2. Demystifying Microservices Applications-ES

El auge de las arquitecturas orientadas a servicios (SOA) y los microservices

#aprendizaje

Definidos brevemente, los microservices son fragmentos de software desplegados en redes de computadoras que se comunican a través de protocolos de red. Sin embargo, esto no es todo; también deben cumplir un conjunto de restricciones adicionales.

Antes de dar una definición más detallada de lo que es una arquitectura de microservices, debemos entender cómo evolucionó la idea de los microservices y qué tipo de problemas fue llamada a resolver. Describiremos los dos pasos principales de esta evolución en dos subsecciones separadas.

El auge de SOA

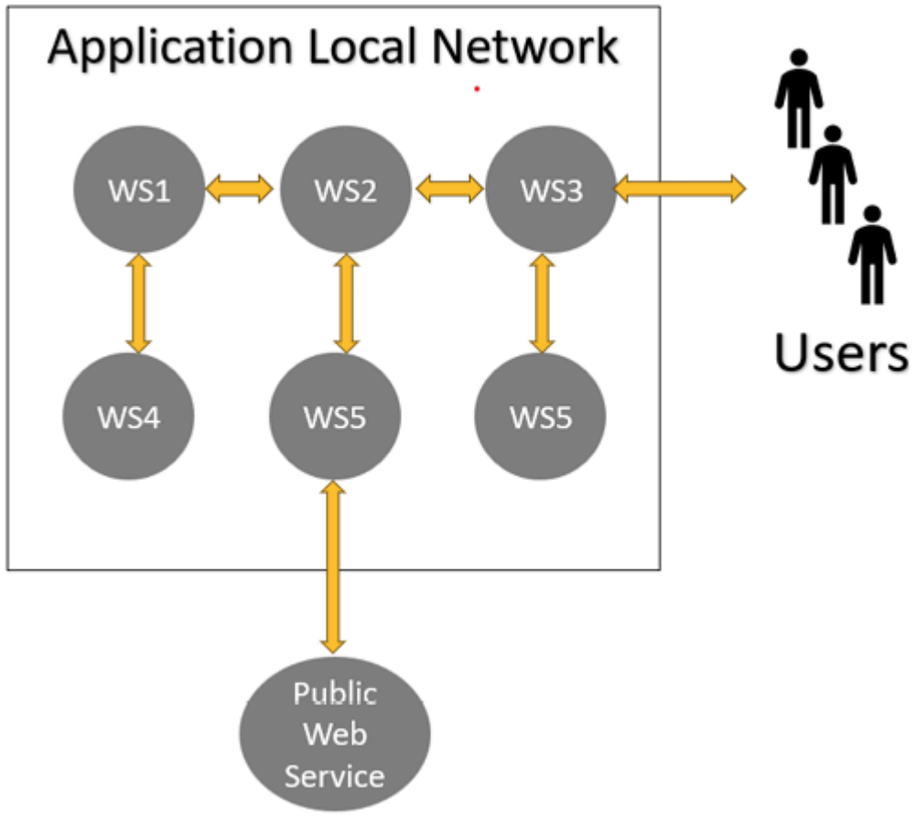
Las llamadas **arquitecturas orientadas a servicios**, o **SOA** (por sus siglas en inglés), fueron el primer paso hacia los microservices. Se trata de arquitecturas basadas en redes de procesos que se comunican entre sí. Inicialmente, las SOA se implementaron como servicios web similares a los que podrías haber experimentado en ASP.NET Core.

En una SOA, diferentes macro-módulos que implementan distintas funcionalidades o roles en las aplicaciones de software se exponen como procesos separados que se comunican entre sí a través de protocolos estándar. La primera implementación de SOA fueron servicios web que se comunicaban mediante el protocolo SOAP basado en XML. Luego, la mayoría de las arquitecturas de servicios web migraron hacia web APIs basadas en JSON, que quizás ya conozcas, ya que los servicios web REST están disponibles como plantillas de proyecto estándar en ASP.NET. La sección *Further reading* contiene enlaces útiles con más detalles sobre servicios web REST.

Las SOA fueron concebidas durante el auge de la creación de software para aplicaciones empresariales como una de las formas de integrar las diversas aplicaciones preexistentes utilizadas por diferentes sucursales y divisiones en un sistema de información empresarial unificado. Dado que las aplicaciones preexistentes estaban implementadas con diferentes tecnologías, y la experiencia en software disponible en las distintas sucursales y divisiones era heterogénea, SOA fue la respuesta a las siguientes necesidades apremiantes:

1. Permitir la comunicación de software entre módulos implementados con diferentes tecnologías y ejecutándose en diferentes plataformas (Linux + Apache, Linux + NGINX o Windows + IIS). De hecho, el software basado en diferentes tecnologías no es compatible a nivel binario, pero aún puede cooperar con otros si cada uno se implementa como un servicio web que se comunica con los demás a través de un protocolo estándar independiente de la tecnología. Entre ellos, vale la pena mencionar el protocolo HTTP REST basado en texto y el protocolo binario gRPC. También vale la pena señalar que el protocolo HTTP REST es un estándar oficial, mientras que actualmente gRPC es solo un estándar de facto propuesto por Google. La sección *Further reading* contiene enlaces útiles para obtener más detalles sobre estos protocolos.
2. Permitir que el versionamiento de cada macro-módulo evolucione de forma independiente de los demás. Por ejemplo, podrías decidir migrar algún servicio web hacia la nueva versión de .NET 9 para aprovechar nuevas características o nuevas bibliotecas disponibles, mientras dejas otros servicios web que no necesitan modificaciones con una versión anterior, digamos, .NET 8.

3. Promover servicios web públicos que ofrezcan servicios a otras aplicaciones. Como ejemplo, piensa en los diversos servicios públicos ofrecidos por Google, como Google Maps, o los servicios de inteligencia artificial ofrecidos por Microsoft, como los servicios de traducción de idiomas.
4. A continuación se muestra un diagrama que resume la SOA clásica.



2.1-SOA

5. Con el tiempo, el sistema de información empresarial y otras aplicaciones SOA complejas conquistaron más mercados y usuarios, por lo que aparecieron nuevas necesidades y restricciones. Las discutiremos en la siguiente subsección.

Hacia las arquitecturas de microservices

A medida que los usuarios y el tráfico de las aplicaciones aumentaron en órdenes de magnitud, la optimización del rendimiento y el balance óptimo de los recursos de hardware entre los distintos módulos de software se convirtieron en una necesidad imperativa. Esto llevó a un nuevo requisito:

> **Important**
>
> Cada módulo de software debe poder escalarse de forma independiente de los demás, de modo que podamos asignar a cada módulo la cantidad óptima de recursos que necesita.

A medida que el sistema de información empresarial ganó un papel central, su operación continua, es decir, un tiempo de inactividad prácticamente nulo, se convirtió en un requisito indispensable, lo que llevó a otra restricción importante:

> **Important**
>
> La arquitectura de microservices debe ser redundante. Cada módulo de software debe tener varias réplicas ejecutándose en diferentes nodos de hardware para resistir fallos de software y de hardware.

Además, para adaptar cada aplicación a un mercado en rápida evolución, los requisitos sobre los tiempos de desarrollo se volvieron más exigentes. En consecuencia, se necesitaron más desarrolladores para desarrollar y mantener cada aplicación con los estrictos hitos establecidos.

Desafortunadamente, gestionar proyectos de software que involucran a más de aproximadamente cuatro personas con la calidad requerida demostró ser prácticamente imposible. Por lo tanto, se añadió una nueva restricción a las SOA:

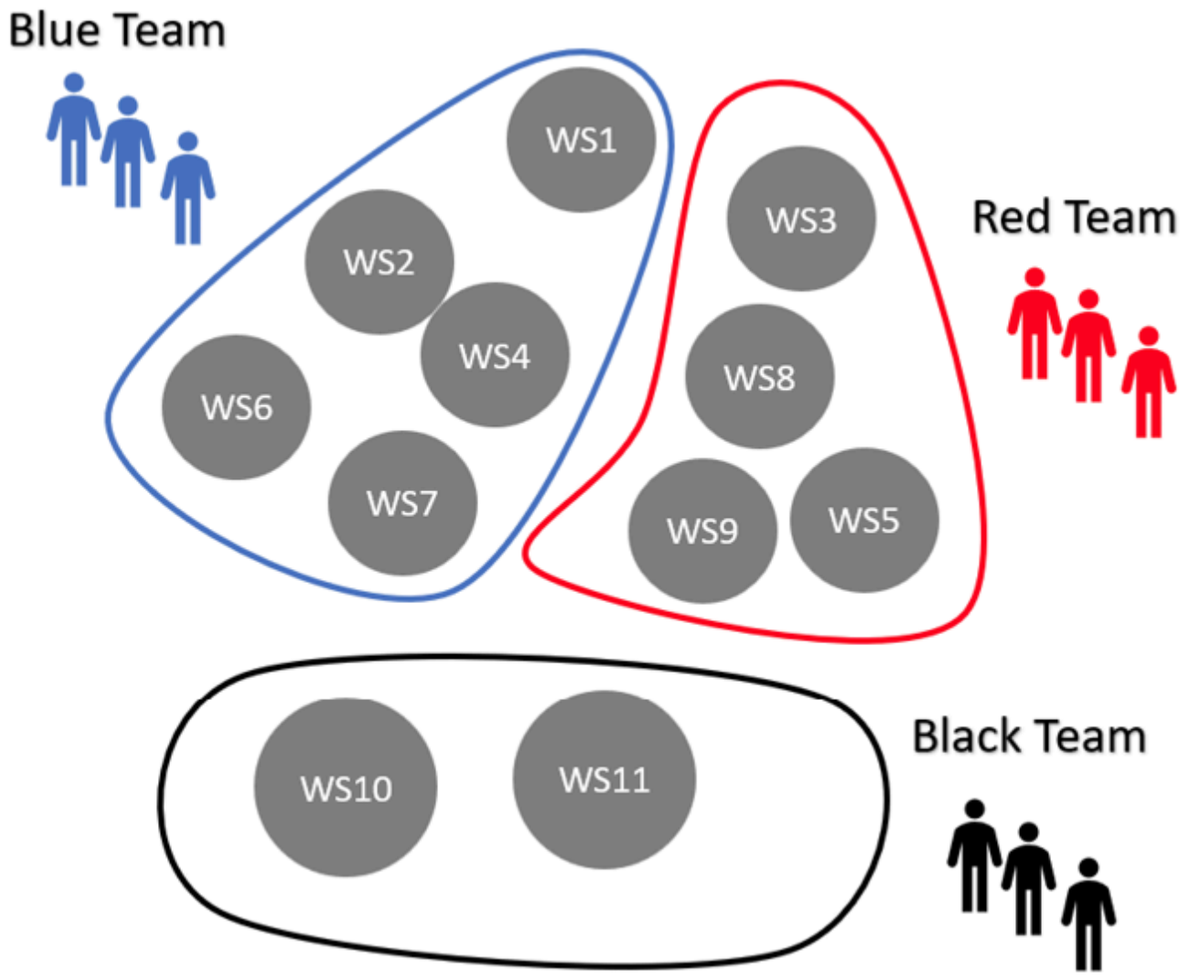
> **Important**
>
> Los servicios que componen una aplicación deben ser completamente independientes entre sí, de modo que puedan ser implementados por equipos separados con interacción mínima.

Sin embargo, el esfuerzo de mantenimiento también necesitaba optimizarse, lo que dio lugar a otra restricción importante:

> **Important**
>
> Las modificaciones a un servicio no deben propagarse a otros servicios. En consecuencia, cada servicio debe tener una interfaz bien definida que no cambie con el mantenimiento del software (o que, al menos, cambie raramente). Por la misma razón, las decisiones de diseño adoptadas en la implementación de un servicio no deben restringir a ningún otro servicio de la aplicación.

Los primeros dos requisitos pueden satisfacerse implementando cada módulo de software como un servicio separado, de modo que podamos asignarle más recursos de hardware simplemente replicándolo en N instancias diferentes según sea necesario para optimizar el rendimiento general y garantizar la redundancia.

1. También necesitamos un nuevo actor, algo que decida cuántas copias de cada servicio usar y en qué hardware ubicarlas. Existen entidades similares llamadas **orquestadores**. Vale la pena señalar que también podríamos tener varios orquestadores, cada uno encargado de un subconjunto de los servicios, ¡o incluso no tener ningún orquestador!
2. En resumen, pasamos de aplicaciones compuestas por servicios web de grano grueso y acoplados a microservices de grano fino y débilmente acoplados, cada uno implementado por un equipo de desarrollo diferente, como se muestra en la siguiente figura.



2.2-Microservices architecture

3. El diagrama muestra microservices de grano fino asignados a diferentes equipos débilmente acoplados. Vale la pena señalar que aunque el acoplamiento débil también era un objetivo inicial de las arquitecturas de servicios web primitivas, tomó tiempo mejorar hasta un buen nivel, alcanzando su punto máximo con la llegada de las técnicas de microservices.
4. El diagrama anterior y los requisitos presentados no definen exactamente qué son los microservices; solo explican el inicio de la era de los microservices. En la siguiente sección, daremos una definición más formal de los microservices que refleje su etapa actual de evolución.

Definición y organización de las arquitecturas de microservices

Definición de una arquitectura de microservices

Una arquitectura de microservices es una arquitectura basada en SOA que satisface todas las restricciones siguientes:

- Los límites de los módulos se definen según el dominio del negocio que requieren. Como discutiremos en las subsecciones siguientes, esto debería garantizar que estén débilmente acoplados.
- Cada módulo se implementa como un servicio replicable, llamado **microservice**, donde replicable significa que se pueden crear varias instancias de cada servicio para garantizar la escalabilidad y la redundancia.
- Cada servicio puede ser implementado y mantenido por un equipo diferente, donde todos los equipos están débilmente acoplados.
- Cada servicio tiene una interfaz bien definida conocida por todos los equipos involucrados en el proyecto de desarrollo.

- Los protocolos de comunicación se deciden al inicio del proyecto y son conocidos por todos los equipos.
- Cada servicio debe depender únicamente de la interfaz expuesta por los demás y de los protocolos de comunicación adoptados. En particular, ninguna decisión de diseño adoptada para un servicio puede imponer restricciones sobre la implementación de los demás.

Dominio del negocio y microservices

Esta restricción tiene el propósito de proporcionar una regla práctica para definir los límites de cada microservice, de modo que los microservices se mantengan débilmente acoplados y puedan ser gestionados por equipos débilmente acoplados. Se basa en la teoría del **diseño dirigido por el dominio** (domain-driven design) desarrollada por Eric Evans (ver *Domain-Driven Design*: https://www.amazon.com/exec/obidos/ASIN/0321125215/domainlanguag-20).

Básicamente, cada dominio del negocio utiliza un lenguaje típico. Por lo tanto, durante el análisis, basta con detectar cambios en el lenguaje utilizado por los expertos con quienes se habla para comprender qué se incluye y qué se excluye de cada microservice.

La lógica detrás de esta técnica es que las personas que interactúan intensamente siempre desarrollan un lenguaje específico reconocido por otros que comparten el mismo dominio del negocio, mientras que la ausencia de dicho lenguaje común es una señal de interacción débil.

De esta manera, el **dominio** de la aplicación o un **subdominio** se divide en los llamados **bounded contexts** (contextos delimitados), cada uno caracterizado por el uso de un lenguaje común. Vale la pena señalar que **dominio**, **subdominio** y **bounded context** son conceptos centrales de DDD. Para más detalles sobre ellos y DDD, puedes consultar la sección *Further reading*, pero nuestra descripción simple debería ser suficiente para comenzar con microservices.

Así, obtenemos la primera división de la aplicación en **bounded contexts**. Cada uno se asigna a un equipo y se define una interfaz formal para cada uno. Esta interfaz se convierte en la especificación de un microservice, y también es todo lo que los demás equipos necesitan saber sobre dicho microservice.

Luego, cada equipo al que se le ha asignado un microservice puede dividirlo aún más en microservices más pequeños para escalar cada uno de forma independiente, verificando que cada microservice resultante intercambie una cantidad aceptable de mensajes con los demás (acoplamiento débil).

La primera división se usa para repartir el trabajo entre los equipos, mientras que la segunda división está diseñada para optimizar el rendimiento de diversas maneras, que detallaremos en la subsección *Organización de microservices*.

Microservices replicables

Debe existir una forma de crear varias instancias del mismo microservice y ubicarlas en el hardware disponible para asignar más recursos de hardware a los microservices más críticos. Para algunas aplicaciones o microservices individuales, esto puede hacerse manualmente; pero, con mayor frecuencia, se adoptan herramientas de software dedicadas llamadas **orquestadores**.

División del desarrollo de microservices entre diferentes equipos

La forma en que se definen los microservices para que puedan asignarse a diferentes equipos débilmente acoplados ya se explicó en la subsección *Dominio del negocio y microservices*. Aquí, vale la pena señalar que los microservices definidos en esta etapa se denominan

microservices lógicos, y luego cada equipo puede decidir dividir cada microservice lógico en uno o más **microservices físicos** por diversas razones prácticas.

Microservices, interfaces y protocolos de comunicación

Una vez que los microservices se asignan a diferentes equipos, es momento de definir sus interfaces y el protocolo de comunicación utilizado para cada tipo de mensaje. Esta información se comparte entre todos los equipos para que cada uno sepa cómo comunicarse con los microservices gestionados por los demás equipos.

Solo las interfaces de todos los microservices lógicos y los protocolos de comunicación asociados deben compartirse entre todos los equipos, mientras que la división de cada microservice lógico en microservices físicos solo se comparte dentro de cada equipo.

La coordinación de los diversos equipos, y la documentación y monitoreo de todos los servicios, se logra con varias herramientas. A continuación se presentan las principales:

- Los **context maps** (mapas de contexto) son una representación gráfica de las relaciones organizacionales entre los diversos equipos que trabajan en todos los bounded contexts de la aplicación.
- Los **catálogos de servicios** recopilan información sobre todos los requisitos, equipos, costos y otras propiedades de los microservices. Herramientas como **Datadog** (https://docs.datadoghq.com/service_catalog/) y **Backstage** (https://backstage.io/docs/features/software-catalog/) realizan diversos tipos de monitoreo, mientras que herramientas como **Postman** (https://www.postman.com/) y **Swagger** (https://swagger.io/) se enfocan principalmente en requisitos formales, como las pruebas y la generación automática de clientes para interactuar con los servicios.

Solo las interfaces de los microservices lógicos son públicas

El código de cada microservice no puede hacer suposiciones sobre cómo se implementa la interfaz pública de todos los demás microservices lógicos. No se puede asumir nada sobre las tecnologías utilizadas (.NET, Python, Java, etc.) y sus versiones, ni sobre los algoritmos y las arquitecturas de datos utilizadas por otros microservices.

Organización de microservices

La primera consecuencia de la independencia en las decisiones de diseño de los microservices es que cada microservice debe tener almacenamiento privado, porque una base de datos compartida causaría dependencias entre los microservices que la comparten. Supongamos que los microservices A y B acceden a la misma tabla de base de datos, T. Ahora, estamos modificando el microservice A para cumplir con los nuevos requisitos de un usuario. Como parte de esta actualización, la solución para A requerirá que reemplacemos la tabla T con dos nuevas tablas, T1 y T2.

En una situación similar, nos veríamos obligados a cambiar también el código de B para adaptarlo al reemplazo de T por T1 y T2. Claramente, la misma limitación no aplica a diferentes instancias del mismo microservice, por lo que ambas pueden compartir la misma base de datos. En resumen, podemos afirmar lo siguiente:

> **Important**
>
> Las instancias de diferentes microservices no pueden compartir una base de datos común.

Desafortunadamente, alejarse de una base de datos de aplicación única inevitablemente conduce a la duplicación de datos y desafíos de coordinación. Más específicamente, el mismo fragmento de datos debe duplicarse en varios microservices, por lo que cuando cambia, el cambio debe comunicarse a todos los microservices que están usando una copia duplicada del mismo.

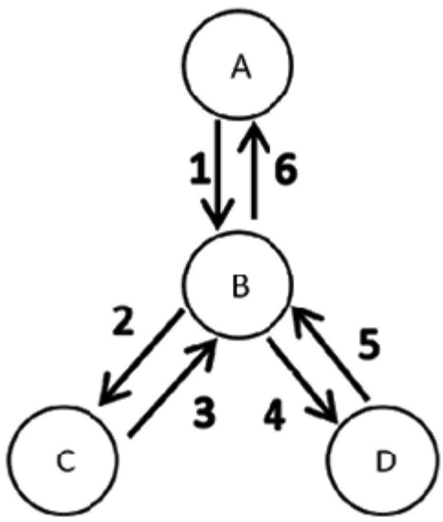
Por lo tanto, podemos establecer otra restricción organizacional:

> **Important**
>
> Los microservices deben diseñarse de manera que se minimice la duplicación de datos, o dicho de otra forma, las duplicaciones deben involucrar la menor cantidad posible de microservices.

Como se mencionó en la sección anterior, si definimos los microservices según el dominio del negocio, la última restricción debería cumplirse automáticamente porque diferentes dominios del negocio generalmente comparten muy pocos datos.

No se derivan otras restricciones directamente de la definición de microservices, pero basta con agregar una restricción trivial de rendimiento sobre el tiempo de respuesta para forzar la organización de los microservices de una manera que se asemeja más a una línea de ensamblaje que al software típico basado en solicitudes del usuario. Veamos por qué.

Una solicitud de usuario que llega al microservice A podría causar, a su vez, una larga cadena de solicitudes emitidas a otros microservices, como se muestra en la siguiente figura:



2.3-Chain of synchronous request-responses

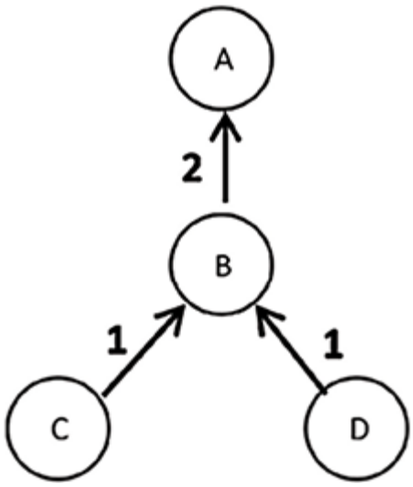
Los mensajes 1-6 son desencadenados por una solicitud al microservice *A* y se envían en secuencia, por lo que sus tiempos de procesamiento se suman al tiempo de respuesta. Además, el microservice *A*, después de haber enviado el mensaje *1*, permanece bloqueado esperando una respuesta hasta que recibe el último mensaje (*6*); es decir, permanece bloqueado durante toda la vida útil del proceso de comunicación encadenado.

El microservice **B** permanece bloqueado dos veces, esperando la respuesta a una solicitud que emitió. La primera vez durante la comunicación *2-3* y la segunda durante la comunicación *4-5*. En resumen, un patrón ingenuo de solicitud-respuesta para la comunicación entre microservices implica altos tiempos de respuesta y un desperdicio del tiempo de cómputo de los microservices.

Las únicas formas de superar los problemas anteriores son evitar dependencias completas entre microservices o almacenar en caché toda la información necesaria para satisfacer cualquier solicitud de usuario en el primer microservice, *A*. Dado que alcanzar independencia

total es básicamente imposible, la solución habitual es almacenar en caché en *A* cualquier dato que necesite para responder solicitudes sin pedir información adicional a otros microservices.

Para lograr este objetivo, los microservices son proactivos y adoptan el enfoque llamado **intercambio asíncrono de datos** (asynchronous data-sharing). Cada vez que actualizan datos, envían la información actualizada a todos los demás microservices que la necesitan para sus respuestas. En pocas palabras, en el ejemplo anterior, los nodos del árbol, en lugar de esperar solicitudes de sus nodos padre, envían datos preprocesados a todos sus posibles invocadores cada vez que sus datos privados cambian, como se muestra en la figura siguiente.



2.4-Data-driven communication

Ambas comunicaciones etiquetadas como *1* se desencadenan cuando los datos de los microservices *C*/*D* cambian, y pueden ocurrir en paralelo. Además, una vez que se envía la comunicación, cada microservice puede volver a su trabajo sin esperar una respuesta. Finalmente, cuando una solicitud llega al microservice *A*, este ya tiene todos los datos que necesita para construir la respuesta sin necesidad de interactuar con otros microservices. En general, los microservices basados en **intercambio asíncrono de datos** preprocesan datos y los envían a cualquier otro servicio que pueda necesitarlos tan pronto como sus datos cambian. De esta manera, cada microservice ya contiene datos precalculados que puede usar para responder inmediatamente a las solicitudes del usuario sin necesidad de comunicaciones adicionales específicas de la solicitud.

En este caso, no podemos hablar de solicitudes y respuestas, sino simplemente de mensajes intercambiados. Las personas que trabajan con aplicaciones web clásicas estarán acostumbradas a comunicaciones de solicitud/respuesta donde un cliente emite una solicitud y un servidor la procesa y envía una respuesta.

Important

En general, en una comunicación de solicitud/respuesta, uno de los actores involucrados, digamos **A**, envía un mensaje que contiene una **solicitud** para realizar un procesamiento específico a otro actor, digamos **B**, luego **B** realiza el procesamiento requerido y devuelve un resultado (la **respuesta**), que también puede ser una notificación de error.

Sin embargo, también podemos tener comunicaciones que no están basadas en solicitud/respuesta. En este caso, simplemente hablamos de mensajes. No hay respuestas sino solo acuses de recibo de que los mensajes han sido recibidos correctamente por el destinatario final o un actor intermedio. A diferencia de las respuestas, los acuses de recibo se envían antes de completar el procesamiento de los mensajes.

Volviendo al **intercambio asíncrono de datos**, a medida que nuevos datos están disponibles, cada microservice hace su trabajo y luego envía los resultados a todos los microservices interesados, y continúa realizando su trabajo sin esperar una respuesta de sus destinatarios.

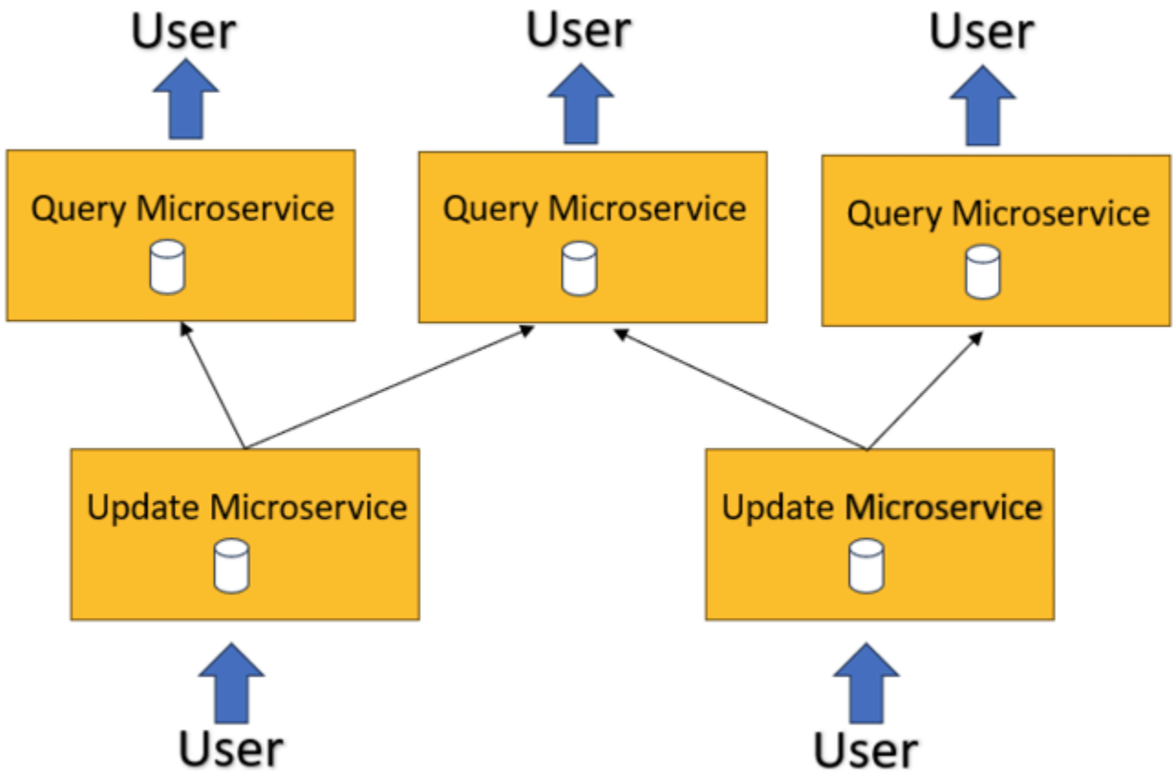
Cada emisor solo espera un acuse de recibo de su destinatario inmediato, por lo que los tiempos de espera no se acumulan como en el ejemplo inicial de solicitud/respuesta encadenada.

¿Qué pasa con los acuses de recibo de mensajes? También causan pequeños retrasos. ¿Es posible eliminar también esta menor ineficiencia? Por supuesto, ¡con la ayuda de la comunicación asíncrona!

> **Important**
>
> En la comunicación síncrona, el emisor espera el acuse de recibo del mensaje antes de continuar su procesamiento. De esta manera, si el acuse de recibo expira o es reemplazado por una notificación de error, el emisor puede realizar acciones correctivas, como reenviar el mensaje.
>
> En la comunicación asíncrona, el emisor no espera ni un acuse de recibo ni una notificación de error, sino que continúa su procesamiento inmediatamente después de enviar el mensaje, mientras que los acuses de recibo o las notificaciones de error se envían a un callback.

La comunicación asíncrona es más efectiva en microservices porque evita completamente los tiempos de espera. Sin embargo, la necesidad de realizar acciones correctivas en caso de posibles errores complica la acción general de envío de mensajes. Más específicamente, todos los mensajes enviados deben agregarse a una cola, y cada vez que llega un acuse de recibo, el mensaje se marca como enviado correctamente y se elimina de esta cola. De lo contrario, si no llega ningún acuse de recibo dentro de un tiempo configurable de `timeout`, o si se genera un error, el mensaje se marca para ser reenviado según algunas políticas de reintento.

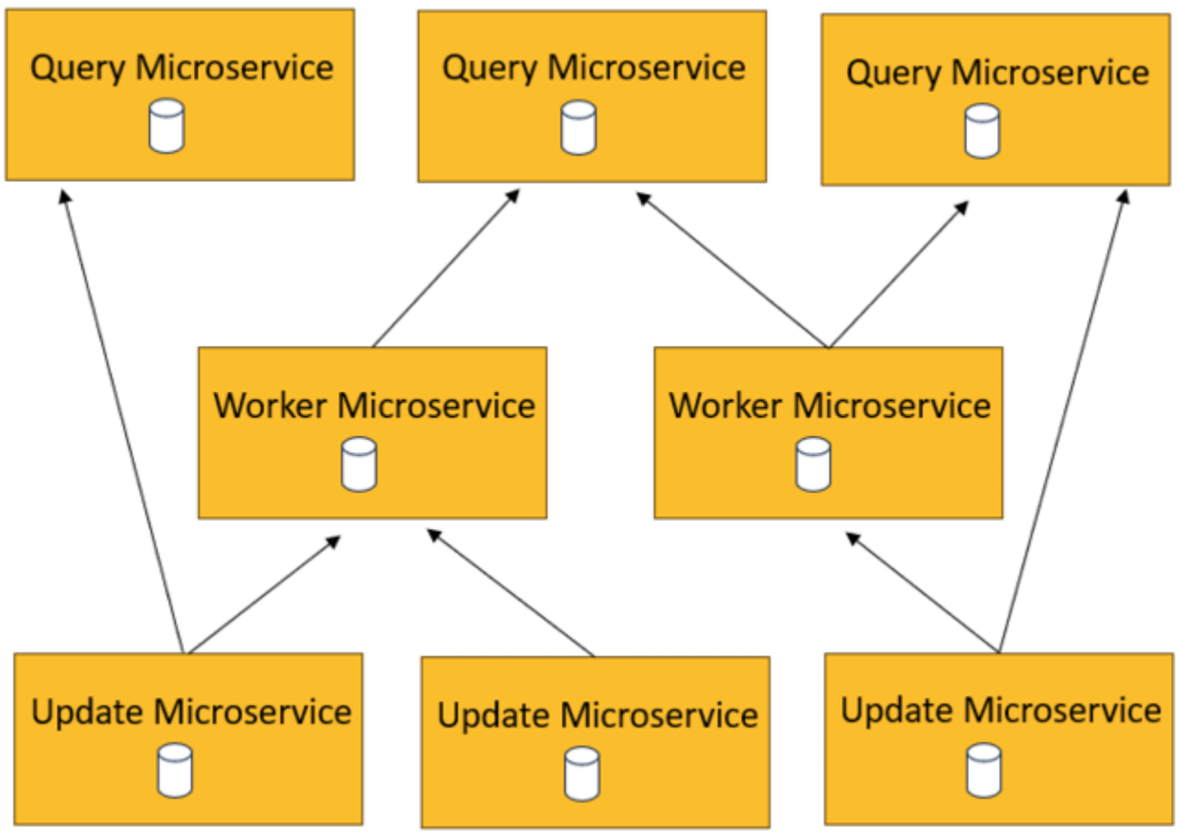
El enfoque de **intercambio asíncrono de datos** de microservices a menudo se acompaña del patrón llamado **Command Query Responsibility Segregation** (**CQRS**). Según CQRS, los microservices se dividen en *microservices de actualización*, que realizan las operaciones CRUD habituales, y *microservices de consulta*, que se especializan en responder consultas que agregan datos de varios otros microservices, como se muestra en la siguiente figura:



2.5-Updates and query microservices

Según el enfoque de **intercambio asíncrono de datos**, cada microservice de actualización envía todas sus modificaciones a los servicios de consulta que las necesitan, mientras que los microservices de consulta precalculan todas las consultas para garantizar tiempos de respuesta cortos. Vale la pena señalar que las actualizaciones basadas en datos se asemejan a una línea de ensamblaje de fábrica que construye todos los resultados de consulta posibles.

Tanto los microservices de actualización como los de consulta se denominan microservices **frontend** porque están involucrados en el patrón habitual de solicitud-respuesta con el usuario. Sin embargo, las actualizaciones de datos en su camino también pueden encontrar microservices que no interactúan en absoluto con un usuario. Se denominan microservices **worker**. La siguiente figura muestra la relación entre microservices worker y frontend.



2.6-Frontend and worker microservices

Mientras que los microservices frontend generalmente responden a varias solicitudes de usuario en paralelo creando un hilo para cada solicitud, los microservices worker solo están involucrados en actualizaciones de datos, por lo que no necesitan paralelizar solicitudes para garantizar tiempos de respuesta bajos al usuario.

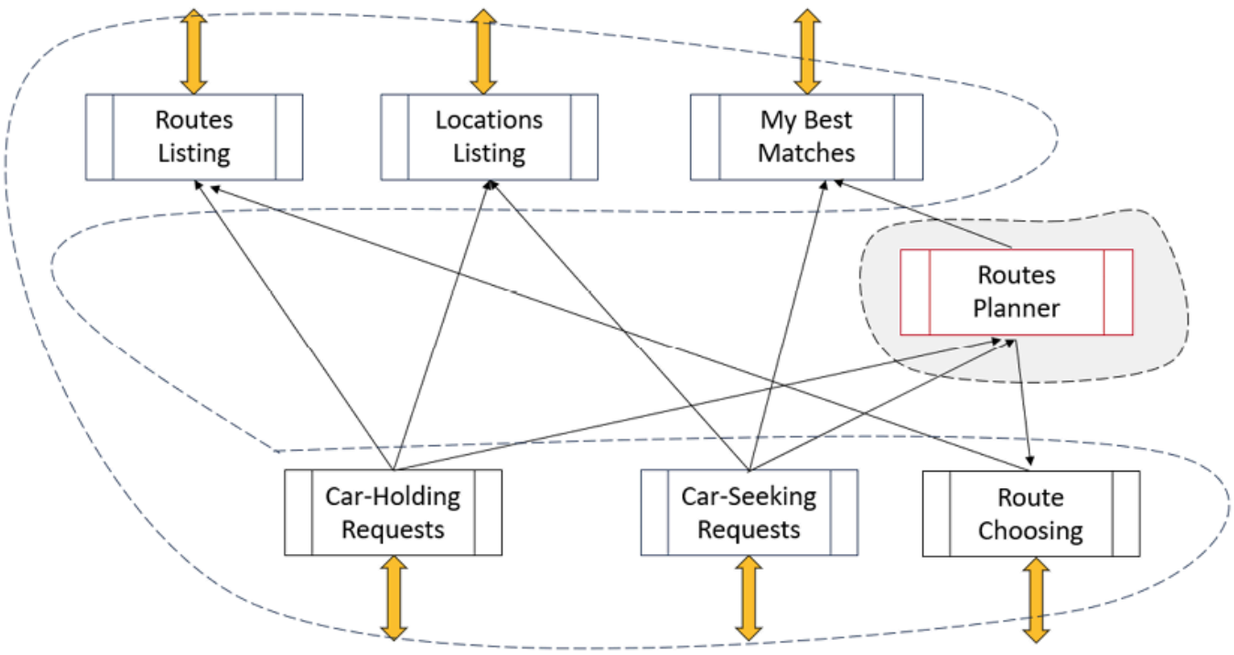
En consecuencia, su operación es completamente análoga a la de las estaciones que componen una línea de ensamblaje. Extraen sus mensajes de entrada de una cola de entrada y los procesan uno tras otro. Los datos de salida se envían a todos los microservices interesados tan pronto como están disponibles. Este tipo de procesamiento se denomina **basado en datos** (data-driven).

Se podría objetar que los microservices worker no son necesarios ya que su trabajo podría ser asumido por los servicios frontend que consumen sus salidas. ¡Este no es el caso! Por ejemplo, imaginemos datos contables que necesitan consolidarse durante un período de tiempo antes de usarse como campos de consultas complejas. Por supuesto, cada microservice de consulta que necesite los datos consolidados podría encargarse de consolidarlos. Sin embargo, esto resultaría en la duplicación del esfuerzo de procesamiento y el almacenamiento necesario para mantener las sumas parciales.

Además, incorporar el procesamiento de consolidación en otros microservices impediría su escalado independiente, con mejor optimización del rendimiento general.

Ejemplo de car-sharing

La siguiente figura muestra un diagrama de comunicación de la parte de gestión de rutas de una aplicación de car-sharing. Las líneas punteadas rodean todos los microservices físicos que pertenecen al mismo microservice lógico. Los microservices de consulta están en la parte superior de la imagen, los microservices de actualización en la parte inferior, y los microservices worker en el medio (con sombreado gris).



2.7-Route-handling subsystem of a car-sharing application

El análisis del lenguaje detectó dos microservices lógicos. El primero habla el lenguaje del usuario que comparte su auto y está compuesto por seis microservices físicos. El segundo se enfoca en la topología, ya que encuentra las mejores rutas entre un origen y un destino y empareja pares intermedios de origen-destino con rutas existentes.

Los propietarios de autos gestionan sus solicitudes con operaciones CRUD en el microservice de actualización `Car-Holding-Requests`, mientras que los usuarios que buscan un auto interactúan de manera similar con el microservice de actualización `Car-Seeking-Requests`. El

microservice `Routes-Listing` lista todos los viajes disponibles con asientos vacíos para nuevos pasajeros, ayudando a quienes buscan auto a elegir la fecha de su viaje. Una vez elegida la fecha, la solicitud se envía a través del microservice `Car-Seeking-Requests`.

Tanto los propietarios de autos como quienes buscan auto interactúan con el microservice de actualización `Route-Choosing`. Quienes buscan auto eligen una de varias rutas disponibles tanto para el origen como para el destino, mientras que los propietarios aceptan a quienes buscan auto seleccionando las rutas que se ajustan a sus orígenes y destinos. Una vez que una ruta es seleccionada por quien busca auto y aceptada por el propietario, todas las demás opciones incompatibles se eliminan de las mejores coincidencias de ambos.

Todas las rutas disponibles tanto para quienes buscan auto como para los propietarios son listadas por el microservice `My-Best-Matches`. El microservice worker `Routes-Planner` calcula las mejores rutas que se ajustan al origen y destino de un propietario de auto que también contengan orígenes y destinos de algunos buscadores de auto. Almacena las solicitudes de buscadores de auto sin emparejar hasta que se añade una ruta que pasa a una distancia aceptable de ellos. Cuando esto sucede, el microservice `Routes-Planner` crea una nueva ruta alternativa para el mismo viaje que contiene el nuevo par origen-destino. Todos los cambios de rutas se envían tanto al microservice `My-Best-Matches` como al `Route-Choosing`.

El microservice `Locations-Listing` gestiona una base de datos de ubicaciones conocidas y se utiliza en varios tipos de sugerencias al usuario, como el autocompletado de orígenes y destinos del usuario y sugerencias de viajes interesantes basadas en estadísticas de preferencias del usuario. Toma su entrada de todas las solicitudes de propietarios y buscadores de auto.

Hemos visto qué tipo de problemas fueron concebidos para resolver los microservices y cómo su adopción añade complejidad al diseño de la aplicación. Además, no es difícil imaginar que probar y mantener una aplicación que se ejecuta en varias máquinas diferentes y depende de patrones complejos de comunicación basada en datos debería ser una tarea compleja y que consume mucho tiempo.

Por lo tanto, es importante evaluar el impacto de usar una arquitectura de microservices en nuestra aplicación para verificar que el costo sea asumible y que las ventajas de la adopción superen las desventajas y los costos adicionales. En la siguiente sección, cubriremos algunos criterios para enfrentar este tipo de evaluación.

¿Cuándo vale la pena adoptar arquitecturas de microservices?

Una aplicación que requiere más de cinco desarrolladores es sin duda un buen candidato para una arquitectura de microservices, ya que los microservices lógicos ayudan a dividir la fuerza de trabajo en equipos pequeños y débilmente acoplados.

Una aplicación de alto tráfico con varios módulos que consumen mucho tiempo también es un buen candidato para una arquitectura de microservices, ya que necesita optimizaciones de rendimiento a nivel de módulo.

Las aplicaciones de bajo tráfico que requieren solo un equipo pequeño de menos de cinco personas para su implementación no son un buen candidato para una arquitectura de microservices.

Decidir cuándo adoptar microservices en todas las demás situaciones que caen entre los casos extremos anteriores no es fácil. En general, requiere un análisis detallado de costos y beneficios.

En cuanto a los costos, usar una arquitectura de microservices requiere un esfuerzo de desarrollo de aproximadamente cinco veces el de una aplicación monolítica usual. Obtuvimos esta escala como un promedio de 7 reescrituras totales de aplicaciones monolíticas con una arquitectura de microservices.

Esto se debe en parte al esfuerzo adicional necesario para manejar comunicaciones confiables, coordinación y gestión detallada de recursos. Sin embargo, la mayor parte de los costos proviene de las dificultades de probar, depurar y monitorear una aplicación distribuida.

Más adelante en el libro, describiremos herramientas y metodologías para manejar eficazmente todos los problemas anteriores, pero el costo adicional que conllevan los microservices permanece.

En cuanto a los beneficios esperados, la ventaja más significativa es la capacidad de enfocar el mantenimiento solo en los módulos críticos, ya que si la interfaz de un microservice no cambia, incluso los cambios más drásticos en su implementación, como migrar a un sistema operativo diferente, a un stack de desarrollo diferente, o simplemente a una versión más nueva del mismo stack, no requerirán ningún cambio en todos los demás microservices.

Podemos decidir reducir al mínimo el mantenimiento de los módulos que no requieren varios cambios críticos para el mercado, mientras nos enfocamos solo en los módulos críticos para el mercado que aumentan el valor percibido de la aplicación o requieren cambios para adaptarse a un mercado en rápida evolución. En resumen, podemos enfocarnos solo en los cambios importantes requeridos por los usuarios, dejando sin modificar todos los módulos que no están involucrados en estos cambios.

Enfocarse en solo unos pocos módulos garantiza un bajo time to market, por lo que podemos satisfacer las oportunidades del mercado tan pronto como aparecen sin el riesgo de lanzar una nueva versión demasiado tarde.

También podemos ajustar el rendimiento rápidamente cuando el tráfico en algunas funcionalidades específicas aumenta, escalando solo los microservices involucrados. Vale la pena señalar que la capacidad de ajustar cada bloque de construcción específico de nuestra aplicación permite un mejor uso del hardware disponible, reduciendo así los costos generales de hardware. Además, la capacidad de ajustar y monitorear microservices específicos simplifica el logro de mejores tiempos de respuesta y, en general, los objetivos de rendimiento.

Habiendo analizado la evolución que llevó a la arquitectura de microservices, así como su naturaleza y organización básica, podemos pasar a los patrones que, aunque no son específicos de los microservices, son comunes en las arquitecturas de microservices.

Patrones comunes de microservices

Ejecución resiliente de tareas

Los microservices pueden ser movidos de una máquina a otra para lograr un mejor balance de carga. También pueden ser reiniciados para restablecer posibles fugas de memoria o para resolver otros problemas de rendimiento. Durante estas operaciones, pueden perder algunos mensajes enviados a ellos, o pueden abortar algún cómputo en curso. Además, pueden ocurrir fallos debido a bugs de software o fallos de hardware.

Dado que las arquitecturas de microservices deben ser confiables (tiempo de inactividad casi nulo), generalmente son redundantes, y se necesita particular cuidado para detectar fallos y aplicar acciones correctivas. Por lo tanto, todas las arquitecturas de microservices deben proporcionar mecanismos tanto para detectar fallos, como simples timeouts, como para corregir operaciones fallidas.

Los fallos se detectan mediante la detección de excepciones inesperadas o timeouts. Dado que el código siempre puede organizarse de manera que convierta los timeouts en excepciones, la detección de fallos siempre puede reducirse a un manejo adecuado de excepciones.

Para resolver este problema, la comunidad de desarrolladores de microservices definió útiles **políticas de reintento** que se pueden adjuntar a excepciones específicas. Generalmente se implementan a través de bibliotecas específicas junto con otros patrones de confiabilidad, pero a veces son ofrecidas de forma nativa por los proveedores de nube.

A continuación se presentan los patrones estándar de confiabilidad utilizados en arquitecturas de microservices:

- **Reintento exponencial**: Ha sido diseñado para superar fallos temporales, como un fallo debido al reinicio de una instancia de microservice. Después de cada fallo, la operación se reintenta con un retraso que aumenta exponencialmente con el número de intentos, hasta que se alcanza un número máximo de intentos. Por ejemplo, primero reintentaríamos después de 10 milisegundos, y si este reintento resulta en un nuevo fallo, se hace un nuevo intento después de 20 milisegundos, luego después de 40 milisegundos, y así sucesivamente. Si se alcanza el número máximo de intentos, se lanza una excepción, donde puede encontrar otra política de reintento o alguna otra estrategia de manejo de excepciones.
- **Circuit break** (corte de circuito): Ha sido diseñado para manejar fallos a largo plazo y generalmente se activa después de que un reintento exponencial alcanza su número máximo de reintentos. Cuando se asume un fallo a largo plazo, el acceso al recurso se prohíbe durante un período de tiempo fijo devolviendo una excepción inmediata sin intentar todas las operaciones requeridas. El tiempo de prohibición debe ser suficiente para permitir la intervención humana o cualquier otro tipo de corrección manual.
- **Aislamiento de barreras** (bulkhead isolation): Ha sido diseñado para prevenir la propagación de fallos y congestión. La idea básica es organizar los servicios y/o recursos en particiones aisladas de modo que los fallos o congestiones que se originen en una partición permanezcan confinados a esa partición, y el resto del sistema continúe funcionando correctamente.

Supongamos, por ejemplo, que varias réplicas de microservices usan la misma base de datos (como es común). Debido a un fallo, una réplica podría empezar a abrir demasiadas conexiones a la base de datos, congestionando también a todas las demás réplicas que necesitan acceder a la misma base de datos.

En este caso, reconocemos que las conexiones a la base de datos son recursos críticos que necesitan aislamiento de barreras. Así, calculamos el número máximo de conexiones que la base de datos puede manejar adecuadamente y las particionamos entre todas las réplicas, asignando, por ejemplo, un máximo de cinco conexiones simultáneas a cada réplica del microservice.

De esta manera, un fallo en una réplica no afecta el acceso adecuado de otras réplicas a la base de datos. Además, si la aplicación está correctamente organizada, las solicitudes que no se pueden atender debido a la réplica fallida eventualmente se reintentarán en una réplica que funcione correctamente, de modo que la aplicación general pueda continuar funcionando adecuadamente. En general, si quisiéramos particionar todas las solicitudes a un recurso compartido, podemos proceder de la siguiente manera:

1. Solo se permite un número máximo de solicitudes salientes simultáneas pendientes similares al recurso compartido; digamos 5, como en el ejemplo anterior de la base de datos. Esto es como poner un límite superior a la creación de hilos.

2. Las solicitudes que excedan el límite anterior se encolan.
3. Si se alcanza una longitud máxima de cola, cualquier solicitud adicional resulta en excepciones lanzadas para abortarlas.

Vale la pena señalar que el patrón de particionamiento y limitación de solicitudes mostrado anteriormente es una forma común de aplicar aislamiento de barreras, pero no es la única. Cualquier estrategia de partición más aislamiento puede clasificarse como aislamiento de barreras. Por ejemplo, se podrían dividir las réplicas de dos microservices que interactúan en dos particiones aisladas de modo que solo las réplicas pertenecientes a la misma partición puedan interactuar. De esta manera, un fallo en una partición no puede afectar a la otra partición.

Junto con las acciones y estrategias para manejar fallos expuestas anteriormente, las arquitecturas de microservices también ofrecen estrategias de prevención de fallos. La prevención de fallos se logra monitoreando consumos anómalos de recursos de hardware y realizando verificaciones periódicas de salud del hardware y software. Para este propósito, los orquestadores monitorean el uso de memoria y recursos de CPU y reinician una instancia de microservice o agregan una nueva instancia cuando caen fuera de un rango definido por el desarrollador. Además, ofrecen la posibilidad de declarar verificaciones periódicas de software que el orquestador puede realizar para verificar si el microservice está funcionando correctamente. La más común de tales verificaciones de salud es una llamada a un **endpoint REST de salud** expuesto por el microservice. De nuevo, si el microservice falla una verificación de salud, se reinicia.

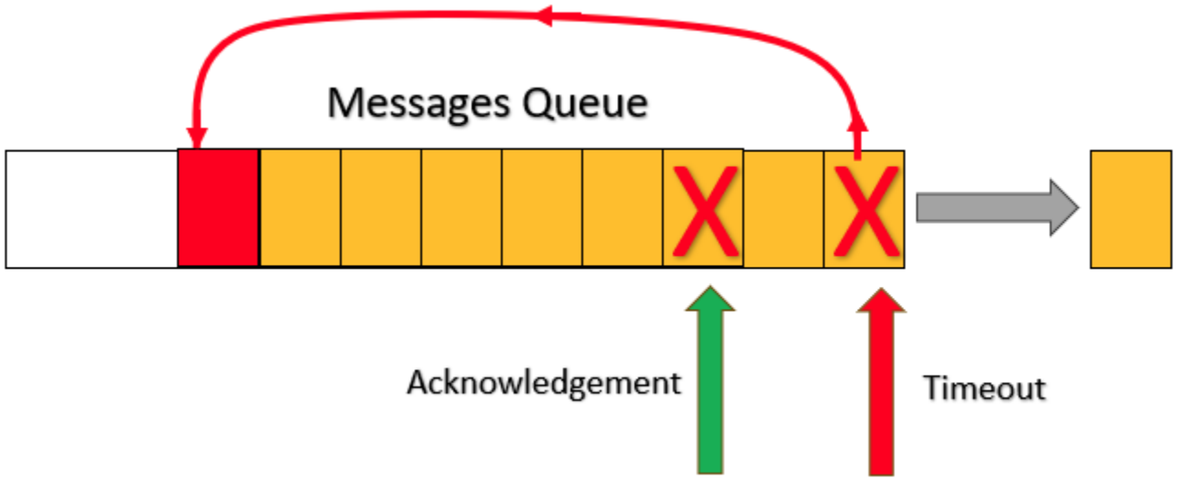
Cuando un nodo de hardware falla una verificación de salud, todos sus microservices se mueven a un nodo de hardware diferente.

Manejo eficaz de la comunicación asíncrona

La comunicación asíncrona con acuse de recibo asíncrono asociado causa tres problemas importantes:

1. Dado que después de la comunicación el microservice emisor pasa a atender otras solicitudes sin esperar el acuse de recibo, debe mantener una copia de todos los mensajes enviados hasta que se detecte un acuse de recibo o un fallo de comunicación, como un timeout, para poder reintentar la operación (con un reintento exponencial, por ejemplo), o tomar otro tipo de acción correctiva.
2. Dado que en caso de un timeout un mensaje puede ser reenviado, el destinatario previsto puede recibir varias copias del mismo mensaje.
3. Los mensajes pueden llegar a un destinatario en un orden diferente al que fueron enviados. Por ejemplo, si dos mensajes que instruyen al destinatario a modificar el nombre de un producto se envían en el orden M1, M2, esperamos que el nombre final sea el contenido en M2. Sin embargo, si el destinatario recibe los dos mensajes en el orden incorrecto, M2, M1, el nombre final del producto será el contenido en M1, causando así un error.

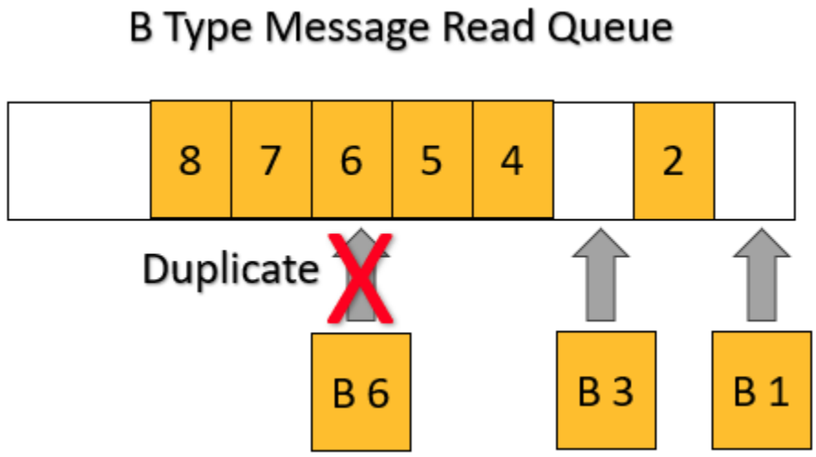
El primer problema se maneja manteniendo todos los mensajes en una cola, como se muestra en la siguiente figura:



2.8-Output message queue

Cuando se recibe un acuse de recibo, el mensaje involucrado se elimina de la cola. Si, por el contrario, se detecta un fallo o timeout, el mensaje se agrega al final de la cola para ser reintentado. Si un reintento debe manejarse con un reintento exponencial, cada entrada de la cola debe contener tanto el número del intento actual como el tiempo mínimo en el que el mensaje puede ser reenviado.

El segundo y tercer problemas requieren que cada mensaje recibido tenga un identificador único y un número de secuencia. El identificador único ayuda a reconocer y descartar duplicados, mientras que el número de secuencia ayuda al destinatario a reconstruir el orden correcto de los mensajes. La siguiente figura muestra una posible implementación.



2.9-Input message queue

Los mensajes solo pueden leerse de la cola de entrada después de que todos los huecos de secuencia antes de ellos hayan sido llenados y leídos, mientras que los duplicados se reconocen y descartan fácilmente.

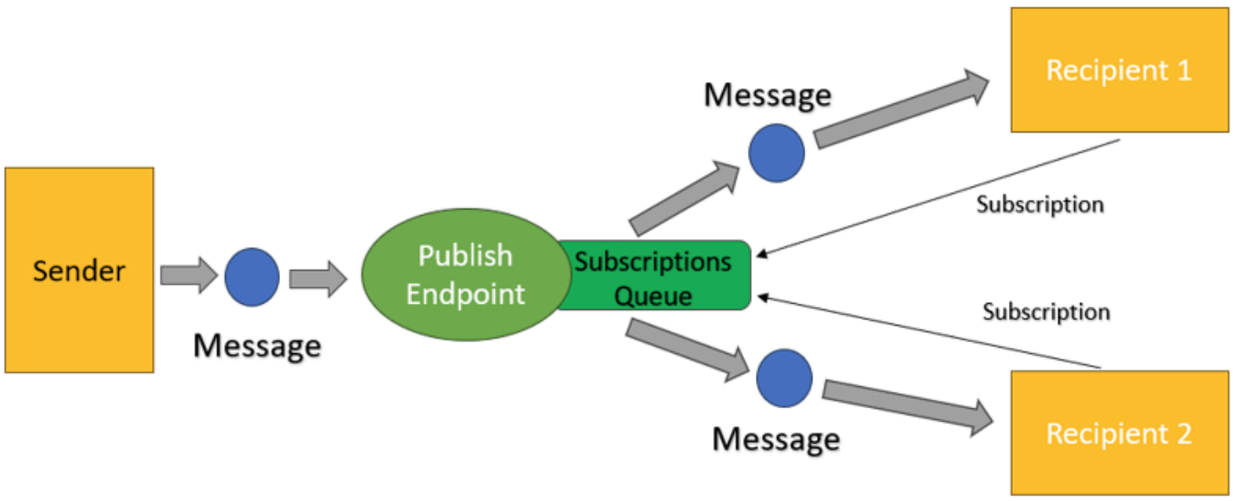
Comunicaciones basadas en eventos

Supongamos que añadimos un nuevo microservice a la aplicación de car-sharing de la *Figura 2.7*, digamos, un microservice worker que calcula estadísticas sobre los viajes de los usuarios. Nos veríamos obligados a modificar todos los microservices de los que necesita entrada, porque todos estos microservices también deben enviar algunos mensajes al microservice recién añadido.

La restricción principal de las arquitecturas de microservices es que las modificaciones a un microservice no deben propagarse a otros microservices, pero simplemente añadiendo un nuevo microservice, ya hemos violado este principio básico.

Para superar este problema, los mensajes que también podrían interesar a microservices recién añadidos se manejan con el patrón **publicador-suscriptor** (publisher-subscriber). Es

decir, el emisor envía el mensaje a un endpoint de publicación en lugar de enviarlo directamente a los destinatarios finales. Luego, cada microservice que está interesado en ese mensaje simplemente se suscribe a este endpoint, de modo que el endpoint de suscripción enviará automáticamente todos los mensajes que reciba. La siguiente figura muestra cómo funciona el patrón publicador-suscriptor.



2.10-Publisher-subscriber pattern

Una vez que un endpoint de publicación recibe un mensaje, lo reenvía a todos los suscriptores que se han añadido a su cola de suscripciones. De esta manera, si añadimos un nuevo microservice, no se requiere ninguna modificación para todos los emisores de mensajes, ya que solo necesitan continuar enviando sus mensajes a los endpoints de publicación adecuados. Es responsabilidad del microservice recién añadido registrarse en los endpoints de publicación apropiados.

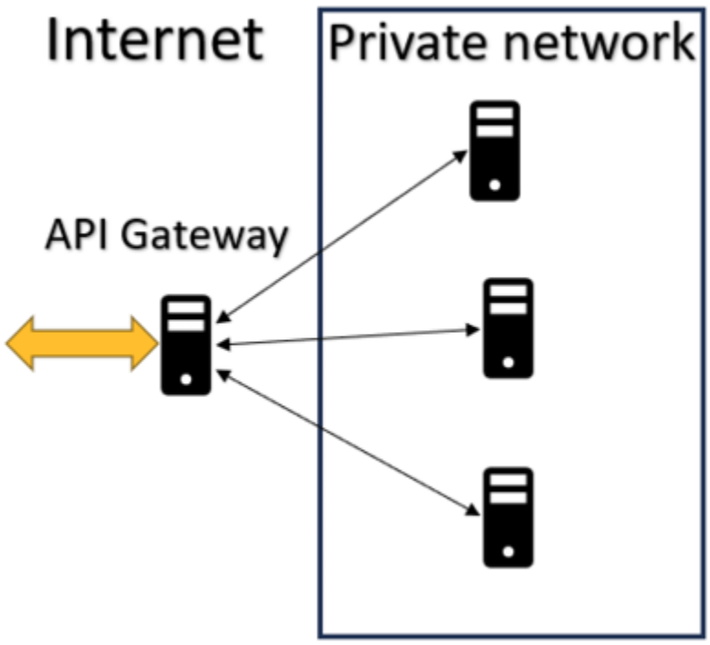
Los endpoints de publicación son manejados por aplicaciones llamadas message brokers que ofrecen este servicio junto con otros servicios de entrega de mensajes. Los message brokers pueden desplegarse como microservices replicables, pero típicamente son ofrecidos como servicios estándar por todos los principales proveedores de nube.

Entre ellos, vale la pena mencionar **RabbitMQ**, que debe instalarse como un microservice, y **Azure Service Bus**, que está disponible como servicio en la nube en Azure.

Interfaz con el mundo exterior

Las aplicaciones de microservices generalmente están confinadas a una red privada y exponen sus servicios a través de direcciones IP públicas o privadas mediante gateways, balanceadores de carga y servidores web. Estos componentes pueden enrutar direcciones externas a microservices internos. Sin embargo, es difícil dejar a la aplicación cliente del usuario la elección del microservice al que enviar cada una de sus solicitudes.

Típicamente, las solicitudes de entrada son todas manejadas por un endpoint único llamado **API gateway** que las analiza y traduce la solicitud a una solicitud apropiada para los microservices internos. De esta manera, la aplicación cliente del usuario no necesita ningún conocimiento de cómo está organizada internamente la aplicación de microservices. Por lo tanto, somos libres de cambiar la organización de la aplicación durante su mantenimiento sin afectar a los clientes que la usan, ya que las traducciones necesarias las realiza el API gateway de la aplicación. Este proceso se conoce como **traducción de interfaz de web API**.



2.11-API gateway

Los API gateways también pueden manejar el versionamiento de la aplicación enviando todas las solicitudes a los microservices que pertenecen a la versión requerida por la aplicación cliente.

Además, típicamente también manejan tokens de autenticación; es decir, tienen las claves para decodificarlos y verificar toda la información del usuario que contienen, como el ID del usuario y sus privilegios de acceso.

Por favor no confundas autenticación con login. El login se realiza una vez por sesión cuando el usuario comienza a interactuar con la aplicación, y lo realiza un microservice dedicado. El resultado de un login exitoso es un token de autenticación que codifica información sobre el usuario y que debe incluirse en todas las solicitudes subsiguientes.

En resumen, los API gateways ofrecen los siguientes servicios:

- Traducción de interfaz de web API
- Versionamiento
- Autenticación

Sin embargo, a menudo también ofrecen otros servicios, como:

- Endpoints de documentación de API, es decir, endpoints que ofrecen una descripción formal de los servicios ofrecidos por la aplicación y cómo solicitarlos. En el caso de comunicación REST, la documentación de API se basa en el estándar **OpenAPI** (ver *Further reading*).
- Caching, es decir, agregar las cabeceras HTTP correctas para manejar el almacenamiento en caché adecuado de todas las respuestas tanto en el cliente del usuario como en los nodos intermedios de la web.

Vale la pena señalar que los servicios anteriores son solo ejemplos comunes de los servicios disponibles en API gateways comerciales o de código abierto que generalmente ofrecen una amplia gama de servicios.

Los API gateways pueden implementarse como microservices ad hoc usando bibliotecas como YARP (https://microsoft.github.io/reverse-proxy/index.html), o pueden usar aplicaciones configurables preexistentes, por ejemplo, el proyecto de código abierto Ocelot (https://github.com/ThreeMammals/Ocelot). Todos los principales proveedores ofrecen potentes API gateways configurables, llamados **sistemas de gestión de API** (para Azure, ver

https://azure.microsoft.com/en-us/products/api-management). Sin embargo, también existen ofertas independientes cloud-native, como Kong (https://docs.konghq.com/gateway/latest/).

Preguntas

1. ¿Cuál es la principal diferencia entre una SOA tradicional y una arquitectura moderna de microservices?
   Las arquitecturas de microservices son de grano fino. Además, cada microservice no debe depender de las decisiones de diseño de otros microservices. Adicionalmente, los microservices deben ser redundantes, replicables y resilientes.
2. ¿Por qué son tan importantes los equipos débilmente acoplados?
   Porque es bastante fácil coordinar equipos débilmente acoplados.
3. ¿Por qué cada microservice lógico debe tener almacenamiento dedicado?
   Es una consecuencia inmediata de la independencia de las decisiones de diseño de un microservice respecto a las decisiones de diseño adoptadas en todos los demás microservices. De hecho, compartir una base de datos común forzaría decisiones de diseño comunes sobre la estructura de la base de datos.
4. ¿Por qué se necesita la comunicación basada en datos?
   Es la única manera de evitar largas cadenas de solicitudes y respuestas recursivas que causarían tiempos de respuesta generales inaceptables.
5. ¿Por qué es tan importante la comunicación basada en eventos?
   Porque la comunicación basada en eventos desacopla completamente los microservices, de modo que los desarrolladores pueden añadir un nuevo microservice sin modificar ninguno de los microservices preexistentes.
6. ¿Los API gateways generalmente ofrecen servicios de login?
   No, los servicios de login son ofrecidos por microservices específicos llamados servidores de autenticación.
7. ¿Qué es el reintento exponencial?
   Una política de reintento que aumenta exponencialmente el retraso entre fallos y reintentos después de cada fallo.

Relacionado

- 00-Practical serverless and microservices with csharp