CACiC 2023
CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN

Implementando estrategias de resiliencia en una arquitectura basada en microservicios

Sergio Leonel Suárez[1,2], Enzo Rucci[3][0000-0001-6736-7358],
Victor Betran[2], and Diego Montezanti[3][0000-0003-0436-0997]

1 Facultad de Informática, UNLP. La Plata (1900), Bs As, Argentina
[2] PedidosYa
{sergio.suarez,victor.betran}@pedidosya.com
3 III-LIDI, Facultad de Informática, UNLP – CIC. La Plata (1900), Bs As, Argentina
{erucci,dmontezanti}@lidi.info.unlp.edu.ar

Resumen En los últimos años, se ha incrementado la adopción de arquitecturas de microservicios para ayudar a superar las limitaciones de los sistemas monolíticos. En los sistemas basados en microservicios, la resiliencia es crucial debido al impacto directo de las fallas en el negocio de una empresa. Debido a esto, han surgido los patrones de diseño para resiliencia como estrategias para gestionar las fallas y mitigar sus efectos negativos. En este trabajo se analiza el comportamiento de algunos patrones para resiliencia dentro del ecosistema de microservicios de PedidosYa, centrándose en el microservicio *Niles*. Para ello, se estudiaron diversos escenarios de fallas y la aplicación de patrones para darles solución. Además, se obtuvieron resultados experimentales que permitieron evaluar el impacto de la aplicación de estos patrones, evidenciando así que *Niles* ha logrado convertirse en uno de los componentes más robustos y utilizados en PedidosYa.

Keywords: Arquitecturas de Microservicios · Resiliencia · Sistemas Monolíticos · Fallos · Patrones de diseño · Sistemas distribuidos

1. Introducción

El término *Arquitectura de microservicios* ha surgido en los últimos años para describir una manera particular de diseñar aplicaciones como conjuntos de servicios desplegables de forma independiente, de manera de dar respuesta a las limitaciones de los sistemas monolíticos [6]. Si bien no existe una definición precisa de este estilo arquitectónico, existen características comunes en torno a la organización, la capacidad empresarial, la implementación automatizada y el control descentralizado de lenguajes y datos [1].

Desafortunadamente, estas arquitecturas también presentan una serie de desventajas. Por ejemplo, cuando ocurre un fallo, resultan afectados tanto el microservicio que lo experimenta como los otros que dependen de él. Otras desventajas son: la complejidad de los sistemas distribuidos, la ausencia de un método específico para la descomposición en microservicios de un sistema de grandes dimensiones, la necesidad de mantener la consistencia, la dificultad en la monitorización del estado de los componentes del ecosistema y la gestión de la seguridad.

La resiliencia es uno de los aspectos no funcionales más buscados, especialmente en los sistemas de gran escala. El manejo de las fallas es una cuestión fundamental, ya

que su impacto tiene relación directa con el negocio de una empresa, traduciendo en pérdidas económicas el tiempo de no disponibilidad de los servicios [3]. Es por ello que entran en juego los patrones de diseño para resiliencia entre microservicios, que son estrategias que ayuden a lidiar con estas fallas y mitigar sus consecuencias negativas.

Este trabajo realiza un análisis de la resiliencia en el ecosistema de microservicios de PedidosYa, una compañía de *delivery* en línea que opera en más de 10 países de América Latina. Esta empresa cuenta con una arquitectura basada mayormente en microservicios, la cual procesa aproximadamente 4 millones de órdenes por semana. El análisis propuesto contempla la aplicación de algunos patrones de resiliencia y las mejoras derivadas de ello, en cuanto a conversiones de negocio y otras métricas.

El resto del artículo se organiza de la siguiente forma. La Sección 2 brinda el marco referencial para el trabajo. La Sección 3 describe los patrones utilizados. La Sección 4 muestra algunos resultados experimentales. Por último, la Sección 5 resume las conclusiones y posibles líneas de trabajo futuro.

2. Marco Referencial

2.1 Fallos en arquitecturas de microservicios

En una arquitectura basada en microservicios existen situaciones que pueden causar que el sistema adquiera un comportamiento inestable. Si bien algunas son comunes a todos los sistemas de software (como una base de datos inconsistente o un recurso agotado), existen casos que son propios de los sistemas distribuidos [5], como son:

- Errores o lentitud en la red: debido a que la interacción entre múltiples servicios es muy frecuente, ante un error de red, un microcorte, el daño de un componente (como el servidor de DNS), o la congestión, el ecosistema resulta afectado.
- Picos de tráfico: dado que cada microservicio es autónomo e independiente, es posible definir sus límites de operación mediante parámetros como la cantidad de requerimientos por minuto que es capaz de soportar. Si un servicio recibe más tráfico del esperado, es posible que se degrade, comenzando a funcionar con lentitud o de forma anómala.
- Priorización incorrecta: no todos los servicios dentro de una compañía tienen el mismo nivel de prioridad. Por ejemplo, un servicio que maneja transacciones monetarias tiene naturalmente una prioridad mayor que un servicio que se encarga de la administración de comentarios. Si las prioridades no están correctamente definidas, el fallo de cualquier servicio podría tener impacto similar. El manejo de prioridades y de errores debe ser definido individualmente sobre cada servicio, de manera de no penalizar a todo el ecosistema ante un fallo de cualquiera de sus elementos.

La resiliencia es una característica que debe ser concebida desde el propio diseño de la arquitectura mediante un correcto modelado de los microservicios. En este contexto, la aplicación de patrones ayuda a construir una arquitectura de microservicios resiliente frente a los fallos.

2.2 Arquitectura de microservicios en PedidosYa y caso de estudio: servicio *Niles*

En cuanto al negocio, la empresa se encuentra dividida en lo que se conocen como verticales, tales como Restaurantes, Farmacias, Mercados y Bebidas. En la vertical de Restaurantes, que es la que más ganancias genera a la empresa, toda la lógica se encuentra dispersa en microservicios, siendo *Niles* uno de los más importantes. *Niles* se encarga de proporcionar al usuario el menú de un restaurante. Realiza múltiples consultas a otros microservicios con el fin de generar el menú, compuesto por secciones con productos. Además, aporta información básica para cada uno de ellos, como su nombre, descripción, imagen, precio y descuentos aplicables, y brinda datos adicionales tales como su popularidad de venta, si es un producto recomendado o si fue marcado como favorito.

Los módulos de visualización dentro de cada aplicación móvil, denominados *shopDetail*, invocan a *Niles* para consultar el menú. El proceso de retorno de un menú es el siguiente: (1) *Niles* recibe el requerimiento de menú para un restaurante; (2) verifica la existencia del menú para el restaurante solicitado en una caché distribuida; (3) en caso de que el menú no exista en la caché, se consultan diferentes servicios de modo de completar la información requerida. Una vez que se obtiene el menú completo, se persiste en la caché para próximas solicitudes; (4) en cualquiera de los casos anteriores, el menú se procesa aplicando diferentes filtros (validaciones de edad, stock disponible, ventanas de horario correctas) y ordenamientos. El microservicio cuenta con una caché centralizada, compartida por múltiples instancias de *Niles*, y una caché de tipo local para guardar información que rara vez cambia, como los países y las categorías de productos. Para poder presentar un menú actualizado, *Niles* recibe eventos de novedades desde otros microservicios tales como *Items-service* y *Battlefront*. Al momento de recibir una actualización, *Niles* elimina de la caché la entrada correspondiente a ese restaurante, el cuál será ingresado nuevamente cuando reciba un requerimiento desde las aplicaciones móviles. El flujo completo de comunicación entre todos los componentes se ilustra en la Figura 1.

2.3 Microservicios asociados

Para brindar su servicio de forma completa, *Niles* interactúa con varios microservicios, que se describen en detalle en [8]. Sin embargo, para realizar los experimentos que permitieron evaluar el impacto de la aplicación de los patrones, únicamente se afectaron las interacciones de *Niles* con dos de ellos, cuyas funcionalidades se describen a continuación:

- *Items-service*: es el microservicio que alimenta al menú. Se encarga de proveer las secciones y productos, así como también sus opciones de configuración. Para cada sección, producto u opciones de productos, administra datos como el nombre, imagen, descripción y precio. Ante cualquier cambio de los datos en un ítem, el servicio envía novedades (mediante tópicos) que llegan a las colas de mensajes de *Niles*, que es quien se encarga de actualizar la caché centralizada.

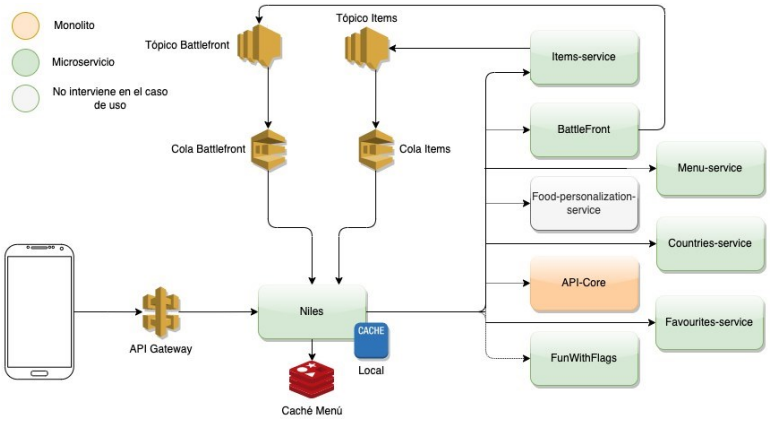

Figura1: Interacción de *Niles* con sus componentes asociados

En caso de que *Items-service* no funcione correctamente (debido a una caída temporal o incidente serio), *Niles* no podría retornar el menú. Si bien la información de *Items-service* se mantiene en una caché centralizada, la misma cuenta con un tiempo de vida limitado por lo que, ante una falla, podría mostrarse un menú desactualizado.

- *Favourites-service*: es el servicio que provee la información de todas las entidades que un usuario marca como sus preferidas. Maneja entidades como restaurantes, productos o configuraciones. *Niles* consume *Favourites-service* para poder agregar o remover productos a la lista de favoritos.

3. Tratamiento de fallos en *Niles*

Un microservicio se implementa como un servidor que se encarga de recibir los requerimientos entrantes, empleando un *pool* de hilos para poder atenderlos. El número de requerimientos que se pueden atender simultáneamente está limitado por la cantidad de hilos disponibles en el *pool*; éstos se reutilizan para nuevos requerimientos a medida que se liberan. Como los microservicios son autónomos y se encargan de una funcionalidad concreta, es común que deban consultar información desde otras entidades externas, como bases de datos, cachés u otros microservicios. Cada vez que un recurso externo es consultado, el *thread* que realiza la consulta queda bloqueado en espera, por lo que no puede atender nuevas peticiones.

3.1 El patrón *Timeout*

En cualquier momento, la entidad consultada puede degradar sus tiempos de respuesta, debido a errores, lentitud en la comunicación o problemas internos; esta situación puede ocurrir frecuentemente. En el caso de que no se establezca un límite de espera para el servicio que consume al recurso que presenta inconvenientes, el hilo que atiende el requerimiento puede quedar bloqueado indefinidamente. Esta situación eventualmente puede desembocar en que todos los *threads* queden bloqueados, y que

incluso se empiecen a encolar peticiones a la espera de que se liberen hilos. Una posible solución consiste en la implementación del patrón *Timeout*, que establece un tiempo de espera máximo al momento de consumir recursos externos; una vez que el lapso se cumple, el servicio consumidor corta la comunicación, liberando a los hilos involucrados para que puedan atender nuevos requerimientos [2]. Un aspecto destacable es que la definición de los tiempos de espera depende de cada microservicio, por lo que se requiere conocer sus indicadores de nivel de rendimiento; (o analizar métricas tales como latencia, *error rate* o tráfico recibido[1], si esta información no está disponible).

Para la entrega de un menú por parte de *Niles*, la petición a *Items-service* es la más importante, ya que sin esa información no es posible recuperar secciones y productos; en tanto, la petición a *Favourites-service* es de relevancia menor, ya que se puede presentar un menú completo sin la información sobre productos marcados como favoritos. Ambas peticiones se ejecutan en paralelo para optimizar recursos y retornar el menú en menos tiempo.

En una situación normal, la solicitud de un menú desde *Niles* se resuelve en 50 ms[2] (ya que *Items-service* responde en 50 ms y *Favourites-service* en 10 ms). Sin embargo, si *Favourites-service* se degrada (y empieza a responder en tiempos altos), ocurren dos problemas. El primero es que, como ambos servicios se ejecutan en paralelo, la respuesta final se produce junto con la del servicio con mayor latencia (en este caso *Favourites-service*), por lo que se penaliza la entrega de un menú debido una petición de escasa relevancia. El segundo (y más grave) es que se pueden bloquear la totalidad de hilos. *Favourites-service* continúa encolando peticiones en espera, por lo que los tiempos de respuesta aumentan aún más. Si se conoce el tiempo de respuesta típico de *Favourites-service* (en este caso, alrededor de 10ms), se puede configurar por ejemplo una espera máxima de unos 15ms. De esta forma, no se penaliza excesivamente el menú (simplemente se entrega sin la información de favoritos) ni tampoco se bloquean hilos a la espera de una respuesta que probablemente nunca llegue.

3.2 El patrón *Retry*

En las arquitecturas distribuidas en general, donde diferentes servicios se comunican constantemente, ocurren con frecuencia errores transitorios a causa de pérdidas momentáneas de conexión a la red o no-disponibilidad temporal de recursos. Normalmente, estos errores se manifiestan durante lapsos cortos de manera de que, si el recurso vuelve a ser consultado a la brevedad, responde correctamente. Una posible solución es la aplicación del patrón *Retry*, que consiste simplemente en reintentar una operación que ha fallado. Según el tipo de error detectado y/o el número de intentos, se pueden realizar diversas acciones. Por ejemplo, si el código del error retornado indica que es un fallo temporal o atípico, la operación puede reintentarse de inmediato, ya que probablemente no vuelva a producirse el mismo error. En tanto, si el error se debió a

[1] Estas métricas son provistas por herramientas que monitorean el desempeño de un servicio, conocidas como APM (*Application Performance Monitoring*)

[2] Este valor se obtiene con la métrica p95, que es la latencia medida en forma de percentil. Una métrica p95 con valor 50ms significa que el 95% de los requerimientos se resuelven en un tiempo menor o igual a 50ms

un problema de conexión o a un pico de peticiones al servicio, es prudente esperar un tiempo antes de reintentar la operación. Finalmente, si el código de error indica que el fallo no es temporal, la operación debería ser cancelada, reportándolo para gestionarlo de manera adecuada. Estas acciones pueden combinarse para crear una política de reintentos ajustada a las necesidades de la aplicación particular.

Debido a que *Niles* no puede generar el menú si *Items-service* (en particular) no está operativo (o fallan peticiones hacia él), existe una fuerte dependencia entre ellos que genera fragilidad en la funcionalidad de *Niles* ante inconvenientes en la red. La aplicación de *Retry* en la comunicación entre ambos servicios permite solucionar estos errores transitorios.

Cabe destacar que, si bien ante un fallo de *Items-service* el usuario podría actualizar la pantalla del móvil, reiniciar la aplicación o cerrar sesión y volver a intentar obtener el menú, existen casos en los que no es posible esa interacción (como procesos *batch*, actualizaciones masivas, liquidaciones de sueldo, etc.), para los cuales el patrón *Retry* es de suma importancia. Por otra parte, es importante configurar y administrar correctamente los reintentos, aumentándolos sólo para los recursos que son vitales para el caso particular. Por ejemplo, no tiene sentido aplicar reintentos para recuperar un producto favorito o una reseña, ya que se penaliza a la funcionalidad principal. Tampoco deberían realizarse reintentos si el código de respuesta HTTP indica que el servidor está sobrecargado (para no saturarlo); o en el caso de una excepción interna (como un envío incorrecto de parámetros) que sea responsabilidad del cliente que realiza la petición.

3.3 El patron Circuit Breaker

El patrón *Circuit Breaker* está inspirado en el funcionamiento de un disyuntor eléctrico, el cual es capaz de interrumpir la corriente en un circuito cuando ocurren determinadas fallas, con el objetivo de proteger a las personas. De manera más abstracta, impide la destrucción total de un sistema ante una falla que afecta a una parte de él. El disyuntor puede restablecerse para restaurar la funcionalidad una vez pasado el riesgo. En el ámbito del software, este concepto se aplica para proteger operaciones relevantes o críticas, mediante un interruptor automático que evita realizarlas cuando el sistema no está en buenas condiciones (en lugar de reintentar) [2] [4] [7].

En el estado normal (cerrado), el disyuntor ejecuta las operaciones como es habitual; si una llamada falla, el disyuntor lo registra, y una vez que la cantidad o la frecuencia de las fallas excede un umbral, el disyuntor se activa y abre el circuito. Cuando el circuito está abierto, las llamadas al disyuntor fallan inmediatamente, por lo que no se intenta ninguna operación. Después de un cierto tiempo, el *Circuit Breaker* decide que la operación tiene cierta probabilidad de éxito y pasa al estado semi-abierto; en él, la siguiente llamada al disyuntor intenta ejecutar nuevamente la operación crítica. Si tiene éxito, el disyuntor se restablece al estado cerrado; pero si la llamada de prueba falla, vuelve al estado abierto hasta que transcurre otro tiempo de espera.

El disyuntor degrada automáticamente la funcionalidad cuando el sistema está bajo estrés. Esto puede impactar en el negocio de la empresa, por lo que es fundamental el criterio para el manejo de las llamadas que se realizan cuando el circuito está abierto. Dependiendo de los casos de uso del sistema, cada disyuntor implementado puede estar

configurado para detectar diferentes tipos de fallas y manejar diferentes parámetros (como, por ejemplo, distintos umbrales). Al igual que con el patrón *Retry*, la configuración debe ser cuidadosa; por ejemplo, si el cliente envía parámetros incorrectos, el *Circuit Breaker* debería permanecer cerrado para no evitar llamadas que tienen que ejecutarse.

Tanto *Niles* como *Items-service* son servicios fundamentales para el negocio, ya que su cáıda impacta directamente sobre la generación de órdenes, por lo cual deben presentar alta disponibilidad y funcionar correctamente en todo momento[3]. *Itemsservice* recibe internamente las novedades de modificaciones de ítems. Debido a que restaurantes grandes pueden producir actualizaciones masivas, las novedades no se dan siempre de manera controlada. Estas novedades actualizan la base de datos propia de *Items-service*, la cual utiliza la técnica de Throttling[4] para controlar estos picos de operaciones: básicamente, la base de datos advierte que se ha llegado al límite de peticiones y agrega dinámicamente mayor capacidad. Sin embargo, este proceso demanda cierto tiempo, durante el cual todas las peticiones realizadas fallan. Mientras tanto, *Niles* continúa solicitando información a *Items-service*, y el agregado de reintentos empeora la situación. Debido a esto, *Items-service* no logra recuperarse y termina afectando a los demás servicios que lo consumen.

La aplicación del *Circuit Breaker* evita saturar a *Items-service* con requerimientos desde *Niles*. Cuando el disyuntor entre ambos pasa a estado abierto, en principio, no podría mostrarse el menú, pero en general la respuesta se brinda desde la caché propia de *Niles*. Así, se evita abrumar a un servicio que ya llegó al límite de requerimientos, pero que estará nuevamente disponible en el corto plazo.

4. Resultados Experimentales

4.1 Diseño experimental

En general, en las comunicaciones entre microservicios, es buena práctica definir *Timeouts* adecuados, configurar reintentos en algunos casos e implementar un *Circuit Breaker* que evite saturar de requerimientos a los microservicios que son consumidos. Debido a esto, se seleccionaron estos patrones para analizar y evaluar el impacto de su aplicación en *Niles* y sus microservicios asociados[5]. Para ello, se realizaron dos procesos experimentales para cada uno de ellos (se puede encontrar un mayor nivel de detalle sobre el diseño experimental en [8]):

1. El primer proceso consistió en enviar a *Niles* una serie de peticiones HTTP para emular el funcionamiento normal. Luego de un determinado tiempo, se sometió a *Niles* a una fuerte ráfaga de peticiones para degradar su funcionamiento. Estas peticiones se realizaron sobre una versión de *Niles* que no incluía la aplicación de

[3] Dentro de la compañía, estos servicios se catalogan como Tier1

[4] https://aws.amazon.com/es/premiumsupport/knowledge-center/dynamodb-tablethrottled

[5] Dentro de PedidosYa, los patrones Timeout, Retry y Circuit Breaker están implementados en una librería interna denominada peya-kator-utils. Los detalles vinculados a la implementación pueden encontrarse en [8]

ningún patrón.

2. El segundo proceso replicó al anterior, pero en este caso sobre una versión de *Niles* que incluía la aplicación del patrón de interés. Para ambos casos se monitorizó el funcionamiento de *Niles*, para luego analizar los datos obtenidos. De esta manera, se pudieron cuantificar los beneficios del uso de cada patrón.

A continuación, se describen varios aspectos vinculados al trabajo experimental.

Instalación de *Niles* utilizando Jarvis, una herramienta de la compañía que permite múltiples operaciones, como la creación y administración de nuevos proyectos y servicios en diferentes lenguajes, librerías, diferentes tipos de bases de datos, tópicos, colas de mensajes, cachés, programación de *cron jobs*, etc.

Ejecución de una serie de peticiones HTTP sobre uno o varios recursos dentro de *Niles*. Mediante Jarvis, se programa una tarea que se lanza a demanda y ejecuta un *script* de la herramienta K6, que permite realizar pruebas de performance. K6 se configura para que, durante el primer minuto de ejecución, intente realizar 200 peticiones (dependiendo de los recursos disponibles en el servidor subyacente); durante el segundo minuto, aumenta para intentar llegar a 500 requerimientos; y, durante los últimos 10 minutos se realizan aproximadamente 700 requerimientos por minuto. Por lo tanto, la duración total de la prueba es de 12 minutos.

Ejecución de fuertes ráfagas de peticiones para degradar un servicio consumido por *Niles*. Para esto se utiliza *Autocannon*, una herramienta de propósito similar a K6. Esta herramienta es más fácil de usar, requiere menos configuraciones y se puede ejecutar localmente, sin utilizar recursos de la compañía. *Autocannon* se configura para realizar 800 peticiones durante 300 segundos al servicio de favoritos de *Niles*, luego de 5 minutos de iniciadas las pruebas con K6. Al enviar una fuerte ráfaga de requerimientos en un período corto (y no hacerlo gradualmente), se evita que el servicio escale horizontalmente, lo que interferiría con el propósito del experimento.

Evaluación del impacto del patrón mediante el análisis de las mejoras generadas en base a su aplicación. Para esto se utilizó la herramienta *DataDog*, la cual ofrece métricas de rendimiento y posee integración con diferentes elementos de infraestructura, como bases de datos, logs y cachés.

4.2 Experimentos y Resultados Obtenidos para *Timeout*

Este experimento se realizó para demostrar cómo, al no configurar correctamente los tiempos de espera en las peticiones HTTP, el menú se ve penalizado por los tiempos altos de respuesta de un servicio de menor prioridad como es *Favouritesservice*. A partir de una instalación de *Niles* sin ningún patrón, se inicia (mediante K6) la realización de peticiones sobre el *endpoint* que ofrece el menú. Luego de aproximadamente cinco minutos, se procede a enviar una ráfaga de peticiones hacia *Favourites-service*.

En la parte izquierda de la Fig. 2 se ve como el tráfico no sigue un patrón regular (alterna periodos de pocas y excesivas peticiones), lo que evidencia que existen hilos bloqueados en espera en determinados momentos. En tanto, a la derecha se observa cómo los tiempos del menú se ven afectados, ya que existen peticiones en espera desde *Favourites-service*.

A continuación, se repite el experimento, pero incorporando la configuración de un límite de espera en cada petición HTTP saliente desde *Niles*. Por lo tanto, cuando *Favourites-service* sufre una ráfaga de requerimientos y se degrada, no se penaliza al menú. A la izquierda de la Fig. 3 se puede observar cómo el tráfico entrante aumenta de forma estable, sin alternar períodos de alto y bajo tráfico. En tanto, a la derecha, se puede ver que el tiempo de latencia no supera los 170 ms (salvo en dos picos), lo que resulta mucho menor a los 6000 ms en promedio sin la aplicación del patrón *Timeout*. Esta mejora en el rendimiento se debe a que se finaliza intencionalmente la comunicación con las dependencias luego del lapso establecido.

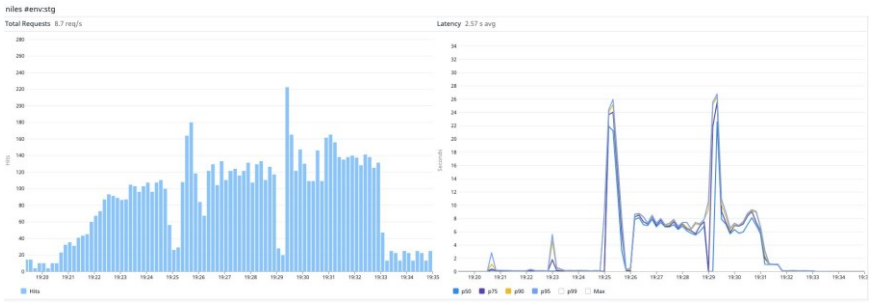

Figura2: Tráfico entrante (izq) y tiempos de respuesta de *Niles* (der)

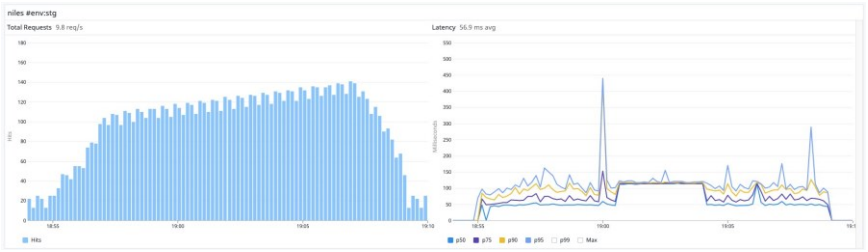

Figura3: Tráfico entrante (izq) y tiempos de respuesta de *Niles* (der) con aplicación de *Timeout*

4.3 Experimentos y Resultados Obtenidos para *Retry*

Para la realización de las pruebas del patrón *Retry* no se ejecutan ráfagas de peticiones sobre *Favourites-service*, pero se realiza una modificación en su código para simular el fallo en una de cada diez peticiones. El objetivo es evaluar el comportamiento de los servicios frente a errores temporales o lentitud en la red con la aplicación de reintentos.

A partir de una instalación de *Niles* sin la aplicación de ningún patrón, se ejecutan las peticiones de prueba utilizando K6. En la parte superior de la Fig. 4 se puede observar el tráfico recibido en *Niles*, donde los errores se marcan en color rojo. Sin embargo, haciendo foco en el tipo de errores (en la parte inferior) se puede ver que presentan código de estado 200, lo que significa que las peticiones a *Niles* finalizan exitosamente (se retorna el menú al usuario), pero en la ejecución ocurrió algún error

(que es el recibido desde *Favourites-service*. Debido a que este servicio no es obligatorio para generar el menú, no se penaliza al mismo.

Mediante el envío de parámetros al cliente que realiza la petición, se configura una cantidad de 3 reintentos por defecto en la comunicación desde *Niles* hacia *Favourites-service*. En la parte superior de la Fig. 5 se puede ver el tráfico recibido por la versión de *Niles* que hace uso del patrón *Retry* (con los errores marcados en rojo), mientras que la parte inferior se focaliza sobre los errores con código de respuesta 200: bien aún existen errores, la diferencia es notable cuando se aplica el patrón, ya que el contador que se aprecia en la Fig. 5 muestra una cantidad total de 54 errores, cuando este valor ascendía a 822 en ausencia de *Retry* (ver Fig. 4).

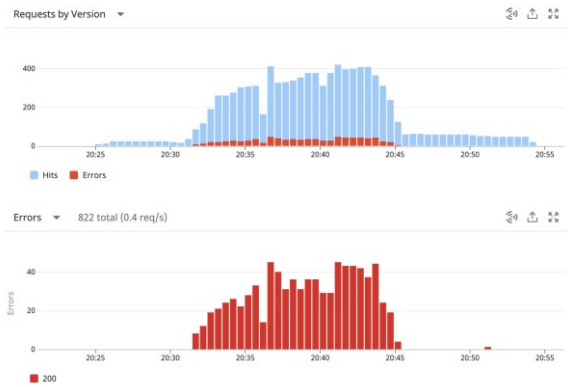

Figura4: Peticiones hacia *Niles* (arriba) y cantidad de errores con código de estado 200 (abajo)

4.4 Experimentos y Resultados Obtenidos para *Circuit Breaker*

El objetivo de estas pruebas es mostrar que la aplicación del patrón evita que el servicio consumidor utilice recursos en realizar peticiones que tienen muchas probabilidades de fallar. La realización de pruebas sigue la misma metodología explicada en la sección 4.1. En la Fig. 6 se puede observar el aumento de tráfico hacia *Niles* (en su versión sin la aplicación de ningún patrón) y los errores provocados intencionalmente en *Favourites-service* (marcados en rojo). En la parte superior, se refleja el tráfico entrante a *Niles* y los correspondientes tiempos de respuesta, mientras que en la parte inferior se ilustran los mismos parámetros para *Favourites-service*. Las peticiones de *Niles* hacia *Favourites-service* son continuas, pero como éste último degrada sus tiempos de respuesta, el menú sufre una penalización. Una vez que se incorpora el *Circuit Breaker*, la Fig. 7 muestra la reducción de las peticiones hacia *Favourites-service* durante su degradación, lo que conduce al ahorro de recursos. Además, en la Fig. 8 se puede observar la disminución de los tiempos de respuesta del menú durante el lapso que no se envían peticiones a *Favourites-service*. Esta mejora se debe a que no se continúa esperando por una respuesta que probablemente no pueda resolverse.

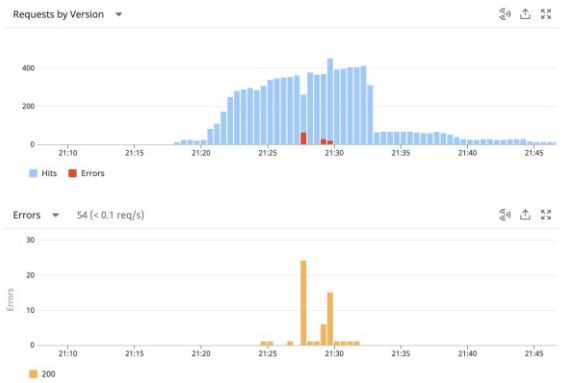

Figura 5: Peticiones hacia *Niles* (arriba), y respuestas código de estado 200 (abajo), con *Retry*

Los rectángulos marcados en la Fig. 8 ilustran los beneficios de la aplicación de *Circuit Breaker*. El rectángulo de la izquierda destaca el lapso de tiempo en el que no se realizan peticiones hacia *Favourites-service*, debido a que el *Circuit Breaker* pasa a estado abierto. El rectángulo del centro permite ver que los tiempos de respuesta de *Favourites-service* durante el incidente superan los 5000 ms; mientras que el rectángulo de la derecha evidencia que el tiempo de respuesta del menú no supera los 500 ms. Esto representa una mejora de 10 veces respecto a no aplicar el disyuntor. Por lo tanto, la aplicación del patrón permite, además del ahorro de recursos, obtener mejoras en el desempeño del servicio si, por ejemplo, se omite la petición para obtener los productos favoritos y se retorna un menú carente de esa información, debido a que el interruptor está abierto.

5. Conclusiones y Trabajo Futuro

En la actualidad, la resiliencia es uno de los aspectos no funcionales más buscados en los sistemas considerando el impacto directo con el negocio de una empresa. En este trabajo se analiza la aplicación de una serie de patrones que se utilizan en la empresa

Figura 6: Tráfico entrante y tiempos de respuesta en *Niles* (arriba) y *Favouritesservice* (abajo)

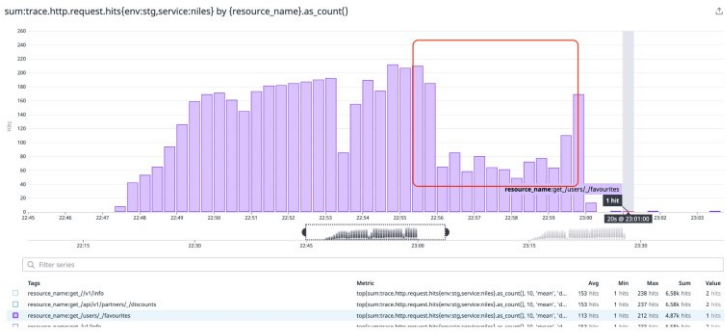

Figura7: Ahorro de recursos con la ayuda del patrón *Circuit Breaker*

PedidosYa para afrontar diversos fallos capaces de afectar el funcionamiento de su ecosistema de microservicios. A partir de los resultados experimentales, se puede concluir que:

- *Timeout* permitió que el tiempo de respuesta para retornar el menú se mantenga razonable, ya que con la inclusión del patrón se disminuyó de aproximadamente 6000 a 170ms (aprox. 35×). Además, al finalizar la comunicación en tiempo acotado, se evita la acumulación de requerimientos pendientes, mejorando así el uso de recursos.
- *Retry* permitió que una mayor cantidad de las peticiones entrantes puedan ser resueltas exitosamente, reduciendo la cantidad de requerimientos erróneos de 822 a 54ms (aproximadamente 15×). *Retry* muestra que, en algunos contextos, un simple reintento puede completar exitosamente una funcionalidad que impacta en las ganancias de la compañía (como la facturación de una orden).

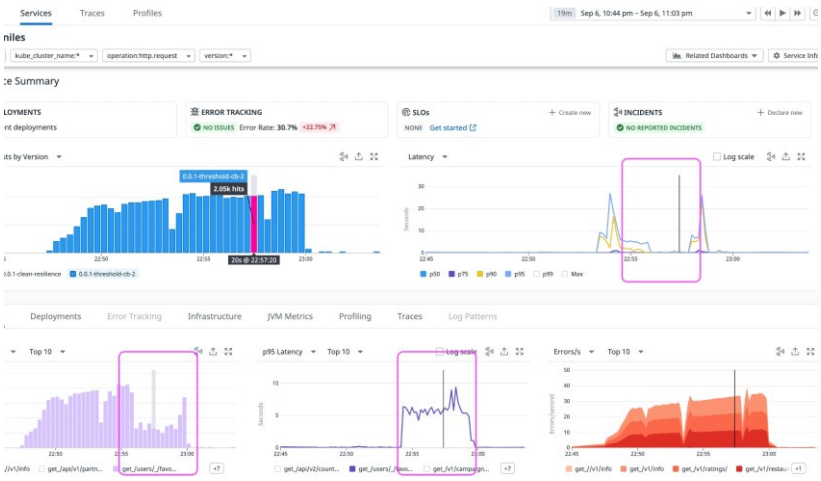

Figura8: Mejoras al aplicar el patrón *Circuit Breaker*

- *Circuit Breaker* facilitó la provisión de un menú disminuido (sin favoritos) ante situaciones de elevadas probabilidades de falla. Además de permitir un ahorro de recursos, evitó realizar peticiones hacia aquellos servicios temporalmente averiados e incluso redujo el tiempo de respuesta del menú en ciertos escenarios.

La implementación de cada uno de estos patrones de resiliencia en *Niles* se encuentran en producción en el ecosistema de PedidosYa, constituyéndolo como uno de los componentes más utilizados y robustos dentro de la compañía. Como trabajo futuro, se propone extender el análisis de la implementación de otros patrones orientados a la resiliencia como *Bulkhead*, *Throttling* o *Queue-Based Load Leveling*, además de considerar la combinación de los ya contemplados.

Referencias

1. Fowler, M.: Microservices, https://martinfowler.com/articles/microservices. html
2. Hofmann, M., Schnabel, E., Stanley, K., Organization, I.B.M.C.I.T.S.: Microservices Best Practices for Java. IBM redbooks, IBM Corporation, International Technical Support Organization (2016), https://books.google.com.ar/books?id=ZiSJAQAACAAJ
3. ION Group: How Do You Manage Non-Functional Requirements?, https://iongroup.com/blog/markets/how-do-you-manage-non-functional-requirements/
4. Nygard, M.T.: Release it! design and deploy production-ready software, Second edition. Pragmatic Bookshelf (2018)
5. Peter Jausovec: Fallacies of distributed systems, https://blogs.oracle.com/developers/post/fallacies-of-distributed-systems
6. Richardson, C.: Microservices Patterns: With examples in Java, chap. Escaping monolithic hell. Manning (2018), https://books.google.com.ar/books?id=UeK1swEACAAJ
7. Richardson, C.: Microservices Pattern: Circuit Breaker, https://microservices.io/patterns/reliability/circuit-breaker.html
8. Suárez, S.L.: Análisis de patrones de resiliencia en una arquitectura basada en microservicios. Tesina de Licenciatura en Sistemas, Universidad Nacional de La Plata (Feb 2023), http://sedici.unlp.edu.ar/handle/10915/149187