¿Son los microservicios la mejor opción? Una evaluación de su eficacia y eficiencia frente a los monolitos

Gonzalez Rodrigo Alejandro, Giménez Sergio Agustín ,Molina Pualuk, Numa Roy, Rodrigo Zalazar[1]

[1]Dpto. Informática, Fac. de Ciencias Exactas Naturales y Agrimensura, Universidad Nacional del Nordeste, Corrientes

gonzlrodrigo@gmail.com, agustin029g@gmail.com, numamolina19@gmail.com, rodrigo.zalazar@comunidad.unne.edu.ar

Abstract: La arquitectura de microservicios ha surgido como una alternativa para el desarrollo de aplicaciones tradicionales basadas en Monolitos, ofreciendo un enfoque altamente modular y escalable. Sin embargo, su implementación y gestión pueden ser complejas sin las herramientas adecuadas.
Como parte de un proyecto colaborativo en el marco de un curso sobre "Arquitectura de Microservicios" en la Licenciatura en Sistemas de Información, exploramos el diseño e implementación de estos. Nuestro objetivo es evaluar su eficacia, y eficiencia como una alternativa a los monolitos.

Keywords: Microservicios, Desarrollo de Aplicaciones, Escalabilidad, Gestión de Servicios, Monolito.

1 Introducción

1.1 Aplicaciones monolíticas

Una aplicación monolítica es una aplicación de software de un solo nivel en la que se combinan diferentes módulos en un solo programa. Por ejemplo, si se compila una aplicación de comercio electrónico, se espera que la aplicación tenga una arquitectura modular alineada con los principios de programación orientada a objetos (OOP). En una aplicación monolítica, los módulos se definen mediante una combinación de construcciones del lenguaje de programación (como los paquetes de Java) y artefactos de compilación (como los archivos JAR de Java).[2]

Beneficios de la aplicación monolítica

La arquitectura monolítica es una solución convencional para compilar aplicaciones. Las siguientes son algunas ventajas de adoptar un diseño monolítico para una aplicación:

- Se puede implementar pruebas de extremo a extremo de una aplicación monolítica mediante herramientas como Selenium.
- Para implementar una aplicación monolítica, simplemente se puede copiar la aplicación empaquetada en un servidor.
- Todos los módulos de una aplicación monolítica comparten memoria, espacio y recursos, de modo que se puede usar una sola solución para abordar problemas cruzados, como el registro, el almacenamiento en caché y la seguridad.
- El enfoque monolítico puede proporcionar ventajas de rendimiento, ya que los módulos pueden llamarse entre sí directamente. Por el contrario, los microservicios suelen requerir una llamada de red para comunicarse entre sí.[3]

Desafíos de la aplicación monolítica

En el contexto de aplicaciones monolíticas, las complejidades se intensifican a medida que crecen en tamaño y complejidad, llegando a un punto donde los desafíos superan a los beneficios. La coordinación extensa de cambios y la dificultad para implementar ajustes en sistemas extensos y complejos son evidentes. La integración continua y la implementación continua (CI/CD) se vuelven desafiantes, requiriendo la reimplementación completa y pruebas manuales extensas. La escalabilidad plantea problemas al asignar recursos a módulos conflictivos, y el riesgo de fallos totales debido a errores en un módulo es inherente. Además, la introducción de nuevas tecnologías implica a menudo la reescritura completa de la aplicación, siendo una tarea costosa en tiempo y recursos financieros.[4]

1.2 Aplicaciones basadas en microservicios

Los microservicios son un enfoque arquitectónico y organizativo para el desarrollo de software donde el software está compuesto por pequeños servicios independientes que se comunican a través de API bien definidas. Los propietarios de estos servicios son equipos pequeños independientes.[1]

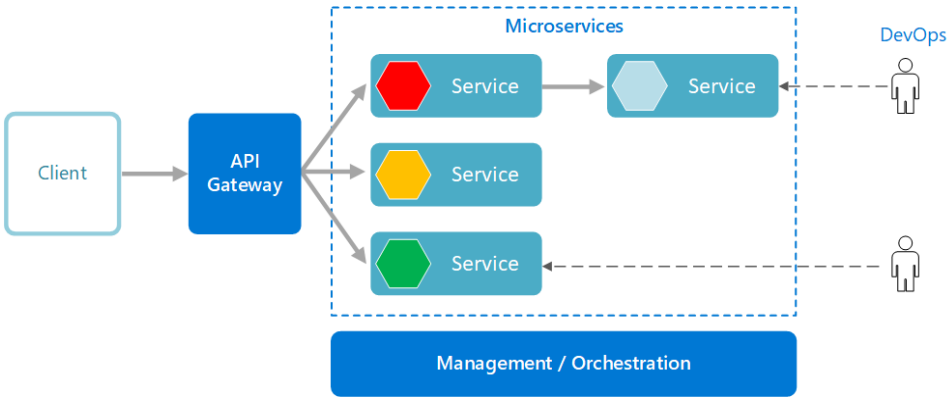


Figura.1 [5] Diseño de un Microservicio estándar

Beneficios de los microservicios

La adopción de la arquitectura de microservicios ofrece ventajas clave al abordar desafíos inherentes a las aplicaciones monolíticas. Al dividir la aplicación en servicios gestionables con límites bien definidos, se facilita el desarrollo y mantenimiento, permitiendo que equipos autónomos trabajan de manera independiente. Esta modularidad posibilita el uso de diferentes lenguajes de programación para cada microservicio, optimizando la aplicación en términos de velocidad, funcionalidad y mantenibilidad. Además, la capacidad de implementar y lanzar servicios de forma independiente mejora la velocidad y la escalabilidad del sistema.

La migración a microservicios resulta beneficiosa para mejorar la escalabilidad, agilidad y velocidad de entrega. También se aplica en la modernización gradual de aplicaciones heredadas y en la extracción de servicios empresariales para su reutilización en distintos canales, contribuyendo a una implementación más eficiente y coherente de funcionalidades comunes.

Desafíos de los microservicios

La implementación de microservicios, a pesar de sus beneficios, presenta desafíos que requieren una consideración cuidadosa en la arquitectura de aplicaciones. La complejidad de un sistema distribuido, con la necesidad de elegir y aplicar mecanismos de comunicación entre servicios, gestionar fallas y abordar la falta de disponibilidad, destaca como un desafío significativo. La gestión de transacciones distribuidas también surge

como una complejidad, ya que las operaciones empresariales distribuidas pueden resultar en la coordinación y reversión complicada de estados en caso de error, potencialmente conduciendo a datos incoherentes.

Las pruebas integrales de aplicaciones basadas en microservicios son más complejas debido a la interacción entre servicios para completar flujos de prueba. La implementación y gestión de una aplicación de microservicios se complica por la multiplicidad de servicios, instancias de entorno y la necesidad de un mecanismo de descubrimiento de servicios. La sobrecarga operativa aumenta al supervisar y gestionar múltiples servicios, generando alertas y enfrentándose a más puntos de falla en la comunicación servicio a servicio. Aunque la arquitectura permite funciones simultáneas, la latencia se incrementa debido a las llamadas de red entre servicios. Es crucial reconocer que no todas las aplicaciones son adecuadas para la descomposición en microservicios, especialmente aquellas que requieren integración estrecha o procesamiento en tiempo real.[8]

Características de los Microservicios

- Los microservicios son pequeños e independientes, y están acoplados de forma imprecisa. Un único equipo reducido de programadores puede escribir y mantener un servicio.
- Cada servicio es un código base independiente, que puede administrarse por un equipo de desarrollo pequeño.
- Los servicios pueden implementarse de manera independiente. Un equipo puede actualizar un servicio existente sin tener que volver a generar e implementar toda la aplicación.
- Los servicios son los responsables de conservar sus propios datos o estado externo. Esto difiere del modelo tradicional, donde una capa de datos independiente controla la persistencia de los datos.
- Los servicios se comunican entre sí mediante API bien definidas. Los detalles de la implementación interna de cada servicio se ocultan frente a otros servicios.
- Admite la programación políglota. Por ejemplo, no es necesario que los servicios compartan la misma pila de tecnología, las bibliotecas o los marcos.[5]

Además de los propios servicios, hay otros componentes que aparecen en una arquitectura típica de microservicios[4]:

Administración e implementación. Este componente es el responsable de la colocación de servicios en los nodos, la identificación de errores, el reequilibrio de servicios entre nodos, etc. Normalmente, este componente es una tecnología estándar, como Kubernetes, en lugar de algo creado de forma personalizada.

Puerta de enlace de API(API Gateway): La puerta de enlace de API es el punto de entrada para los clientes. En lugar de llamar a los servicios directamente, los clientes llaman a la puerta de enlace de API, que reenvía la llamada a los servicios apropiados en el back-end.

Definiciones:

Integración continua (CI): Práctica de desarrollo de software que consiste en integrar y verificar el código fuente de manera automática y frecuente, para detectar errores y conflictos de manera temprana.

Implementación continua (CD): Práctica de desarrollo de software que consiste en automatizar el proceso de despliegue y entrega de software, para que los cambios se puedan implementar de manera rápida y segura.

Escalabilidad: Capacidad de un sistema para manejar un aumento en la carga de trabajo o el número de usuarios sin degradar su rendimiento o funcionalidad.

Cohesión: Grado en que los elementos de un módulo o componente están relacionados y trabajan juntos para lograr un objetivo común.

Acoplamiento: Grado en que los elementos de un módulo o componente dependen entre sí. Un acoplamiento alto significa que los cambios en un elemento pueden afectar a otros elementos.

API: Interfaz de programación de aplicaciones, conjunto de reglas y protocolos que permiten a los programas comunicarse entre sí.

API REST: Interfaz de programación de aplicaciones basada en el protocolo HTTP y los principios de la arquitectura REST (Representational State Transfer). Las API REST permiten a los clientes acceder y manipular recursos en un servidor a través de operaciones HTTP estándar, como GET, POST, PUT y DELETE. Las respuestas de la API REST suelen estar en formato JSON o XML.

Contenedores (Docker): Los contenedores son entornos aislados y portátiles que contienen todo lo necesario para que una aplicación se ejecute, incluyendo el código, las bibliotecas y las dependencias. Docker facilita la creación y gestión de entornos de desarrollo y producción, y permite una mayor flexibilidad y escalabilidad en el despliegue de aplicaciones.

Imagen en un contenedor: Plantilla de solo lectura que contiene todo lo necesario para ejecutar una aplicación en un contenedor de Docker. Las imágenes se crean a partir de un archivo de configuración llamado Dockerfile, que especifica las dependencias, bibliotecas y comandos necesarios para construir la imagen. Al crear un contenedor a partir de una imagen, se crea una instancia en ejecución de la aplicación con su propio sistema de archivos y recursos aislados.

2 Instrumentación

Para el despliegue del estudio en cuestión se dispuso de los siguientes paquetes de software y hardware:

- Pc de escritorio con microprocesador Ryzen 5 2600 6 núcleos 12 hilos a 3.4 GHz, 16 GB de ram DDR4 a 2400 Mhz, Caché L1 total 576 KB Caché L2 total 3MB Caché L3 total 16MB
- Fedora 38 Workstation
- Docker
- PostgreSQL
- Apache Jmeter
- Golang 1.20 o superior
- Partición con formato Btrfs
- Grafana
- Prometheus

3 Desarrollo

Diseño:

Para llevar a cabo nuestra investigación, nos centraremos en un monolito que se compone de una API REST construida en golang y usa como base de datos PostgreSQL, diseñada para la gestión de tareas o pendientes . A pesar de que este proyecto es relativamente sencillo, es lo suficientemente complejo como para demostrar la mayoría de los puntos que distinguen un monolito de un microservicio. Primero debemos entender que funciones cumplen que son de manera básica un crud de usuarios que se puede acceder desde el endpoint /users y uno de tareas accesibles desde el endpoint /taks y cada módulo con su tabla respectiva en la base datos

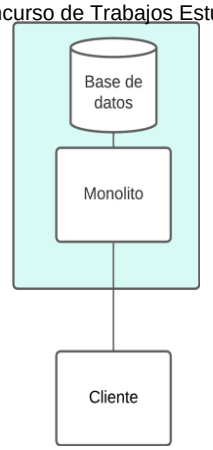


Figura.2

Dado que los microservicios representan una arquitectura de software, es fundamental descomponer nuestro monolito en sus componentes más pequeños y relativamente independientes.
Para eso vamos a seguir los siguientes pasos[9]:

1. Es necesario identificar las partes del código (dominios) que están altamente cohesionadas y bajo acopladas con el resto, puesto que son susceptibles de ser separadas a un microservicio aparte.
2. Separar los dominios en distintos proyectos. Se pueden utilizar lenguajes de programación distintos para cada proyecto, dependiendo de las necesidades de cada dominio. Nuestro consejo es que se trate de homogeneizar, a no ser que haya casos en los que de verdad merezca la pena implementar un dominio en un lenguaje de programación específico.
3. Obtener un grafo de las dependencias existentes en el sistema legado monolítico entre los dominios identificados. Se puede utilizar alguna herramienta que analice el proyecto.
4. A partir de las dependencias, diseñar una interfaz (diseño API first) para cada uno de los dominios. Estas APIs (pueden ser REST, GraphQL, gRPC…) serán la nueva puerta de entrada a cada uno de los mismos. Esto significa que si el dominio de tiendas necesita algo del dominio de productos, deberá utilizar alguna de las operaciones que ofrece el dominio de productos en su API.
5. El momento de migrar los datos a los nuevos esquemas de datos de cada uno de los diferentes microservicios. Este paso puede llegar a ser muy costoso, a la par que complicado.
6. Una vez todos y cada uno de los microservicios son sencillos, hacen lo que tienen que hacer (y no más) con la ayuda (o no) de una o varias BBDD, es hora de integrarlos. Esto es, hacer que interactúen entre ellos, convirtiendo las dependencias de módulos que había en el monolito en llamadas a sus APIs.
7. Se deberían implementar tests unitarios para cada uno de los microservicios (entendiendo el microservicio como unidad), además de tests de integración que comprueben que saben trabajar en conju
8. nto. Estos últimos deberían tener en cuenta las particularidades que tiene todo sistema distribuido: pérdida de mensajes, particiones de red, caída de los microservicios,etc.

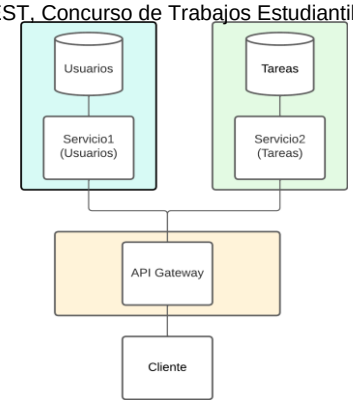


Figura.3

Siguiendo la guía previa para descomponer un monolito en microservicios, hemos obtenido un diseño que muestra claramente dos dominios distintos: uno enfocado en la gestión de tareas y otro en usuarios. En nuestro caso, hemos optado por mantener el lenguaje de programación Go (Golang) debido a su capacidad completa para construir cada servicio. Cambiar de lenguaje podría complicar las pruebas y comparaciones de rendimiento en el futuro.

No hemos identificado dependencias significativas entre los módulos, lo que nos ha permitido evitar la interconexión entre ellos. Para aumentar la independencia y evitar problemas de sincronización, dividimos la base de datos en dos partes. Posteriormente, empleamos un API Gateway para crear una interfaz común para todos los clientes, independientemente de los cambios en los microservicios, todo conectado a través de Api Rest.

Despliegue:

Monolito

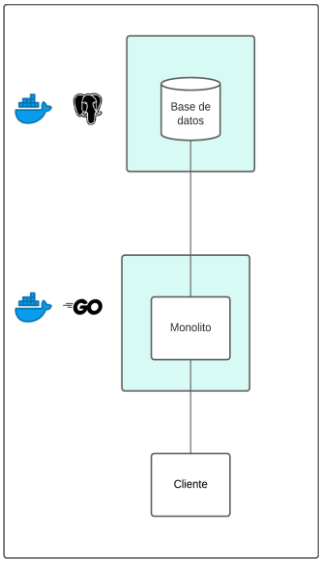


Figura.4

1. Montamos un contenedor con la imagen de postgres uno en el puerto 5432:5432.
2. Creamos la imagen de la aplicación en un contenedor en el puerto 3000:3000 verificando que se conecte correctamente con su base de datos respectiva.

Microservicio

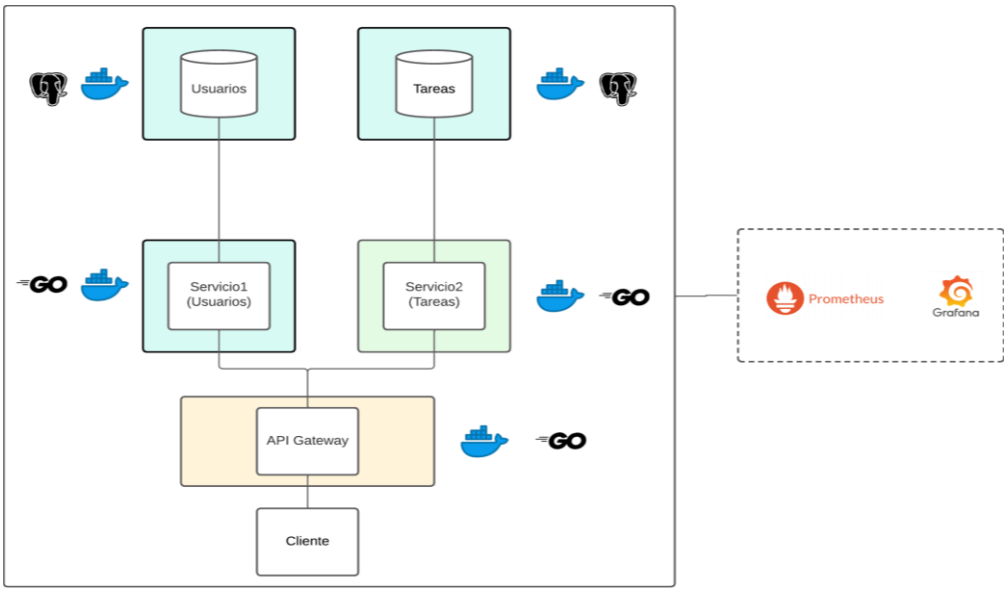


Figura.5

Para el despliegue de la aplicación seguimos los siguientes pasos:

1. Montamos dos contenedores con la imagen de postgres uno en el puerto 5432:5432 para la base de datos de tareas y otro en el puerto 5435:5432 para la base de datos de usuarios.
2. Creamos la imagen del microservicio de tareas y ejecutamos un contenedor con el puerto 4000:4000 verificando que se conecte correctamente con su base datos respectiva, y hacemos lo mismo con el microservicio de usuarios pero en el puerto 5000:5000.
3. Creamos una imagen del api gateway y lo hacemos correr el puerto 8080:8080.
4. Creamos dos postgre exported cada uno en un contenedor y enlazamos con a cada uno con su base de datos, con los puertos con los puertos 9187:9187 y 9188:9187

5. Generamos un archivo prometheus.yml donde guardamos todas las fuentes de métricas entre estas los dos progress exported y el api gateway
6. Corremos un contenedor docker con con prometheus usando la configuración anteriormente en el puerto 9090:9090

Las fuentes, código y ejemplos del montaje de los sistemas se encuentra en estos dos repositorios:
https://github.com/RodrigoGonzalez78/microservicios_task_managment
https://github.com/RodrigoGonzalez78/tasks_management_backend.git

4 Comparativa

Para la comparativa decidimos medir la el rendimiento de ambas arquitectura bajo el mismo sistema utilizando la herramienta jmeter y generando una prueba de estrés que se compone de cuatro pruebas de 30 segundos cada uno donde luego se promedia automáticamente por la herramienta, ejemplo pruebas de 2000 para /users seria 4 pruebas de 2000 peticiones en 30 segundos todo esto para evitar anomalías e irregularidades.

Monolito	Peticiones	servicio	Tiempo de respuesta máximo	Tiempo de Respuesta promedio	Tasa de error	Peticiones por segundo
	10000	/users	1533	105	8.67%	216
		/task	1535	116	8.02%	216
	5000	/users	2005	104	2.44%	141.9
		/task	2012	116	2.46%	141.8
	2000	/users	1730	131	0.62%	53.4
		/task	1750	123	0.60%	53.4

Tabla.1

Microservicios sin api gateway	Peticiones	servicio	Tiempo de respuesta máximo	Tiempo de Respuesta promedio	Tasa de error	Peticiones por segundo
	10000	/users	17022	469	14.41%	222.4
		/task	37129	1558	18.96%	221.4
	5000	/users	2519	215	4.53%	92
		/task	2561	212	3.75%	92.1
	2000	/users	2896	261	2.39%	62.1
		/task	2786	210	0.30%	62.1

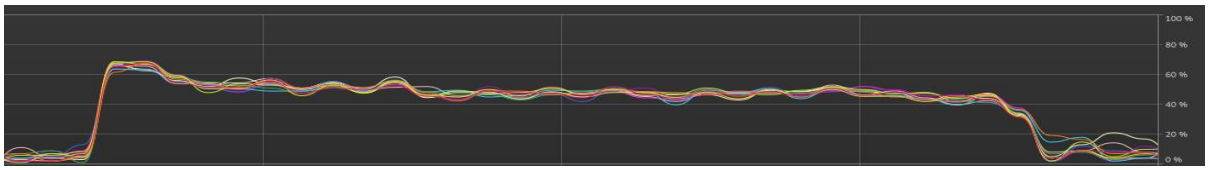
Tabla.2

Microservicio con Api Gateway	Peticiones	servicio	Tiempo de respuesta máximo	Tiempo de Respuesta promedio	Tasa de error	Peticiones por segundo
	10000	/users	11816	667	20%	241
		/task	41305	3773	13.25%	228
	5000	/users	2010	212	4.05%	92.1
		/task	1916	202	3.19%	92.1
	2000	/users	1242	204	0.0%	59
		/task	1417	188	0.0%	59.1

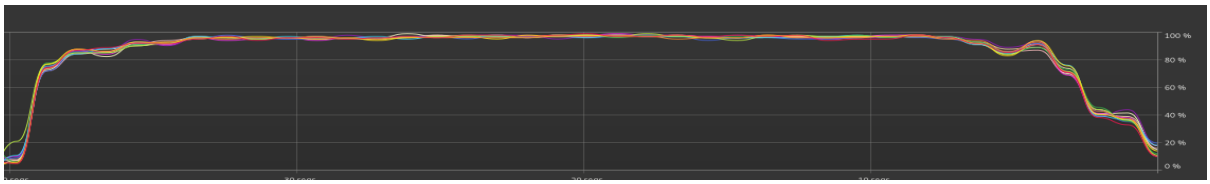
Tabla.3

Consumo de CPU de cada arquitectura:

Monolito:



Microservicio:



Usamos como referencia el consumo de cpu como es el recurso donde se nota una mayor diferencia entre ambas arquitecturas, en esta comparación podemos ver un aumento significativo al usar una arquitectura de microservicio pasando de un 40-65% a un 100% igualmente esto puedo variar dependiendo que funcion cumpla nuestro sistema y qué tecnologías usamos.

5 Conclusiones y posibles mejoras

La comparativa de rendimiento entre el monolito y los microservicios utilizando JMeter revela diferencias notables. El monolito exhibe tiempos de respuesta más rápidos, una menor tasa de error y es capaz de manejar un mayor número de peticiones por segundo.

En contraste, tanto el "Microservicio sin API Gateway" como el "Microservicio con API Gateway" muestran tiempos de respuesta más prolongados y tasas de error comparativamente más elevadas.

Estos resultados apuntan a que, en este contexto particular, el monolito supera en rendimiento a los microservicios. Sin embargo, es interesante destacar que la presencia de un API Gateway parece influir positivamente en la eficiencia del microservicio.

6 Referencias

[1] https://aws.amazon.com/es/microservices/
[2] https://cloud.google.com/architecture/microservices-architecture-introduction?hl=es-419
[3] Introducción a los microservicios | Cloud Architecture Center
[4] https://cloud.google.com/architecture/microservices-architecture-introduction?hl=es-419#monolith_challenges
[5] https://learn.microsoft.com/es-es/azure/architecture/guide/architecture-styles/microservices
[6] https://www.atlassian.com/es/microservices/microservices-architecture/building-microservices
[7] Beneficios de los microservicios, https://cloud.google.com/architecture/microservices-architecture-introduction?hl=es-419#microservices-benefits
[8] https://cloud.google.com/architecture/microservices-architecture-introduction?hl=es-419#microservices_challenges
[9] https://www.hiberus.com/crecemos-contigo/migracion-microservicios/