Extendiendo las Estrategias de Resiliencia de una Arquitectura Basada en Microservicios

Sergio Leonel Suárez[1,2], Enzo Rucci[3,4], Victor Betran[2], and Diego Montezanti[3]

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

Resumen En la última década, se ha incrementado la adopción de arquitecturas de microservicios para ayudar a superar las limitaciones de los sistemas monolíticos. Dado que la resiliencia se torna crucial debido al impacto directo de las fallas en el negocio de una empresa, han surgido patrones de diseño para resiliencia como estrategias para gestionar las fallas y mitigar sus efectos negativos. En un trabajo previo se analizó el comportamiento de algunos patrones para resiliencia dentro del ecosistema de microservicios de PedidosYa, particularmente en el microservicio *Niles*. Para ello, se estudiaron diversos escenarios de fallas y se aplicaron 3 patrones de manera individual (*Timeout*, *Retry* y *Circuit Breaker*) para darles solución. Este trabajo extiende al anterior mediante la incorporación de un nuevo patrón (*Bulkhead*) y la combinación de los ya estudiados previamente. Los resultados experimentales muestran que la implementación de patrones de resiliencia en *Niles* permiten aumentar su robustez, siendo una de las razones por las que se encuentran en producción en el ecosistema de PedidosYa.

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

1. Introducción

En la última década, el término *arquitectura de microservicios* ha emergido para describir una forma específica de diseñar aplicaciones como conjuntos de servicios que se pueden desplegar de manera independiente, abordando así las limitaciones de los sistemas monolíticos [8]. Aunque no existe una definición precisa para este estilo arquitectónico, se reconocen características comunes como la organización, la capacidad empresarial, la automatización y el control descentralizado de lenguajes y datos [1].

No obstante, estas arquitecturas también presentan diversas desventajas, relacionadas con la complejidad propia de los sistemas distribuidos, la falta de métodos

para la descomposición en microservicios y las dificultades ligadas a la consistencia, la monitorización y la seguridad. Sin embargo, en este trabajo, el foco está puesto en la resiliencia, que es uno de los aspectos no funcionales más buscados, especialmente en sistemas a gran escala, ya que cuando ocurre un fallo, no solo se ve afectado el microservicio que falla, sino también aquellos que dependen de él. El manejo de fallos es crucial, ya que su impacto puede traducirse en pérdidas económicas debido al tiempo de inactividad de los servicios. En consecuencia, en los últimos años se ha suscitado un gran interés de la comunidad en el abordaje de la resiliencia en arquitecturas de microservicios [3,13,12,4,5,9,2].

Como parte de las estrategias utilizadas para manejar estas fallas y mitigar sus consecuencias negativas, se emplean patrones de diseño para la resiliencia entre microservicios. En un trabajo anterior [11] analizamos 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, que procesa aproximadamente 4 millones de órdenes por semana, cuenta con una arquitectura basada mayormente en microservicios. El análisis contempló la aplicación de 3 patrones de resiliencia (*Timeout*, *Retry* y *Circuit Breaker*) en forma individual y las mejoras derivadas de ello, en cuanto a conversiones de negocio y otras métricas. Este trabajo extiende al anterior incorporando al patrón *Bulkhead* y a la combinación de los 3 ya estudiados previamente, lo que permite enriquecer y robustecer el análisis realizado.

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

2. Contexto

2.1. Algunos fallos en arquitecturas de microservicios

En una arquitectura de microservicios, pueden surgir situaciones que hagan que el sistema se vuelva inestable. Aunque algunos problemas son comunes a todos los sistemas de software, como una base de datos inconsistente o la falta de recursos, existen problemas particulares de los sistemas distribuidos [7], como pueden ser:

- Problemas de red o lentitud: Dado que la interacción entre servicios es constante, cualquier error de red, interrupción breve, fallo en un componente (como el servidor DNS) o congestión es capaz de afectar todo el sistema.
- Sobrecarga de tráfico: Cada microservicio es independiente y tiene límites operacionales específicos, como la cantidad de solicitudes por minuto que puede manejar. Si un servicio recibe más tráfico del esperado, puede experimentar degradación, funcionando más lentamente o de manera anómala.
- Priorización inadecuada: No todos los servicios en una empresa tienen el mismo nivel de importancia. Por lo tanto, si las prioridades no estuvieran bien definidas, el fallo de cualquier servicio podría tener impactos del mismo orden.

Es crucial definir las prioridades y el manejo de errores de forma individual para cada servicio, para evitar que el fallo de un componente afecte a todo el sistema.

La resiliencia es un aspecto que debe ser integrado desde el diseño inicial de la arquitectura, mediante un modelado adecuado de los microservicios. En este contexto, el uso de patrones de diseño es fundamental para construir una arquitectura capaz de tener un comportamiento adecuado frente a los fallos.

2.2. El servicio *Niles* dentro de la arquitectura de microservicios de PedidosYa

La empresa se organiza en distintas verticales, como Restaurantes, Farmacias, Mercados y Bebidas. La vertical de Restaurantes, que genera los mayores ingresos, distribuye toda su lógica en microservicios, siendo *Niles* uno de los más destacados. *Niles* se encarga de proporcionar a los usuarios el menú de un restaurante, realizando múltiples consultas a otros microservicios para construir el menú, que incluye secciones con productos junto con información básica como nombre, descripción, imagen, precio y descuentos aplicables, además de datos adicionales como popularidad de ventas, recomendaciones y favoritos.

Dentro de cada aplicación móvil, un módulo de visualización (denominado *shopDetail*), consulta a *Niles* para obtener el menú. Para ello, *Niles* recibe la solicitud de menú para un restaurante y verifica si el menú solicitado existe en una caché distribuida; si el menú no se encuentra en la caché, consulta diversos servicios para completar la información requerida. Una vez que se obtiene el menú completo, se guarda en la caché para solicitudes futuras. En cualquier caso, se procesa el menú mediante la aplicación de diferentes filtros (validaciones de edad, disponibilidad de stock, horarios correctos) y ordenamientos.

Niles cuenta con una caché centralizada, compartida por varias instancias, además de una caché local para almacenar información que cambia raramente, como países y categorías de productos. Para mantener el menú actualizado, *Niles* recibe eventos de actualización de otros microservicios como *Items-service* y *Battlefront*. Cuando se recibe una actualización de novedad, *Niles* elimina de la caché la entrada correspondiente a ese restaurante, y sólo vuelve a agregarla cuando reciba una nueva solicitud desde una aplicación móvil. La Fig. 1 ilustra el flujo completo de comunicación entre todos los componentes.

2.3. Algunos microservicios relacionados

Para poder cumplir con su tarea y proporcionar el menú completo, *Niles* interactúa con otros microservicios, que se describen en detalle en [10]. Sin embargo, los experimentos que se realizaron para evaluar el impacto de la aplicación de los patrones de resiliencia únicamente afectaron las interacciones de *Niles* con tres de ellos. A continuación se describen sus principales funcionalidades:

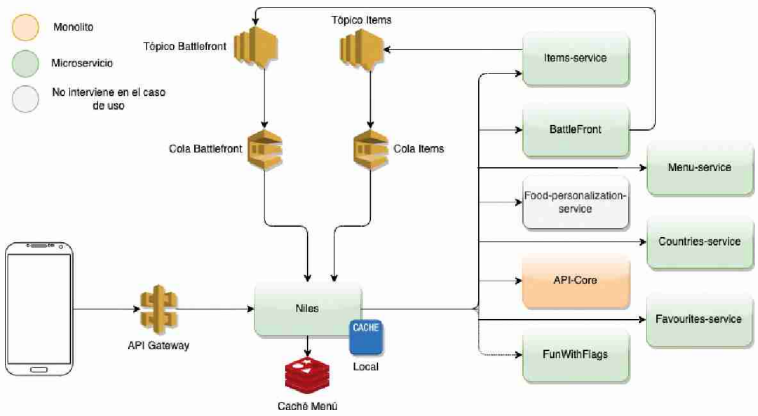

Figura 1: Interacción de *Niles* con sus componentes asociados

- *Items-service* es el microservicio responsable de alimentar el menú, proporcionando secciones, productos y sus opciones de configuración. Administra datos como el nombre, imagen, descripción y precio para cada sección, producto y opción de producto. Cuando se produce un cambio en algún dato, el servicio envía actualizaciones (a través de tópicos) que llegan a las colas de mensajes de *Niles*, el cual se encarga de actualizar la caché centralizada. Si *Items-service* no funciona bien (debido a una caída temporal o un incidente grave), *Niles* podría no mostrar el menú (o mostrarlo desactualizado).
- *Battlefront* es el servicio encargado de la administración de descuentos, promociones, beneficios y campañas publicitarias. Como cada descuento (por restaurante o producto) puede ser creado, modificado, eliminado o dejar de estar vigente, *Battlefront* envía novedades por medio de mecanismos sincrónicos de mensajes (tópicos y colas de mensajes), las cuales son recibidas desde *Niles* para aplicar estos cambios sobre el menú.
- *Favourites-service* es el servicio encargado de proporcionar información sobre todas las entidades que un usuario marca como favoritas. Maneja entidades como restaurantes, productos y configuraciones, por lo que *Niles* utiliza *Favourites-service* para agregar o eliminar productos de la lista de favoritos.

3. Estrategias para lidiar con fallos en *Niles*

Un microservicio se implementa como un servidor que recibe solicitudes y utiliza un grupo de hilos (*thread pool*) para gestionarlas. La cantidad de solicitudes que se pueden atender simultáneamente está limitada por el número de hilos disponibles en el *pool*, los cuales se pueden reutilizar para nuevas solicitudes una vez que se liberan. Dado que cada microservicio es autónomo (y responsable de una funcionalidad específica), a menudo puede requerir información de entidades externas, como bases de datos, cachés u otros microservicios. Cada vez que se consulta un

recurso externo, el hilo solicitante se bloquea en espera, quedando impedido para pueda atender nuevas solicitudes durante ese tiempo.

3.1. El patrón *Bulkhead*

Este patrón está inspirado en los mamparos (*bulkheads*) de los barcos, los cual son particiones de metal que se pueden sellar para dividir el casco de la nave en compartimentos separados. Una vez que se cierran las escotillas, el mamparo evita que el agua fluya de un área hacia otra, para que el efecto de una única ruptura en el casco no pueda propagarse y terminar produciendo el hundimiento del barco [6]. Esta misma idea puede emplearse en el diseño de arquitecturas de software. Al particionar un sistema, se puede evitar que una falla que afecta a una parte del mismo se propague hacia las demás funcionalidades.

Para el caso de de *Niles*, la implementación del patrón *Bulkhead* requiere de la explicación de tres términos relacionados. Un *webpool* es un espacio donde se despliegan diferentes aplicaciones, pudiendo contener sus propias variables de entorno, además de variar en cuanto a los recursos asignados a cada servicio (como capacidad de memoria y CPU, cantidad mínima y máxima de instancias, etc.). A su vez, cada *webpool* pertenece a un *cluster*, lo que permite diferenciar ambientes de instalación (como por ejemplo los comúnmente conocidos como ambiente de pruebas y ambiente de producción). Al crear varios *webpools* y asignarlos al mismo *cluster*, es posible desplegar un microservicio en producción con diferentes configuraciones, la cual es precisamente la idea del patrón *Bulkhead*. Por último, en PedidosYa, es habitual utilizar el término *worker* para referir a un *webpool* que contiene desplegada una versión de un servicio, con el fin de realizar tareas en segundo plano, actualizaciones masivas, tareas programadas, manejo de novedades por colas de mensajes y otras tareas similares.

En el caso de *Niles*, se despliegan dos instancias del servicio en *webpools* separados. Una de ellas atiende normalmente las solicitudes HTTP, mientras que la otra no lo hace, sino que se encarga de recibir las novedades desde los tópicos de *Items-service* y de *Battlefront*. La versión de ambas instancias es exactamente la misma, pero ejecutan diferentes bloques de código según el valor de variables de entorno del *webpool*. Como eventualmente ocurrirán picos de novedades (por ejemplo, debido a actualizaciones masivas), el funcionamiento de *Niles* se podría ver afectado en cuanto a la generación de un menú. Sin embargo, al separar las diferentes funcionalidades, se logran aislar errores y evitar una degradación en la función del servicio de retornar el menú de determinado restaurante.

3.2. La combinación *Timeout-Retry-Circuit Breaker*

La aplicación de estos 3 patrones, de manera individual, fue explicada en detalle en el trabajo previo [11]. Aquí, el foco está puesto en la combinación de ellos, en la que cada patrón cumple su función encadenando acciones:

- El patrón *Timeout* es el primero en aplicarse para definir un tiempo de espera límite para las peticiones y evitar esperas largas, penalización de funcionalidades principales y encolado innecesario de peticiones.
- El patrón *Retry*, en caso de necesidad, realiza reintentos ante fallos transitorios, como pueden ser errores de red o agotamiento de tiempo de espera límite.
- El patrón *Circuit Breaker*, mediante la administración de sus estados internos, pasa al estado "abierto" cuando el cliente de la petición HTTP falla repetidamente, llegando al umbral configurado.

Para cada caso de uso, es necesario realizar un análisis que tenga en cuenta los tiempos de respuesta de cada servicio, evaluar la cantidad de reintentos y realizar combinaciones entre tiempos de espera y reintentos para no penalizar a la funcionalidad principal del servicio. Del mismo modo, es necesario analizar el comportamiento de cada uno de los *Circuit Breakers* existentes, definir respuestas por defecto en caso de que el disyuntor se encuentre en estado "abierto", y retornar una petición fallida en caso de que no sea posible funcionar sin una dependencia.

4. Resultados Experimentales

4.1. Diseño de experimentos

Para poder analizar el impacto de la aplicación de los patrones mencionados en *Niles*[1], se realizaron dos procesos experimentales para cada uno de ellos (un mayor nivel de detalle sobre el diseño de los experimentos puede encontrarse en [10]):

1. El primer proceso consistió en emular el funcionamiento normal de *Niles*, mediante el envío de una serie de peticiones HTTP. Luego de un lapso, *Niles* fue sometido a una fuerte ráfaga de peticiones con el objetivo de degradar su funcionamiento. Estas peticiones se realizaron sobre una versión de *Niles* que no incluía la implementación de ninguno de los patrones mencionados.
2. En segunda instancia se repitió el procedimiento, pero utilizando una versión de *Niles* que incluía la aplicación del patrón de interés. El análisis de los datos obtenidos mediante la monitorización del funcionamiento de *Niles* permitió 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 para el despliegue y desarrollo de experimentos y servicios.

Ejecución de una serie de peticiones HTTP sobre uno o varios recursos dentro de *Niles*. Utilizando Jarvis, se programa una tarea que se lanza a demanda y que se vale de la herramienta K6 [2]. En particular, K6 se configura para intentar realizar 200 peticiones durante el primer minuto de ejecución (dependiendo

[1] Dentro de PedidosYa, los patrones *Timeout*, *Retry* y *Circuit Breaker* están implementados en una librería interna denominada *peya-kator-utils* [10]

[2] Disponible en `https://k6.io/`

de los recursos disponibles en el servidor subyacente); 500 peticiones en el segundo minuto; y 700 peticiones por minuto durante los últimos 10 minutos. 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* [3], la cual se configura para realizar una alta cantidad de peticiones al servicio o recurso involucrado en la prueba particular durante un lapso determinado. El comando de *Autocannon* se lanza manualmente un cierto tiempo después del inicio de las pruebas con K6. Al enviar una fuerte ráfaga de solicitudes en un período corto (en lugar de hacerlo gradualmente), se evita el escalado horizontal del servicio, lo cual representaría una interferencia con el propósito del experimento.

Evaluación del impacto del patrón mediante el análisis de las mejoras obtenidas tras su aplicación. Para ello, se utilizó *DataDog* [4].

4.2. Experimentos y Resultados Obtenidos para *Bulkhead*

Como se mencionó anteriormente, la funcionalidad principal de *Niles* es generar el menú de un restaurante. Además, también ofrece la posibilidad de entregar productos para realizar *Cross-Selling* [5]. El procedimiento experimental consistió en la ejecución de la misma cantidad de peticiones de la sección 4.1 (utilizando K6), pero sobre cada recurso disponible (menú y *Cross-Selling*). Como el objetivo es provocar la degradación de un recurso, las ráfagas de peticiones se realizaron sobre el recurso de *Cross-Selling*, para poder distinguir entre el comportamiento de *Niles* cuando todas las funcionalidades conviven en el mismo servidor (en ausencia del patrón *Bulkhead*) y cuando lo hacen en diferentes (con *Bulkhead* implementado). Las ráfagas de peticiones se emitieron durante 5 minutos, y comenzaron 3 minutos después del inicio del experimento con K6.

En la Fig. 2 se puede observar cómo, al momento de iniciar las ráfagas, *Niles* presenta errores (marcados en rojo). Como la ráfaga de peticiones es fuerte, el servicio deja de estar disponible por varios minutos. Si bien las ráfagas de peticiones se realizan sobre el recurso de *Cross-Selling*, el servicio se ve afectado de forma completa, impactando en la funcionalidad del menú. Por lo tanto, los resultados obtenidos muestran cómo las peticiones constantes a un recurso de menor relevancia pueden impactar en la funcionalidad principal del servicio.

Para la aplicación del patrón *Bulkhead*, en lugar de contar con un único *webpool*, se cuenta con dos de ellos. Cada uno de los *webpools* contiene una instancia idéntica de

[3] Herramienta similar a K6 pero más fácil de usar, que requiere menos configuraciones y puede ejecutarse localmente sin utilizar recursos de la compañía. Disponible en `https://github.com/mcollina/autocannon`

[4] Herramienta que proporciona métricas de rendimiento e integra diversos elementos de infraestructura, como bases de datos, registros y cachés. Disponible en `https://www.datadoghq.com/`

[5] En marketing, se llama *Cross-Selling* a la táctica mediante la cual se intenta vender productos complementarios a los que consume o pretende consumir un cliente.

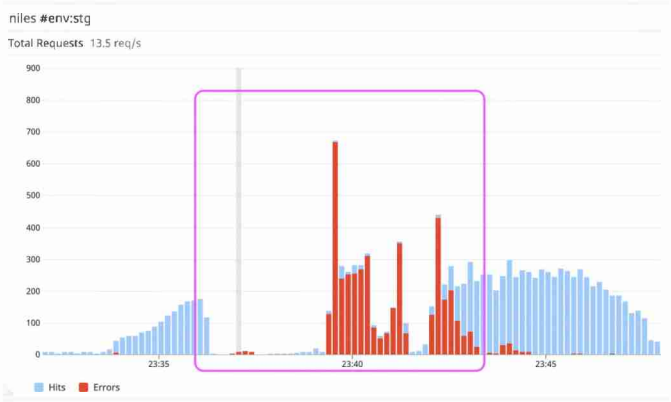

Figura 2: Errores en todos los recursos que ofrece *Niles*

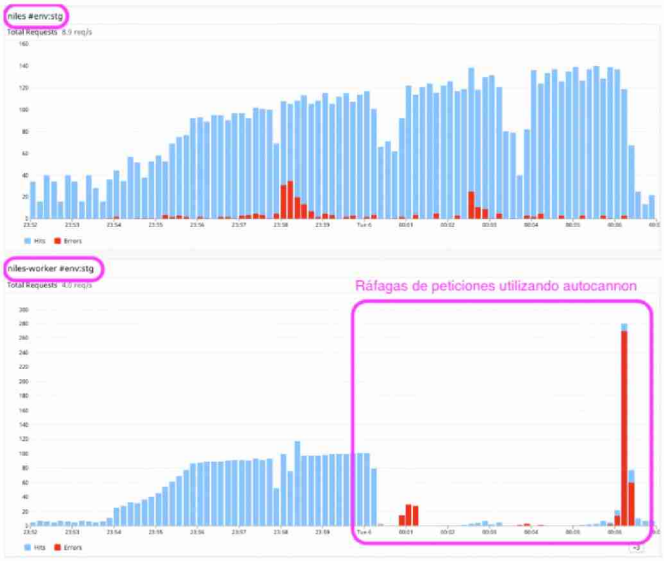

Figura 3: Tráfico y errores en *Niles*: sobre el recurso del menú (arriba) y sobre el recurso de *Cross-Selling* (abajo) en *webpools* separados

la misma versión de *Niles*. La cantidad de peticiones a realizar es la misma que en el paso previo, pero ejecutándose en cada uno de los dos recursos que provee *Niles*. En la Fig. 3 (arriba) se pueden visualizar las peticiones realizadas sobre ambos recursos (menú y *Cross-Selling*) junto a las ráfagas provocadas para degradar el servicio. En el *webpool worker*, la ráfaga de peticiones es lo suficientemente fuerte para dejar al servicio no disponible, incluso al nivel de no permitir más peticiones. Como se puede observar en la Fig. 3 (abajo), existe un pico de tráfico donde la gran mayoría de peticiones fallan. Durante el lapso de ejecución de ráfagas de

peticiones (utilizando *Autocannon*), se procedió a solicitar la información del menú a las instancias de *Niles* que no recibieron ráfagas de peticiones. En este caso fue posible recibir la respuesta correctamente, permitiendo visualizar exitosamente el menú. En consecuencia, resulta claro que el patrón *Bulkhead* permite aislar los fallos, de manera de poder seguir ofreciendo la funcionalidad deseada.

4.3. Experimentos y Resultados Obtenidos para la combinación *Timeout-Retry-Circuit Breaker*

Para realizar el experimento, se utilizó la misma estrategia aplicada para el estudio del patrón *Retry* en forma individual [11]. El recurso degradado por las ráfagas es *Favourites-service*; además, se realizó una modificación para que uno de cada diez requerimientos falle al azar. En este caso, las ráfagas de peticiones se emitieron durante 5 minutos, y comenzaron 5 minutos después del inicio del experimento con K6.

El primer punto crítico a considerar es que no existe una finalización de la comunicación desde *Niles* hacia ningún servicio al no implementarse el patrón *Timeout*. Esta situación conduce a que, ante cualquier petición de menor relevancia (como puede ser la que se realiza hacia *Favourites-service*), se penalice al tiempo total de respuesta. Este comportamiento se ilustra en la Fig. 4) (der), la cual muestra diferentes métricas de latencia [6].

El segundo punto crítico para el análisis está relacionado con el hecho de que, al no contar con la implementación del patrón *Retry*, toda falla transitoria se traduce en un error, como se puede observar en Fig. 4) (izq). Por lo tanto, ante problemas de intermitencias en la red, o de respuestas con tiempos elevados, las peticiones no pueden resolverse.

Por último, en ausencia de implementación del patrón *Circuit Breaker*, no existe un mecanismo que evite seguir realizando requerimientos hacia un servicio, incluso cuando éste presente altas probabilidades de falla. Esta situación lleva a que, aunque *Favourites-service* haya sido al degradado con una ráfaga de peticiones, *Niles* continúe solicitándole información. En la Fig. 5 se observa cómo, en una etapa de fallos constantes en el servicio de favoritos (rectángulo de la derecha), el consumidor continúa realizando peticiones (rectángulo de la izquierda).

Al aplicar la combinación de patrones en *Niles*, se analizan los puntos críticos mencionados en las pruebas sin patrones. En la Fig. 6 (der) se puede apreciar cómo al implementar el patrón *Timeout*, es *Niles* quien finaliza la comunicación, al tener establecidos tiempos límite de espera. De esta manera, se evita el encolamiento de hilos y la correspondiente penalización en los tiempos del menú. Esto se puede ver reflejado en la parte derecha de la imagen, donde el tiempo más alto registrado es de 6,5 segundos, que resulta mucho menor a los tiempos observados en la Fig. 4 (der).

[6] En el análisis de pruebas de cargas de trabajo es habitual calcular el percentil `p` de latencia de la aplicación. Tomando todos los tiempos de solicitud-respuesta, la latencia del percentil `p` es el valor en el que se completan el `p`% de las solicitudes.

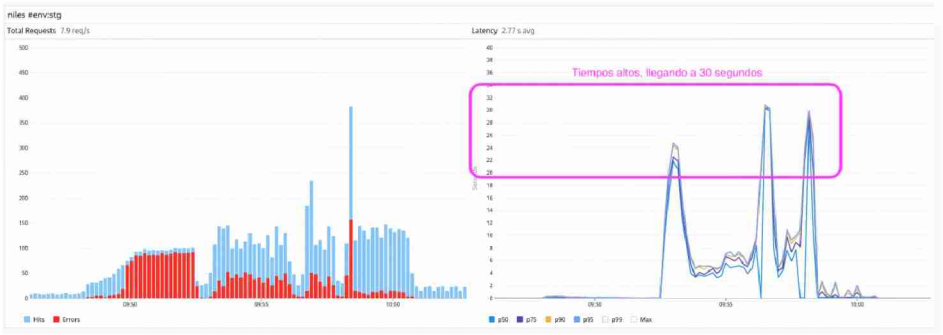

Figura 4: Tráfico a *Niles* (izq) y tiempos de respuesta del menú (derecha)

Figura 5: Peticiones constantes sobre un servicio que se encuentra atravesando un período de fallos

Ante la falla del intento anterior, se reintenta 3 veces más, en la búsqueda de lograr dar respuesta al requerimiento. Sin embargo, debido a la configuración empleada, estos reintentos fallan, dando lugar a la ejecución de *Circuit Breaker*. Gracias a la inclusión de este mecanismo, el número total de solicitudes con fallas se reduce considerablemente, como se observa en Fig. 6 (izq). Más aún, en la Fig. 7 se puede ver el impacto de incluir el patrón *Circuit Breaker* en el funcionamiento de *Niles*. En particular, se puede notar cómo *Niles* evita continuar realizando peticiones hacia el servicio en alerta cuando *Favourites-service* es degradado debido a las ráfagas iniciadas mediante la utilización de *Autocannon*.

Por lo tanto, mediante la combinación de los patrones de diseño *Timeout*, *Retry* y *Circuit Breaker*, se logra una mejora en los siguientes aspectos:

- Se evita el encolamiento de hilos en esperas largas, debidas a la latencia de servicios que son consumidos.
- Se evita la excesiva penalización de tiempos sobre la funcionalidad principal (por ejemplo, impidiendo largas esperas por un producto favorito, cuando lo indispensable es la información básica del menú).
- Los reintentos ayudan a resolver fallas transitorias por lentitud en red, o incidentes en ella.

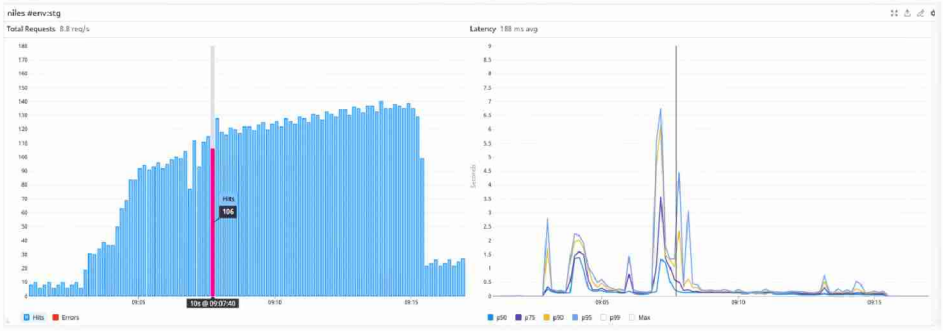

Figura 6: Patrón *Timeout*: tiempo de respuesta con pico de 6,5 segundos

Figura 7: *Circuit breaker* en estado abierto evita peticiones hacia *Favourites-service*

- Se posibilita el ahorro de recursos, evitando realizar peticiones hacia un servicio que se encuentra en estado de alerta, con altas probabilidades de que un requerimiento falle.

5. Conclusiones y Trabajo Futuro

Actualmente, la resiliencia es uno de los aspectos no funcionales más valorados en los sistemas, debido a su impacto directo en el negocio de una empresa. Este trabajo extiende la evaluación de varios patrones, utilizados por la empresa PedidosYa, para manejar diversos fallos que podrían afectar el funcionamiento de su ecosistema de microservicios. A partir de los resultados experimentales, se pueden extraer las siguientes conclusiones:

- *Bulkhead* permite el aislamiento de los fallos que afectan a una funcionalidad particular de *Niles*, evitando la propagación hacia otros servicios, y permitiéndole así responder al problema de la priorización de sus funcionalidades.
- Es posible diseñar y aplicar una combinación de patrones en Niles, para enfrentar escenarios más complejos de fallas. Si bien esto aumenta la complejidad de su implementación, se obtiene el beneficio del aumento en la robustez del sis-

tema, ya que se contempla la prevención y mitigación de una mayor cantidad de casos de fallas.

La implementación de patrones de resiliencia en *Niles* permiten aumentar su fiabilidad, siendo una de las razones por las que se encuentran en producción en el ecosistema de PedidosYa. A futuro, se propone extender el análisis de la implementación de otros patrones orientados a la resiliencia, como *Throttling* o *Queue-Based Load Leveling*, de forma de enriquecer el presente estudio.

Referencias

1. Fowler, M.: Microservices, `https://martinfowler.com/articles/microservices.html`
2. Giedrimas, V., Omanovic, S., Alic, D.: The aspect of resilience in microservices-based software design. In: Software Technologies: Applications and Foundations: STAF 2018 Collocated Workshops, Toulouse, France, June 25-29, 2018, Revised Selected Papers. pp. 589–595. Springer (2018)
3. Heorhiadi, V., Rajagopalan, S., Jamjoom, H., Reiter, M.K., Sekar, V.: Gremlin: Systematic resilience testing of microservices. In: 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS). pp. 57–66. IEEE (2016)
4. Jagadeesan, L.J., Mendiratta, V.B.: When failure is (not) an option: Reliability models for microservices architectures. In: 2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). pp. 19–24. IEEE (2020)
5. Mendonca, N.C., Aderaldo, C.M., Cámara, J., Garlan, D.: Model-based analysis of microservice resiliency patterns. In: 2020 IEEE International Conference on Software Architecture (ICSA). pp. 114–124. IEEE (2020)
6. Nygard, M.T.: Release it! design and deploy production-ready software, Second edition. Pragmatic Bookshelf (2018)
7. Peter Jausovec: Fallacies of distributed systems, `https://blogs.oracle.com/developers/post/fallacies-of-distributed-systems`
8. Richardson, C.: Microservices Patterns: With examples in Java, chap. Escaping monolithic hell. Manning (2018)
9. Surendro, K., Sunindyo, W.D., et al.: Circuit breaker in microservices: State of the art and future prospects. In: IOP Conference Series: Materials Science and Engineering. vol. 1077, p. 012065. IOP Publishing (2021)
10. 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`
11. Suárez, S.L., Rucci, E., Betran, V., Montezanti, D.M.: Implementando estrategias de resiliencia en una arquitectura basada en microservicios. In: Actas del XXIX Congreso Argentino de Ciencias de la Computación (CACIC 2023). pp. 467–488 (Oct 2023), `http://sedici.unlp.edu.ar/handle/10915/164996`
12. Yang, T., Lee, C., Shen, J., Su, Y., Feng, C., Yang, Y., Lyu, M.R.: Microres: Versatile resilience profiling in microservices via degradation dissemination indexing. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 325–337 (2024)
13. Yin, K., Du, Q.: On representing resilience requirements of microservice architecture systems. International Journal of Software Engineering and Knowledge Engineering **31**(06), 863–888 (2021)