambos acaban dependiendo de Redis que es el que mantiene el estado.

Esta solución se ha desarrollado con SSE pero con WebSockets es totalmente análogo con la única salvedad que también se puede escribir en el canal de websocket con lo que se podrían resumir los endpoints a uno sólo.

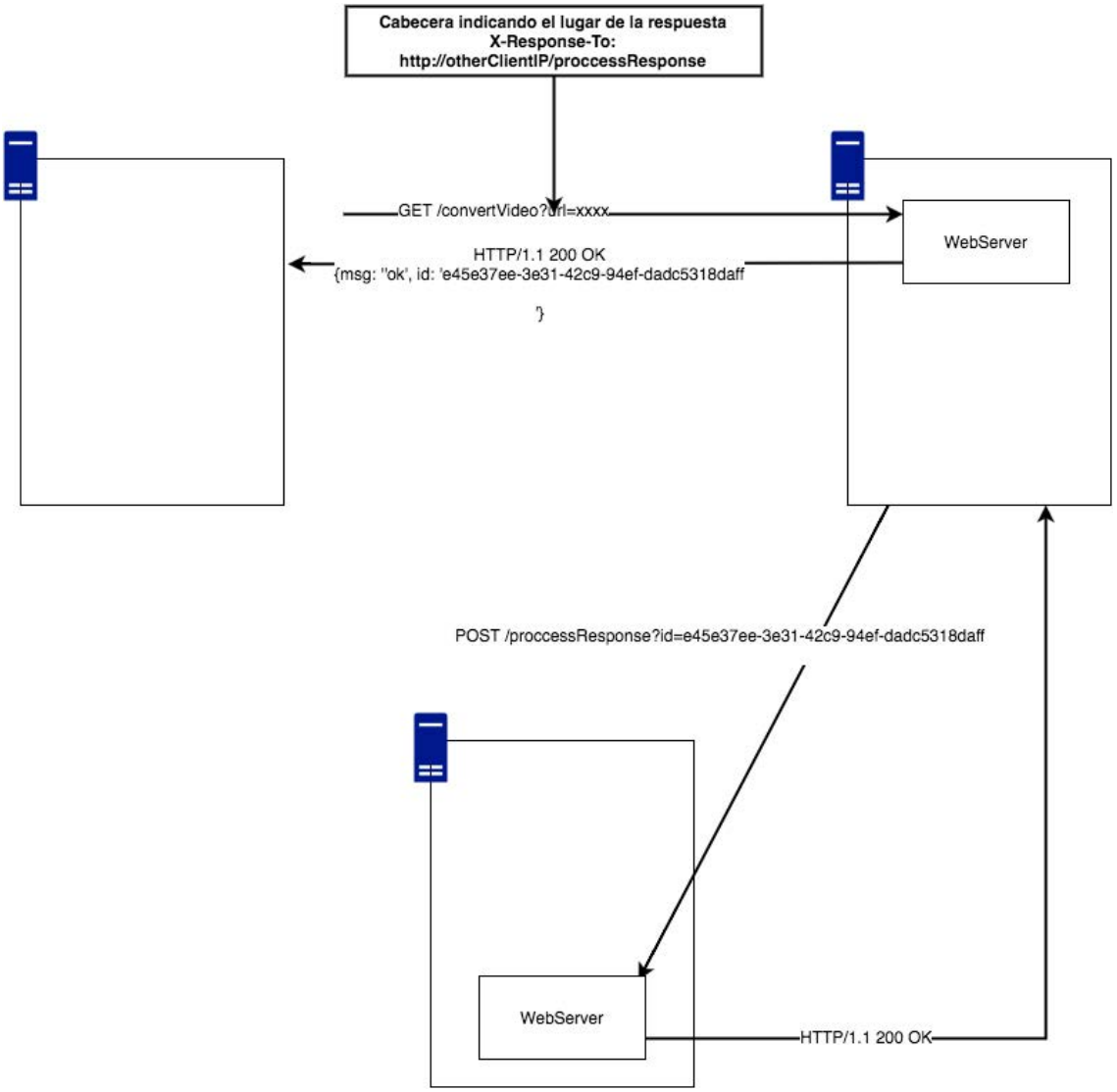
El servidor escribe la respuesta donde el cliente le indique

En algunos casos los microservicios pueden ser consumidos por un navegador que actúa como cliente. Normalmente si no hay una conexión previamente abierta y el cliente es un navegador el servidor no tiene forma de localizar el cliente para entregarle de vuelta los datos. El cliente no tiene por qué ser un navegador y lo normal es que no lo sea. Puede ser interesante de alguna forma que junto con la petición el cliente le indique al servidor dónde quiere recibir la respuesta, de forma que puede generar nuevas opciones de comunicación que incluso incluyan a más de dos participantes.

- Escribe la respuesta de vuelta en el cliente. Tiene un servidor web que escucha. Implica varios procesos en el cliente con ambos roles: servidor y cliente



- Escribe la petición en un tercer participante



Problema: evolucionar el API

Otro problema importante que hay que tener en cuenta con la comunicación entre servicios es que para la comunicación entre ambos se establece un contrato que define cómo debe recibir los datos y cómo debe entregarse la respuesta. Este contrato son las interfaces de la OOP. Los contratos no permanecen inalterados en el tiempo y lo normal es que no todas las partes se puedan adaptar a los cambios en los contratos inmediatamente y tengan que convivir diferentes versiones de los contratos en el tiempo. Estos cambios acaba siendo una pesadilla sobre todo para los desarrolladores de frontend.

Hoy existen dos estilos predominantes para estas interacciones HTTP: REST y los endpoints a medida. En ambos casos el contrato pertenece al servicio que es que decide cómo cambiarlo y los clientes se tienen que adaptar o fallar. Hay bastante literatura sobre sistemas de versionado de URLs para sobrevivir a estos cambios

REST

Hay varias debilidades en un sistema REST típico, algunas que son especialmente problemáticas para aplicaciones móviles:

- Para obtener grafos de objetos complicados se requieren múltiples interacciones entre el cliente y el servicio. Para aplicaciones móviles que operan en condiciones de red variables, estos ciclos de interacciones son altamente indeseables.
- Invariablemente, campos e información adicional se van añadiendo a los endpoint REST según van cambiando los requerimientos. Los clientes viejos también reciben estos datos a causa de que la especificación de cómo se recuperan los datos se encuentra en el servidor y no en los clientes. Con el tiempo estos datos transmitidos (payload) van creciendo para todos los clientes. Cuando esto se convierte en un problema, una solución suele ser añadir versionado a los endpoint REST. El versionado de los endpoint REST complica el servidor que termina con código duplicado y código espagueti. Otra posible solución es generar múltiples vistas del mismo endpoint REST, esto al final no suele ofrecer la flexibilidad necesaria.

- Los endpoint REST suelen ser débilmente tipados y suelen carecer de metadatos legibles por máquinas. Los lenguajes no tipados tienen su utilidad, pero los fuertemente tipados garantizan su corrección y tienen muchas más oportunidades para el desarrollo de herramientas. Los desarrolladores suplen la falta de metadatos consultando documentación que con frecuencia es incompleta o está desfasada.

La mayoría de las APIs REST orientadas a terceros no suelen estar en un estado ideal, y tampoco lo están las APIs REST de uso interno de la empresa. Las consecuencias de la opacidad del API y las excesivas consultas suele ser más severa en las APIs internas porque evolucionan más rápido.

Las APIs REST inevitablemente acaban construyendo endpoints a medida. Esto acaba acoplando los datos a una vista en particular, lo que viola uno de los principales objetivos de REST.

Endpoints ad hoc

Muchas aplicaciones no formalizan un contrato entre el cliente y el servicio. Los desarrolladores de producto acceden al servidor a través de endpoints a medida y escriben código personalizado para obtener los datos que necesitan en ese momento. Los servidores definen procedimientos y devuelven datos. Esta aproximación tiene la virtud de la simplicidad, pero tiene el inconveniente de que los sistemas acaban siendo inmantenibles según avanza el tiempo.

- Estos sistemas típicamente definen un endpoint personalizado para cada vista. Para sistemas con una gran superficie, rápidamente crece hacia una pesadilla y endpoints abandonados, herramientas inconsistentes y código masivamente duplicado en el servidor.
- Como en REST, los payloads de los endpoints a medida suelen crecer de forma constante. Los clientes desplegados no se pueden romper y se acaba teniendo un número grande de versiones. En estas circunstancias es difícil eliminar datos de los endpoints.
- Los endpoints a medida tienden a crear procesos torpes para los desarrolladores de frontend. Primero se suelen forzar a crear un endpoint a medida en el servidor y luego entonces se adapta el cliente.
- Como en los sistemas REST, los sistemas con endpoints personalizados no tiene un sistema de tipos personalizado lo que

elimina las posibilidades de instrospección y de desarrollo de herramientas. Algunos sistemas de endpoints a medida son fuertemente tipados como Protocol Buffers, Thrift o XML. Estos sistemas no son tan expresivos y flexibles como GraphQL y tienen los otros inconvenientes de los endpoints a medida.

GraphQL como alternativa a REST

GraphQL es un lenguaje de consulta de datos y un runtime diseñado y usado por Facebook para solicitar y entregar datos a aplicaciones web y móviles desde el año 2012. Facebook liberó este lenguaje en 2015.

Una consulta GraphQL es una cadena interpretada por un servidor que devuelve datos en un formato especificado. Un ejemplo de consulta.

```
{
  user(id: 3500401) {
    id,
    name,
    isViewerFriend,
    profilePicture(size: 50)  {
      uri,
      width,
      height
    }
  }
}
```

Una respuesta de ejemplo.

```
{
  "user" : {
    "id": 3500401,
    "name": "Jing Chen",
    "isViewerFriend": true,
    "profilePicture": {
      "uri": "http://someurl.cdn/pic.jpg",
      "width": 50,
      "height": 50
    }
  }
}
```

Incluso este ejemplo sencillo muestra varios principios de diseño:

- Jerárquico: la mayoría de desarrollos de producto implican trabajar con vistas jerárquicas. La propia consulta GraphQL es un conjunto jerárquico de campos
- Centrado en el producto: GraphQL se centra en los requerimientos de la vista y de los ingenieros de frontend.

- Consultas específicas para el cliente: En GraphQL las especificación de las consultas está codificada en el cliente en vez de en el servidor. Las consultas tienen una granularidad de campo. En la gran mayoría de aplicaciones escritas sin GraphQL, es el servidor el que determina los datos devueltos en sus endpoints. Una consulta GraphQL, por el contrario, devuelve exactamente lo que el cliente solicita y no más.
- Retrocompatible: En un mundo con aplicaciones nativas desplegadas sin obligación de actualizar, la retrocompatibilidad es un reto. Facebook, por ejemplo, hace releases en ciclos de 2 semanas y mantiene esas versiones por dos años. Esto significa que hay un mínimo de 52 versiones de clientes interrogando a los servidores en un momento dado. El hecho de que el cliente especifique las consultas hace que se simplifique mucho garantizar la retocompatibilidad.
- Código estructurado: normalmente los lenguajes con la granularidad de campo imponen un motor de almacenamiento. GraphQL por el contrario impone una estructura en el servidor y expone unos campos que son sustentados por cualquier código. Así se gana en flexibilidad en la parte del servidor.
- Protocolo en la capa de aplicación: No requiere un transporte en particular. Se pasa una cadena y es interpretada por el servidor.
- Fuertemente tipado: Al ser un lenguaje fuertemente tipado se puede verificar que la consulta es sintácticamente correcta y válida antes de su ejecución, por ejemplo en tiempo de desarrollo, y el servidor puede tener garantías de la forma y naturaleza de las respuestas. También hace más sencillo escribir buenas herramientas de cliente.
- Instrospectivo: Las herramientas y los clientes pueden consultar el sistema de tipos usando el propio lenguaje GraphQL.

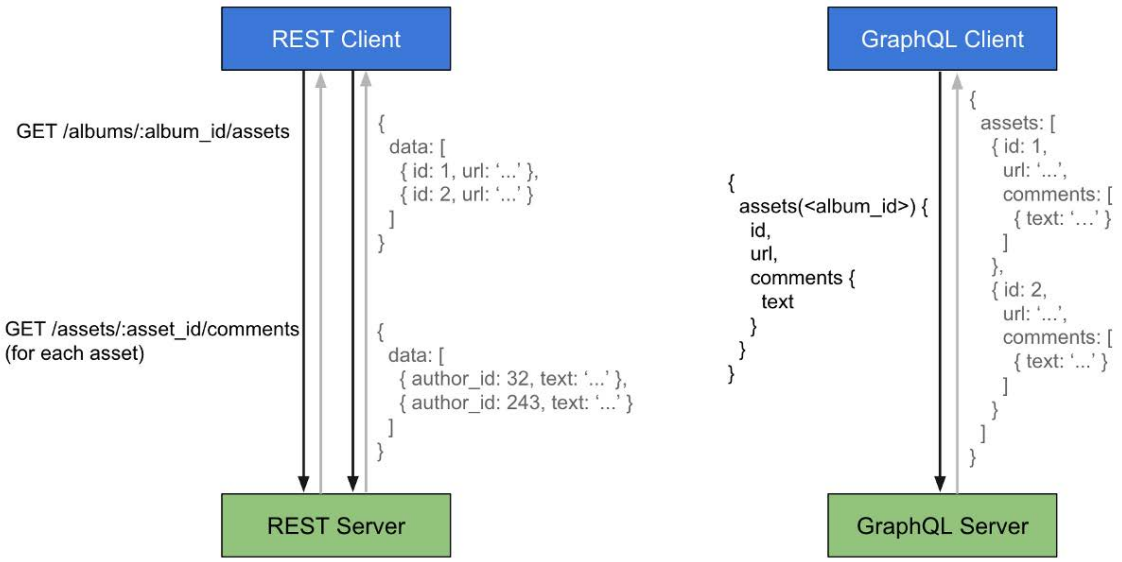


Ilustración 22, ejemplo de llamada GraphQL vs REST

La potencia de GraphQL está en que en vez de definir la estructura de las respuesta en la propia respuesta, se deja esta flexibilidad a los clientes.

¿Esta muerto REST?

En realidad no, GraphQL es muy nuevo y no tienen una gran adopción. Cada día hay más sistemas en producción que están apostando por GraphQL y están demostrando que es muy útil, pero la industria ha invertido mucho en las APIs REST y no van a desaparecer de un día para otro. Además, como hay mucha gente que no tiene un modelo de madurez elevado de REST, y lo implementa de cualquier forma, tienen una flexibilidad total (con los problemas que eso conlleva). GraphQL puede convivir con REST. Algunas APIs usarán REST, otras GraphQL y otras ambos.

GraphQL puede implementarse como un envoltorio a REST, transformando peticiones GraphQL en REST.

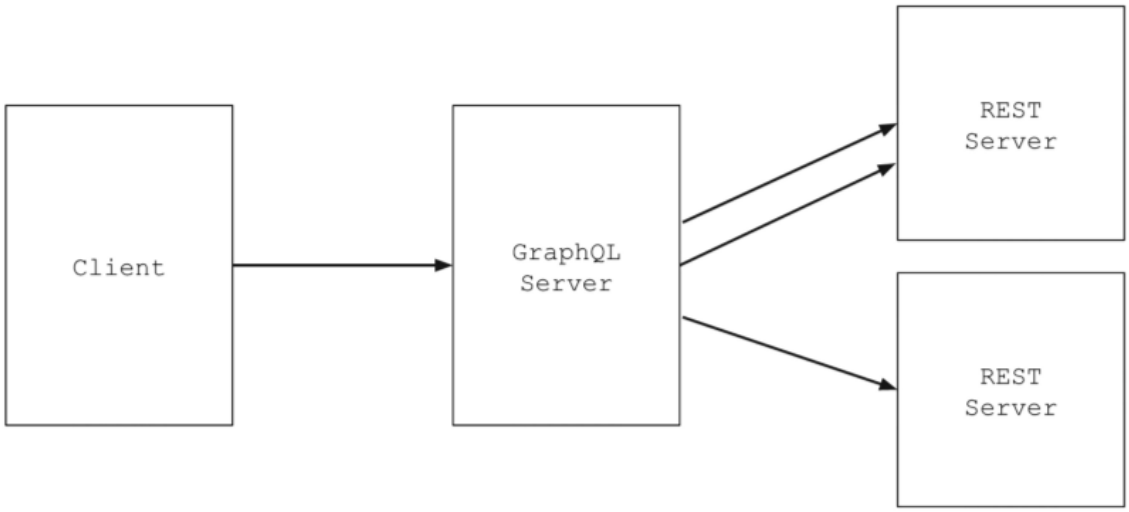


Ilustración 23, GraphQL como wrapper REST

Problema: gestionar subscripciones con GraphQL

En una aplicación REST normal, las suscripciones se pueden entender como una generalización del problema de procesar una petición de forma asíncrona, solo que en vez de generarse una respuesta a la petición se generan varias respuestas y se seguirán recibiendo hasta que el cliente decida dejar de hacerlo. Esto tiene dos implicaciones fundamentales:

- Que el servidor tiene que mantener el estado puesto, ya que, tiene que saber qué cliente tiene subscritos
- El servidor tiene que tener un mecanismo de subscripción y desubscripción

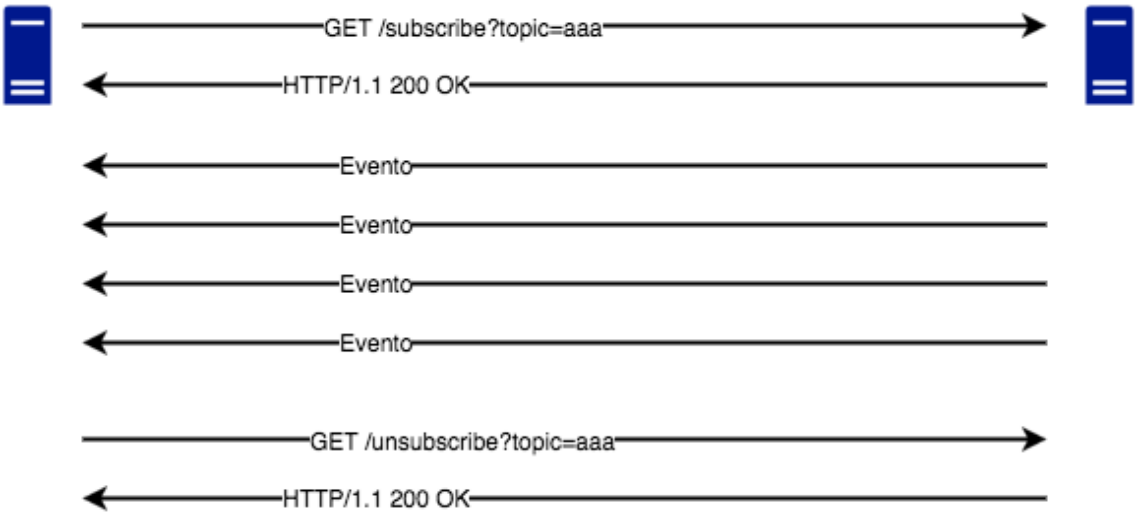


Ilustración 24, Flujo de subscripción, desubscripción

Facebook usa las subscripciones en GraphQL y tienen un post de su blog de graphql.org explicando a grandes rasgos cómo funcionan pero no están puestas en la especificación ni hay una implementación de referencia, aunque dicen que lo van a hacer.

Con GraphQL el sistema de subscripciones se complica más porque, como el valor que toman los datos de vuelta lo define el cliente, hace que el estado que tiene que manejar el servidor sea bastante más complicado, puesto que en cierto modo hay que personalizarlo para cada cliente.

Imaginemos que nuestra API tiene una subscripción para enterarnos cuándo un usuario pulsa en el Like de una historia, hasta aquí lo que sería un topic de un patrón pub/sub normal. En GraphQL esto puede ser asi:

```
subscription StoryLikeSubscription($input: StoryLikeSubscribeInput) {
  storyLikeSubscribe(input: $input) {
    story {
      likers { count }
      likeSentence { text }
    }
  }
}
```

Ilustración 25, Subscripción en GraphQL

Para empezar ya hay una diferencia importante, y es que el topic acepta parámetros, luego es como si tuviésemos tantos topic como parámetros para cada subscripción. Además un cliente que hubiese pasado los mismos parámetros que otro no tiene por qué recibir la misma respuesta, por ejemplo:

```
subscription StoryLikeSubscription {
  storyLikeSubscribe(id:10) {
    story {
      likers { count }
      likeSentence { text }
    }
  }
}
```

Ilustración 26, Ejemplo de subscripción GraphQL

```
subscription StoryLikeSubscription {
  storyLikeSubscribe(id:10) {
    story {
      likers { count }
    }
  }
}
```

Ilustración 27, Ejemplo de subscripción con mismo parámetro y diferente valor de retorno

Los dos retos a la hora de manejar las subscripciones en GraphQL son el número de topics (por los parámetros) y como poder tener que devolver a cada cliente una respuesta diferente, nos hace plantearnos si cada vez que se dispara un evento hay que volver a recalcular todas las subscripciones para todos los clientes.

Implementación de subscripciones en GraphQL

Supongamos que queremos subscribirnos a los eventos que se generan cada vez que se convierte un vídeo por parte de cualquier usuario de nuestra aplicación (por mantener el ejemplo de la conversión de vídeos del problema anterior)

Nuestra request de GraphQL para la subscripción tendría esta forma:

```
subscription {
  newVideoConverted {
    video {
      id
      name
      url
    }
  }
}
```

Ilustración 28, Request GraphQL para subscribirnos a nuevos videos

La respuesta esperada sería de esta forma

Ilustración 29, Respuesta esperada en GraphQL

Cuando uno se subscribe a un evento no tiene porqué haber datos disponibles en ese momento con lo cual la primera respuesta puede ser “null” u otro valor asignado, como pudiera ser una confirmación de que la subscripción se ha realizado correctamente.

Ilustración 30, Respuesta inmediata tras una subscripción

Como cada vez que se dispara un evento hay que volver a ejecutar la consulta, debemos guardar la query del usuario en algún sitio, esto parece sencillo pero tiene su trampa. La información relativa a una query la conoce el cliente y la librería de GraphQL una vez que la parsea, pero no es conocida en puntos intermedios del proceso previo al parseo puesto que en eso punto se ve como cadena de texto. Además no vale solamente con guardar la query tal cual, hay que guardar las variables, el nombre de la operación y el valores raíz en el servidor. En el servidor cuando una query se resuelve tiene una valores raíz o de contexto asociados que deben poder ser almacenados con cada query de los clientes y poder ser regenerados de forma consistente en cada evento. Estos valores de contexto pueden ser referencias a la base datos, caches, etc.

```
const Query = new GraphQLObjectType({
  name: 'Query',
  fields: () => ({
    viewer: {
      type: User,
      resolve(parent, args, {db}) {
        return db.get(`
          SELECT * FROM User WHERE name = 'freiksenet'
        `);
      }
    }
  })
});
```

Ilustración 31, Resolución de una query GraphQL en servidor con referencias la BD

Otro problema que surge es que GraphQL permite gestionar varios campos en una única petición de subscripción. Por ejemplo:

```
subscription {
  newVideoConverted {
    video {
      id
      name
      url
    }
  }
  videoDeleted {
    videoId
  }
}
```

Ilustración 32, Subscripcion con múltiples campos

¿Cómo hay que tratar una subscripción con múltiples campos? Podemos plantear que es una única subscripción y nos daríamos de alta y baja en conjunto. O podemos tratarlos por separado, si los tratamos en conjunto se puede complicar la gestión puesto que habría que almacenar la función de resolución para ambos.

Veamos un ejemplo de Schema GraphQL que soporta subscripciones. El ejemplo simplemente lleva la cuenta de un contador y notifica sus cambios a los clientes subscritos.

```
const schema = new GraphQLSchema({
    query: new GraphQLObjectType({
        name: 'RootQueryType',
        fields: {
            count: {
                type: GraphQLInt,
                description: 'The count!',
                resolve: (parent) => {
                    return count;
                }
            }
        }
    }),
    mutation: new GraphQLObjectType({
        name: 'RootMutationType',
        fields: {
            updateCount: {
                type: GraphQLInt,
                description: 'Updates the count',
                resolve: (parent, args) => {
                    count += 1;
                    events.emit('updatedCount');
                    return count;
                }
            }
        }
    }),
    subscription: new GraphQLObjectType({
        name: 'RootSubscriptionType',
        fields: {
            updatedCount: {
                type: GraphQLInt,
                description: 'Updates the count',
                resolve: (parent, args, bla, queryAST) => {
                    if (('operation' in parent) && parent.operation === 'subscribe') {
                        events.emit('graphql:subscribe', parent.socket.id, queryAST);
                        return '';
                    }
                    if (('operation' in parent) && parent.operation === 'unsubscribe') {
                        events.emit('graphql:unsubscribe', parent.socket.id, queryAST);
                        return '';
                    }
                    return count;
                }
            }
        }
    })
});
```

Ilustración 33, Ejemplo de Schema GraphQL con subscripciones

En la operaciones de subscripción se dispara un evento de tipo subscripción que guarda el contexto (en este caso la referencia al cliente de socket.io) y la query de la subscripción en forma de AST (abstract sintax tree).

```
import {EventEmitter2} from 'eventemitter2';
import {graphql} from 'graphql';
import schema from './schema';

let subscriptionStorage = {};

export const events = new EventEmitter2({
    wildcard: true,
    maxListeners: 500 // pq si
});

events.on('graphql:subscribe', (socketId, query) => {
    if (!(query.fieldName in subscriptionStorage)) {
        subscriptionStorage[query.fieldName] = {};
    }
    subscriptionStorage[query.fieldName][socketId] = query.operation.loc.source.body;
});

events.on('graphql:unsubscribe', (socket, query) => {
    delete subscriptionStorage[query.fieldName][socketId];
});

events.on('updatedCount', () => {
    if (subscriptionStorage.updatedCount) {
        Object.keys(subscriptionStorage.updatedCount).forEach(subscription => {
            graphql(schema, subscriptionStorage.updatedCount[subscription], {})
                .then(resolved => {
                        console.log(JSON.stringify(resolved, null, 2));
                    }
                );
        });
    }

});
```

Ilustración 34, Gestión de eventos GraphQL

En el ilustración anterior se aprecia que cada vez que alguien se subscribe se guarda el nombre de la query y el identificador del cliente (el identificador de socket.io). Cada vez que se ejecuta una actualización del contador se vuelve a resolver la query de GraphQL.

Se puede ver una implementación más completa en los repositorios de GitHub

https://github.com/farconada/graphql-subscriptions

https://github.com/eyston/graphql-todo-subscriptions

Las subscripciones en GraphQL son mucho más complicadas que con REST puesto que como el esquema de la respuesta lo define el cliente hay que mantener un estado bastante complejo para cada cliente. Actualmente las subscripciones no están bien resueltas. En REST en cambio como el cliente no decide el formato de los datos que se le devuelve hace que no haya que ejecutar la consulta para cada cliente en cada evento y que además se pueda cachear fácilmente.

9. Conclusiones

Una aplicación monolítica al final acaba siendo un gran problema, así que es bueno descomponer ese problema en problemas mas pequeños. No vale descomponer las aplicaciones de cualquier forma. Los microservicios se crean como una forma de descomposición basada en el concepto de los bounded context (contexto delimitado) y con una filosofía de no compartir nada. La comunicación de los microservicios se realiza a través de su interfaz o contrato de servicio con los clientes.

Los diferentes microservicios se utilizan entre ellos y por el cliente final, si bien el cliente final (navegador o aplicación) no suele acceder a los microservicios directamente sino que lo hace a través de un API Gateway. Normalmente los clientes finales se comunican con HTTP. API Gateway se suele implementar con proxys tipo Apache o Nginx, aunque también hay otros como Kong (https://getkong.org). El lenguaje nativo de estos API gateways es HTTP.

Hay diversos mecanismos para la comunicación de servicios. De los más completos para poder escalar y desacoplar las diferentes partes de una aplicación son los sistema de gestión de colas tipo RabbitMq, pero ello conlleva un gestionar un midleware complejo. En una primera descomposición de la aplicación la comunicación de los microservicios se puede emplear HTTP como elemento de comunicación usando APIs tipo REST o GraphQL.

La principal limitación de HTTP como lenguaje de comunicación es cómo implementar llamadas asíncronas y subscripciones. HTTP es de tipo Request/Response y no Publish/subscribe. Aunque hay implementaciones de Publish/Subscribe no es su forma de comunicación natural y sin un broker de mensajería no va a tener nunca las funcionalidades que ofrecen brokers tipo RabbitMQ.

GraphQL parece la evolución a REST, aunque todavía no está muy desarrollado. La especificación de subscripciones todavía no existe aunque ya se sabe que estará y hay diversas iniciativas para implementarla. La gran ventaja de GraphQL es que deja el contrato de servicio entre cliente y servidor en manos de los clientes de forma que hace mucho más fácil la evolución de la API puesto que se reducen los endpoints y la pesadilla del versionado de las APIs REST.

Esto también genera nuevas complicaciones como hemos visto en el caso de las subscripciones.

10. Bibliografía

- Building Microservices, Designing Fine-Grained Systems. Sam NewMan. O'Reilly
- Patterns of Enterprise Application Architecture. Martin Fowler
- Microservices vs. Service-Oriented Architecture. Mark Richards. O'Reilly
- https://www.nginx.com/blog/introduction-to-microservices/
- https://www.nginx.com/blog/building-microservices-using-an-api-gateway/
- https://www.nginx.com/blog/building-microservices-inter-process-communication/
- https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/
- https://www.nginx.com/blog/event-driven-data-management-microservices/
- https://www.nginx.com/blog/deploying-microservices/
- https://www.nginx.com/blog/refactoring-a-monolith-into-microservices/
- https://www.nginx.com/blog/connecting-applications-part-nginx-microservices/
- https://www.nginx.com/blog/microservices-soa/
- http://blog.christianposta.com/microservices/why-microservices-should-be-event-driven-autonomy-vs-authority/?utm_content=buffer25562&utm_medium=social&utm_source=twitter.com&utm_campaign=buffer
- https://auth0.com/blog/2015/11/07/introduction-to-microservices-part-4-dependencies/
- https://www.thoughtworks.com/insights/blog/microservices-evolutionary-architecture
- https://www.innoq.com/en/blog/why-restful-communication-between-microservices-can-be-perfectly-fine/
- https://medium.com/chute-engineering/graphql-in-the-age-of-rest-apis-b10f2bf09bba#.4bx174qpb
- https://edgecoders.com/restful-apis-vs-graphql-apis-by-example-51cb3d64809a#.aczewnojf

- https://keywordbrain.com/blog/understanding-graphql-server/
- http://blog.startifact.com/posts/graphql-and-rest.html
- https://blog.jacobwgillespie.com/from-rest-to-graphql-b4e95e94c26b#.y84j9fl64
- http://hueypetersen.com/posts/2015/10/28/experimenting-with-graphql-subscriptions/
- http://davidandsuzi.com/writing-a-basic-api-with-graphql/
- https://www.reindex.io/blog/how-facebooks-graphql-will-change-backend-development/
- https://gist.github.com/OlegIlyenko/a5a9ab1b000ba0b5b1ad
- https://kadira.io/blog/graphql/subscriptions-in-graphql
- http://graphql.org/blog/subscriptions-in-graphql-and-relay/
- https://cometd.org/
- https://blogs.oracle.com/slc/entry/introduction_to_bayeux_protoco
- https://blogs.oracle.com/slc/entry/introduction_to_bayeux_protoco
- https://thrift.apache.org/
- https://avro.apache.org/
- https://developers.google.com/protocol-buffers/
- http://www.grpc.io/
- https://www.rabbitmq.com/
- http://graphql.org/

11. Anexo I: Containers para microservicios

En el desarrollo de este proyecto se han realizado diversas pruebas de descomposición de aplicaciones en microservicios y algunos de ellos se han ejecutado en containers Docker (https://www.docker.com/).

Docker es un proyecto de código abierto que automatiza el despliegue de aplicaciones dentro de contenedores de software, proporcionando una capa adicional de abstracción y automatización de Virtualización a nivel de sistema operativo en Linux. Docker utiliza características de aislamiento de recursos del kernel de Linux, tales como cgroups y espacios de nombres (namespaces) para permitir que "contenedores" independientes se ejecuten dentro de una sola instancia de Linux, evitando la sobrecarga de iniciar y mantener máquinas virtuales.

El soporte del kernel de Linux para los espacios de nombres aísla de vista, en su mayoría, una aplicación del entorno operativo, incluyendo árboles de proceso, red, ID de usuario y sistemas de archivos montados, mientras que los cgroups del kernel proporcionan aislamiento de recursos, incluyendo la CPU, la memoria, el bloque de E / S y de la red.

Usando Docker para crear y gestionar contenedores puede simplificar la creación de sistemas altamente distribuidos, permitiendo múltiples aplicaciones, las tareas de los trabajadores y otros procesos para funcionar de forma autónoma en una única máquina física o en varias máquinas virtuales. Esto permite que el despliegue de nodos se realice a medida que se dispone de recursos o cuando se necesiten más nodos, lo que permite una plataforma como servicio (PaaS - Plataform as a Service) de estilo de despliegue y ampliación de los sistemas como Apache Cassandra, MongoDB o Riak. Docker también simplifica la creación y el funcionamiento de las tareas de carga de trabajo o las colas y otros sistemas distribuidos.

Docker hoy en día se impone como el estándar de facto en cuanto a containers. A raíz de esto surge la duda sobre el tamaño de los containers. ¿Container grandes o pequeños? ¿es un container para un microservicio o un conjunto de containers pueden formar un único microservicio? Yo soy de la opinión de que un container debe contener un microservicio puesto que una unidad de ejecución. Hay una tendencia natural en Docker a separar cada

proceso en un container diferente entre otras cosas por una limitación tecnológica de Docker que no ejecuta un sistema de init dentro del contenedor. No parece lógico que si una aplicación necesita, por ejemplo httpd de Apache y ejecutar tareas periódicas con cron o un syslog haya que crear 3 contenedores para ello.

Hay varios intentos de paliar esto, por ejemplo lanzando como proceso principal un gestor de procesos como supervisord (http://supervisord.org/) y que se encargue este de gestionar. Estos gestores de procesos son muy limitados y no llegan con mucho a la funcionalidad que ofrecen verdaderos gestores como systemd (https://www.freedesktop.org/wiki/Software/systemd/). No tener un gestor de servicios nativo en el container hace que no sea suficiente con instalar el RPM correspondiente en el container para que el servicio se arranque. A día de hoy hay varios intentos de ejecutar systemd dentro Docker pero ello conlleva una rebaja de la seguridad del container para que funcione.

En cuanto a la seguridad de los container existe otro punto a debatir. Los containers se crean como una forma de encapsulación de servicios, por un lado como una forma de estandarizar los procesos para que sean fácilmente transportables y por otra como medida de seguridad como si fuesen un chroot más poderoso. Dentro de los containers determinadas funcionalidades del kernel de Linux se ejecutan en namespaces, por ejemplo los PIDs de los procesos y las interfaces de red. Linux, en las principales distribuciones, implementa SELinux (https://selinuxproject.org/page/Main_Page) como un sistema de Mandatory Access Control (MAC) y Role Base Access Control (RBAC). Esto sirve por ejemplo para tener un control fino de las funcionalidades de un proceso dentro del sistema. Se pueden definir políticas del tipo, el servicio MySQL no puede ejecutar binarios fuera de la carpeta /usr/bin o no puede enviar correos. SELinux funciona etiquetando los archivos del sistema. Linux no tiene un sistema de namespaces para SELinux y cada container de Docker se ejecuta etiquetado de la misma forma para todos los archivos del container, esto implica que si en un mismo container tenemos un servidor web y un MySQL no podemos establecer diferencias a nivel de SELinux entre un proceso y otro, lo que puede entenderse como una rebaja de la seguridad.

Como SELinux tiene cierta complejidad para definir las políticas, las principales distribuciones de Linux ofrecen políticas ajustadas para cada tipo de

proceso con un nivel seguridad y granularidad bastante bueno. Cuando ejecutamos containers ya no se aplican están políticas personalizadas para cada procesos sino que se ejecuta una política genérica para containers y queda en manos del usuario ampliarla.

12. Anexo II: Cabecera X-Response-To

En el transcurso de este trabajo se ha visto una forma de conversación de microservicios en el que hay tres participantes: el cliente, el servidor y el receptor de la respuesta. Para que el servidor sepa quien es el receptor de la respuesta hay varias opciones. O dejarlo pregrabado en la configuración o dejar que sea el cliente el que le indique donde quiere recibir la respuesta. Esta última forma es muy interesante por las posibilidades que puede ofrecer.

Una forma de que el cliente informe al servidor del receptor de la respuesta puede ser empleando una cabecera HTTP, por ejemplo X-Response-To. Determinar el valor de esta cabecera puede ser algo más complicado de establecer. Por ejemplo el valor de X-Response-To puede ser una URL de tipo http://… pero cómo indicar que método emplear GET, POST, PUT…. El método no va codificado en la URL. También se puede entender que siempre debería ser un POST puesto que es una entrega de datos.

La dirección de la respuesta no tiene porqué ser solamente HTTP, puesto que también podría ser AMQP por ejemplo amqp://user:pass@host:10000/vhost (https://www.rabbitmq.com/uri-spec.html). Si hay que establecer parámetros adicionales que no encajen en un esquema de URI se podrían definir otras cabeceras complementarias derivadas de X-Response-To.

Se ve interesante desarrollar este esquema de cabeceras y plasmarlo de alguna forma.

Por otro lado la comunicación con 3 partes en la que una de las partes recibe datos que no ha solicitado puede implicar ciertos problemas de seguridad. Por un lado la autenticación que se hace más compleja y por otro lado se pueden generar problemas de denegación de servicio, por ejemplo si a partir de una request sencilla se escriben muchos datos de respuesta en un tercero de forma que se amplifica el ataque.