UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Programa de Apoyo al Egreso de Profesionales en Actividad

TÍTULO: Análisis de patrones de resiliencia en una arquitectura basada en microservicios

AUTOR: APU Sergio Leonel Suarez

DIRECTOR ACADÉMICO: Dr Diego Montenzanti y Dr Enzo Rucci

DIRECTOR PROFESIONAL: Ing. Victor Betran

CARRERA: Licenciatura en Sistemas

Resumen

En el desarrollo de software, la resiliencia es uno de los aspectos no funcionales más importantes, especialmente en grandes compañías, ya que el impacto de las fallas tiene relación directa con su negocio. Por lo tanto, en los sistemas basados en microservicios surgen los patrones de diseño para resiliencia, los cuales permiten la implementación de estrategias para el manejo de estas fallas y la mitigación de sus efectos negativos. Por este motivo, en esta tesina se propuso como objetivo analizar el comportamiento de una serie de patrones utilizados para proveer resiliencia frente a diversos fallos capaces de afectar el funcionamiento del ecosistema de microservicios de la empresa PedidosYa.

Palabras Clave

Microservicios, Resiliencia, PedidosYa, Patrones de Diseño, Timeout, Retry, Circuit Breaker, Bulkhead, Combinación de Patrones, Requerimientos no funcionales, Tolerancia a fallas, Sistemas distribuidos.

Conclusiones

Habiendo analizado el comportamiento de Niles y sus microservicios asociados ante un conjunto de escenarios típicos de fallas, tanto en ausencia como en presencia de distintos patrones de resiliencia, se considera que se ha cumplido con el objetivo planteado originalmente en esta tesina. La implementación de cada uno de estos patrones de resiliencia en el microservicio Niles se encuentran en producción en el ecosistema de PedidosYa, logrando ser uno de los componentes más utilizados y robustos dentro de la compañía.

Trabajos Realizados

Inicialmente, se estudió la arquitectura de microservicios de PedidosYa, para comprender diferentes escenarios típicos de fallos. Luego, se analizó el funcionamiento del microservicio Niles. detallando su operación y los servicios de los que depende. Se hizo foco en el manejo de errores vinculados con Niles, y se estudiaron los patrones que se utilizan para la resiliencia, abarcando definición, conceptos asociados, funcionamiento e implementación dentro de la compañía. Por último, se obtuvieron resultados experimentales que permitieron analizar y evaluar el impacto de la aplicación de los patrones en cuestión. En particular, se analizó el comportamiento de Niles en presencia de fallos tanto, con y sin la aplicación de los patrones estudiados, de manera de poder cuantificar su incidencia

Trabajos Futuros

Por un lado, se planea continuar estudiando patrones habituales para la resiliencia en microservicios, tales como *Throttling* y *Queue-Based Load Leveling.*

Por otro lado, se propone impulsar la aplicación de patrones de resiliencia en toda la compañía, así como también la práctica de *Chaos Engineering*

Fecha de la presentación: Diciembre de 2022

Resumen

En lo que respecta a soluciones de software, los últimos años han sido testigos de un incremento en la implementación de arquitecturas de microservicios para dar respuesta a algunas de las limitaciones de los sistemas monolíticos tradicionales. En los sistemas basados en microservicios, el impacto de las fallas que ocurren y que se propagan por la cadena de dependencias tiene relación directa con el negocio de una empresa. Por lo tanto, la resiliencia es uno de los aspectos no funcionales más importantes, especialmente en grandes compañías. Debido a esto han surgido los patrones de diseño para resiliencia, los cuales permiten la implementación de estrategias para el manejo de las fallas y la mitigación de sus efectos negativos. Por este motivo, en esta tesina se propuso como objetivo analizar el comportamiento de una serie de patrones utilizados para proveer resiliencia frente a diversos fallos capaces de afectar el funcionamiento del ecosistema de microservicios de la empresa PedidosYa.

A lo largo de este trabajo, se estudió parte de la arquitectura de microservicios de PedidosYa, comprendiendo diferentes escenarios típicos de fallos que afectan a estas arquitecturas. En particular, se analizó el microservicio Niles (que es el encargado de retornar el menú de un restaurante), detallando su operación y los servicios de los que depende para cumplir su funcionalidad. Se hizo foco en el tratamiento de errores vinculados con Niles, y se estudiaron los patrones que son frecuentemente utilizados para la resiliencia, abarcando su definición, conceptos asociados, su funcionamiento y su implementación dentro de PedidosYa. Posteriormente, se obtuvieron resultados experimentales que permitieron analizar y evaluar el impacto de la aplicación de los patrones en cuestión. En particular, se analizó el comportamiento de Niles en presencia de fallos tanto, con y sin la aplicación de los patrones estudiados, de manera de poder cuantificar su incidencia.

Habiendo analizado el comportamiento de Niles y sus microservicios asociados ante un conjunto de escenarios típicos de fallas, tanto en ausencia como en presencia de distintos patrones de resiliencia, se considera que se ha cumplido con el objetivo planteado originalmente en esta tesina. La implementación de cada uno de estos patrones de resiliencia en el microservicio Niles se encuentran en producción en el ecosistema de PedidosYa, logrando ser uno de los componentes más utilizados y robustos dentro de la compañía.

Índice

Índice de figuras

Lista de acrónimos

API: interfaz de programación de aplicaciones (*application programming interface*)
CI/CD: integración continua y entrega continua (*continuous integration and continuous delivery*)
SLI: indicadores de nivel de servicio *(Service-level indicator)*
APM: application performance monitoring
DNS: sistema de nombres de dominio (Domain Name System)
AWS: Amazon Web Services
CPU: unidad de proceso central (Central Processing Unit)
JSON: notación de objeto de JavaScript (JavaScript Object Notation)
URL: localizador de recursos uniforme (Uniform Resource Locator)
HTTP: protocolo de transferencia de hipertexto (Hypertext Transfer Protocol)

Capítulo 1

Introducción

En primer lugar, se presenta la motivación de esta tesina (Sección 1.1). Luego se enuncian los objetivos y la metodología a emplear (Sección 1.2) junto a los resultados obtenidos (Sección 1.3). Por último, se describe la organización del documento (Sección 1.4).

1.1. Motivación

Previo a la denominada “era del Cloud Computing", las arquitecturas monolíticas representaban la forma tradicional de diseñar el software de negocio en empresas desarrolladoras de sistemas informáticos [1]. Sin embargo, la constante demanda de soluciones flexibles y escalables, sobre los cuales la velocidad de entrega debe ser llevada al límite, evidencia los problemas y desventajas de estas arquitecturas [2]:

- Aumento de complejidad: a medida que los sistemas monolíticos crecen, aumenta su complejidad, lo que dificulta la tarea de agregar nueva funcionalidad, analizar flujos existentes o incluso eliminar código.
- Lentitud del proceso de desarrollo, en cuanto a la incorporación de nuevas características y su ensamblado con la funcionalidad existente.
- Lentitud en el proceso de instalación y, por consiguiente, complejidad para regresar a versiones previas en caso de fallos.
- Fuerte dependencia con la tecnología seleccionada para su desarrollo, debido a que el componente monolítico se encuentra desarrollado en un único lenguaje de programación, por lo que su discontinuación (parcial o total) podrían afectar al sistema completo.

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. Si bien no existe una definición precisa de este estilo arquitectónico, existen ciertas características comunes en torno a la organización, la capacidad empresarial, la implementación automatizada y el control descentralizado de lenguajes y datos [3].

Sin embargo, 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 de las desventajas comunes de las arquitecturas de microservicios 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 incertidumbre sobre el momento adecuado para adoptar este enfoque, la necesidad de mantener la consistencia, la dificultad en la monitorización del estado de los componentes del ecosistema y la mayor exigencia para gestionar la seguridad.

La resiliencia es uno de los aspectos no funcionales más buscados en los sistemas, especialmente en los de gran escala [4]. El manejo de las fallas se torna una cuestión fundamental, ya que su impacto tiene relación directa con el negocio de una empresa, dado que cada segundo en que el servicio no está disponible se traduce en pérdidas económicas [5]. Es por ello que entra en juego el concepto de patrones de diseño para la resiliencia entre microservicios, que se utilizan en la implementación de estrategias para lidiar con estas fallas y mitigar sus consecuencias negativas.

En el desarrollo de este trabajo, se propone realizar 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 los mencionados patrones de resiliencia y las mejoras obtenidas en cuanto a conversiones de negocio[1] y otras métricas al aplicarlos. También se analizará la posible combinación de patrones con el fin de lograr un sistema con un alto grado de cobertura frente a fallos.

1.2. Objetivos y metodología

El objetivo general de este trabajo consiste en analizar el comportamiento de una serie de patrones utilizados para proveer resiliencia en arquitecturas de microservicios, frente a diversos fallos típicos capaces de afectar el funcionamiento del ecosistema de la empresa PedidosYa.
Para lograr el cumplimiento de este objetivo general, se abordará la concreción de los siguientes objetivos específicos:

- Comprender diversos escenarios típicos de fallos que se pueden presentar en arquitecturas basadas en microservicios.
- Presentar los patrones frecuentemente utilizados para implementar estrategias de resiliencia frente a los escenarios de fallos previamente descritos.
- Comprender los aspectos fundamentales que se requieren para una implementación eficiente de cada uno de los patrones de resiliencia en un entorno real.
- Evaluar la aplicación de los distintos patrones en diversas situaciones reales en las que se presentan fallos, e incluso la factibilidad de aplicar una combinación de ellos frente a situaciones complejas.
- Poner de manifiesto la necesidad de aplicar técnicas de resiliencia mediante la comparación del comportamiento del sistema real frente a fallos, tanto en presencia de patrones de resiliencia como cuando no se ha implementado ninguna estrategia al respecto.

La metodología que se lleva a cabo es idéntica para cada uno de los patrones a analizar. Se inicia con una explicación del patrón y conceptos asociados, su funcionamiento,

[1] La tasa de conversión es el porcentaje de personas que han visitado un sitio web y han llevado a cabo un objetivo específico.

implementación, una demostración de su aplicación en un caso concreto, una posible combinación de patrones y las métricas resultantes.

1.3. Resultados obtenidos

- Documentación del diseño e implementación de diversos patrones de resiliencia en una arquitectura de microservicios concreta. Obtención de métricas sobre su funcionamiento.
- Cuantificación de las ventajas y desventajas de implementar patrones de resiliencia y aplicarlos en una arquitectura de microservicios, para dar tratamiento a las fallas que se presentan.
- Análisis de la conveniencia de implementar patrones de resiliencia en una arquitectura de microservicios concreta.

1.4. Organización del documento

A continuación se describe la forma en que está organizado el resto del presente documento.

Capítulo 2: se presenta el marco teórico, definiendo el concepto de arquitectura de microservicios y sus principales características, ventajas y desventajas. Además, se focaliza sobre la capacidad de recuperación frente a fallas, es decir, la resiliencia.

Capítulo 3: se definen diferentes estrategias de resiliencia en arquitecturas de microservicios, que se utilizan dentro del ecosistema de Pedidos Ya. En particular, se expone el caso de uno de los microservicios más importantes, sus interacciones con otros microservicios y la aplicación de los patrones mencionados, ya sea en forma aislada o combinada.

Capítulo 4: se presenta el trabajo experimental realizado y los resultados obtenidos para evaluar la efectividad de los patrones aplicados en el caso de estudio elegido.

Capítulo 5: se presentan las conclusiones de esta tesina y las posibles acciones a implementar en un futuro.

Capítulo 2

Marco teórico

En primer lugar, se describen a las arquitecturas basadas en microservicios (Sección 2.1). En particular, se realiza una definición de esta arquitectura, una comparación con la arquitectura monolítica, y luego se detallan las características y beneficios de la arquitectura basada en microservicios. Luego se enuncian los desafíos de los microservicios (Sección 2.2), y cómo la resiliencia toma importancia en estas arquitecturas (Sección 2.3). A continuación, se enumeran los fallos más comunes en esta clase de arquitecturas de software (Sección 2.4). Para finalizar el capítulo, se introduce el concepto de patrones de resiliencia (Sección 2.5).

2.1. Arquitecturas basadas en microservicios

En este capítulo se describen diferentes conceptos relacionados a las arquitecturas basadas en microservicios, de forma de brindar las definiciones y el marco conceptual necesarios para comprender de manera más adecuada el trabajo presentado.

2.1.1. Tendencia del software como servicio

Hace algunas décadas, las arquitecturas monolíticas representaban la forma tradicional de diseñar el software de negocio en empresas desarrolladoras de sistemas informáticos.
Los sistemas distribuidos han ido adoptando soluciones más cercanas al tipo de “grano fino”, cambiando las pesadas aplicaciones monolíticas por otras basadas en pequeños microservicios [6], debido a la constante demanda de sistemas más flexibles y escalables, donde la entrega de funcionalidades debe ser rápida.

En la Figura 2.1 se puede observar la transformación de un sistema monolítico en una arquitectura de microservicios.

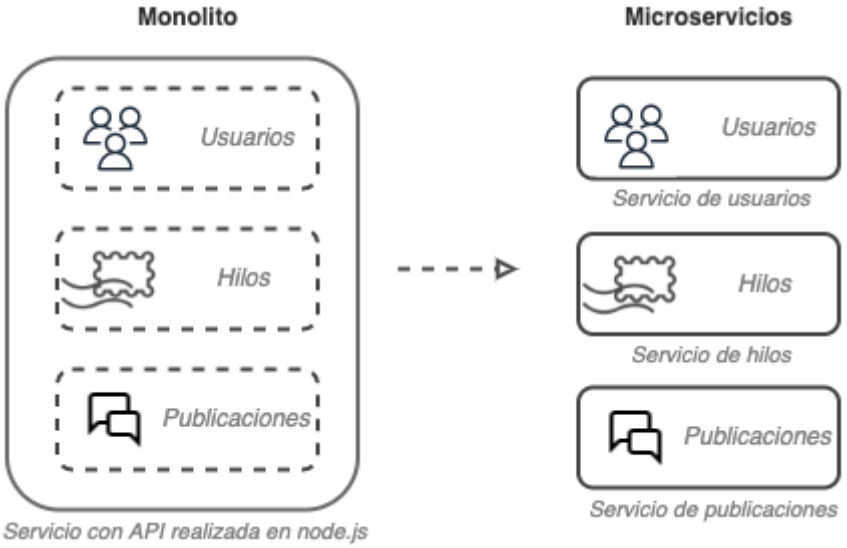

Figura 2.1 - Transformación de una aplicación monolítica en microservicios

El intento por satisfacer las demandas de la industria sobre la rápida entrega de soluciones de software ha contribuido a mejoras en la automatización de la infraestructura, en la forma de ejecutar pruebas y en las diferentes técnicas de entrega continua de software.

2.1.2.Definición de arquitectura de microservicios

Los microservicios son un enfoque arquitectónico y organizativo para el desarrollo de software, en el que éste está compuesto por pequeños servicios independientes que se comunican a través de APIs bien definidas, y donde los propietarios de estos servicios son pequeños equipos independientes [7].

Las arquitecturas de microservicios permiten que las aplicaciones sean más fáciles de escalar y más rápidas de desarrollar, facilitando la innovación y acelerando los tiempos de comercialización de las nuevas características.

El principal objetivo de los sistemas basados en microservicios es descomponer grandes proyectos de software en pequeñas unidades desacopladas que se comunican entre sí mediante una interfaz sencilla. Constituye un diseño arquitectónico contrario al tradicional enfoque monolítico sobre las aplicaciones, en el que todo se crea en una única pieza [8].

2.1.3.Arquitectura monolítica en comparación con la de microservicios

Con las arquitecturas monolíticas, todos los procesos están estrechamente asociados y se ejecutan como un solo servicio. Esto significa que, si un proceso de una aplicación experimenta un pico de demanda, se debe escalar toda la arquitectura. Agregar o mejorar las características de una aplicación monolítica se vuelve más complejo a medida que crece la base de código. Esta complejidad limita la experimentación y dificulta la implementación de nuevas ideas. Las arquitecturas monolíticas aumentan el riesgo de no-disponibilidad de

la aplicación, ya que muchos procesos dependientes y estrechamente vinculados aumentan el impacto de un error en un proceso.

Con una arquitectura de microservicios, una aplicación se crea a partir de componentes independientes, que ejecutan cada proceso de la aplicación como un servicio. Estos servicios se comunican a través de una interfaz bien definida mediante APIs ligeras. Los servicios se crean de acuerdo con las necesidades de la empresa, y cada servicio desempeña una sola función. Debido a que se ejecutan de forma independiente, cada servicio se puede actualizar, implementar y escalar para satisfacer la demanda de funciones específicas de una aplicación.

En la Figura 2.2 se puede observar cómo difieren las estructuras en arquitecturas monolíticas y de microservicios [9].

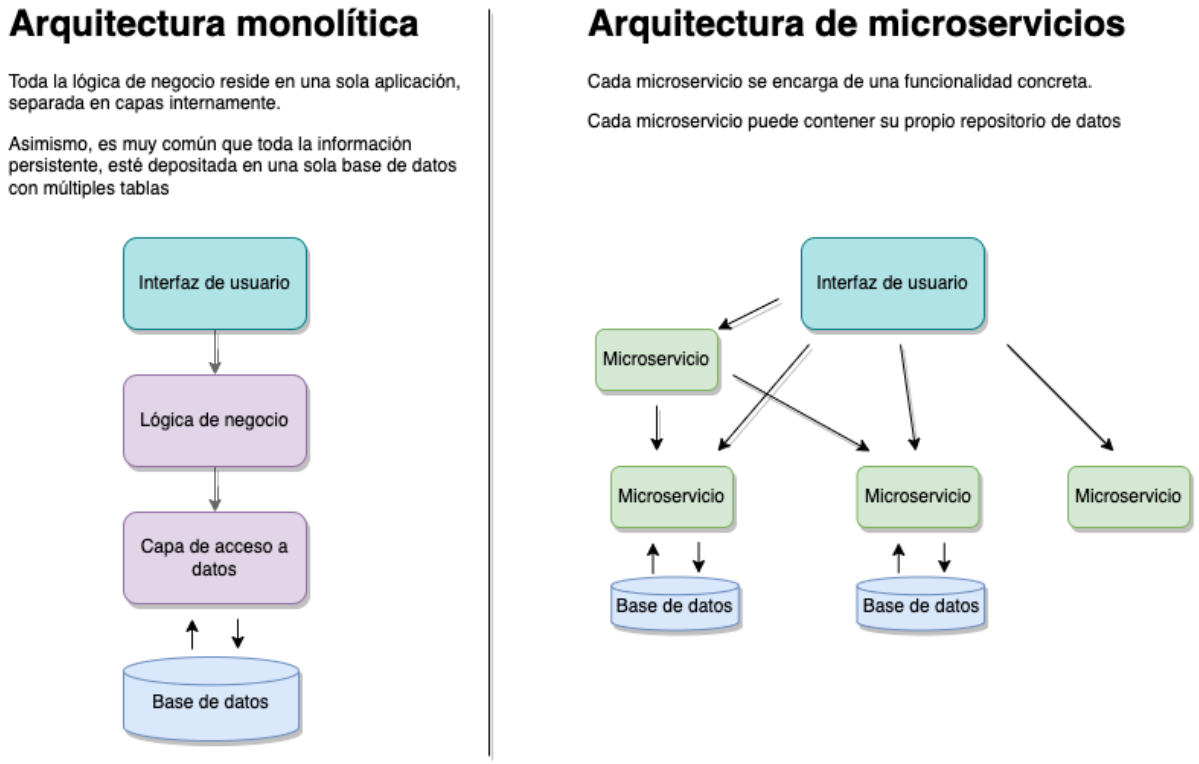

Figura 2.2 - Comparación entre una arquitectura monolítica y una arquitectura de microservicios

2.1.4.Características de los microservicios

Si bien no existe una definición precisa de este estilo arquitectónico, existen ciertas características comunes en torno a la organización, la capacidad empresarial, la implementación automatizada y el control descentralizado de lenguajes y datos.
Se listan algunas de las principales características de los microservicios [10]:

- Son desarrollados en base a una capacidad de negocio concreta, ofreciendo una funcionalidad única.

- Deben ser tratados como una “caja negra”, ya que su implementación no requiere ser conocida por ningún servicio externo, y exponen su funcionalidad mediante uno o más puntos de acceso, más conocidos como *endpoints*.
- Los límites entre los distintos servicios están claramente definidos, lo que permite lograr independencia respecto de la implementación de cada servicio, derivando a su vez en menor acoplamiento y mayor cohesión.
- Al comunicarse únicamente a través de una interfaz, pueden ser mantenidos, desplegados, desarrollados y probados de manera independiente.
- Son escritos y mantenidos por un equipo reducido de programadores.
- No es necesario que los servicios compartan el mismo *stack* tecnológico, las librerías o los frameworks.

2.1.5.Beneficios de los microservicios

Las arquitecturas de microservicios ofrecen múltiples beneficios [11]:

- Permiten la entrega y el despliegue continuos de aplicaciones grandes y complejas: es más simple agregar funcionalidades, realizar modificaciones e iterar el microservicio[2] debido a que éste es más pequeño. En consecuencia, el tiempo de instalación de los microservicios suele ser menor, lo que a su vez reduce el tiempo de entrega.
- Los servicios son pequeños y fáciles de mantener, ya que el código es más sencillo de entender para un desarrollador. La pequeña base de código no ralentiza el IDE o la herramienta utilizada para escribir código, lo que genera que los desarrolladores sean más productivos. Además, el desarrollo inicial de un servicio suele requerir menos tiempo que el correspondiente al de un gran monolito, lo que acelera la etapa de implementación.
- Los servicios pueden escalar de forma independiente: cuando un servicio dentro de la arquitectura necesita más recursos, como CPU o memoria, es relativamente simple añadir una nueva instancia sólo del microservicio en cuestión. En cambio, en un sistema monolítico, todos los recursos y funcionalidades se encuentran concentradas en un solo lugar.
- Permiten la autonomía de los equipos: un grupo reducido de personas son los responsables de uno o varios microservicios. Este grupo se encarga de escribir el código y de sus pruebas unitarias, la instalación y el monitoreo de los mismos.
- Permiten una fácil experimentación y adopción de nuevas tecnologías: la arquitectura de microservicios elimina cualquier compromiso a largo plazo con una pila de tecnología. En principio, al desarrollar un nuevo servicio, los desarrolladores son libres de elegir el lenguaje y los *frameworks* que mejor se adapten al mismo, sin condicionamientos debidos a decisiones que puedan haberse tomado en el pasado. De hecho, puede resultar práctico reescribir pequeños servicios utilizando mejores lenguajes y tecnologías.Si una prueba de una nueva tecnología falla, se puede desechar ese trabajo sin arriesgar todo el proyecto, a diferencia de lo que ocurre con las arquitecturas monolíticas, en las cuales las elecciones tecnológicas iniciales restringen severamente las posibilidades de usar diferentes lenguajes y *frameworks* en el futuro.
- Por último, presentan un mejor comportamiento en cuanto al aislamiento de fallas. Por ejemplo, si un servicio no cuenta con suficiente memoria, este problema sólo afecta a ese servicio en particular, con lo cual los otros componentes de la arquitectura pueden

[2] Iterar el microservicio hace referencia al agregado progresivo de funcionalidad.

seguir atendiendo solicitudes con normalidad. En tanto, un componente de una arquitectura monolítica que se comporta incorrectamente es capaz de derribar todo el sistema.

En la Figura 2.3 se puede observar cómo dentro de una empresa existen múltiples equipos autónomos. Cada equipo puede administrar varios microservicios, donde cada uno tiene su propio repositorio de código. Para realizar la instalación en diferentes ambientes (ambiente de prueba o productivo), se utilizan herramientas de integración continua y entrega continua (CI/CD).

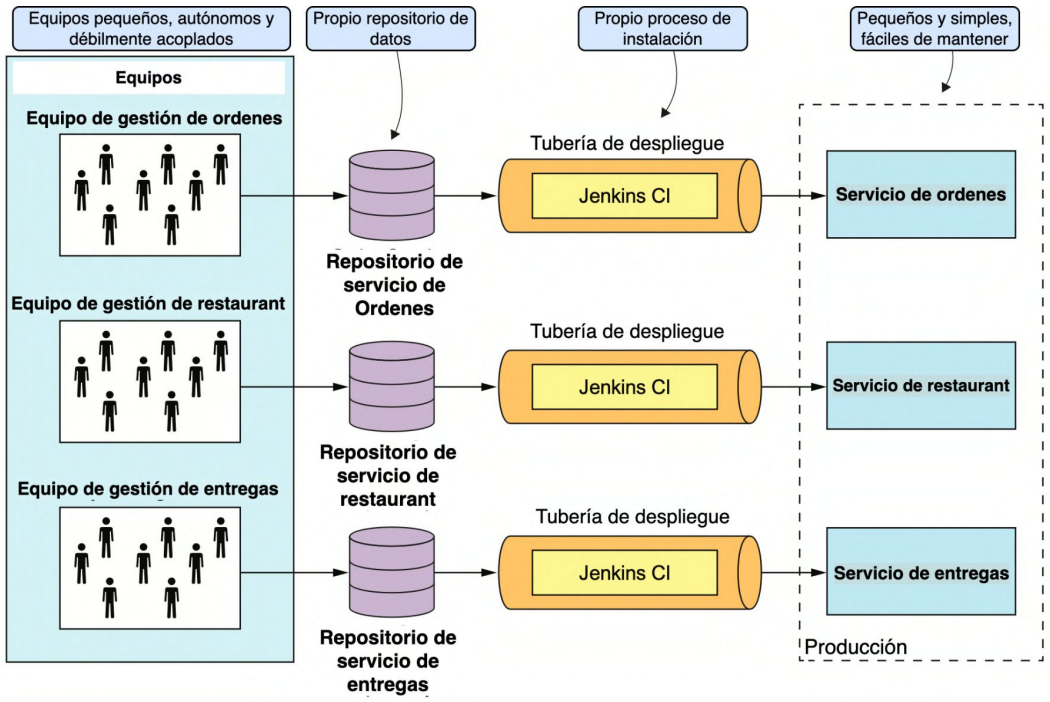

Figura 2.3 - Proceso de instalación de un microservicio

2.2. Desafíos de los microservicios

La arquitectura de los microservicios también presenta una serie de desventajas y desafíos. Por ejemplo, cuando ocurre un fallo, podrían resultar afectados tanto el microservicio que lo experimenta como los otros que dependen de él. Algunas de las desventajas o desafíos más comunes se listan a continuación [12]:

- Complejidad de los sistemas distribuidos: los desarrolladores deben ser capaces de lidiar con la complejidad que requiere un sistema distribuido. Entre los desafíos, se pueden mencionar la implementación de mecanismos de comunicación entre los distintos servicios, de forma tal que sean capaces de manejar fallas parciales (fallos temporales por lentitud o errores en la red).
- Ausencia de un método específico para la descomposición: no existe un método concreto y bien definido para descomponer un sistema en servicios. El trabajo de diseño

que requiere la etapa de descomposición es extremadamente desafiante, ya que un mal diseño puede provocar el desaprovechamiento de los beneficios de esta arquitectura.

- Incertidumbre sobre el momento adecuado para adoptar este enfoque: en una primera etapa, pueden no manifestarse los problemas que se resolverán mediante una implementación basada en microservicios, por lo que una solución monolítica parece la más natural. Sin embargo, en etapas más avanzadas, cuando las aplicaciones comienzan a crecer rápidamente en tamaño y complejidad, empieza a tener sentido descomponerlas funcionalmente en un conjunto de microservicios. El problema consiste en determinar en qué punto del ciclo de vida de la aplicación se debe comenzar a utilizar el enfoque basado en microservicios.
- Necesidad de mantener la consistencia frente a los problemas que surgen cuando una gran cantidad de servicios independientes interactúan entre sí para satisfacer un objetivo en común.
- Dificultad en la monitorización del estado: el hecho de tener múltiples sistemas desplegados al mismo tiempo complejiza considerablemente la tarea de detección y seguimiento de fallas.
- Mayor exigencia para gestionar la seguridad: cuantos más servicios independientes coexisten, mayor es el desafío de garantizar la seguridad, debido a su encapsulamiento y posibles interacciones.

2.3. La resiliencia como requerimiento no funcional

El concepto de resiliencia, aplicado a una arquitectura de microservicios, hace referencia a la capacidad que tiene un sistema para continuar brindando determinados niveles de calidad de servicio, aun cuando una parte del mismo esté fuera de servicio temporal o permanentemente. Se dice que una arquitectura es resiliente cuando evita los fallos en cascada y tiene la capacidad de mantener sus datos en un estado consistente [13].
La resiliencia es uno de los aspectos no funcionales más buscados en los sistemas, especialmente en los de gran escala. El manejo de las fallas se torna una cuestión fundamental, ya que su impacto tiene relación directa con el negocio de una empresa, dado que cada segundo en que un servicio no está disponible se traduce en pérdidas económicas.

2.4. 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 situaciones son comunes a todos los sistemas de software (como por ejemplo una base de datos inconsistente, un recurso agotado o una instalación deficiente), existen casos que son propios de los sistemas distribuidos [14]. A continuación se listan los fallos más comunes en arquitecturas de microservicios:

2.4.1. Errores o lentitud en la red

En un ecosistema de microservicios, la interacción entre múltiples servicios es muy frecuente. Ante un error de red, un microcorte, el daño de un componente(por ejemplo el

servidor de DNS), o la congestión en la transferencia de datos, el ecosistema resulta afectado.

2.4.2. Picos de tráfico

Dado que los microservicios son autónomos e independientes, es posible definir sus límites de operación mediante métricas, como por ejemplo, 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.

2.4.3. 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 un servicio aleatorio, podría tener el mismo impacto que cualquier otro. 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.

2.5. Concepto de patrones para resiliencia

Al adoptar un enfoque basado en microservicios, existen diferentes categorías de patrones de diseño sobre los cuales se puede establecer una estructura sólida que permita al ecosistema escalar o evolucionar de manera segura.

Existen patrones de seguridad (como por ejemplo, Access Token), de consistencia de datos (SAGA), de instalación o *deployment* (Service Mesh o Sidecar), de comunicación (Backend for Frontend o Gateway), de *testing* (Consumer-side Contract Test), de *refactoring* (Anti-corruption layer) y de resiliencia, entre otros [15].

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 de resiliencia ayuda a construir una arquitectura resistente a fallos. Estos patrones de diseño para la resiliencia entre microservicios se basan en implementar estrategias para lidiar con estas fallas y mitigar sus efectos negativos.

Capítulo 3

Patrones de Resiliencia en una Arquitectura de Microservicios de PedidosYa

En primer lugar, se describe el ecosistema de microservicios en PedidosYa (Sección 3.1). Luego, se presenta el caso de estudio, haciendo foco en el microservicio Niles, y los servicios que son consumidos para cumplir su objetivo (Sección 3.2). Posteriormente, se analiza en detalle el tratamiento de fallos en Niles, estudiando una serie de patrones de resiliencia, para poder definirlos, implementarlos y aplicarlos al caso de estudio (Sección 3.3).

3.1. El ecosistema de microservicios de Pedidos Ya

PedidosYa[3] es una compañía uruguaya de delivery en línea con presencia en varios países de América Latina. Su sede central está ubicada en Montevideo, Uruguay, pero opera también en Argentina, Bolivia, Chile, Costa Rica, Panamá, Nicaragua, Paraguay, Guatemala, Perú, República Dominicana, Ecuador, Venezuela, El Salvador y Honduras. En la actualidad pertenece a Delivery Hero.

Desde su creación en 2009, PedidosYa ha tenido un rápido crecimiento. En 2019 se transformó en unicornio, al alcanzar una valoración de 1.000 millones de dólares, según sus múltiples puntos de ventas y la participación dentro del grupo Delivery Hero [16]. En la actualidad, PedidosYa procesa unas 4 millones de órdenes por semana, de las cuales aproximadamente la mitad provienen de Argentina.

PedidosYa cuenta con una arquitectura basada mayormente en microservicios, pero aún conserva algunas funcionalidades implementadas en dos sistemas monolíticos, que próximamente serán reemplazados para alcanzar un ecosistema completo basado en microservicios.

En cuanto al negocio, la empresa se encuentra dividida en lo que se conocen como verticales[4], tales como Restaurantes, Farmacias, Mercados y Bebidas. En la vertical que más ganancias genera a la empresa (Restaurantes), toda la lógica de negocio se encuentra dispersa en microservicios, de los cuales uno de los más importantes recibe el nombre de Niles.

[3] https://www.pedidosya.com/

[4] Vertical: Un segmento dentro de una industria que se compone de clientes y negocios similares.

En la Figura 3.1 se muestra el flujo completo, desde que un usuario ingresa al sistema, hasta que realiza su pedido y monitorea su entrega por medio de mapas web.

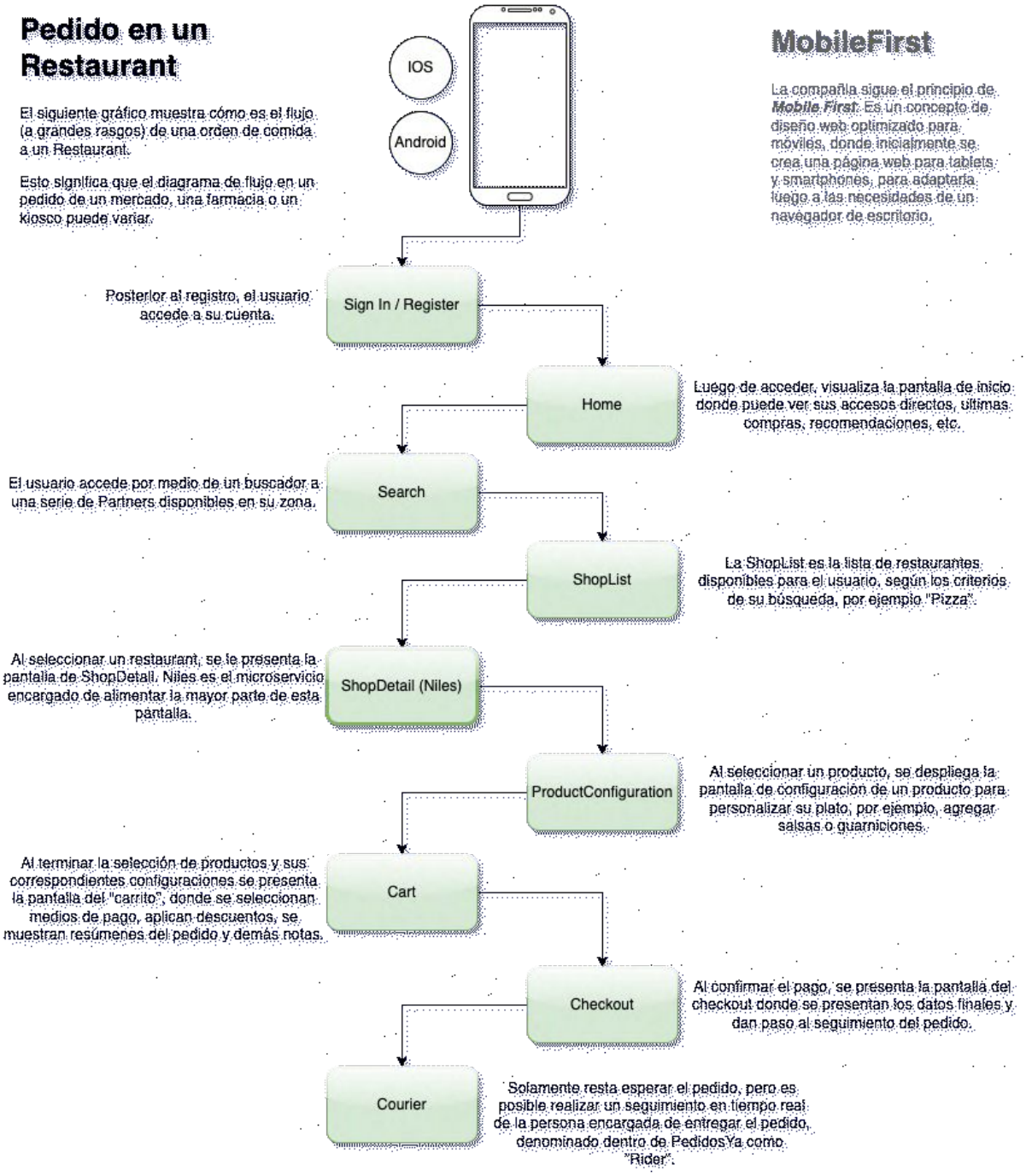

Figura 3.1 - Flujo de una orden en un restaurante

3.2. Caso de estudio: el servicio Niles

3.2.1. Niles

Niles[5] es el microservicio encargado 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 y aportando información básica para cada uno de ellos, como su nombre, descripción, imagen, precio y descuentos aplicables, y completando 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*[6], invocan a Niles para consultar el menú. Las aplicaciones móviles realizan sus requerimientos a través de un *proxy*, denominado *client-api-gateway*, el cuál se encarga de redireccionar parámetros, realizar validaciones de seguridad y distribuir carga, entre otras cosas. La interacción entre la aplicación móvil y Niles se ilustra en la Figura 3.2.

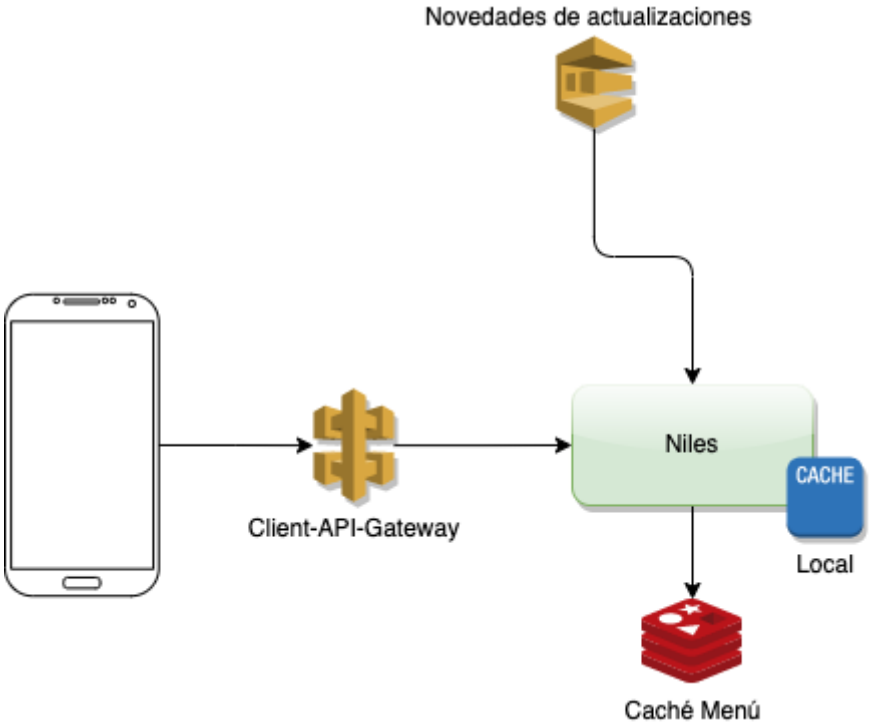

Figura 3.2 - Comunicación entre una aplicación móvil y Niles

El proceso de retorno de un menú es el siguiente:

- Niles recibe el requerimiento de menú para un restaurante.
- Se verifica la existencia del menú para el restaurante solicitado en la caché distribuida.

[5] Niles hace referencia al mayordomo de la *sitcom "The Nanny"*.
[6] ShopDetail: módulo de visualización de PedidosYa, para aplicaciones móviles, responsable de mostrar el menú de un restaurante.

- 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.
- En cualquiera de los casos anteriores, el menú se procesa aplicando diferentes filtros (validaciones de edad, stock disponible, ventanas de horario correctas) y ordenamientos.

Como se mencionó previamente, 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.

Con el objetivo de presentar un menú actualizado, Niles recibe eventos de novedades desde otros microservicios tales como *Items-service* y *Battlefront* (que se describen brevemente en la siguiente sección). Al momento de recibir una actualización, Niles procede a eliminar 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 3.3.

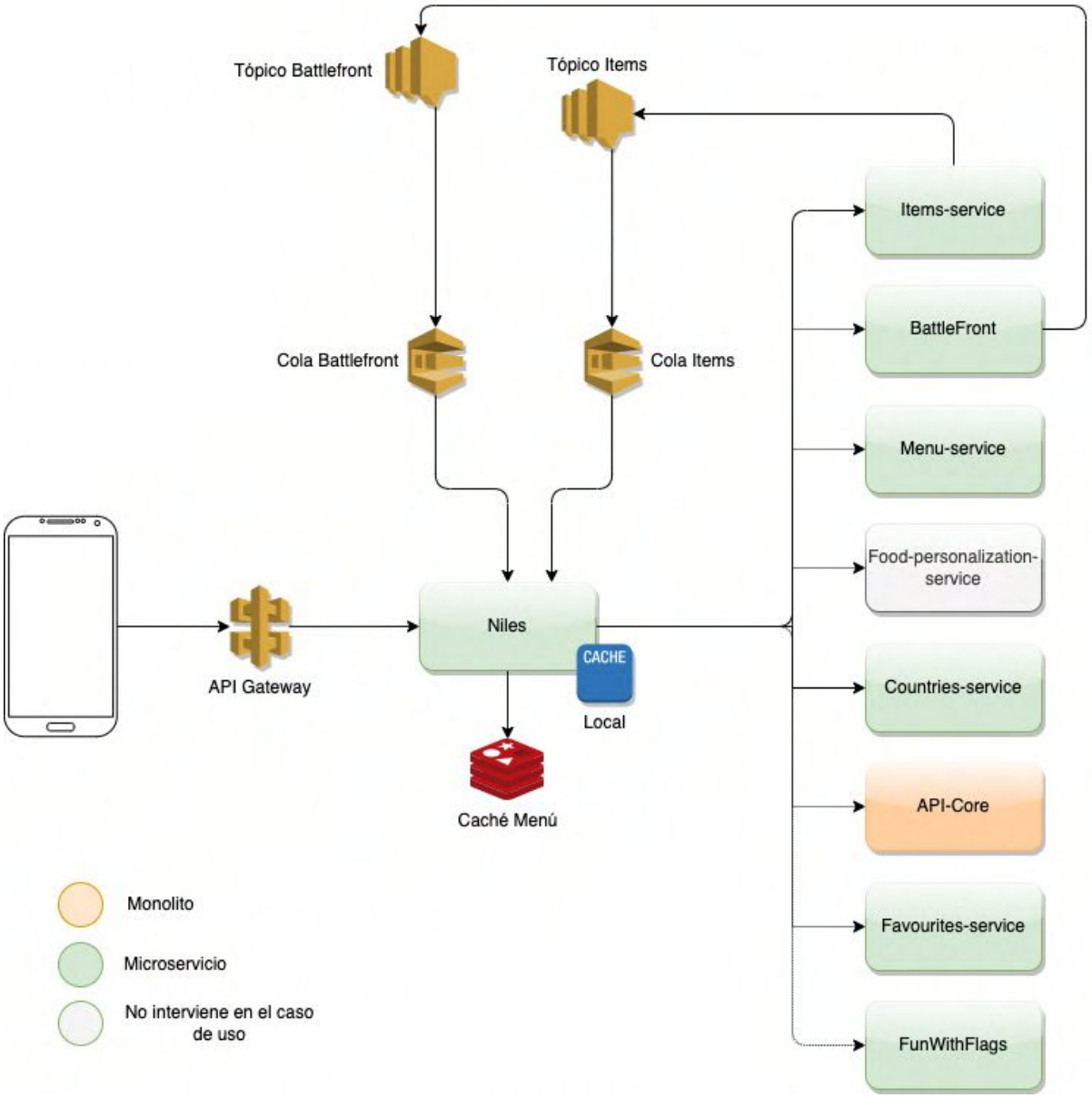

Figura 3.3 - Interacción de Niles con todos los demás componentes

3.2.2.Microservicios asociados

A continuación, se describen las principales funcionalidades de cada uno de los microservicios con los que interactúa Niles.

- Items-service: es el microservicio que alimenta al menú. *Items-service* es el encargado de proveer tanto las secciones de un menú, como los productos que lo completan y sus opciones de configuración.

Para cada sección, producto u opciones de productos, administra datos como el nombre, imagen, descripción y precio. Además maneja distintos estados, que varían según modificadores como lapsos de tiempo, stock y actualizaciones.

Ante cualquier cambio de los datos en un ítem, *Items-service* envía novedades utilizando tópicos, los cuales llegan a las colas de mensajes de Niles, quien se encarga de actualizar la caché centralizada.

En caso de que *Items-service* no funcione correctamente (debido a lapsos de caída o incidentes serios), Niles no puede proporcionar su funcionalidad completa. Si bien se cuenta con una caché centralizada que mantiene un espejo de la información de *Items-service*, la misma cuenta con un tiempo de vida de seis horas por lo que, ante una falla, podría mostrarse un menú desactualizado.

- Battlefront: es el servicio encargado de la administración de descuentos en la vertical de *food*[7]. Maneja los descuentos semanales por restaurante, los descuentos por promociones diarias por producto, los beneficios con respecto al costo de entrega (interactuando con el módulo de Pedidos Ya Plus[8]), y la administración de campañas publicitarias, como por ejemplo, mundiales de fútbol o recitales importantes.
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.

- Menú-service: es el microservicio que se encarga de proveer el puntaje de rating por productos. Si bien su nombre es ambigüo, y su objetivo pareciera ser el mismo que el de Niles, no es así. El rating sobre cada producto proporciona información sobre si el mismo será marcado como "Más vendido" dentro del menú. Esta información la genera a partir de la cantidad de órdenes generadas que contienen al producto.

- Food-personalization-service: es el microservicio que se ocupa de personalizar secciones dentro del menú. Según diferentes parámetros, genera secciones dinámicas basándose en patrones, como por ejemplo: la sección de productos más comprados en el último mes por el usuario en determinado restaurante.

- Countries-service: es el servicio que ofrece la lista de países, entregando información como su nombre, la zona horaria, la moneda y su abreviatura en código.

- API-Core: es uno de los dos sistemas monolíticos que existen hoy en el ecosistema de Pedidos Ya. Si bien el objetivo a nivel compañía es migrar a un ecosistema 100% de microservicios, aún queda funcionalidad importante en estos sistemas monolíticos.
Niles requiere interactuar con API-Core para recuperar la información de las categorías de comida, por ejemplo: pizzas, empanadas, hamburguesas.
Dentro del menú, estás categorías brindan información que sirve para decorar al producto, permitiendo realizar filtros en tiempo real dentro de la aplicación.

- 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, configuraciones de productos, etc.

[7] La vertical de restaurantes también es denominada vertical de *food*.
[8] Pedidos Ya Plus es un servicio de suscripción preferencial que permite a los usuarios acceder, a través de un pago mensual, a determinados beneficios en el uso de su cuenta.

Niles consume *Favourites-service*[9] para poder interactuar con la entidad de producto, es decir, poder agregar o remover productos a la lista de favoritos.

- FunWithFlags: es el servicio de la compañía que se encarga de funcionar como "Feature flag" [17]. Feature flag es una herramienta de software que permite utilizar banderas para modificar funcionalidades existentes. Se utiliza para realizar diferentes experimentos, con el fin de analizar el impacto de la utilización del menú. Dentro de Niles se utiliza, por ejemplo, para mostrar diferentes tipos de ordenamiento en cuanto a secciones y productos, habilitar funcionalidades como la oportunidad de administrar favoritos, mostrar campañas al inicio del menú, dar prioridad a productos con imágenes de alta calidad, etc.

3.3. Tratamiento de fallos en Niles

Los fallos en Niles son tratados mediante la implementación de patrones de resiliencia tal como se menciona en la sección 2.5. Concepto de patrones para resiliencia. Para explicar el tratamiento de fallos se deben tener una consideraciones generales (Sección 3.3.1), para luego entrar en el detalle de una serie de patrones listados a continuación:

- Patrón *Timeout* (Sección 3.3.2)
- Patrón *Retry* (Sección 3.3.3)
- Patrón *Circuit Breaker* (Sección 3.3.4)
- Patrón *Bulkhead* (Sección 3.3.5)

Luego de presentar la definición, la implementación y la aplicación en un caso de uso para cada uno de los patrones previamente mencionados, se evalúa la combinación de los patrones *Timeout, Retry* y *Circuit Breaker* (Sección 3.3.6).

3.3.1. Consideraciones generales

Dentro de PedidosYa, la mayoría de los microservicios están desarrollados en lenguaje Kotlin[10]. Una de las librerías más conocidas para realizar peticiones HTTP (*ktor-client-core*)[11], provee un cliente poco intuitivo para implementar limitaciones de tiempos. La siguiente línea de código demuestra cómo crear un cliente HTTP utilizando la librería *ktor-client-core*.

```
HttpClient() {}.get<String>(host = "google.com")
```

Para poder estandarizar buenas prácticas en cuanto al desarrollo de software, se implementó una librería interna llamada *peya-ktor-utils,* la cuál provee diferentes mecanismos de resiliencia. Esta librería se basa fuertemente en el uso de otra librería, realizada en Java, llamada resilience4j[12].

[9] favourites es la traducción inglesa de favoritos. Se utiliza esta variante debido a que existe un servicio obsoleto llamado favorites.
[10] https://kotlinlang.org/
[11] https://mvnrepository.com/artifact/io.ktor/ktor-client-core-jvm
[12] https://resilience4j.readme.io/

En la librería interna *peya-ktor-utils* se puede definir una lista de proveedores, donde cada proveedor es un decorador[13] que engloba una operación. Esta operación puede ser desde una petición HTTP hasta una lectura a una base de datos o una búsqueda en una caché.

Para la creación de un cliente HTTP, se debe crear una subclase de la superclase *AbstractClient*, configurando la resiliencia, y utilizando los métodos que provee la superclase. El siguiente fragmento de código muestra la implementación de la superclase AbstractClient.

```
abstract class AbstractClient(
    open val config: HttpClientConfiguration,
    open val resilience: ResilienceProviders) {

    protected fun get(
        uri: String,
        params: Map<String, Any?>
        headers: Map<String, Any?>
    )

    protected fun post(
        uri: String,
        params: Map<String, Any?>,
        headers: Map<String, Any?>,
        body: Any?
    )

    protected fun put(
        uri: String,
        params: Map<String, Any?>,
        headers: Map<String, Any?>,
        body: Any?
    )

    protected fun patch(
        uri: String,
        params: Map<String, Any?>,
        headers: Map<String, Any?>,
        body: Any?
    )

    protected fun delete(
        uri: String,
        params: Map<String, Any?>,
        headers: Map<String, Any?>,
    )
}
```

Como se puede observar en la clase *AbstractClient*, es necesario enviar dos parámetros al constructor de la misma:

[13] Hace referencia a una función que envuelve a otra.

1. El primero es una configuración, con datos como el host al que se realizará el requerimiento HTTP. En el siguiente fragmento de código se puede ver la creación del objeto de configuración.

```
HttpClientConfiguration(
    host = "items-service.live.peja.co"
)
```

2. El segundo parámetro es un objeto de tipo ResilienceProviders, el cual contiene una lista de proveedores.

Un microservicio está compuesto por un servidor, el cual se encarga de recibir los requerimientos entrantes. Este servidor emplea un *pool* de hilos para poder recibir requerimientos simultáneos. El número de requerimientos que pueden ser atendidos a la vez está limitado por la cantidad de hilos disponibles en el *pool*, como se puede observar en la Figura 3.4. A medida que se liberan hilos, son reutilizados para responder a nuevos requerimientos.

Figura 3.4 - Servidor y grupo de hilos

Cuando el servidor recibe un nuevo requerimiento, se utiliza un hilo para realizar el trabajo necesario. Como los microservicios son autónomos e independientes, y se encargan de una única funcionalidad concreta, es común que necesiten consultar y recuperar información desde otras entidades o recursos externos, ya sean bases de datos, cachés u otros microservicios.

Cada vez que un microservicio, una entidad o un recurso externo son consultados, el *thread* que realiza dicha consulta queda en espera (bloqueado), por lo que no puede atender nuevas peticiones, como se ilustra en la Figura 3.5:

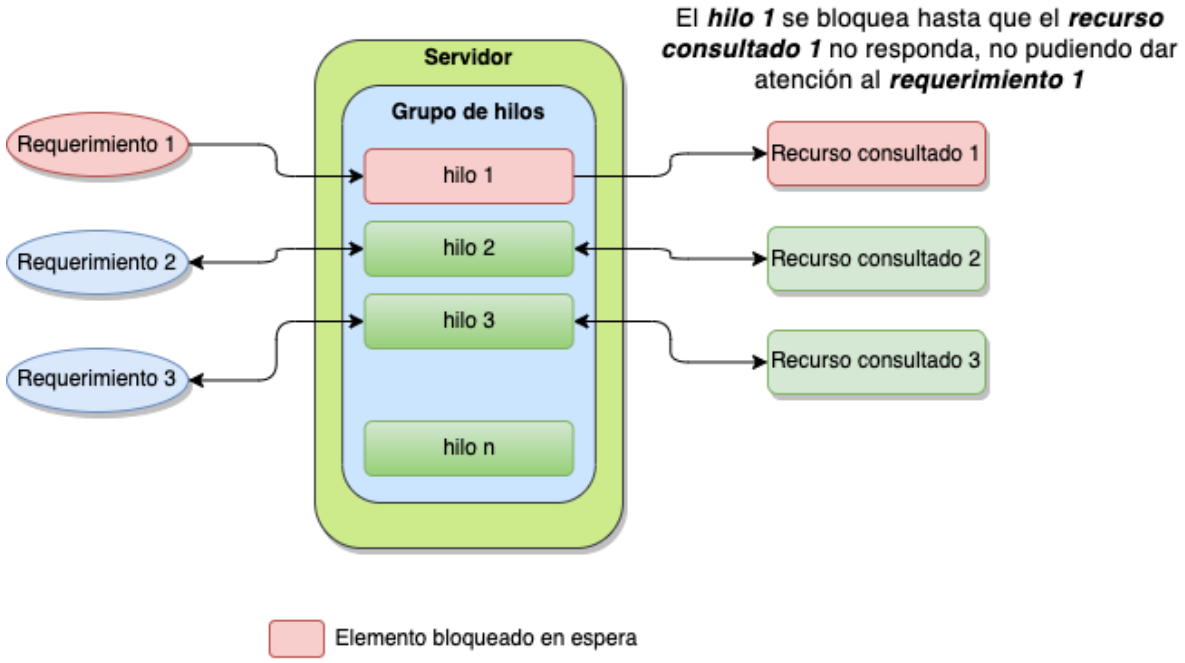

Figura 3.5 - Hilo bloqueado en espera

3.3.2. El patrón *Timeout*

3.3.2.1.Definición

En cualquier momento, el recurso o entidad consultada puede presentar degradación en sus tiempos de respuesta, debidos a inconvenientes por errores o lentitud en la red o en algún componente en la comunicación, o incluso a problemas internos.

En el caso de que no se establezca un límite de espera entre el servicio que consume y el recurso consumido que presenta anomalías, el hilo que atiende el requerimiento original puede quedar bloqueado indefinidamente. Esta situación eventualmente puede desembocar en que todos los *threads* queden bloqueados, o peor aún, que se empiecen a encolar peticiones a la espera de que se liberen hilos, tal y como se ilustra en la Figura 3.6:

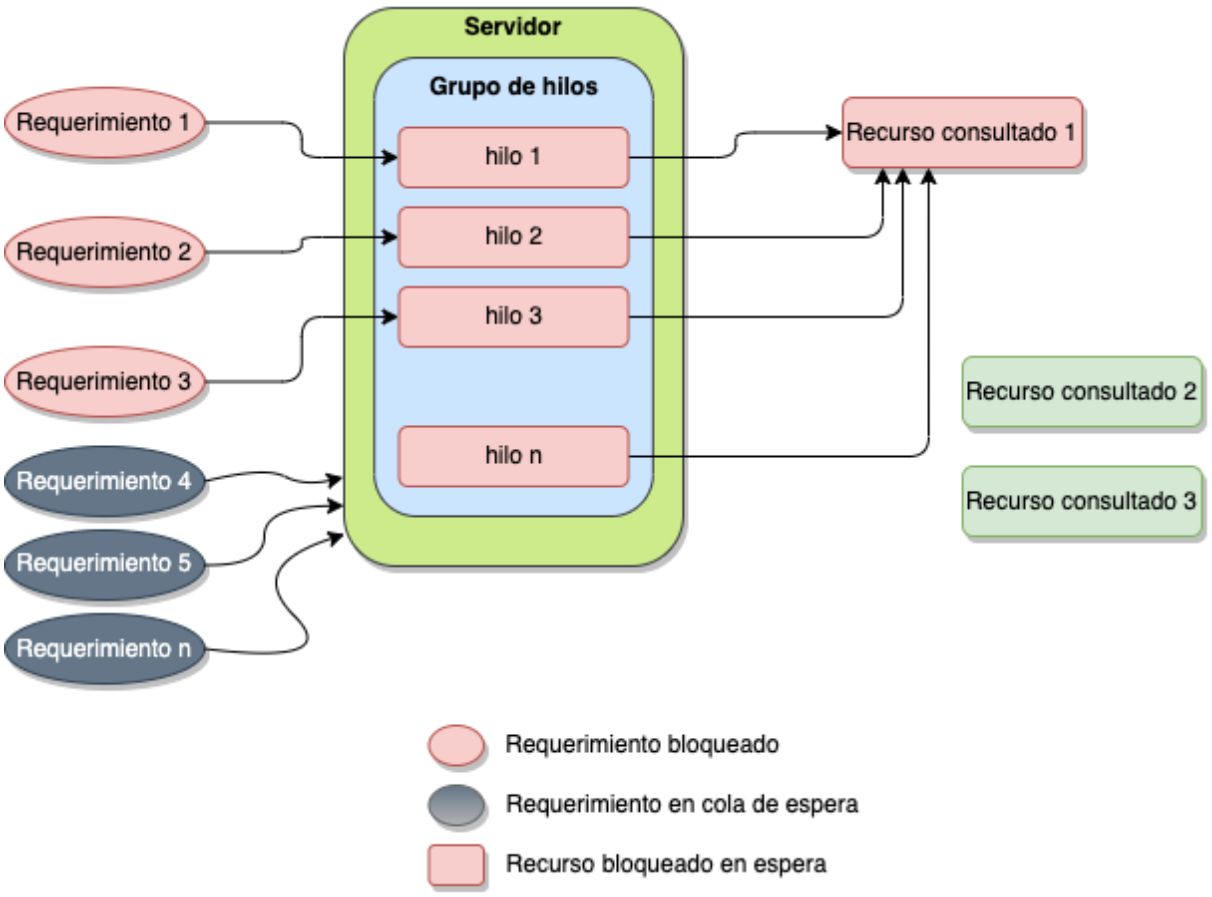

Figura 3.6 - Totalidad de hilos bloqueados

Al agotar todos los hilos disponibles, los nuevos requerimientos que arriban se alojan en una cola de espera. Los hilos en espera continúan consumiendo los recursos del servidor que atiende los requerimientos, pudiendo llegar incluso hasta el punto de saturarlo y/o de completar la capacidad de la cola causando, en el peor de los casos, la caída completa del servidor.

El patrón *Timeout* permite prevenir que los hilos queden bloqueados indefinidamente en espera. Una vez que el tiempo de espera se cumple, el *thread* involucrado es liberado para poder atender nuevos requerimientos [18].

Es importante aceptar que las demoras ocurren, y que incluso pueden hacerlo muy frecuentemente. Por ello resulta necesario definir una manera de tratarlas, estableciendo un tiempo de espera máximo cuando tengan que consumir recursos externos. De esta manera, el servicio consumidor corta la comunicación, liberando hilos y procediendo con un plan alternativo en caso necesario [19].

3.3.2.2. Implementación

La implementación del patrón *Timeout* consiste en definir un tiempo de espera al momento de consultar recursos externos, tales como una base de datos o un microservicio.
Muchos proveedores de servicios suelen proporcionar clientes para su consumo (por ejemplo, para realizar consultas a una bases de datos, para almacenamiento, etc.), y que en la propia implementación de estos clientes se definan mecanismos para establecer tiempos

de espera[14]. Sin embargo, en muchas ocasiones estos clientes no tienen la opción de definir un tiempo límite de espera, o éste es difícil de configurar.

Dentro de las estrategias que ofrece la librería *peya-ktor-utils*, se establece una manera de poder definir tiempos límites al consumo de recursos externos.
Para la implementación del patrón *Timeout*, es necesario agregar una instancia de TimeoutConfiguration a la lista de *resilience providers*, indicando el nombre definido para despachar métricas y *logs* a las aplicaciones de monitoreo, y el tiempo de espera límite, como se puede ver en la siguiente porción de código.

```
ResilienceProviders(
   providers = listOf(
       TimeoutPolicyFactory.create(
           TimeoutConfiguration(
               name = "timeout-cliente",
               timeoutDuration = Duration.ofMillis(500)
           )
       )
   )
)
```

Por último, al crear una clase que hereda de AbstractClient, se obtiene un cliente que implementa el patrón *Timeout*, como se muestra a continuación en el siguiente fragmento de código, donde resaltado en verde se puede ver como se le envía un parámetro al constructor del cliente HTTP, el cual contiene la configuración para implementar el patrón *Timeout*.

```
class ItemsClient(
   override val config: HttpClientConfiguration,
   override val resilience: ResilienceProviders
) : AbstractClient(config, resilience) {
   suspend fun getSections(partnerId: Int) {
       val uri = "/v2/sections?partnerId=$partnerId"
       return this.get(uri)
   }
}
```

En la Figura 3.7 se puede observar la forma en la que el sistema reporta la ocurrencia de una operación que excede el tiempo definido como máximo.

[14] Amazon es un ejemplo de proveedor de clientes: https://aws.amazon.com/es/sdk-for-java/

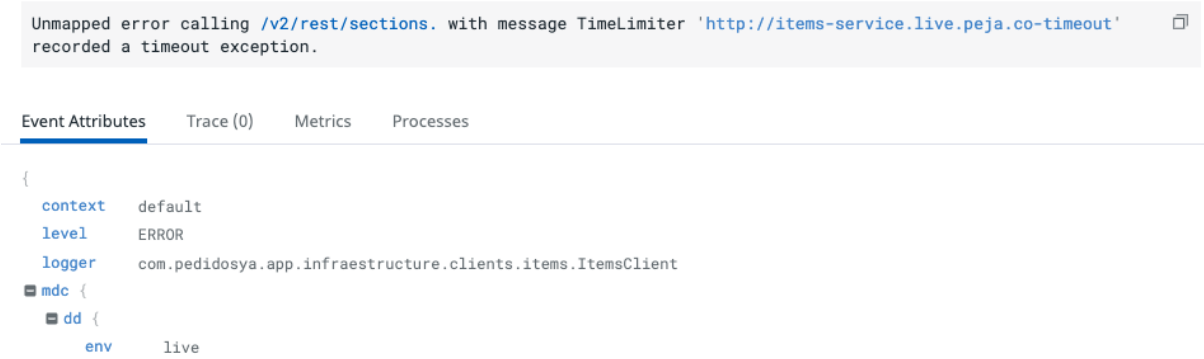

Figura 3.7 - Líneas de *log* resultantes frente a una petición que ha alcanzado el límite de tiempo

Un aspecto destacable es que la definición de los tiempos de espera no es arbitraria, sino que depende de cada microservicio, por lo que resulta necesario conocer el SLI que provee el mismo. El SLI hace referencia a las métricas que miden el rendimiento del servicio. En determinados casos, el dueño del servicio no es capaz de proporcionar esta información, y en tal situación, el consumidor debe de poder analizar diferentes métricas tales como latencia[15], *error rate*[16] y tráfico que recibe. Está información la ofrecen herramientas conocidas como APM, las cuales son aplicaciones para monitorear el desempeño de un servicio, como por ejemplo DataDog[17] o NewRelic[18] [20].

3.3.2.3. Aplicación al caso de estudio

El hecho de no definir tiempos de espera máximos en operaciones que consumen recursos externos es una mala práctica; por lo tanto, resulta necesario configurar límites de tiempo acordes a cada una de ellas.
Como se explicó previamente, Niles entrega el menú de un restaurante, realizando consultas de datos a diferentes microservicios. 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 al servicio *favourites-service* es (de alguna manera) la menos importante, ya que se puede presentar un menú completo sin la información que nos indica si un producto fue marcado como favorito.

Se definirán 3 suposiciones para demostrar la aplicación al caso de estudio:

- Suposición 1: el servicio *Items-service* tiene definido un SLI, donde la latencia, medida con la métrica *p95*[19], es de un promedio de 50 milisegundos. Esto significa que el 95% de los requerimientos recibidos, se resuelven en una duración menor a los 50 milisegundos.

[15] Métrica que representa los tiempos de respuestas en el servicio, los cuales son medidos en percentiles, tales como p50, p95, p99.
[16] Métrica que define el porcentaje promedio de errores en las peticiones recibidas.
[17] https://www.datadoghq.com/
[18] https://newrelic.com/
[19] p50, p95 y p99 son algunas de las métricas de latencia, denominadas percentiles.

- Suposición 2 : el servicio *favourites*, tiene definido un SLI donde la latencia, también en la métrica *p95*, es de un promedio de 10 milisegundos.
- Suposición 3: la tercera suposición consiste en no definir tiempos límite desde Niles hacia ninguno de los microservicios nombrados. Debido a que 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 milisegundos (*Items-service* resuelve su petición en 50 milisegundos y *favourites* en 10 milisegundos), como se puede observar en la Figura 3.8:

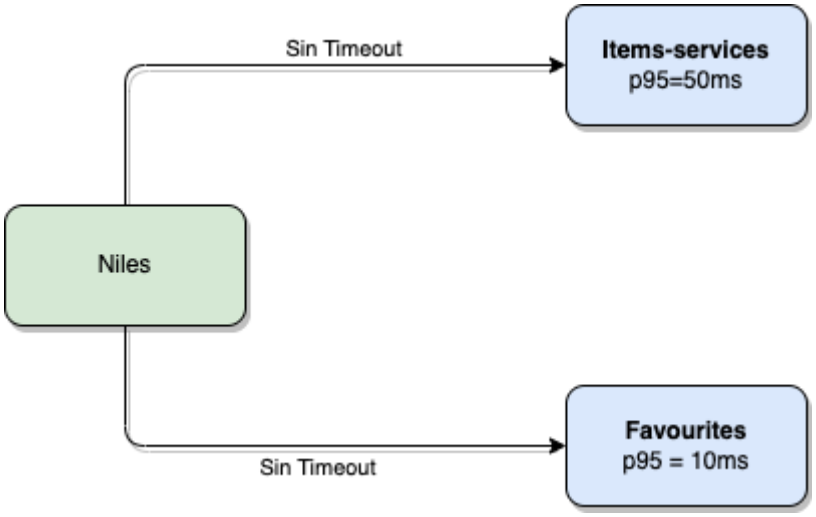

Figura 3.8 - Niles sin timeouts a servicios externos

En este escenario, suponemos que se produce una degradación de *Favourites-service*. Este servicio, por razones desconocidas, comienza a responder en tiempos altos del orden de 1 segundo.

La primera consecuencia consiste en la penalización del menú. 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* (1 segundo). Es decir, se penaliza la entrega de un menú por el hecho de averiguar si contiene productos marcados como favoritos.

La segunda (y más grave) consecuencia, es que se puede producir el caso que se describe en la Figura 3.9, donde la totalidad de hilos están bloqueados. *Favourites-service* se continúa degradando debido a que sigue encolando peticiones en las colas de espera, y los tiempos de respuesta continúan aumentando. En la imágen se puede observar como Niles consume sus recursos (*threads*) en información que no es relevante para el menú.

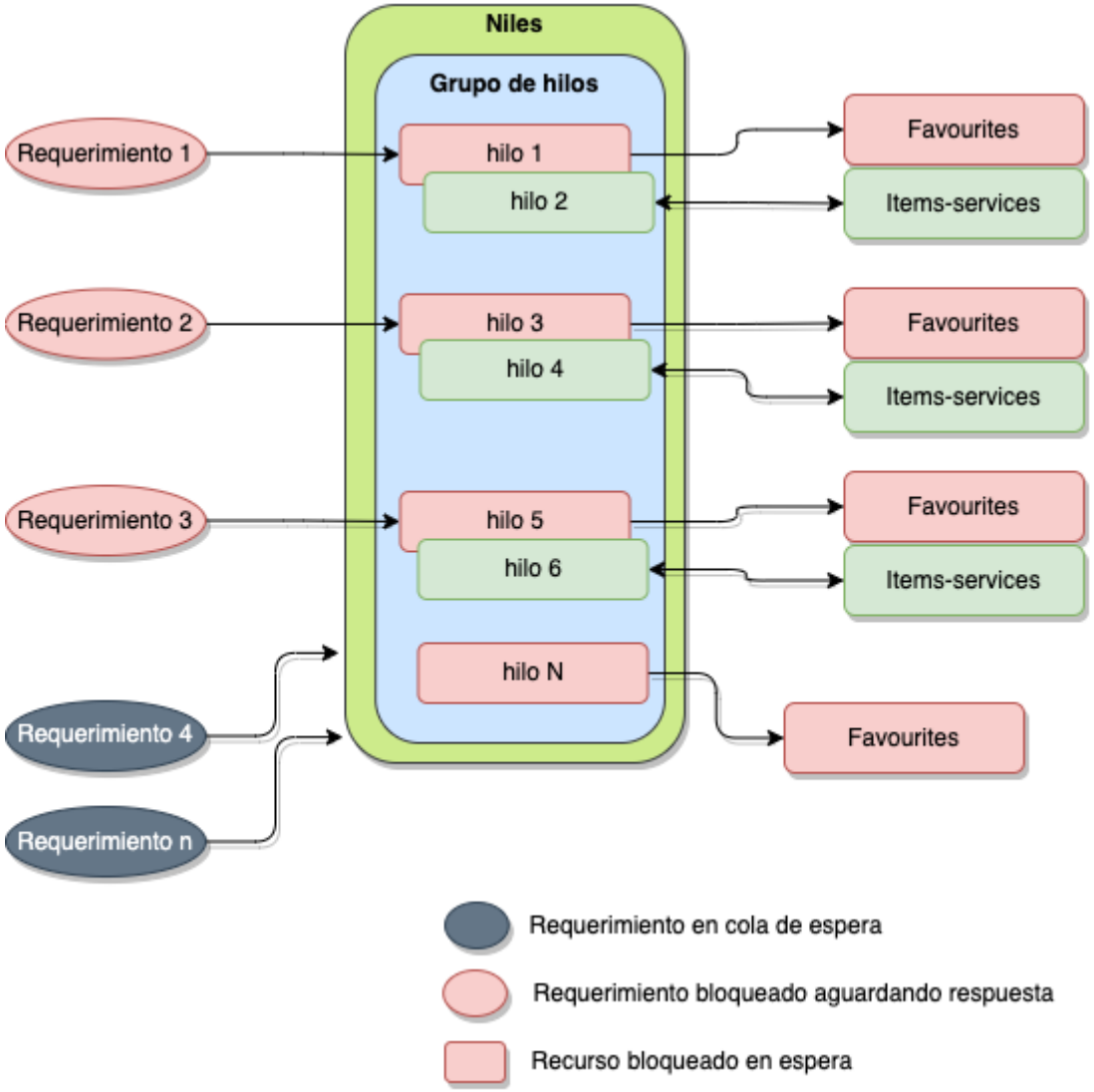

Figura 3.9 - Niles sin recursos disponibles para atender nuevos requerimientos

La solución a este problema requiere conocer de manera aproximada los tiempos normales en los que responde el servicio de *favourites* para definir adecuadamente un límite de tiempo. Si el percentil 95 indica que los tiempos de respuesta en general no exceden a los 10 milisegundos, un tiempo prudente de espera serían 15 milisegundos.
Con el nuevo tiempo de espera definido, no se penaliza el menú, y tampoco se bloquean hilos a la espera de una respuesta que probablemente nunca llegue. En la Figura 3.10 se puede observar cómo la comunicación es finalizada desde Niles cuando el servicio de *favourites* no responde en los tiempos esperados, liberando así los recursos.

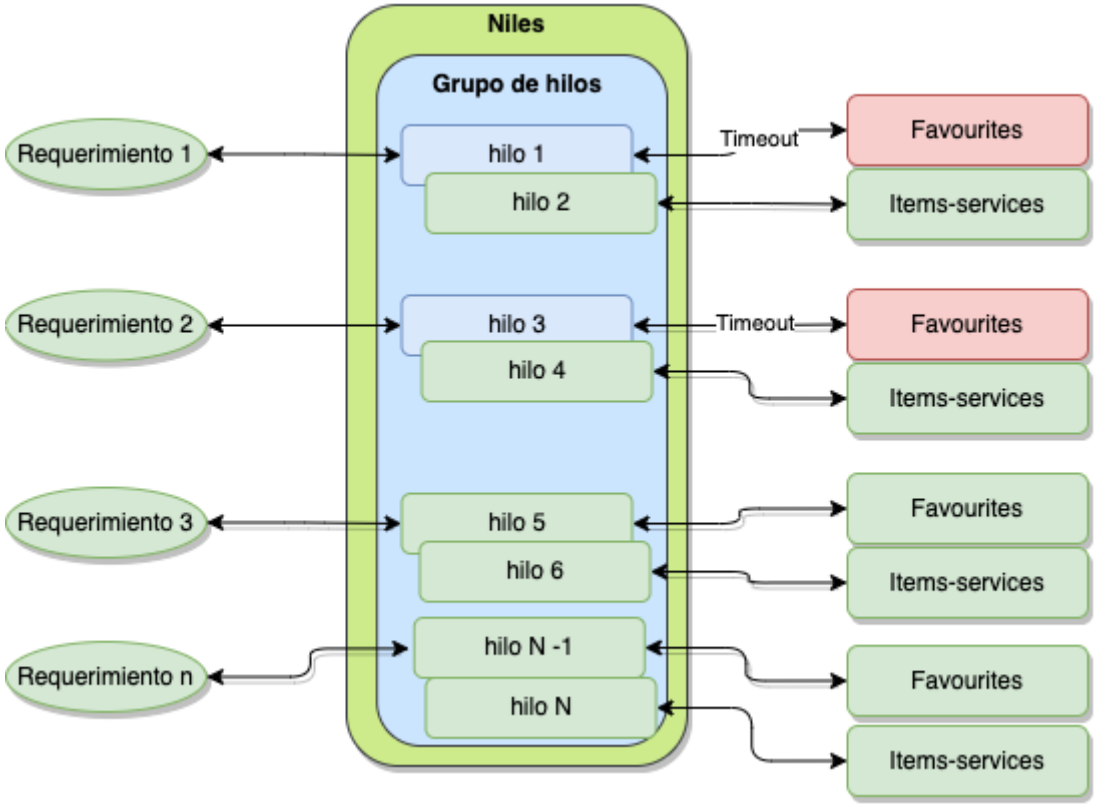

Figura 3.10 - Niles finaliza la comunicación con el servicio degradado, liberando recursos

3.3.3.El patrón *Retry*

3.3.3.1.Definición

En arquitecturas basadas en microservicios, y en aplicaciones distribuidas en general, donde diferentes servicios y recursos externos se comunican constantemente, se pueden producir errores temporales o transitorios (*transient failures*), como se ilustra en la Figura 3.11. Estos fallos pueden estar causados por diferentes motivos, siendo los más comunes las pérdidas momentáneas de conexión a la red o la no disponibilidad temporal de servicios.

Normalmente estos errores se manifiestan durante cortos lapsos de tiempo de manera de que, si el servicio o recurso vuelve a ser invocado o consultado a la brevedad, responde de manera correcta.

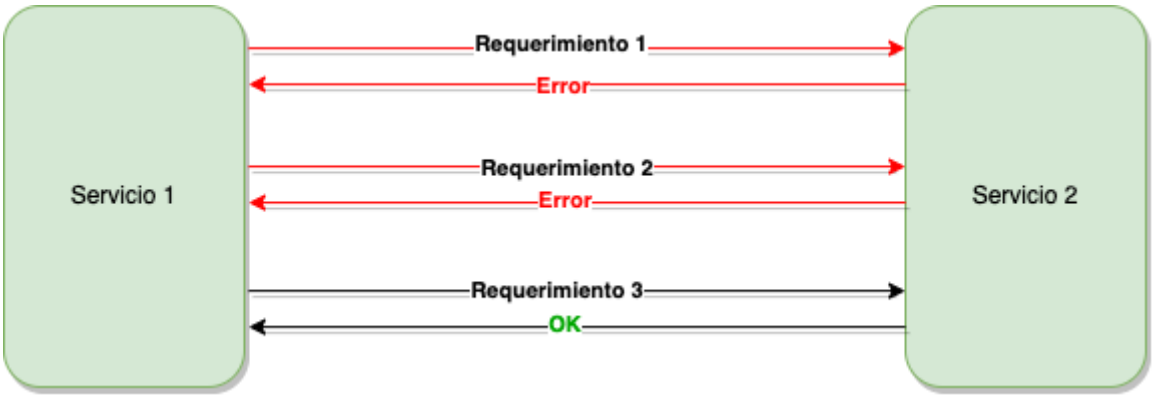

Figura 3.11 - Fallos transitorios en la comunicación

Debido a que estos errores son bastante frecuentes, es necesario gestionarlos de manera correcta para minimizar su impacto en el sistema. Una posible solución a este problema es la aplicación del patrón *Retry*.

La idea del patrón *Retry*, tal y como indica su nombre, consiste simplemente en reintentar una operación que ha fallado. Para explicarlo de una manera más detallada, es necesario añadir que, según el tipo de error detectado y/o el número de intentos, se pueden realizar diversas acciones:

- Reintentar de inmediato: si el código del error retornado indica que es un fallo temporal o atípico, la aplicación puede reintentar inmediatamente la misma operación, ya que probablemente no vuelva a producirse el mismo error.
- Reintentar tras un tiempo de espera: si el error se ha debido a un problema de conexión a la red, o bien por un pico de peticiones al servicio, es prudente dejar pasar un tiempo antes de volver a intentar la operación.
- Cancelar: si el código de error retornado indica que el fallo no es temporal, la operación debería ser cancelada, por lo que se debería reportar el error o gestionarlo de manera adecuada.

Estas acciones pueden combinarse para crear una política de reintentos ajustada a las necesidades de nuestra aplicación. En la Figura 3.12 se observan los posibles caminos que se pueden recorrer cuando se implementa el patrón *Retry*:

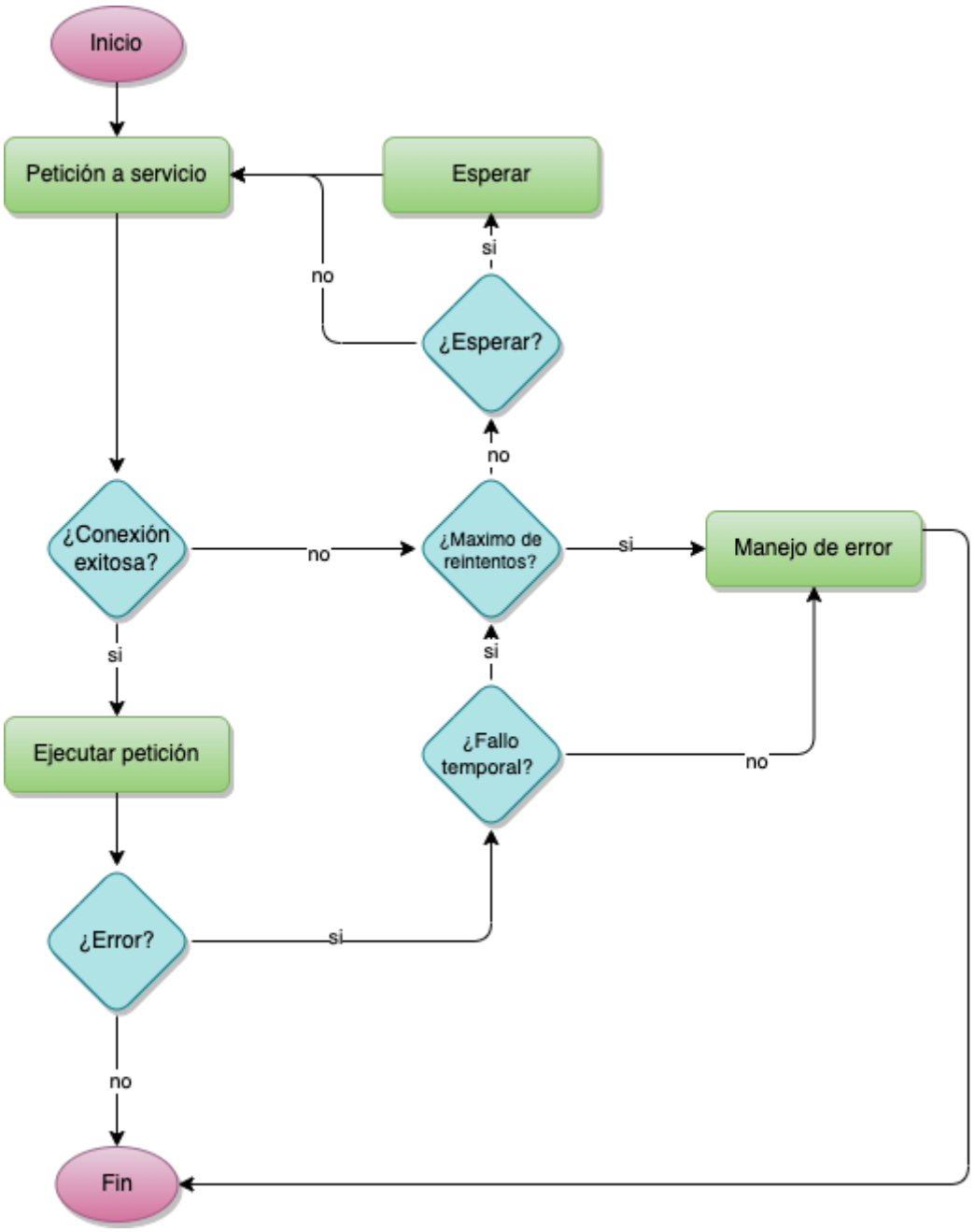

Figura 3.12 - Posibles flujos de ejecución con reintentos

3.3.3.2.Implementación

La implementación del patrón *Retry* es similar a la configuración realizada para el patrón *Timeout*, ya que se utiliza la librería *peya-ktor-utils*. Al momento de crear un nuevo cliente de peticiones HTTP, se precisan dos objetos. El primer objeto es una configuración que consta, entre otras cosas, del *host* sobre el cual se realiza la petición, como se puede observar en la siguiente línea de código:

```
HttpClientConfiguration(
    host = "items-service.live.peja.co"
)
```

El segundo objeto, de la misma manera que en la sección anterior, es una lista de *resilience providers*. En este caso la clase a instanciar es RetryConfiguration, como se aprecia en el siguiente fragmento de código:

```
ResilienceProviders(
   providers = listOf(
       RetryPolicyFactory.create(
           RetryConfiguration(
               name = "retry-configuration",
               maxAttempts = 3,
               retryOnStatuses = listOf(500, 502),
               ignoreExceptions = listOf(BadRequestException::class.java),
               exponentialBackoff = ExponentialBackoff(Duration.ofMillis(30), 1.5, 0.5),
           )
       )
   )
)
```

A continuación se detalla cada uno de los parámetros necesarios para instanciar una configuración de reintentos:

- *MaxAttempts*: hace referencia a la cantidad de reintentos, antes de fallar definitivamente la operación.
- *RetryOnStatuses*: indica los códigos de estados de respuesta HTTP[20] frente a los cuales se realizarán reintentos. Esto significa que si el código de respuesta de una petición HTTP es 500, o 502 (como se aprecia en el código), se realizarán reintentos; sin embargo, frente a un código 429 (Too Many Requests), no se ejecutarán reintentos.
- *IgnoreExceptions*: como se indicó previamente, la idea de los *resilience providers* es encapsular una operación o bloque de código. La ocurrencia de una determinada excepción durante la ejecución de cualquier bloque de código, con la resiliencia incorporada, se utiliza como propiedad para decidir si se deben realizar o no reintentos.
- *ExponentialBackoff*: es un algoritmo que determina los periodos en los que se realizarán los reintentos, que consiste en un mecanismo para no realizar reintentos en periodos fijos. Para ellos, se implementa la técnica de retraso exponencial, que vuelve a intentar una operación, con un tiempo de espera que aumenta exponencialmente hasta que se alcanza un número máximo de reintentos [21].

Es importante realizar una configuración correcta y evitar reintentos en determinados casos. Por ejemplo, si el código de respuesta de una petición HTTP es 429, indica que se están realizando más peticiones de las que soporta el servidor, y en el caso de realizar reintentos, es posible saturarlo. Del mismo modo, si el bloque de código que encapsula la operación de reintento produce una excepción interna (por ejemplo, si se envían incorrectamente los parámetros necesarios para realizar la operación, es decir, un código de estado 400, o la excepción BadRequestException[21] en java/kotlin), también deben evitarse los reintentos, ya que es un problema del cliente de la petición.

[20] https://developer.mozilla.org/es/docs/Web/HTTP/Status
[21] https://docs.oracle.com/javaee/7/api/javax/ws/rs/BadRequestException.html

3.3.3.3.Aplicación al caso de estudio

Como se mencionó previamente,Niles realiza múltiples peticiones HTTP a otros servicios; sin embargo, si en particular *Items-service* no está operativo, o falla alguna petición hacia él, el menú no podrá ser generado.
Esta fuerte dependencia genera fragilidad en la funcionalidad de Niles ante cualquier inconveniente en la red, como por ejemplo latencia alta, microcortes, problemas temporales de DNS, etc. En la Figura 3.13 se observan intentos de comunicación entre ambos servicios mientras la red sufre de errores transitorios; por lo tanto algunas de las peticiones HTTP desde Niles, no llegan a ser recibidas por *Items-service*.

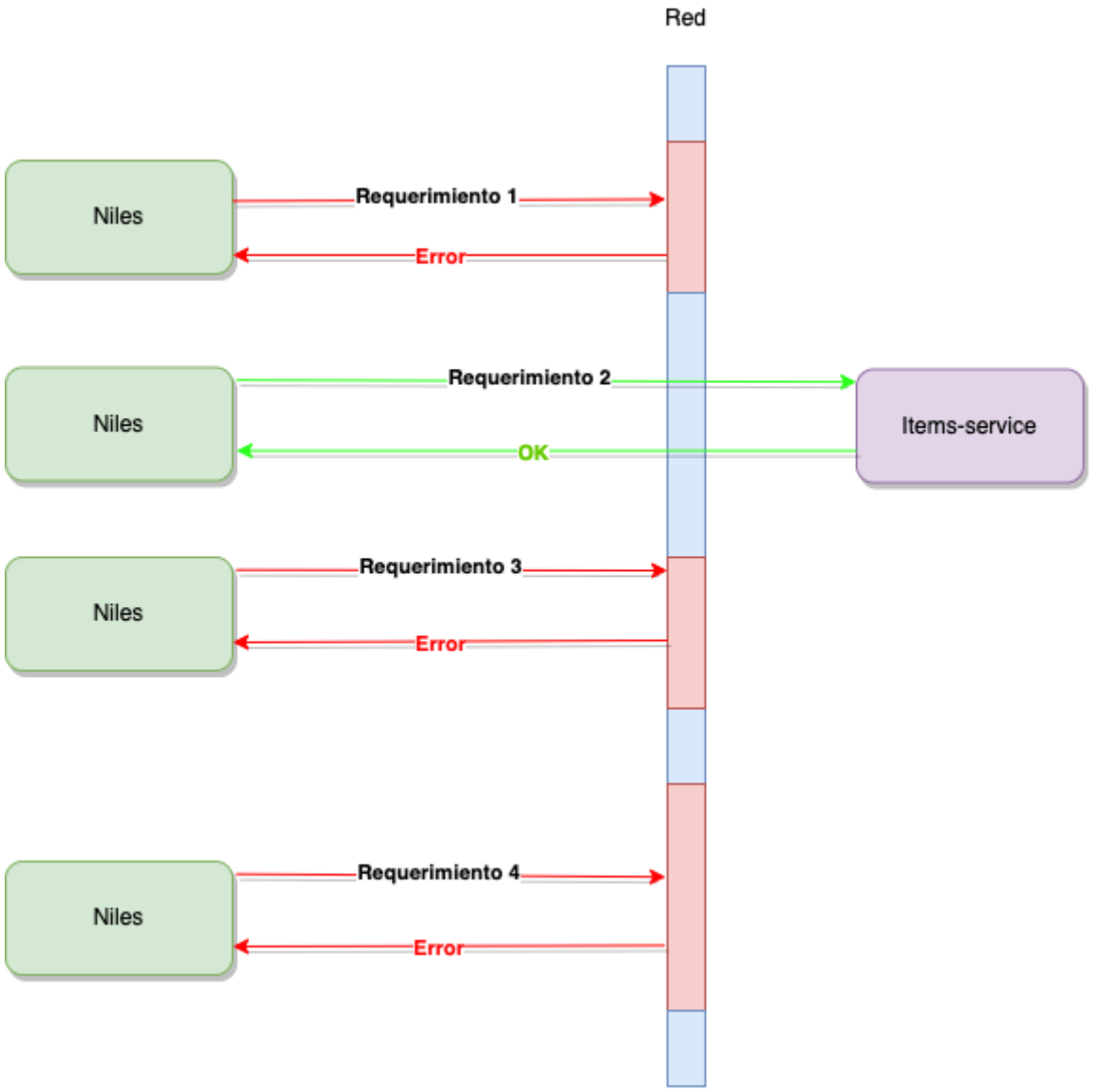

Figura 3.13 - Fallos transitorios en la red

Al aplicar el patrón *Retry* en la comunicación entre Niles e *Items-service*, es posible solucionar estos errores transitorios, tal como se observa en la Figura 3.14:

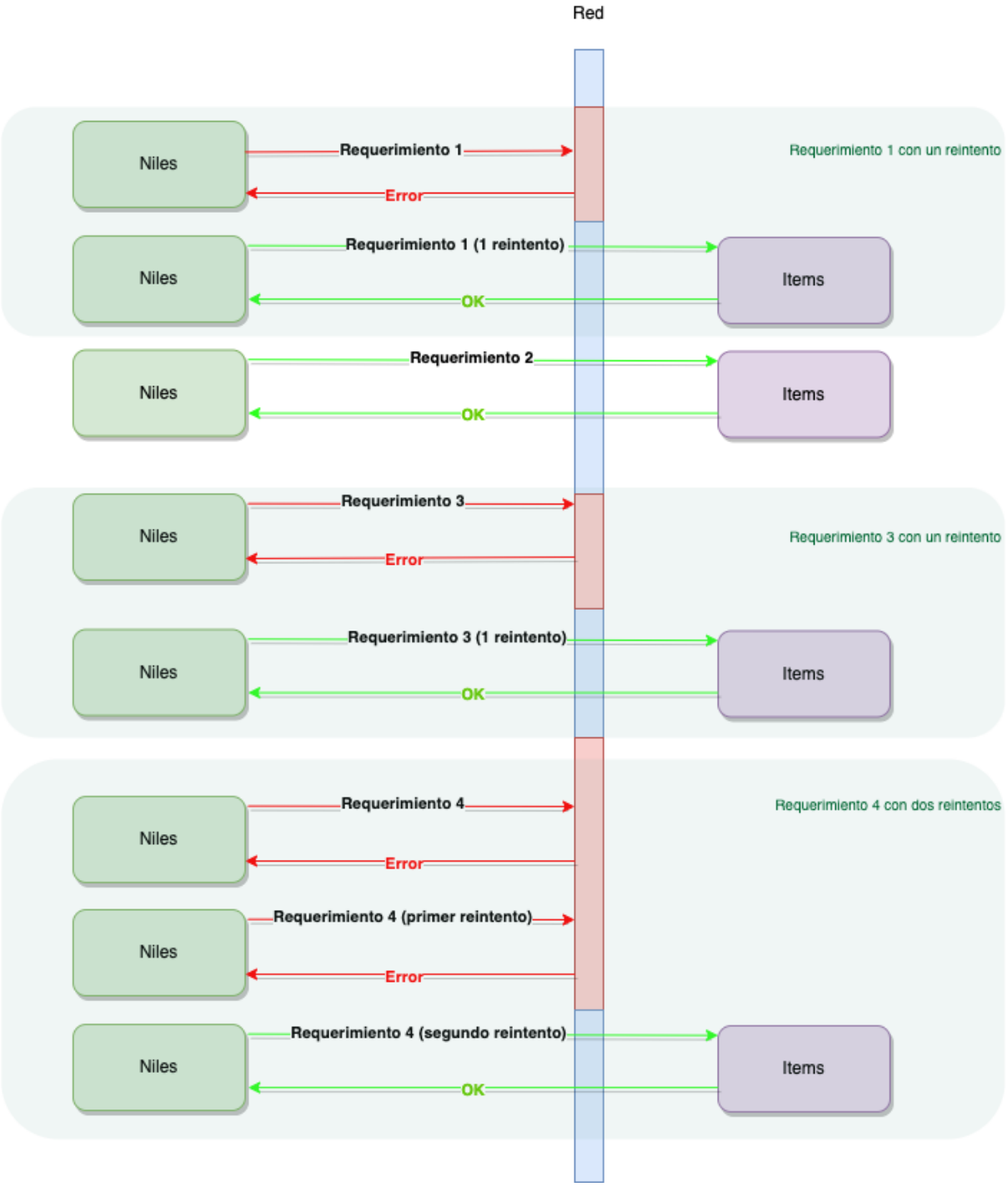

Figura 3.14 - Patrón *Retry* implementado entre Niles e *Items-service*

En el ejemplo de la figura, la comparación entre una comunicación sin reintentos, y una comunicación que aplica el patrón *Retry* nos permite observar que, en el primer caso, tres de los cuatro requerimientos fallan, y en el segundo caso, los cuatro requerimientos finalmente resultan exitosos.

Al solicitar el menú de un restaurante, ante la ocurrencia de un fallo de *Items-service* (y por consiguiente de la creación del menú), el usuario puede actualizar la pantalla del celular, cerrar y volver a abrir la aplicación o cerrar sesión y volver a intentar obtener el menú, lo cual, en un caso como el presentado, resuelve el incidente. Sin embargo, existen casos donde no existe esa posibilidad de interacción, como por ejemplo procesos *batch*, actualizaciones masivas, liquidaciones de sueldo, etc. En esos casos, el patrón *Retry* es de suma importancia. Por otra parte, es importante administrar correctamente la aplicación de reintentos, y sólo aumentar la cantidad de ellos para los recursos que son vitales en el caso de uso particular. Esto significa que, por ejemplo, no tendría sentido aplicar múltiples reintentos para recuperar un producto favorito, una *review*[22] de un producto o un dato menor, ya que se penalizaría a la funcionalidad principal.

3.3.4.El patrón *Circuit Breaker*

3.3.4.1.Definición

La industria de la energética adoptó la inclusión de los fusibles como una solución parcial al problema del calentamiento resistivo debido a la conexión de numerosos electrodomésticos a la red eléctrica domiciliaria, causante a su vez de numerosos accidentes[22]. El propósito de un fusible es el de quemarse antes que la instalación sufra una sobrecarga; por lo tanto, está diseñado para mantener bajo control la falla general. Sin embargo, pese a su utilidad, este es un dispositivo desechable, de un único uso.

La solución a esta desventaja de los fusibles consiste en la utilización de un disyuntor, el cual es un dispositivo capaz de interrumpir o abrir un circuito eléctrico cuando ocurren fallas de aislamiento en un equipo o en una instalación eléctrica. Su principal objetivo es la seguridad de las personas, evitando que las mismas resulten afectadas por corrientes eléctricas al entrar en contacto con el equipamiento que falla.

El principio del disyuntor es el mismo que el de los fusibles: detecta el exceso de uso, falla primero y abre el circuito. De manera más abstracta, el disyuntor existe para permitir que un subsistema (un circuito eléctrico) falle (consumo de corriente excesivo, posiblemente debido a un cortocircuito) sin destruir todo el sistema (la casa). Además, una vez que ha pasado la situación de riesgo, el disyuntor puede restablecerse para restaurar la funcionalidad completa del sistema.

Es posible aplicar el mismo concepto al software, protegiendo operaciones relevantes, o incluso críticas, por medio de un componente que pueda eludir las llamadas cuando el sistema no está en buenas condiciones. A diferencia de los reintentos, en este caso existen interruptores automáticos para evitar operaciones, en lugar de volver a ejecutarlas [23][24][25].

En el estado normal ("cerrado"), el disyuntor ejecuta las operaciones como es habitual; pueden ser llamadas a otro sistema u operaciones internas que están sujetas a tiempo de espera o a otro error de ejecución (como por ejemplo, peticiones HTTP, envíos de mensajes a colas, consultas a bases de datos, etc). Si la llamada tiene éxito, no sucede nada extraordinario. Sin embargo, si falla (según la configuración especificada para el *Circuit*

[22] Review de un producto, hace referencia a un comentario u opinión acerca del producto.

Breaker), el disyuntor registra esa falla. Una vez que el número de fallas (o frecuencia de fallas, en casos más sofisticados) 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 hay ningún intento de ejecutar la operación real. Después de un lapso de tiempo adecuado, el interruptor automático[23] decide que la operación tiene cierta probabilidad de éxito, por lo que pasa al estado "medio abierto" *(half-open)*. En este estado, se permite que la siguiente llamada al interruptor automático intente ejecutar nuevamente la operación crítica (es decir, una similar a la que produjo la apertura del disyuntor). Si la llamada tiene éxito, el disyuntor se restablece y vuelve al estado "cerrado", es decir, listo para una operación más rutinaria. Sin embargo, si esta llamada de prueba falla, el disyuntor vuelve al estado "abierto" hasta que transcurre otro tiempo de espera.

A la técnica descrita se la denomina *Circuit Breaker* y sus posibles estados se pueden visualizar en la Figura 3.15:

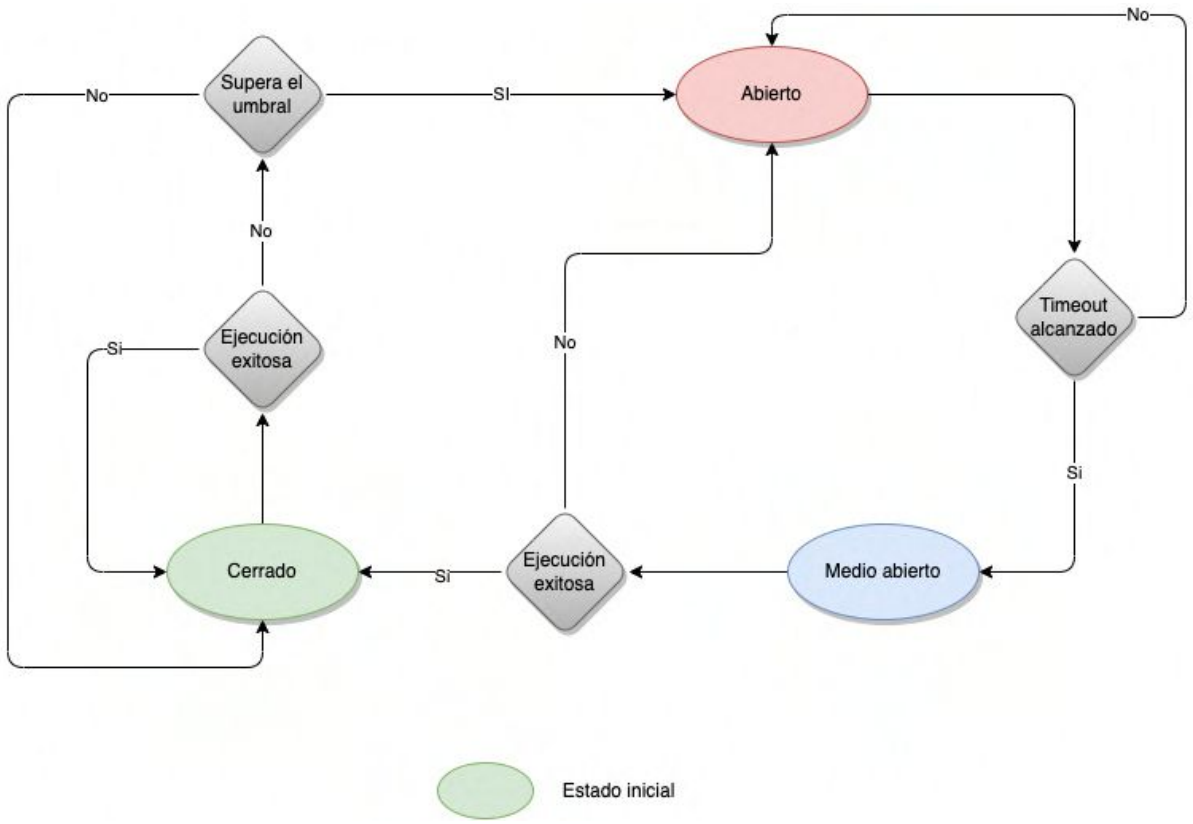

Figura 3.15 - Posibles estados de un *circuit breaker*

El estado de circuito abierto debería estar indicado por algún tipo de excepción, que sea diferente a la que retorna el error original cuando esos errores aún no son suficientes para activar al disyuntor. Esto permitiría brindar una mejor retroalimentación al usuario, retornando por ejemplo una respuesta por defecto o un modo específico de manejo de esa excepción particular.

[23] El circuit breaker es conocido como disyuntor, o interruptor automático.

Dependiendo de los casos de uso del sistema, cada disyuntor implementado puede estar configurado para detectar diferentes tipos de fallas, y manejar diferentes parámetros, por ejemplo, se puede optar por tener un umbral más bajo para las fallas de "tiempo de espera agotado para llamar al sistema remoto" que para los errores de "conexión rechazada".

Los disyuntores son una forma de degradar automáticamente la funcionalidad cuando el sistema está bajo estrés. Esto puede tener un impacto en el negocio del sistema. Por lo tanto, es fundamental involucrar a los diferentes actores al momento de decidir cómo manejar las llamadas realizadas cuando el circuito está abierto. Por ejemplo, ¿debería un sistema minorista aceptar un pedido si no puede confirmar la disponibilidad de los artículos del cliente? ¿Qué sucede si no puede verificar la tarjeta de crédito o la dirección de envío del cliente? Por supuesto, esta conversación no es exclusiva del uso de un *Circuit Breaker*, pero discutir el interruptor automático puede ser una forma más eficaz de abordar el tema que solicitar un documento de requisitos.

3.3.4.2.Implementación

La librería *peya-ktor-utils* soporta la implementación del *Circuit Breaker*, siguiendo los mismos pasos que los patrones *Timeout* y *Retry*.

Como se vió en las implementaciones de los patrones anteriores, al momento de la creación de un cliente de peticiones HTTP, se necesitan dos objetos. El primero no varía de los anteriores, y en cuanto al segundo, para la implementación del *Circuit Breaker* se necesita una instancia de la clase CircuitConfiguration, tal como ilustra el siguiente fragmento de código:

```
ResilienceProviders(
   providers = listOf(
       CircuitBreakerPolicyFactory.create(
           CircuitConfiguration(
               name = "circuit-configuration",
               failureRateThresholdPercentage = 0.0f,
               waitDurationInOpenState = Duration.ofMillis(5000),
               permittedNumberOfCallsInHalfOpenState = 20,
               slidingWindowSize = 200,
               responseErrorValues = listOf(500, 502),
               recordedExceptions = arrayOf(
                   IOException::class.java,
                   TimeoutException::class.java
               ),
               ignoreExceptions = arrayOf(
                   HttpNotFoundException::class.java
               )
           )
       )
   )
)
```

A continuación se detalla cada uno de los parámetros necesarios para instanciar una configuración del *Circuit Breaker*:

- *Name*: es el nombre que se utiliza como prefijo para los *logs*, métricas y trazas propias del *Circuit Breaker*, como los cambios de estado o los errores.
- *FailureRateThresholdPercentage*: es el porcentaje de errores que se tiene en cuenta para cambiar el estado del circuito de cerrado a abierto. Está fuertemente relacionado al valor de *SlidingWindowSize*.
- *WaitDurationInOpenState*: tiempo que se mantiene el circuito abierto, antes de pasar al estado *half-open*.
- *PermittedNumberOfCallsInHalfOpenState*: cantidad de requerimientos a realizar en el estado *half-open*, para pasar al estado cerrado o volver al estado anterior (abierto).
- *SlidingWindowSize*: es la cantidad de requerimientos sobre la cual se evalúa el porcentaje de fallas *FailureRateThresholdPercentage*.
- *ResponseErrorValues*: indica los códigos de respuesta HTTP sobre los cuales se considerará que una respuesta es fallida.
- *RecordedExceptions*: similar al campo *ResponseErrorValues,* pero con respecto a las excepciones producidas dentro del bloque de código ejecutado.
- *IgnoreExceptions*: indica las excepciones que no son consideradas como errores, para evitar abrir el circuito en caso de que se provoquen las mismas; generalmente aquí se agregan las excepciones conocidas como BadRequestException o NotFoundException).

Como en el patrón *Retry*, es necesario ser cuidadoso con la configuración; por ejemplo, si el código de respuesta para una petición HTTP es 400 (BadRequest) significa que el cliente es

quien está enviando algún parámetro incorrecto, por lo que el *Circuit Breaker* no debería entrar en estado abierto, ya que se evitarían llamadas que realmente deberían ejecutarse.

3.3.4.3.Aplicación al caso de estudio

En las comunicaciones entre microservicios resulta bastante habitual la necesidad de aplicar un *Circuit Breaker*. En general, es buena práctica definir siempre tiempos de espera adecuados *(timeouts)*, configurar reintentos en algunos casos e implementar un *Circuit Breaker* que evite saturar de requerimientos a los microservicios que son consumidos.

El caso de estudio a analizar hace foco en la comunicación entre Niles e *Items-service*. Ambos servicios están catalogados como Tier1[24], por lo cual deben presentar una alta disponibilidad y un correcto funcionamiento en cualquier momento.

Como se explicó anteriormente, *Items-service* es el microservicio que contiene el repositorio de todas las entidades básicas referidas a ítems de restaurantes, tales como inventarios, secciones, productos, grupos de opciones, etc. Este servicio es consumido por muchos otros, como se observa en la Figura 3.16:

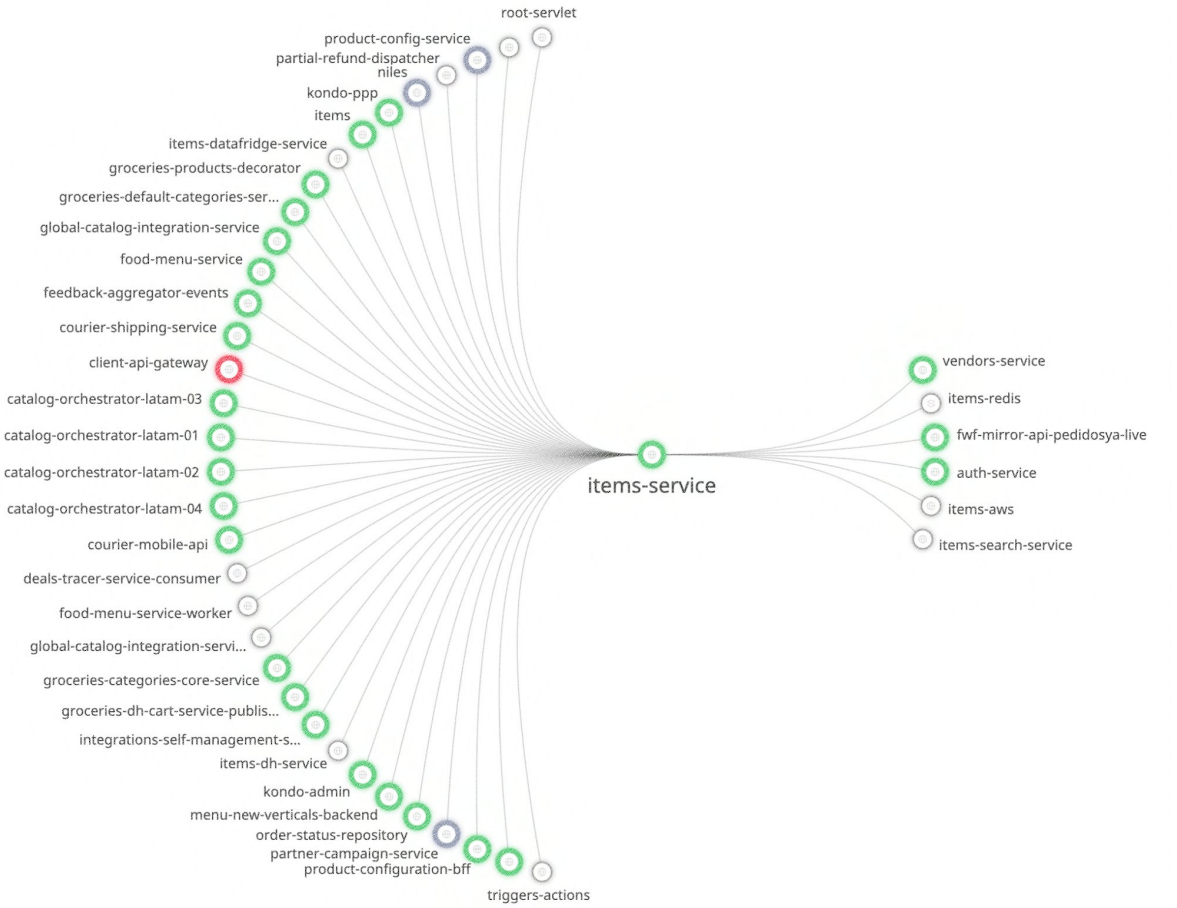

Figura 3.16 - Mapa de dependencias con consumidores *(izq)* y consumidos (der) de *Items-service*

[24] Tier1 es una clasificación que indica que son servicios cuya caída impacta directamente sobre la generación de órdenes, por lo que son fundamentales para el negocio. Existen también servicios Tier 2, 3 y 4.

Items-service recibe internamente las novedades de modificaciones de ítems, tal como lo hace Niles (la diferencia radica en que *Items-service* recibe novedades de otros elementos que no son necesarios para el menú, como por ejemplo grupos de opciones y opciones de un producto). 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 es un DynamoDB de AWS[25]. Esta base de datos utiliza la técnica de Throttling[26] para controlar estos picos de operaciones. Básicamente, DynamoDB advierte que se ha llegado al límite de procesamiento de lecturas/escrituras, con lo que procede a agregar mayor capacidad, pero esta operación demanda algunos minutos. Como la base de datos es única para el servicio, todas las peticiones realizadas fallan hasta que DynamoDB agregue dinámicamente más capacidad. Mientras tanto, Niles continúa solicitando información de secciones y productos a *Items-service*, y el agregado de reintentos (mediante el patrón *Retry*) empeora la situación.

Debido a la cantidad de requerimientos denegados desde su base de datos, *Items-service* no logra recuperarse, y por consiguiente, termina afectando a los demás servicios que lo consumen, como se observa en la Figura 3.17:

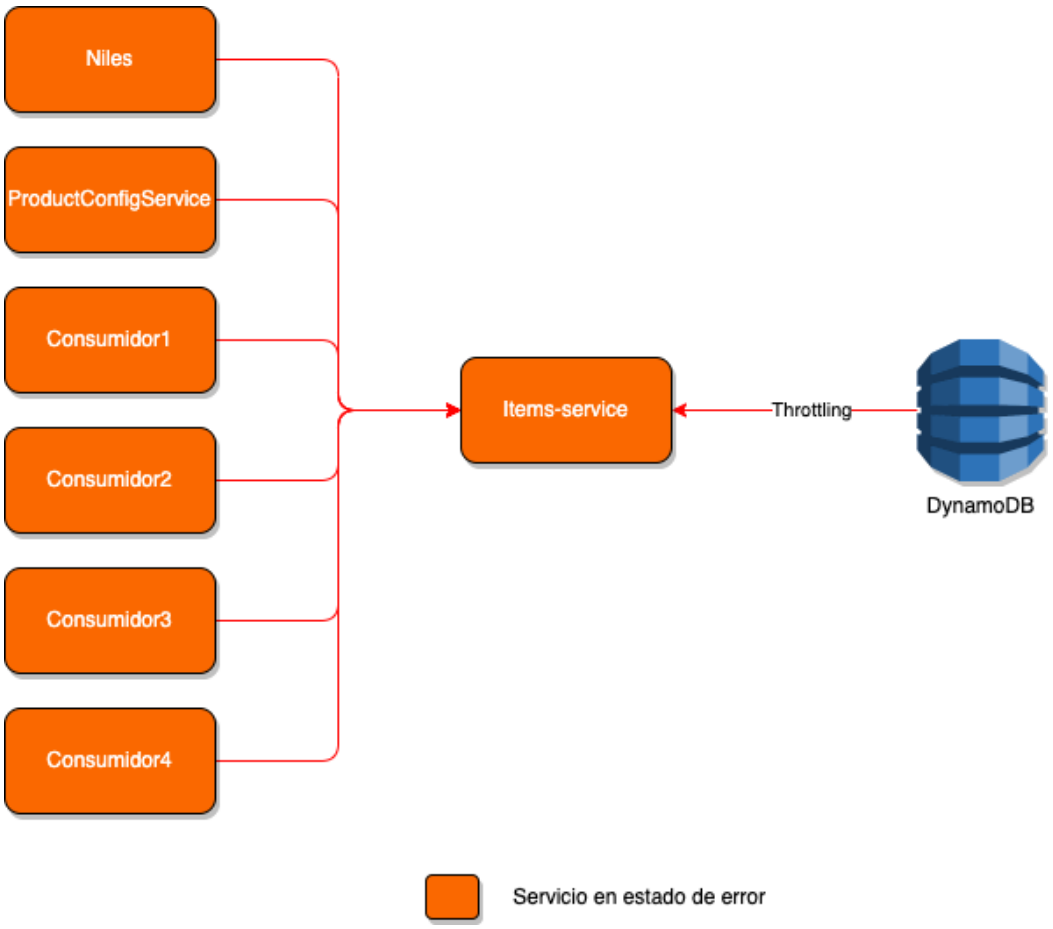

Figura 3.17 - Errores en todos los consumidores de *Items-service*

[25] https://aws.amazon.com/es/dynamodb/
[26] https://aws.amazon.com/es/premiumsupport/knowledge-center/dynamodb-table-throttled/

Con la aplicación del patrón *Circuit-Breaker*, se evita saturar a *Items-service* con requerimientos salientes de Niles, sabiendo que es muy posible que éstos sigan fallando por un lapso corto de tiempo.
Cuando el disyuntor entre Niles e *Items-service* cambia al estado abierto, en principio, no podría mostrarse el menú, aunque en muchos casos la caché propia que tiene Niles responderá sin problemas. Con esta estrategia se evita abrumar a un servicio que ya llegó al límite de requerimientos establecido por su base de datos, pero del cual se sabe que estará disponible en el corto plazo; el *Circuit Breaker* decidirá cuándo el servicio está disponible para recibir requerimientos nuevamente, y pasar así al estado cerrado.

3.3.5.El patrón *Bulkhead*

3.3.5.1.Definición

En un barco, los mamparos (*Bulkheads*) son particiones de metal que se pueden sellar para dividir el barco en compartimentos separados. Una vez que se cierran las escotillas, el mamparo evita que el agua se mueva de una sección a otra y de esta forma, el efecto de una única ruptura en el casco no puede propagarse de forma de terminar produciendo el hundimiento del barco [26]. La función del mamparo es que se cumpla el principio de contención de daños, como se puede ver en la Figura 3.18.

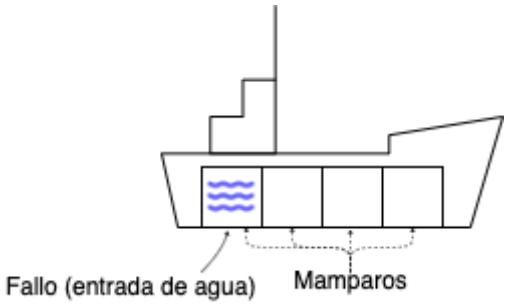

Figura 3.18 - Fallo (entrada de agua) que es aislado por medio de un mamparo

Esta misma técnica puede ser empleada en el diseño de arquitecturas de software. Al particionar un sistema, se puede evitar que una falla en una parte del mismo afecte a todas las demás funcionalidades. La forma más común de implementar mamparos es mediante redundancia física. Por ejemplo, si hay cuatro servidores independientes, una falla en el hardware de uno de ellos no es capaz de afectar a los demás. Del mismo modo, si hay dos instancias de aplicación ejecutándose en un servidor, y una de ellas falla, la otra podrá continuar su ejecución.

A gran escala, un servicio podría implementarse a partir de varias granjas independientes de servidores, con algunas de éstas reservadas para uso de aplicaciones críticas, y otras disponibles para usos que no lo son. Por ejemplo, un sistema de compra de pasajes de avión podría proporcionar servidores dedicados para el registro de clientes; esta funcionalidad no se vería afectada si otros servidores compartidos fueran abrumados con consultas de "estado de vuelo" (como sucede a veces durante condiciones climáticas severas).

En la Figura 3.19 se puede ver como dos servicios (S1 y S2) consumen a otro servicio (S3). Debido a que S1 y S2 dependen de un servicio común, existe un grado de vulnerabilidad de cada uno respecto del otro. Si S1 se ve abrumado por requerimientos, o existe algún defecto que desencadene un error en S3, S2 y sus consumidores también resultarán afectados. Este tipo de acoplamiento invisible produce que el diagnóstico de problemas (particularmente de rendimiento) en S2 sea muy difícil. Por otra parte, la programación de mantenimiento, actualización o instalación de S3 también requiere de coordinación con ambos consumidores, pudiendo resultar difícil encontrar una ventana que sea adecuada para ambos.

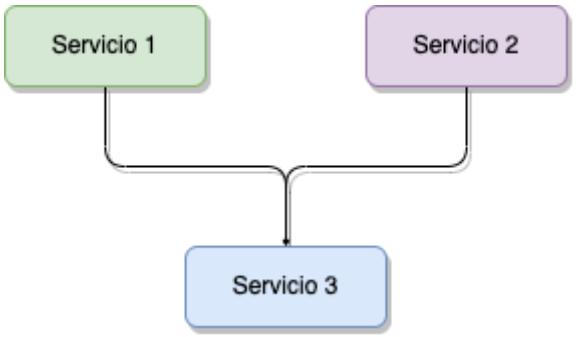

Figura 3.19 - Dos servicios que consumen a uno en común

Suponiendo que los servicios consumidores son elementos críticos con SLI´s estrictos, una opción más segura sería particionar al servicio S3, tal como se muestra en la Figura 3.20.

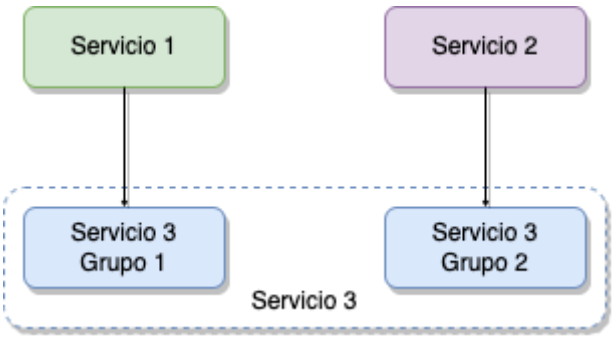

Figura 3.20 - Partición del servicio común en dos grupos

A una escala más pequeña, la vinculación de un proceso a una CPU o grupo de CPU´s es un ejemplo de partición a través de mamparos. Esto garantiza que el sistema operativo programe los subprocesos de ese proceso sólo en la CPU o grupo de CPU´s designado; por lo tanto, si un proceso se torna inestable y no libera a la CPU, sólo puede afectar a la CPU vinculada, pero no consume los recursos de toda la máquina completa, aislando así el problema.

El *Bulkhead* es efectivo para mantener un grado de funcionalidad del servicio, incluso ante fallas. Es especialmente útil en arquitecturas orientadas a microservicios, donde la degradación de un solo servicio puede tener repercusiones en toda la empresa, como se muestra en la Figura 3.21:

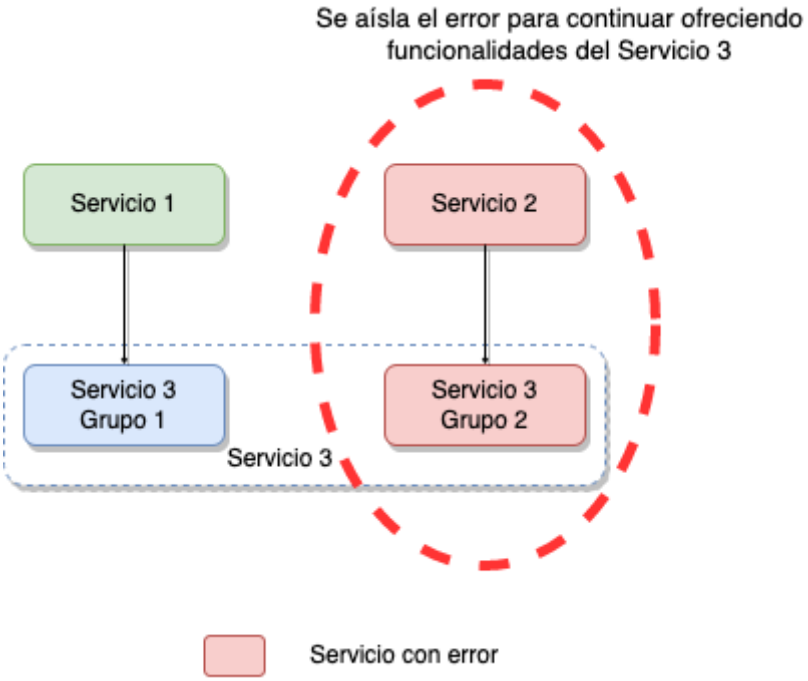

Figura 3.21 - Error que puede ser aislado, permitiendo continuar con el funcionamiento

Es posible implementar esta idea en diferentes niveles, como por ejemplo: particionar grupos de subprocesos dentro de una aplicación, grupos de CPU´s dentro de un servidor o grupos de servidores dentro de un *cluster*. En una arquitectura de microservicios, ya desde la etapa de diseño se debe tener en cuenta el modo de aislar las diferentes partes, evitando así las fallas en cascada.

3.3.5.2.Implementación

La implementación del patrón *Bulkhead* para el caso de estudio de Niles es realizada por el equipo de infraestructura de PedidosYa.
Dentro de la compañía, los equipos de desarrollo cuentan con una herramienta de *self-service*[27] denominada *Jarvis*, la cuál permite múltiples operaciones como la creación y administración de nuevos proyectos y servicios en diferentes lenguajes, librerías, *clusters* donde instalar servicios (incluso para diferentes regiones), bases de datos de diferentes tipos, tópicos, colas de mensajes, cachés, *jobs* programados, etc.
La pantalla principal de *Jarvis* es una consola de comandos, como se muestra en la Figura 3.22:

[27] https://www.techtarget.com/searchstorage/definition/self-service-cloud-computing

Figura 3.22 - Jarvis

Webpool es uno de los componentes que proporciona *Jarvis*. Un *webpool* es un espacio donde se despliegan las diferentes aplicaciones; cada *webpool* pertenece a un *cluster*, lo que permite diferenciar ambientes de instalación (por ejemplo, los comúnmente conocidos como ambiente *staging*[28] y ambiente de producción).

Cada *webpool* puede contener su propias variables de entorno, variar en cuanto a capacidad de memoria y CPU de cada servicio, cantidad mínima y máxima de instancias permitidas, etc., como se puede ver en la Figura 3.23:

[28] *Staging*: ambiente de pruebas

Figura 3.23 - Características de un *webpool*

Jarvis permite crear varios *webpools* y asignarlos al mismo *cluster*. De este modo es posible desplegar un microservicio en producción con diferentes configuraciones, que es precisamente la idea del patrón *Bulkhead*: particionar servidores en un *cluster*.

La página web de *Jarvis* permite visualizar de una manera simple las configuraciones de un servicio; además, en la página principal de un servicio, se observan los *webpools* en los que la aplicación está ejecutándose, tal como ilustra la Figura 3.24:

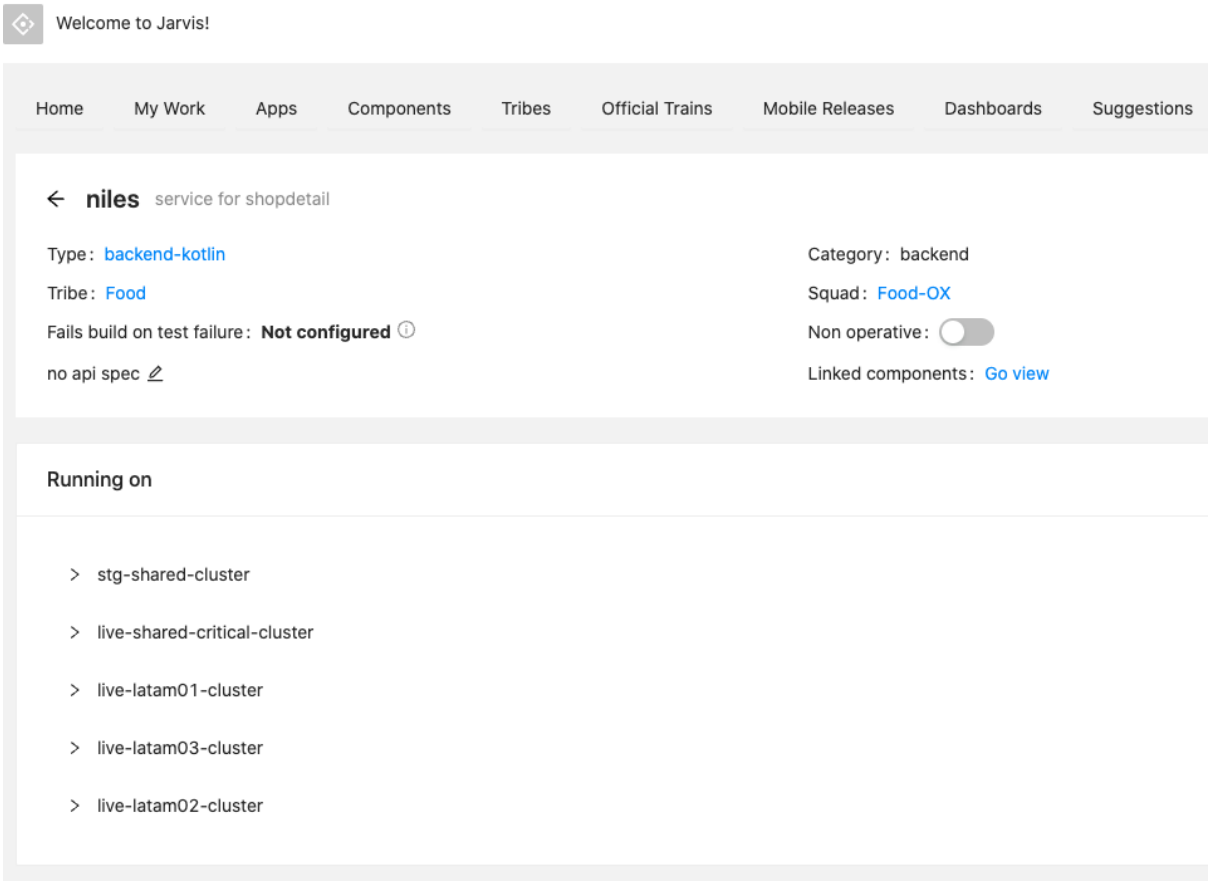

Figura 3.24 - Varios *webpools* en diferentes *clusters*

En este caso, se puede ver como Niles está desplegado en diferentes *webpools* (y en diferentes *clusters*), donde cada uno hace referencia a una región. Esto ayuda a realizar despliegues de manera progresiva, evitando errores y facilitando el regreso a una versión anterior en caso de fallos.

3.3.5.3.Aplicación al caso de estudio

En PedidosYa, es un caso de uso común utilizar el concepto de *worker*, el cual hace referencia 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 despliega el servicio en un *webpool* separado, que no atiende requerimientos HTTP, sino que se encarga de recibir las novedades desde los tópicos de *Items-service* y de *Battlefront*. La versión es exáctamente la misma, pero ejecutan diferentes bloques de código, ya que consumen variables de entorno del *webpool*, como se puede apreciar en la Figura 3.25:

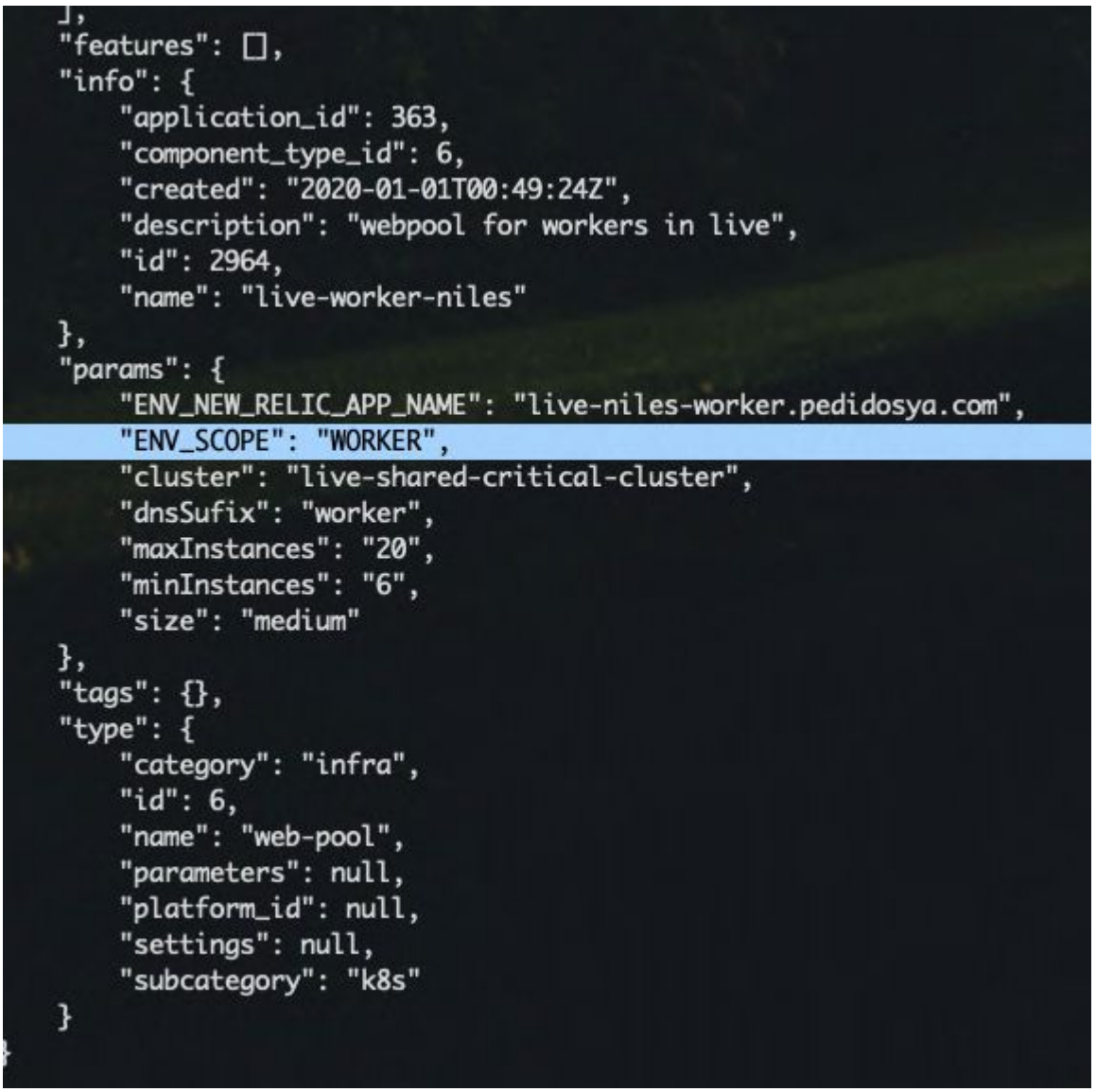

Figura 3.25 - Propiedades del *webpool* utilizado para instancias de tipo *worker*

En el siguiente fragmento de código se puede observar cómo se realizan diferentes bifurcaciones dependiendo de la variable de entorno SCOPE.

```
// Inicia las clases que reciben mensajes desde colas
fun Application.startConsumers() {
   if (System.getEnvVar("SCOPE") == "WORKER") {
       Consumer.start()
   }
}
```

De esta manera, se aísla funcionalidad en diferentes tipos de instancias: las instancias que reciben requerimientos HTTP por medio de *endpoints* y las instancias de tipo *worker* que se encargan de recibir novedades desde colas de mensajes para actualizar la información guardada en caché.

Eventualmente, ocurren picos de novedades debido a las actualizaciones masivas previamente mencionadas, como se puede apreciar en la Figura 3.26. Sin la implementación del *Bulkhead*, estos picos podrían afectar al funcionamiento de Niles en cuanto a la generación de un menú. Sin embargo, al separar las diferentes funcionalidades, se logran aislar errores, como por ejemplo el consumo excesivo de recursos que provocaría una degradación en la función del servicio de retornar el menú de determinado restaurante.

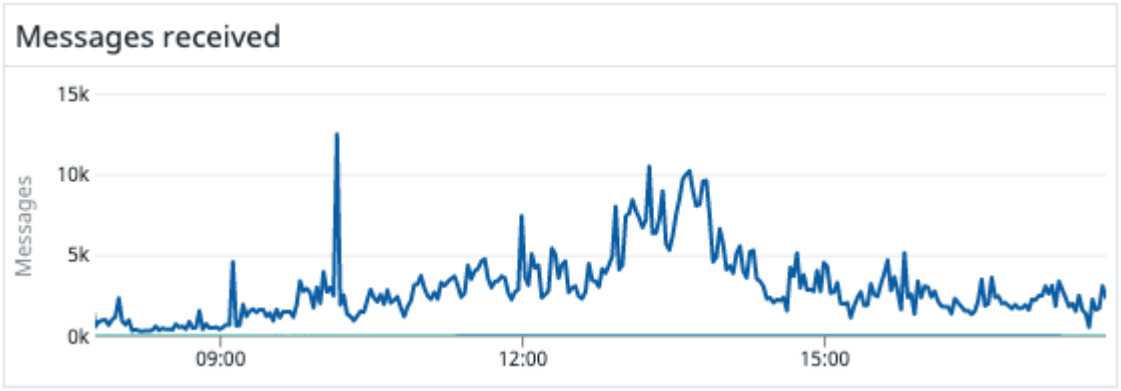

Figura 3.26 - Pico de 13000 novedades por minuto

3.3.6.Combinación de patrones

Es habitual que los patrones se usen en forma combinada, lo que posibilita el tratamiento de diferentes tipos de fallos. A continuación, se describe una combinación posible entre los patrones vistos.

3.3.6.1.La combinación *Timeout*, *Retry* y *Circuit Breaker*

Una posible combinación de patrones puede ser la aplicación conjunta de los patrones *Timeout*, *Retry* y *Circuit Breaker*. De hecho, la combinación mencionada es la que se utiliza por defecto en todos los clientes creados utilizando la librería *peya-ktor-utils*, resultando difícil encontrar un caso de uso donde no sea aplicable.

Cada patrón cumple su función encadenando acciones:

- El patrón *Timeout* es el primero en aplicarse, el cual define un tiempo de espera límite en las peticiones, evitando esperas largas, penalización de funcionalidades principales y encolado innecesario de peticiones.
- El patrón *Retry*, en caso de ser necesario, realiza reintentos ante fallos transitorios, como pueden ser errores de red o fallos por tiempo de espera límite alcanzado.
- El patrón *Circuit Breaker*, administrando internamente sus estados, pasa al estado “abierto” cuando el cliente de la petición HTTP falla repetidamente, llegando al umbral configurado.

Es necesario analizar cada caso de uso: teniendo en cuenta los tiempos de respuesta de cada servicio, se debe 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 estado del circuito sea

“abierto”, y retornar una petición fallida en caso de que no sea posible funcionar sin una dependencia.

Capítulo 4

Resultados Experimentales

En primer lugar, se describe el diseño experimental, el cual representa el proceso para analizar y evaluar el impacto de la aplicación de los patrones de resiliencia (Sección 4.1). Luego, se describe el trabajo experimental y se analizan los resultados obtenidos para los patrones *Timeout* (Sección 4.2), *Retry* (Sección 4.3), *Circuit Breaker* (Sección 4.4), *Bulkhead* (Sección 4.5), y una posible combinación de ellos (Sección 4.6).

4.1. Diseño Experimental

Para poder analizar y evaluar el impacto de la aplicación de los patrones, se realizaron dos procesos experimentales para cada uno de ellos:

1. El primer proceso consistió en enviar a Niles una serie de peticiones HTTP sobre uno o varios de sus recursos para emular el funcionamiento normal. Luego de un determinado tiempo, se sometió a Niles a una fuerte ráfaga de peticiones con el propósito de degradar su funcionamiento. Estas peticiones se realizaron a una versión de Niles que no incluía la aplicación del patrón en cuestión.
2. El segundo proceso consistió en replicar el anterior, pero en este caso a una versión de Niles que sí incluía la aplicación del patrón de interés. Para ambos casos se monitorizó el funcionamiento de Niles, para luego analizar los datos experimentales obtenidos. De esta manera, se pudieron cuantificar las ventajas y desventajas del uso de los patrones mencionados.

A continuación se describen varios aspectos vinculados al trabajo experimental.

Instalación de Niles: se utiliza la herramienta *Jarvis*. Existe la posibilidad de que para alguno de los patrones se deban realizar configuraciones o instalaciones adicionales en otros servicios, como por ejemplo *Favourites-service*.

Ejecución de una serie de peticiones HTTP: se realizan sobre uno o varios recursos disponibles dentro de Niles. Para ejecutar estas peticiones se utiliza el mecanismo provisto por *Jarvis*, el cual consiste en programar un *cron job*[29], que puede ser lanzado a demanda. Esta clase de tareas programadas es utilizada para pruebas de performance, habitualmente durante la madrugada, para no afectar al ecosistema de Pedidos Ya.

La tarea programada ejecuta un *script* de la herramienta K6[30], la cual es utilizada para realizar pruebas de performance, ofreciendo la funcionalidad de ejecutar requerimientos de

[29] https://en.wikipedia.org/wiki/Cron
[30] https://k6.io/

manera gradual y configurable. La configuración por defecto para las pruebas de estos experimentos es la que se puede apreciar en el siguiente fragmento de código; en caso de requerir variaciones, se realizarán las aclaraciones pertinentes:

```
{duration: '1m', target: 200}
{duration: '1m', target: 500}
{duration: '10m', target: 700}
```

Esta configuración (en formato JSON) significa que durante el primer minuto de ejecución, K6 hará lo posible para realizar 200 peticiones (lo cual dependerá de los recursos disponibles en el servidor donde se ejecuta K6). Durante el segundo minuto, la cantidad de requerimientos se aumenta para intentar llegar a 500. Finalmente, 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: el objetivo es degradar un servicio que es consumido por Niles, o alguna característica dentro del mismo servicio. Se realizará este proceso aproximadamente al minuto 5 de iniciadas las pruebas con K6.

Para realizar esta operación, se utiliza una herramienta de similar propósito a K6, llamada autocannon[31]. Esta herramienta resulta más simple de utilizar que K6, requiere menos configuraciones y no utiliza recursos de la compañía (para ejecutar las pruebas con K6 es necesario ejecutar procesos dentro de la infraestructura de PedidosYa tales como instancias de máquinas virtuales). Además, autocannon puede ser ejecutado localmente.

La configuración por defecto para realizar los experimentos es la se observa en la porción de código a continuación, y en caso de no ser la misma en determinado experimento, se detallarán los nuevos parámetros a utilizar.

```
autocannon  --renderStatusCodes -c 800 -d 300 -m "GET"
"https://favourites-service.dev.peja.co/users/1/favourites"
```

A continuación se explican los parámetros enviados al comando:

-c : hace referencia a la cantidad de conexiones concurrentes a usar.
-d : hace referencia a la duración en segundo que se ejecutarán peticiones.
-m : hace referencia al método[32] utilizado en la petición.
--renderStatusCodes : es una bandera que se envía para mostrar una grilla con los resultados de las peticiones, indicando la cantidad de requerimientos y su código de respuesta HTTP.
El último parámetro enviado es la URL sobre la cual se realizarán las peticiones.

El objetivo de ejecutar el comando autocannon es enviar una fuerte ráfaga de requerimientos que sature algún servicio o recurso utilizado por Niles. Si se enviaran

[31] https://github.com/mcollina/autocannon
[32] https://developer.mozilla.org/es/docs/Web/HTTP/Methods

peticiones de manera gradual, el servicio escalaría horizontalmente; es decir, aumentando en cantidad de instancias, lo que permitiría recibir más tráfico, interfiriendo con el propósito del experimento.

Análisis y evaluación del impacto del patrón: diagnóstico de problemas y evaluación de mejoras en base a la aplicación del patrón a experimentar. Para este trabajo, se utiliza la herramienta DataDog[33], la cuál además de ser un poderoso APM, el cuál brinda métricas de rendimiento, posee integración con diferentes elementos de infraestructura tales como bases de datos, cachés. DataDog también muestra e indexa *logs*, permite visualizar tráfico entrante a los servicios, y cuenta la funcionalidad de alertas y monitores, para enviar notificaciones en caso de incidentes o casos anormales.

4.2. Trabajo Experimental y Resultados Obtenidos para *Timeout*

4.2.1. Trabajo Experimental

Para ejecutar las pruebas del patrón *Timeout* y así poder obtener resultados experimentales se respetan todos los pasos indicados en la sección 4.1. Diseño Experimental.

4.2.2. Resultados en ausencia de *Timeout*

La ejecución de este experimento busca demostrar cómo, al no configurar correctamente los tiempos de espera en las peticiones HTTP, el menú que ofrece Niles se ve penalizado por los tiempos altos de un servicio de menor prioridad como es *Favourites-service*.

Cada petición del menú se verá penalizada en cuanto a tiempos, que se corresponderá con la espera de la respuesta de *Favourites-service*. Justamente este último consumirá tiempo innecesario en medio de un periodo de latencia alta.

Siguiendo la configuración enumerada en la sección 4.1. Diseño experimental, se procede a realizar la instalación del Niles sin la aplicación de ningún patrón. El próximo paso es iniciar la ejecución de K6 realizando peticiones sobre el endpoint que ofrece el menú dentro de Niles. Aproximadamente a los cinco minutos de inicio de la ejecución, se procede a enviar una ráfaga de peticiones a *Favourites-service*. El resultado de esta ejecución se ilustra en la Figura 4.1, donde se puede observar claramente que ante la fuerte ráfaga de requerimientos (imagen de la izquierda), los tiempos de respuesta crecen significativamente (imagen de la derecha).

[33] https://www.datadoghq.com/

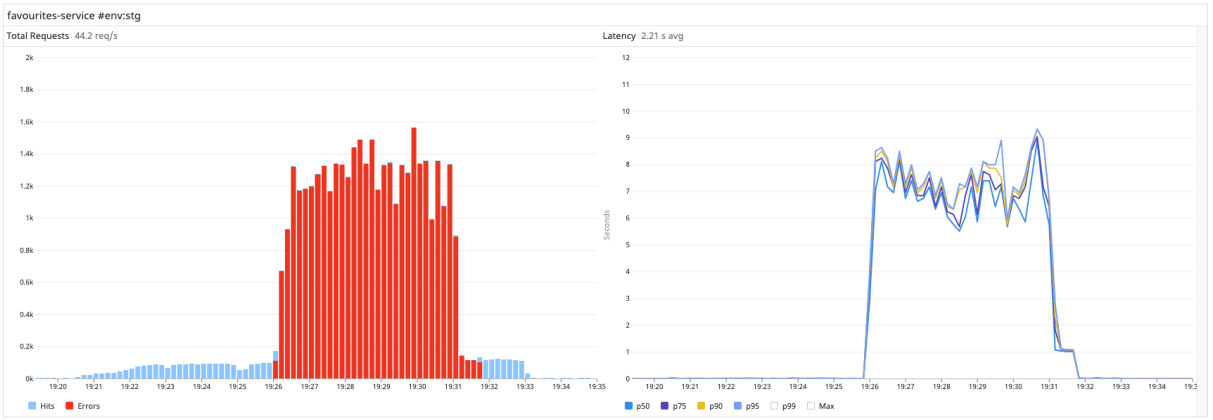

Figura 4.1 - Tráfico entrante (izq) y tiempos de respuesta de *Favourites-service* (der)

Si se analiza el estado de Niles, se puede ver cómo se ve afectado en consecuencia de la degradación de *Favourites-service*, tal como lo muestra la Figura 4.2:

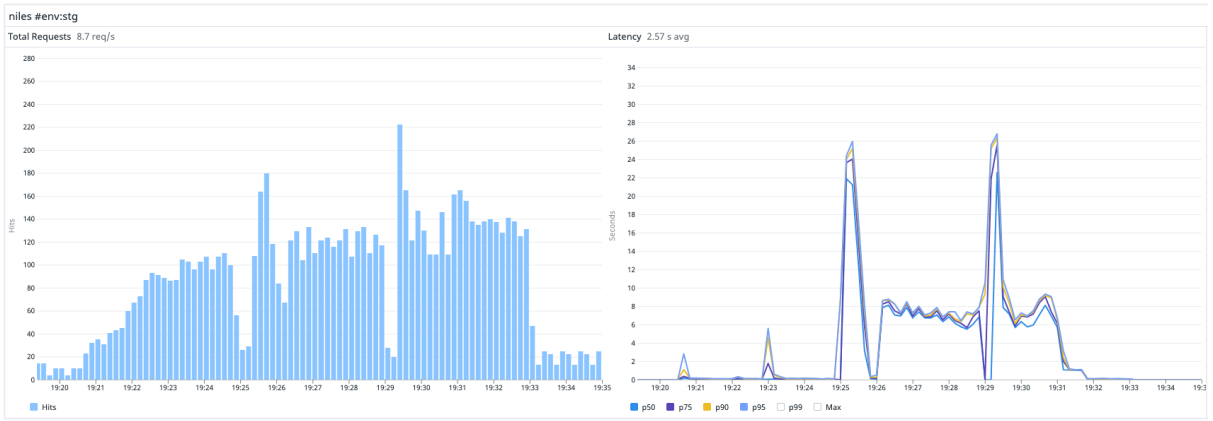

Figura 4.2 - Tráfico entrante (izq) y tiempos de respuesta de Niles (der)

En la Figura 4.2 (izq) se ve como el tráfico no sigue un patrón usual (alterna periodos de pocas y excesivas peticiones), lo que evidencia que existen hilos en espera de ser ejecutados en algunos momentos. En la derecha de la Figura 4.2 se observa cómo los tiempos del menú se ven afectados ya que existen peticiones en espera desde *Favourites-service*.

Para concluir con la primera parte del experimento (ausencia del patrón *Timeout*), se muestra en la Figura 4.3 el tiempo consumido de cada servicio en las peticiones del menú:

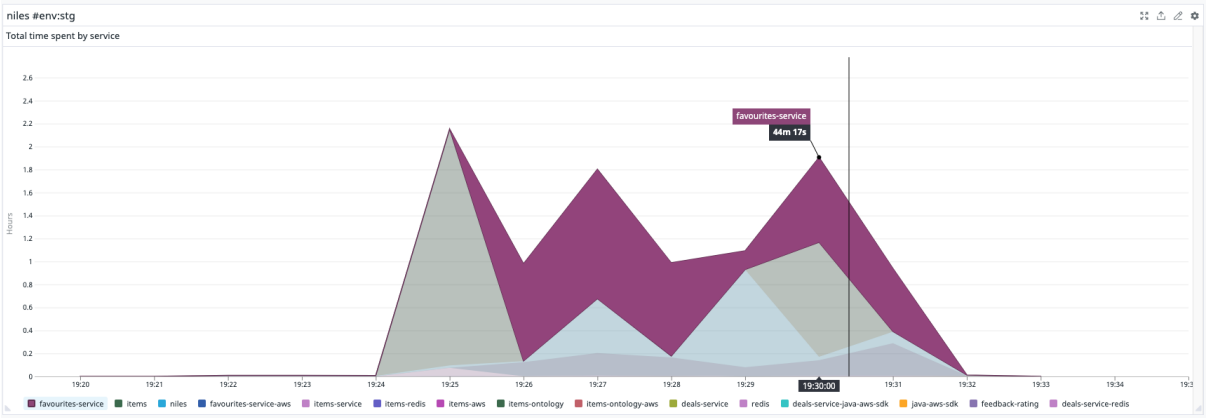

Figura 4.3 - Tiempo consumido por los diferentes servicios, donde Favourites-service consume un gran porcentaje

En esta figura queda claro como el tiempo de respuesta en la petición de un menú se ve penalizado por un servicio de menor importancia, como es *Favourites-service*. La causa de este mal funcionamiento se debe a que no existe una correcta priorización en cuanto a funcionalidades mandatorias y a la ausencia de un mecanismo que permita tomar decisiones ante una espera excesiva.

4.2.3.Resultados en presencia de *Timeout*

Para iniciar con el experimento, se configura un tiempo límite de espera en cada petición HTTP saliente desde Niles, tal como se muestra en el siguiente fragmento de código (resaltado en verde):

```
class FavouritesClient(
    override val config: HttpClientConfiguration,
    override val client: Client,
    override val registry: MeterRegistry,
    override val resilience: ResilienceProviders = ResilienceProviders(
      listOf(TimeoutPolicyFactory.create(TimeoutConfiguration("timeout-fav", Duration.ofMillis(100))))
    )
)
```

El siguiente paso es repetir el proceso experimental seguido en la sección anterior, sólo que esta vez en presencia del patrón *Timeout*. En la Figura 4.4 se ilustran los requerimientos realizados hacia *Favourites-service* utilizando *autocannon*. Debido a la configuración utilizada, los resultados son prácticamente idénticos a los mostrados en la Figura 4.1, lo cual resulta esperable.

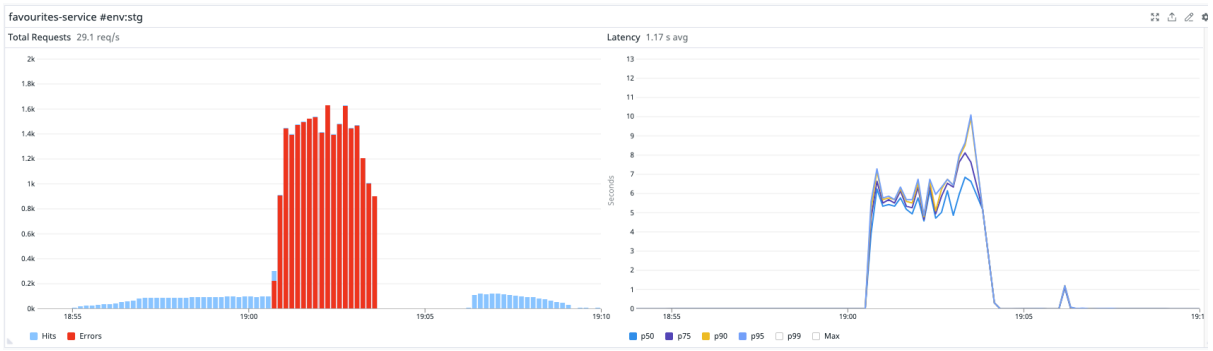

Figura 4.4 - Tráfico entrante a *Favourites-service* con ráfagas de peticiones (izq) y tiempos de respuesta (der)

En la Figura 4.5 (izq) se puede observar cómo el tráfico entrante aumenta de forma estable, sin alternar periodos de poco y mucho tráfico. A la derecha, se puede ver que el tiempo de latencia no supera los 170 milisegundos (salvo por dos picos), lo que resulta mucho menor a los 6000 milisegundos en promedio sin la aplicación del patrón en cuestión. Esta mejora en el rendimiento se debe a que se finaliza intencionalmente la comunicación con las dependencias luego de cumplir un tiempo determinado.

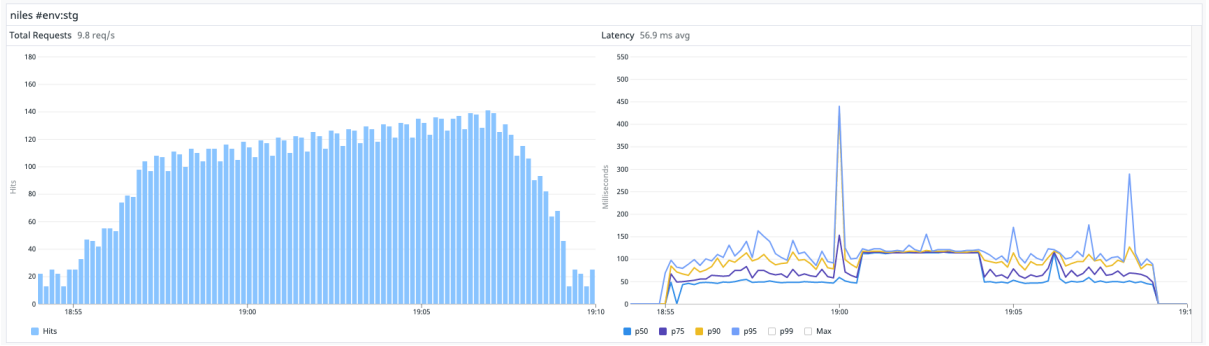

Figura 4.5 - Tráfico entrante (izq) y tiempos de respuesta (der) en Niles con aplicación de *Timeout*

En lo que respecta al tiempo consumido por cada servicio, se muestra en la Figura 4.6 que con la aplicación del patrón *Timeout*, incluso en medio de un incidente en *Favourites-service*, este no consume más tiempo del debido.

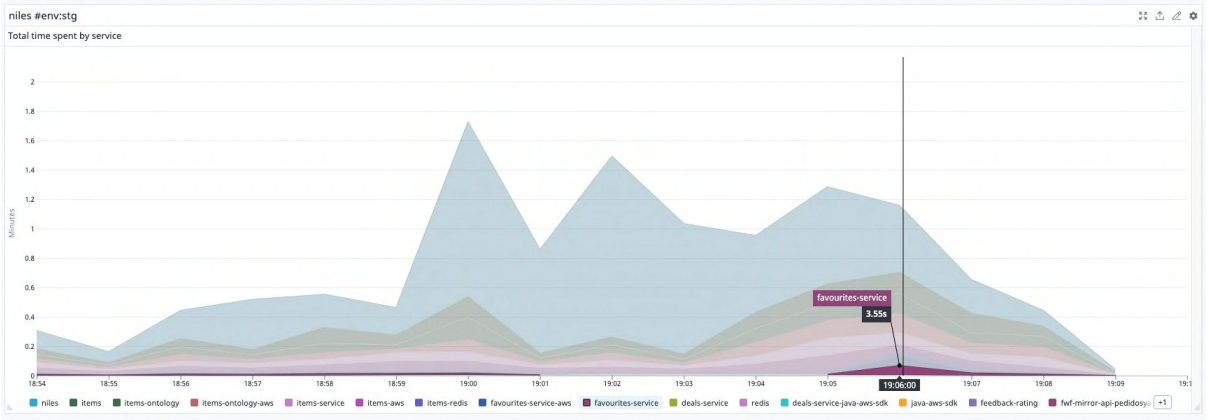

Figura 4.6 - Tiempo consumido por *Favourites-service* cuando se aplica el patrón *Timeout*

En la Figura 4.7 se puede notar la relación entre Niles y *Favourites-service* cuando el patrón *Timeout* está presente. Cuando *Favourites-service* sufre una ráfaga de requerimientos y se ve degradado, éste no penaliza al menú, ya que la comunicación se finaliza luego de 100 milisegundos.

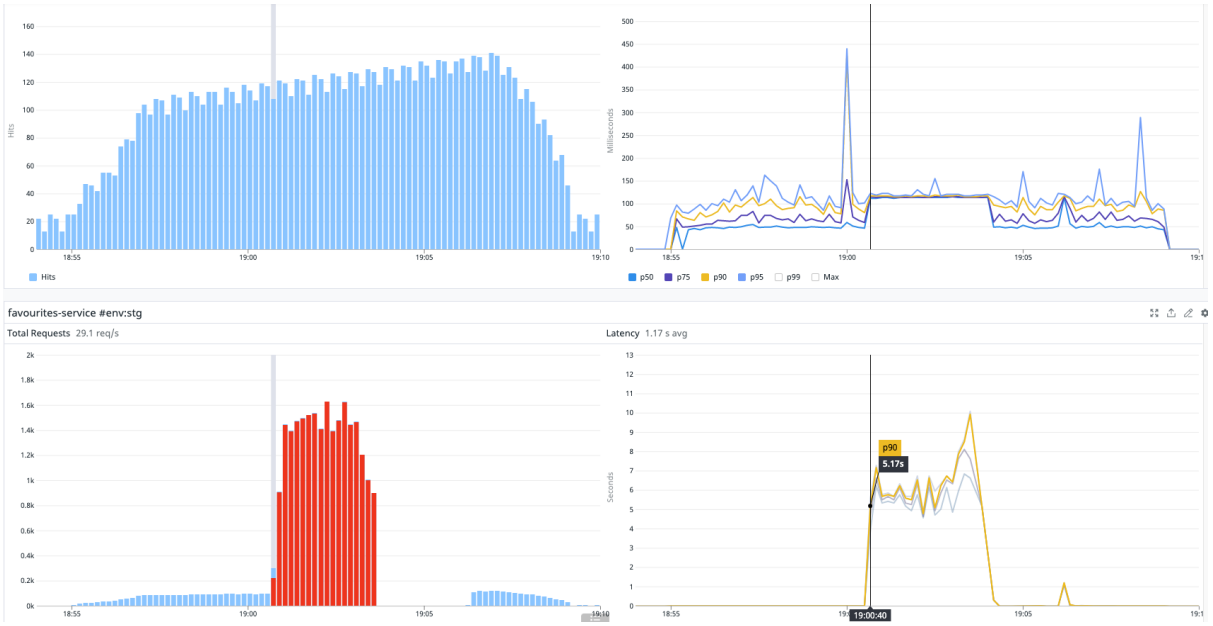

Figura 4.7 - Comportamiento de Niles y sus tiempos de respuesta (arriba) y comportamiento de Favourites-service con sus tiempos de respuesta (abajo) en presencia de *Timeout*

4.3. Trabajo Experimental y Resultados Obtenidos para *Retry*

4.3.1. Trabajo Experimental

Para ejecutar las pruebas del patrón *Retry*, y obtener resultados experimentales, se ejecutan algunos de los pasos mencionados en la sección 4.1.Diseño Experimental. Para este caso en particular no se ejecutarán rafagas de peticiones sobre *Favourites-service*, pero adicionalmente, se realiza una modificación en el código de *Favourites-service* para simular el fallo en una de cada diez peticiones. El error obtenido en la petición fallida es el que se muestra en la Figura 4.8:

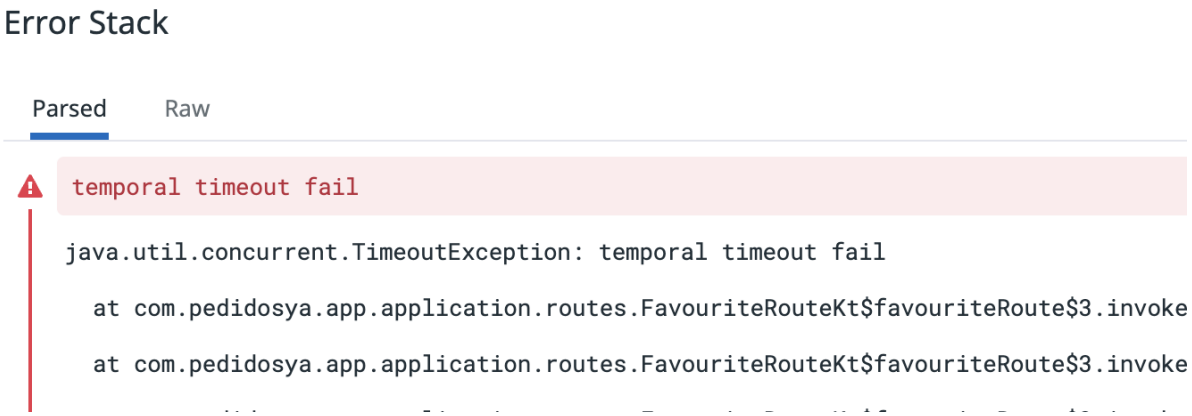

Figura 4.8 - Fallo de petición en *Favourites-service*

El objetivo es demostrar cómo se comportan los servicios con respecto a la aplicación del patrón *Retry* y los fallos transitorios como una respuesta lenta o errores temporales en la red.

4.3.2.Resultados en ausencia de *Retry*

Siguiendo los pasos de la sección 4.1.Diseño Experimental se procede a instalar una versión de Niles sin la aplicación del patrón *Retry*. Luego, se ejecuta el proceso de pruebas utilizando K6. En la Figura 4.9 se puede observar el tráfico recibido en Niles, donde los errores se marcan en color rojo:

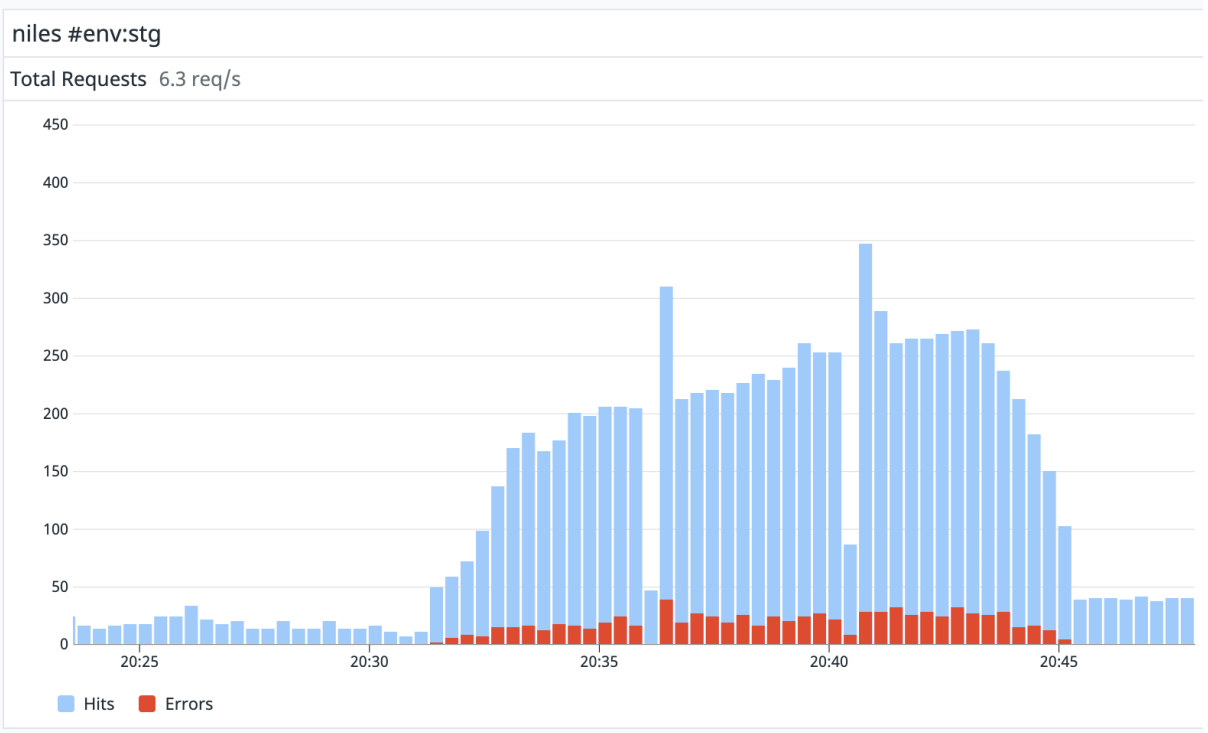

Figura 4.9 - Tráfico entrante en Niles

Si se hace foco en el tipo de errores, como se muestra en la Figura 4.10, se puede ver que están catalogados como errores pero con código de estado número 200. Esto significa que las peticiones recibidas en Niles, finalizan con estado exitoso (se retorna el menú al

usuario), pero en la ejecución existió algún tipo de error. Este error es el recibido desde *Favourites-service* y, al no ser obligatorio para generar el menú, no penaliza al mismo. Aun así, este tipo de errores son visibles desde Datadog.

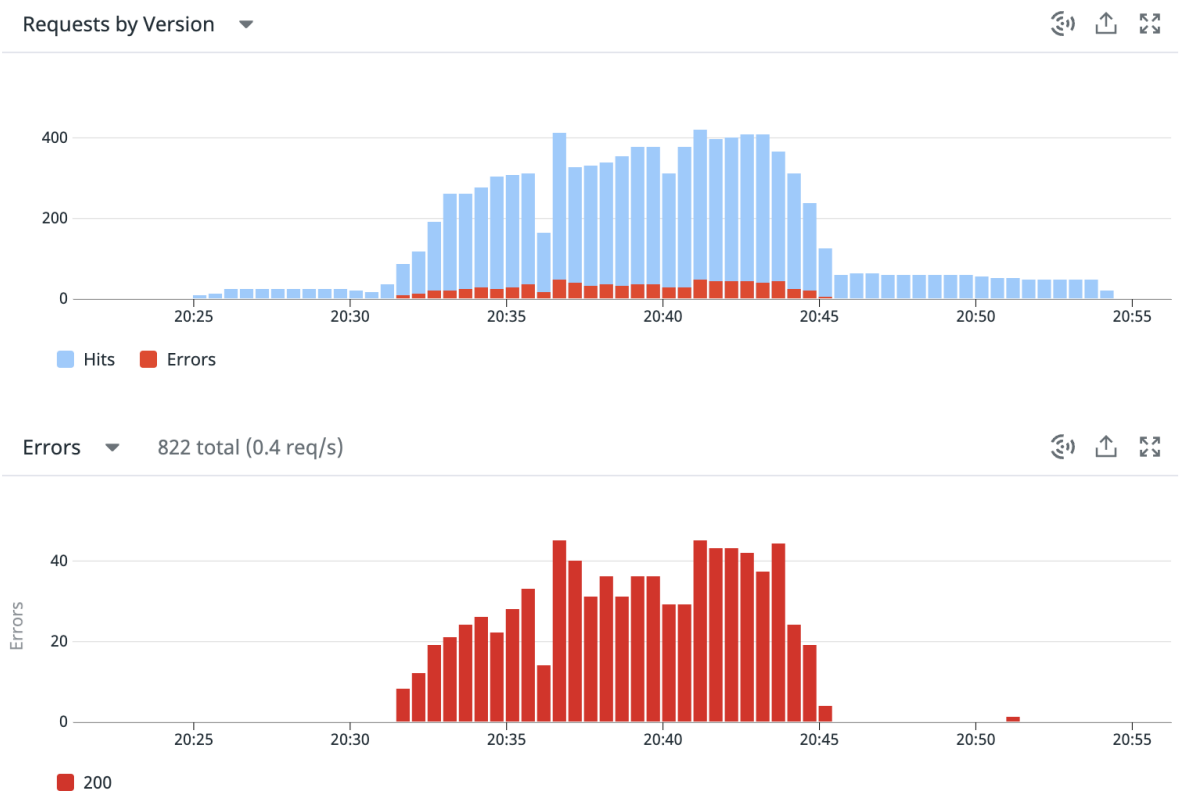

Figura 4.10 - Peticiones hacia Niles (arriba) y cantidad de errores con código de estado 200 (abajo)

4.3.3. Resultados en presencia de *Retry*

Para realizar la configuración de reintentos en la comunicación desde Niles hacia *Favourites-service*, se envía por parámetros al cliente que realiza la petición HTTP, el correspondiente *Provider* para que administre los reintentos, tal como se puede ver en el siguiente fragmento de código (resaltado en verde).

```
class FavouritesClient(
    override val config: HttpClientConfiguration,
    override val client: Client,
    override val registry: MeterRegistry,
    override val resilience: ResilienceProviders = ResilienceProviders(
        listOf(RetryPolicyFactory.create(RetryConfiguration("retry-fav")))
    )
)
```

La configuración por defecto indica que la cantidad de reintentos es de 3 como lo indica el siguiente fragmento de código (resaltado en verde).

```
data class RetryConfiguration(
    val name: String,
    val maxAttempts: Int = 3,
    val retryOnStatuses: List<Int> = listOf(HttpStatus.SC_INTERNAL_SERVER_ERROR,
HttpStatus.SC_BAD_GATEWAY),
    val failAfterMaxRetries: Boolean = false,
    val retryOnResult: (Any) -> Boolean = { response -> (response as Response).status.value in
retryOnStatuses },
    val ignoreExceptions: List<Class<out Throwable>> = listOf(
        HttpNotFoundException::class.java),
    val exponentialBackoff: ExponentialBackoff = ExponentialBackoff(Duration.ofMillis(30), 1.5, 0.5)
)
```

Luego de ejecutar las pruebas de carga en la versión de Niles que hace uso del patrón *Retry*, se puede ver el tráfico recibido y los errores marcados en rojo en la Figura 4.11.

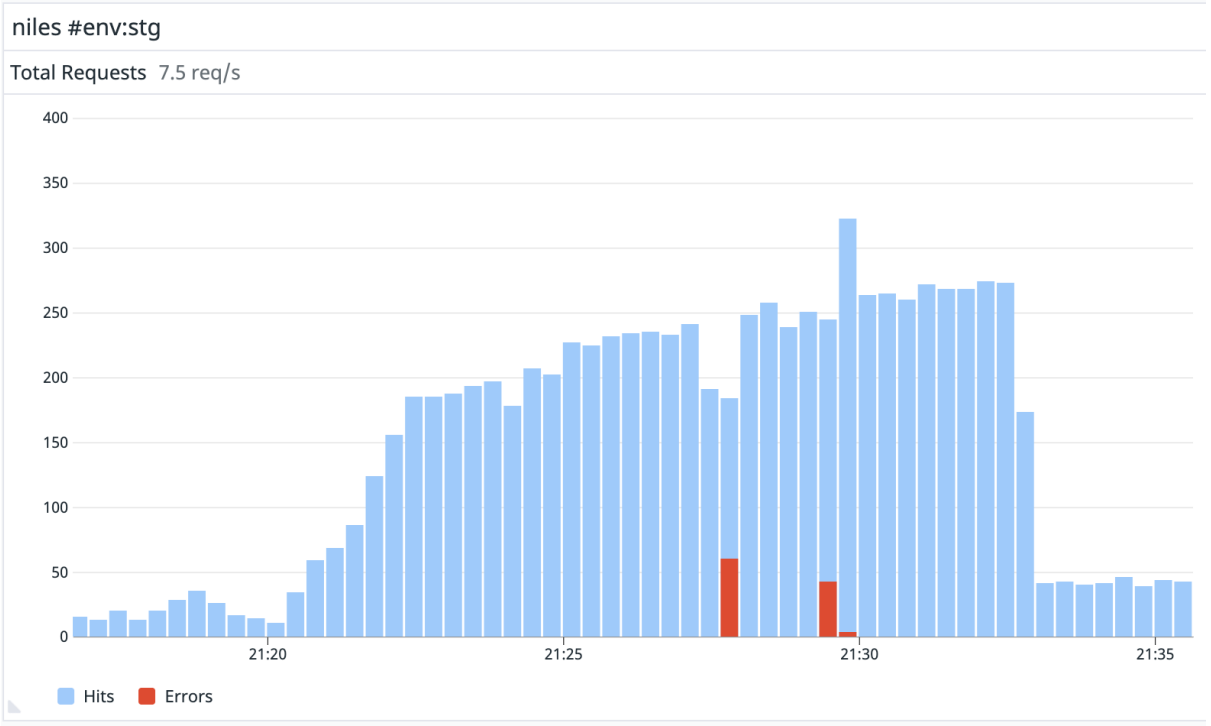

Figura 4.11 - Tráfico entrante en Niles cuando se aplica el patrón *Retry*

Focalizando sobre los errores ocurridos, la Figura 4.12 hace evidente el estado de respuesta con código 200:

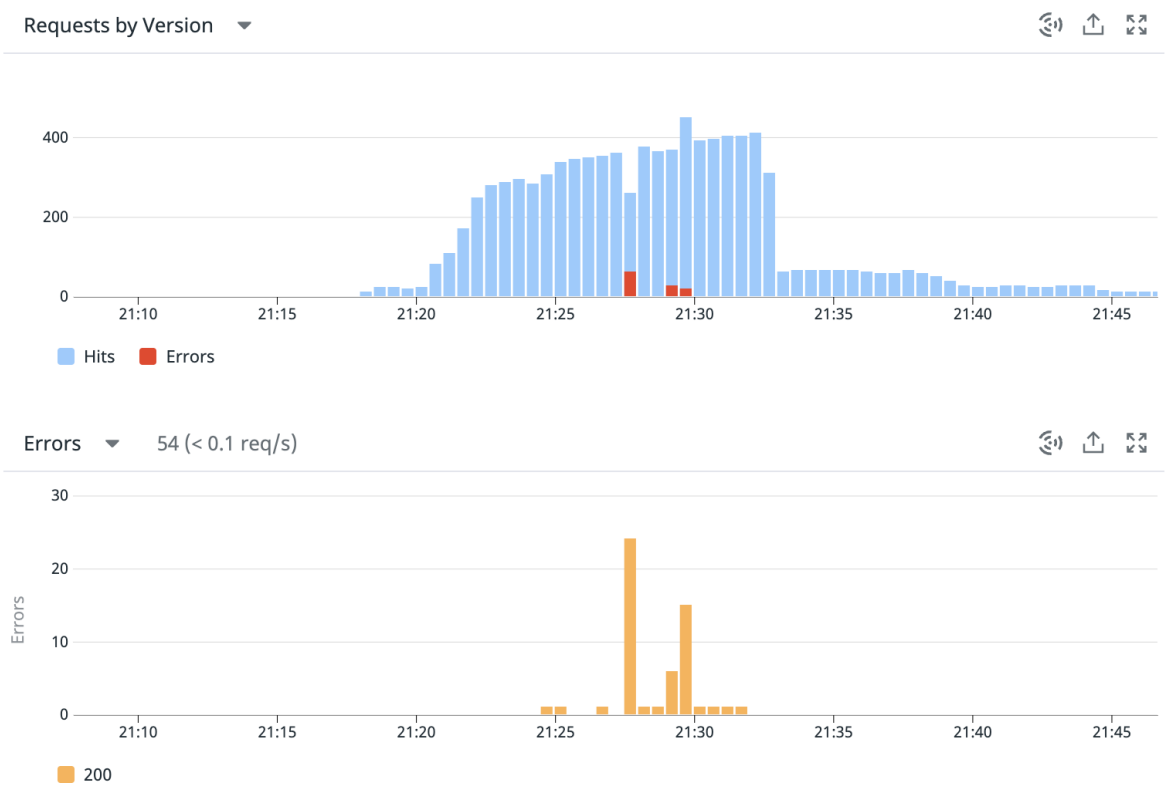

Figura 4.12 - Peticiones entrantes hacia Niles (arriba), y respuestas con error, código de estado 200 (abajo) cuando se aplica al patrón *Retry*

Si bien aún existen errores, la diferencia es notable cuando se aplica el patrón, ya que el contador que se puede apreciar en la Figura 4.12 muestra una cantidad total de errores de 54, cuando en ausencia de *Retry* este valor ascendía a 822 (ver Figura 4.10).

Para concluir el experimento, se muestra en la Figura 4.11 una comparación de ambas versiones (sin patrón aplicado y con la aplicación del mismo), para hacer visible la cantidad de errores en cada una. Resulta evidente la menor ocurrencia de errores cuando el patrón *Retry* está presente.

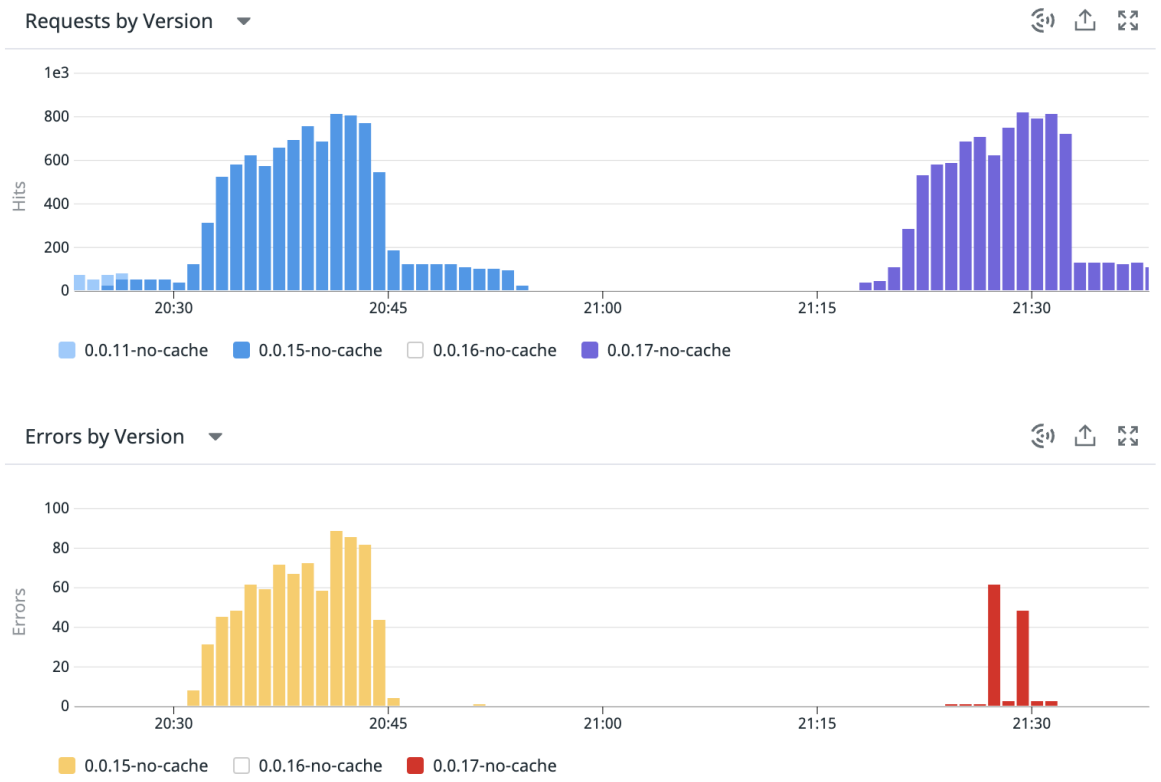

Figura 4.11 - Comparación de errores cuando se aplica el patrón *Retry*

4.4. Trabajo Experimental y Resultados Obtenidos para *Circuit Breaker*

4.4.1. Trabajo Experimental

El proceso llevado a cabo para obtener resultados experimentales para el patrón *Circuit Breaker* es exactamente el mismo que el utilizado en la sección 4.1.Diseño Experimental.
El objetivo de las pruebas es demostrar cómo ante la aplicación del patrón, el servicio consumidor es capaz de no utilizar recursos realizando peticiones que tienen muchas probabilidades de fallar.

4.4.2. Resultados en ausencia de *Circuit Breaker*

Los resultados de la ejecución del experimento sin el patrón *Circuit Breaker* son iguales a los de la sección 4.2.2.Resultados en ausencia de *Timeout* ya que es una versión libre de aplicación de patrones.

En la Figura 4.12 se puede observar el aumento de tráfico hacia Niles y los errores provocados intencionalmente en *Favourites-service* (marcados en color rojo). En la parte superior, se refleja el tráfico entrante a Niles y sus tiempos de respuesta, mientras que en la parte inferior se ilustra lo correspondiente a *Favourites-service*.

Figura 4.12 - Tráfico entrante y tiempos de respuesta en Niles (arriba) y *Favourites-service* (abajo)

En la Figura 4.13 (izq) se puede observar cómo las peticiones salientes desde Niles hacia *Favourites-service* son continuas, más allá del estado de éste último. Como *Favourites-service* se ve degradado, su tiempo de respuesta aumenta, penalizando al menú, como se puede notar a la derecha de la Figura 4.13.

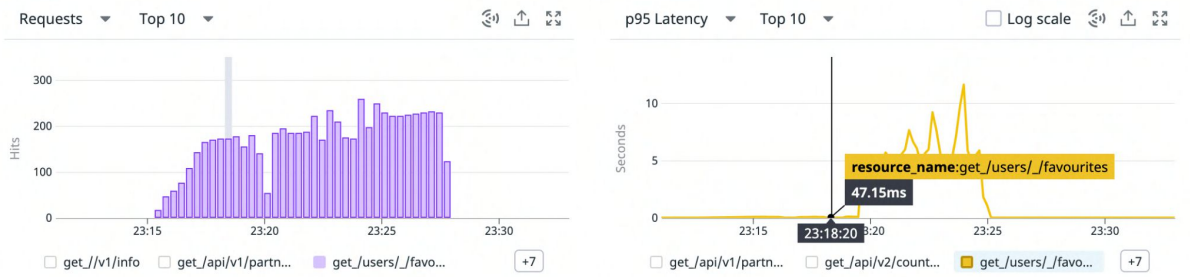

Figura 4.13 - Peticiones desde Niles hacia *Favourites-service* (izq) y tiempo de respuesta de *Favourites-service* (der)

4.4.3.Resultados en presencia de *Circuit Breaker*

Para aplicar el patrón *Circuit Breaker* en la comunicación entre Niles y *Favourites-service*, se agrega el siguiente código al cliente HTTP (resaltado en verde):

```
class FavouritesClient(
    override val config: HttpClientConfiguration,
    override val client: Client,
    override val registry: MeterRegistry,
    override val resilience: ResilienceProviders = ResilienceProviders(
        listOf(CircuitBreakerPolicyFactory.create(CircuitConfiguration("circuit-fav")))
    )
)
```

Luego de ejecutar la prueba de carga, se pueden observar una serie de resultados. En primer lugar, la Figura 4.14 exhibe cómo se reduce el envío de peticiones hacia *Favourites-service* durante su degradación, lo que conduce a un importante ahorro de recursos.

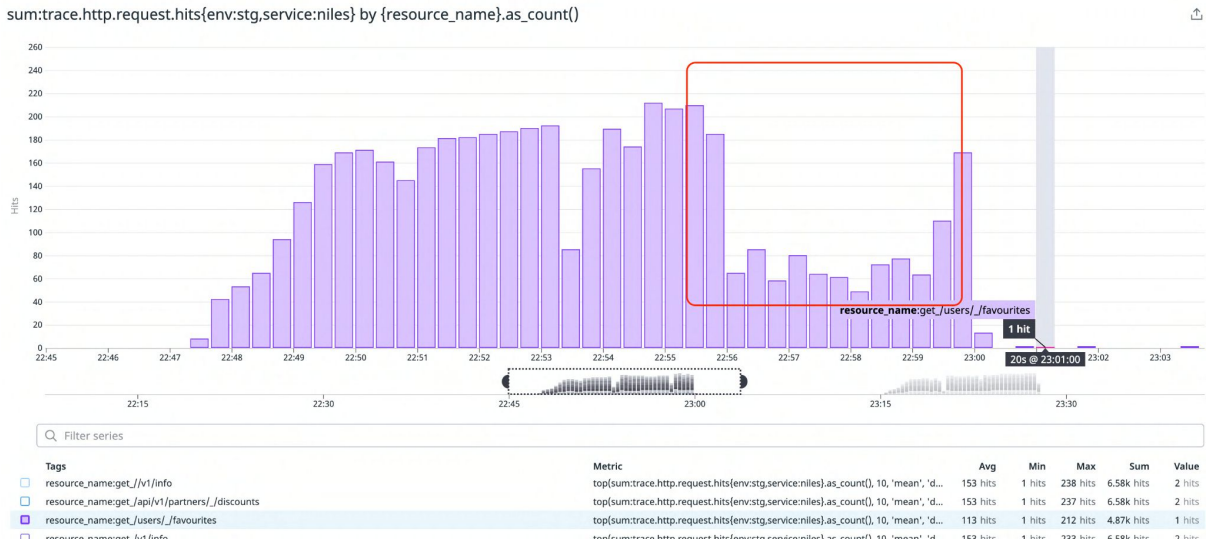

Figura 4.14 - Ahorro de recursos con la ayuda del patrón *Circuit Breaker*

En la figura anterior se puede notar que el lapso donde el tráfico disminuye es durante el horario 22:56hs y 22:59hs aproximadamente. Esta información se puede reforzar observando los registros de alerta de Niles, como se muestra en las Figuras 4.15 y 4.16:

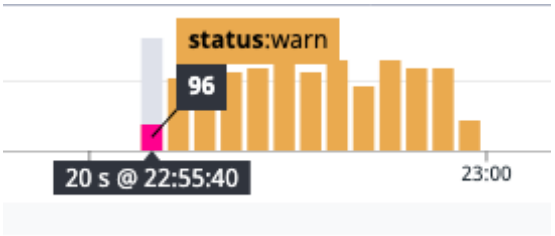

Figura 4.15 - Lapso de tiempo en el que se observan registros de alerta

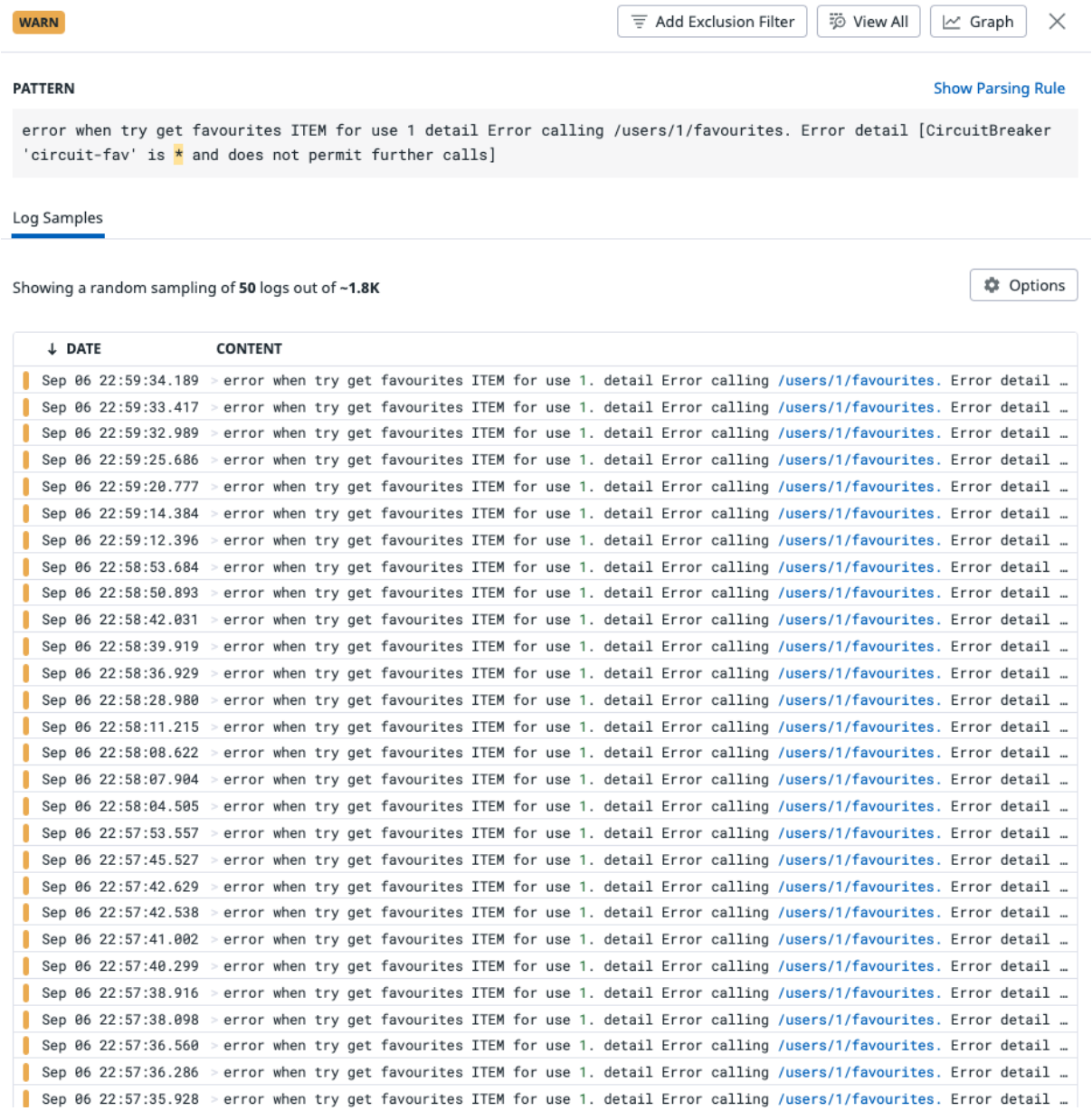

Figura 4.16 - Detalle de registros de alerta en Niles

Como segundo resultado, se puede observar en la Figura 4.17 que durante el lapso que no se envían peticiones hacia *Favourites-service*, los tiempos de respuesta del menú son bajos. Este buen funcionamiento se debe a que no se continúa esperando por una respuesta que posiblemente falle o nunca llegue a resolverse.

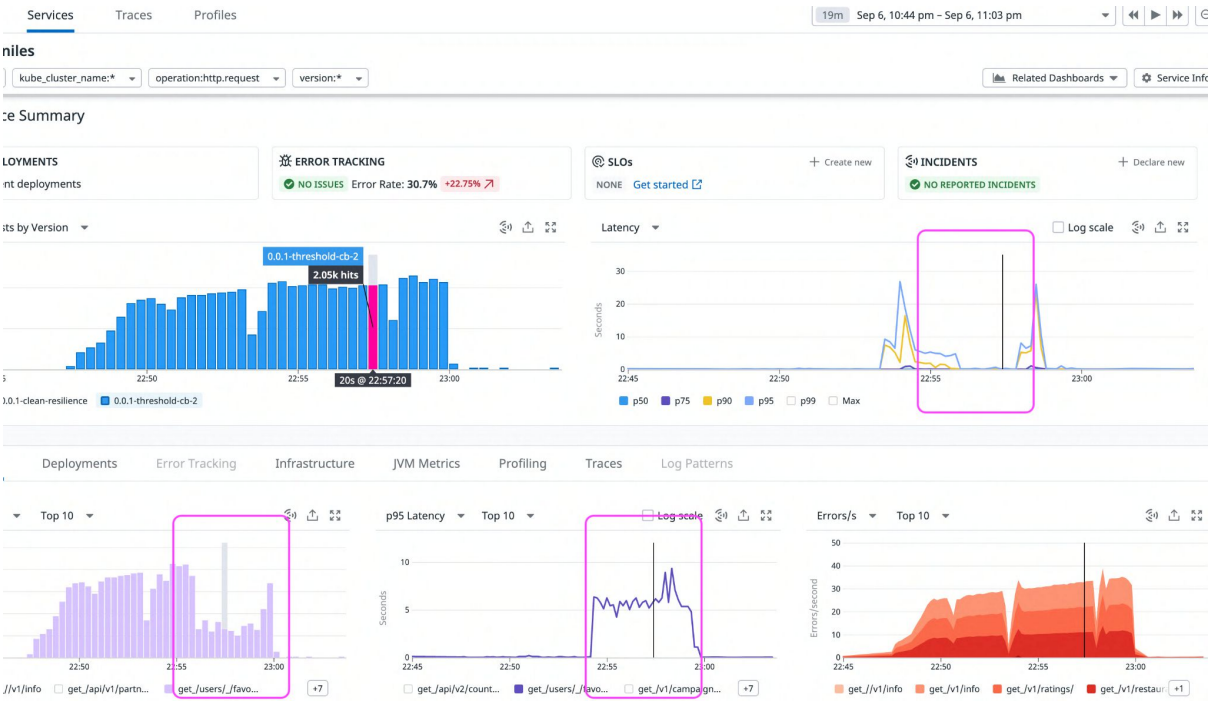

Figura 4.17 - Beneficios en cuanto a tiempos al aplicar el patrón *Circuit Breaker*

En la Figura 4.17 se han marcado 3 rectángulos en color violeta que facilitan la identificación de los beneficios de la aplicación de *Circuit Breaker*. En el rectángulo de la izquierda, se puede notar el lapso de tiempo donde no se realizan peticiones hacia *Favourites-service*, debido a que el *Circuit Breaker* pasa a un estado *abierto*. En el rectángulo del centro, se puede ver que los tiempos de respuesta de *Favourites-service* durante el incidente provocado superan los 5000 milisegundos. Por último, en el rectángulo de la derecha, se puede observar el tiempo de respuesta del menú, el cuál no supera los 500 milisegundos. Este tiempo de respuesta representa una mejora de 10 veces, respecto a no aplicar el patrón en cuestión.

La aplicación del patrón además de permitir ahorrar recursos, ofrece mejoras en cuanto a la performance del servicio debido a que es posible utilizar respuestas por defecto en determinados casos de uso, esto significa que cuando el estado del circuito es *abierto*, en lugar de realizar una petición para obtener los productos favoritos, se omite y se retorna un listado vacío de favoritos.

4.5. Trabajo Experimental y Resultados Obtenidos para *Bulkhead*

4.5.1. Trabajo Experimental

Como se mencionó anteriormente, la funcionalidad principal de Niles es generar el menú de un restaurante. Además de esta, también ofrece la posibilidad de entregar productos para realizar *Cross Selling*[34].

El proceso experimental difiere levemente del usado para los patrones anteriores. Se ejecutarán la misma cantidad de peticiones que la sección 4.1.Diseño Experimental

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

utilizando K6, pero ahora sobre cada recurso disponible. Es decir, además de realizar peticiones al menú, también se efectuarán al recurso de *Cross Selling*. Se detalla la configuración de peticiones en el siguiente fragmento de código:

```
{duration: '1m', target: 200}
{duration: '1m', target: 500}
{duration: '10m', target: 700}
```

Al igual que en los experimentos anteriores, el objetivo es provocar la degradación de un recurso. En este caso, se realizarán ráfagas de peticiones sobre el recurso de *Cross Selling* para poder visualizar el comportamiento del servicio Niles cuando todas la funcionalidades conviven en el mismo servidor (en ausencia del patrón *Bulkhead*) y cuando lo hacen en diferentes (en presencia del patrón *Bulkhead*).

Durante el lapso de 12 minutos se ejecutarán peticiones a los dos siguientes *endpoints*:

https://niles.dev.peja.co*/v1/niles/partners/%partnerId%/menus*
https://niles.dev.peja.co*/v2/niles/partners/%partnerId%/cross-selling*

Resulta importante aclarar que el servidor donde está instalado Niles es el mismo para atender ambas peticiones. Luego de 3 minutos de iniciado el proceso de realizar peticiones con K6, se lanzan ráfagas de peticiones sobre el recurso de *cross-selling* utilizando el siguiente comando:

```
autocannon --renderStatusCodes -c 1200 -d 300 -p3 -m "GET" \
"https://niles.dev.peja.co/v1/niles/partners/34114/cross-selling"
```

4.5.2.Resultados en ausencia de *Bulkhead*

Al finalizar el periodo de prueba, se analiza el comportamiento de Niles. En la Figura 4.18 se puede observar como al momento de iniciar las ráfagas de peticiones, Niles presenta errores (marcados en rojo).

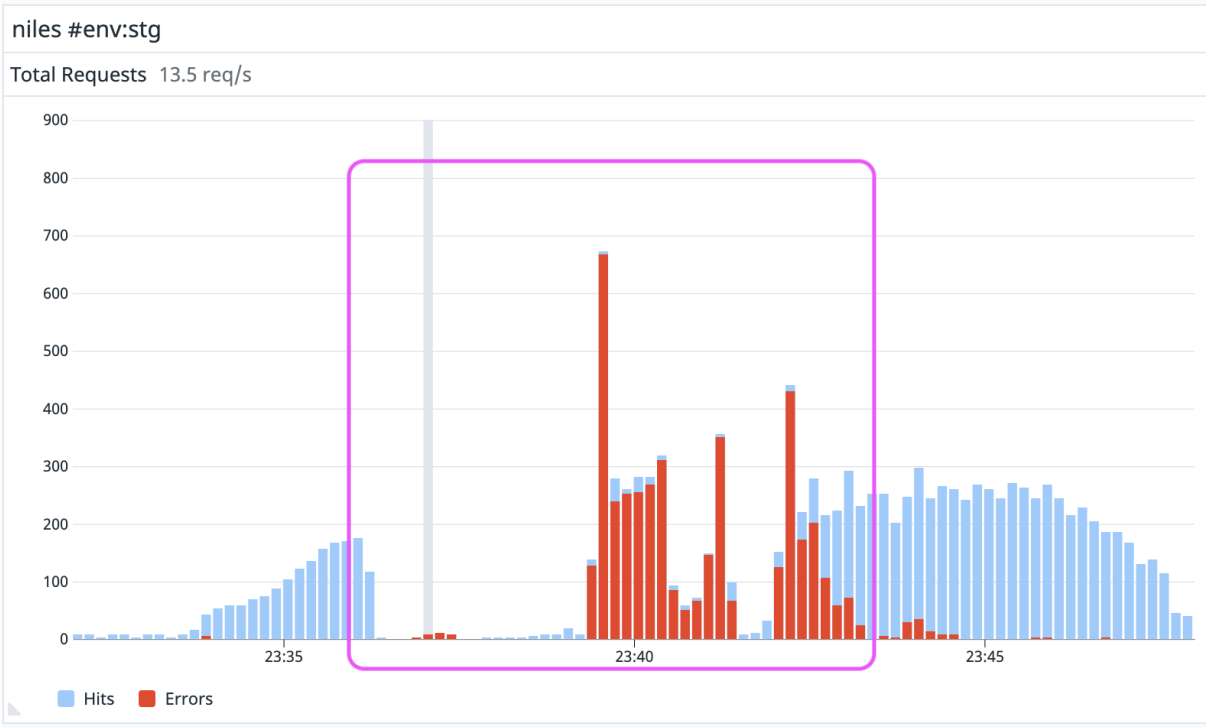

Figura 4.18 - Errores en todos los recursos que ofrece Niles

Como la ráfaga de peticiones es fuerte, esta deja al servicio no 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. En la Figura 4.19 se puede apreciar como al solicitar el menú, la respuesta es errónea (con código de error 503).

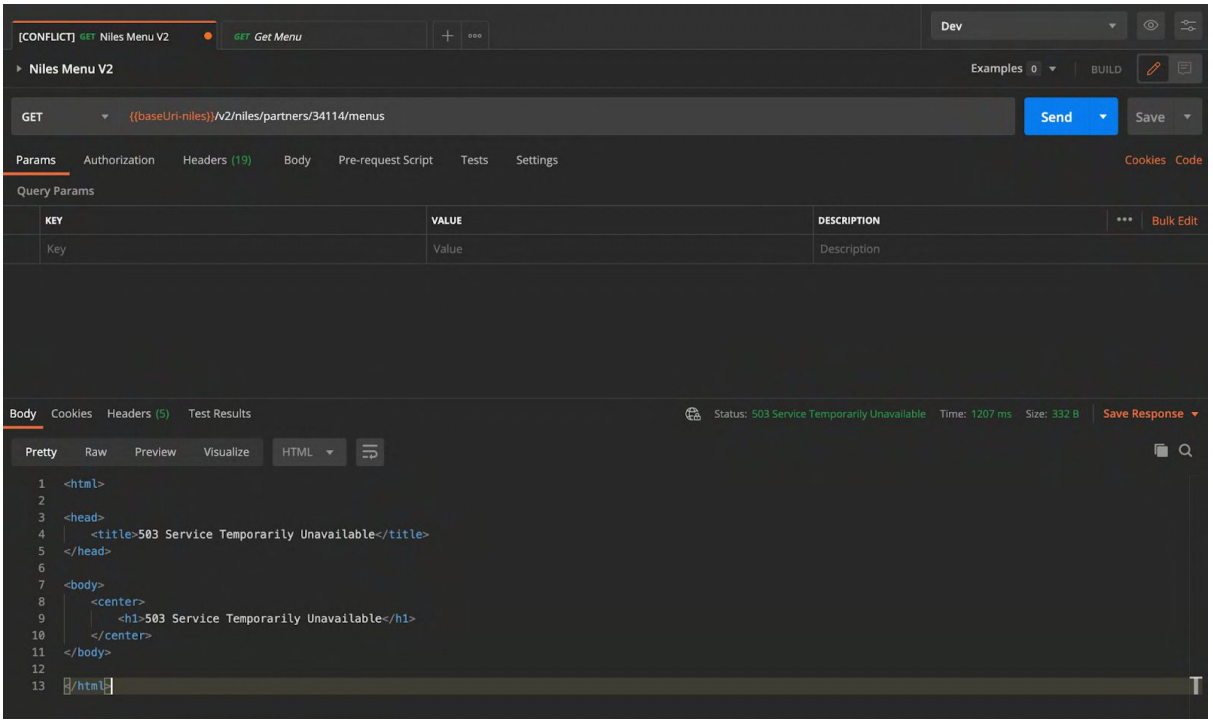

Figura 4.19 - Petición fallida durante la degradación del servicio utilizando Postman[35]

En resumen, los resultados obtenidos muestran cómo las peticiones constantes a un recurso de menor prioridad pueden impactar en la funcionalidad principal del servicio.

4.5.3.Resultados en presencia de *Bulkhead*

Para la aplicación del patrón *Bulkhead*, se siguen los pasos detallados en la sección 3.3.4.2.Implementación, la cual consta básicamente de la creación de un nuevo *pool* contenedor de aplicaciones.

A diferencia del caso anterior, se cuenta con dos *webpools* en lugar de uno sólo. Cada uno de los *webpools* contiene exactamente una instancia de la misma versión de Niles. Además, gracias a *jarvis*, se les asigna diferente DNS.

A partir de esta nueva configuración, las peticiones se realizan contra dos instancias diferentes de Niles: la primera ***niles.dev***, procesa las peticiones del menú; la segunda ***niles-worker.dev***, procesa las peticiones de *cross-selling*. De esta manera, las peticiones son realizadas sobre los dos siguientes *endpoints*:

https://niles.dev.peja.co*/v1/niles/partners/%partnerId%/menus*
https://niles-worker.dev.peja.co*/v2/niles/partners/%partnerId%/cross-selling*

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. Aproximadamente a los tres minutos de

[35] Postman es una herramienta para realizar peticiones HTTP.

iniciadas las peticiones desde K6, se realizan ráfagas de requerimientos sobre el recurso de *cross-selling*. En la Figura 4.20 se pueden visualizar las peticiones realizadas sobre ambos recursos (menú y *cross-selling*) junto a las ráfagas provocadas para degradar el servicio.

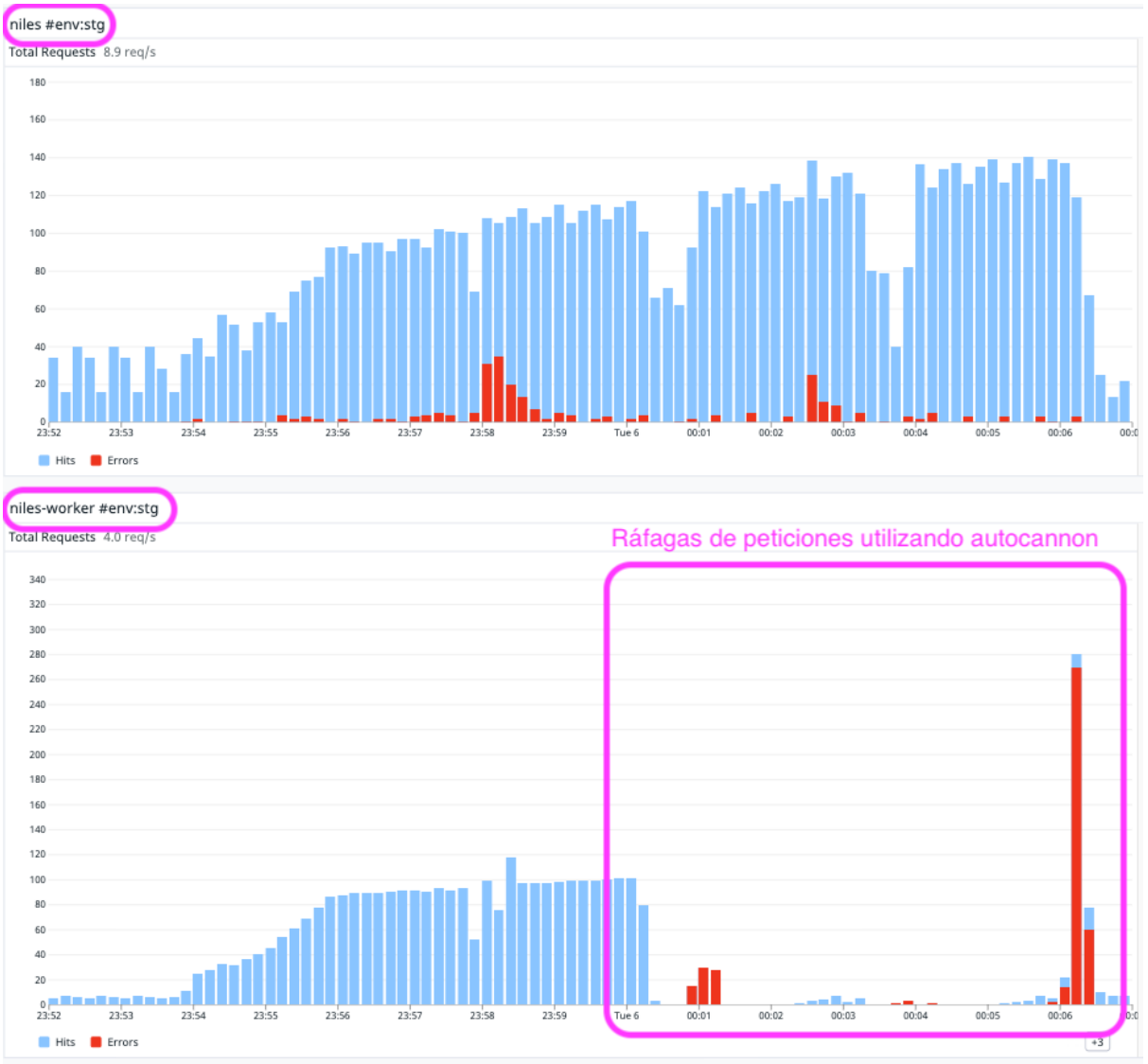

Figura 4.20 - Tráfico y errores en Niles sobre el recurso del menú (arriba) y tráfico y errores en Niles en otro *webpool* asociado a *cross selling* (abajo)

En el *webpool worker*, la ráfaga de peticiones es lo suficientemente fuerte para dejar a el servicio no disponible, incluso al nivel de no permitir más peticiones. Luego, como se ve en la Figura 4.20 (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 procede a solicitar la información del menú a las instancias de Niles que no recibieron ráfagas de peticiones. Como se muestra en la Figura 4.21, la respuesta es recibida correctamente, siendo posible visualizar el menú.

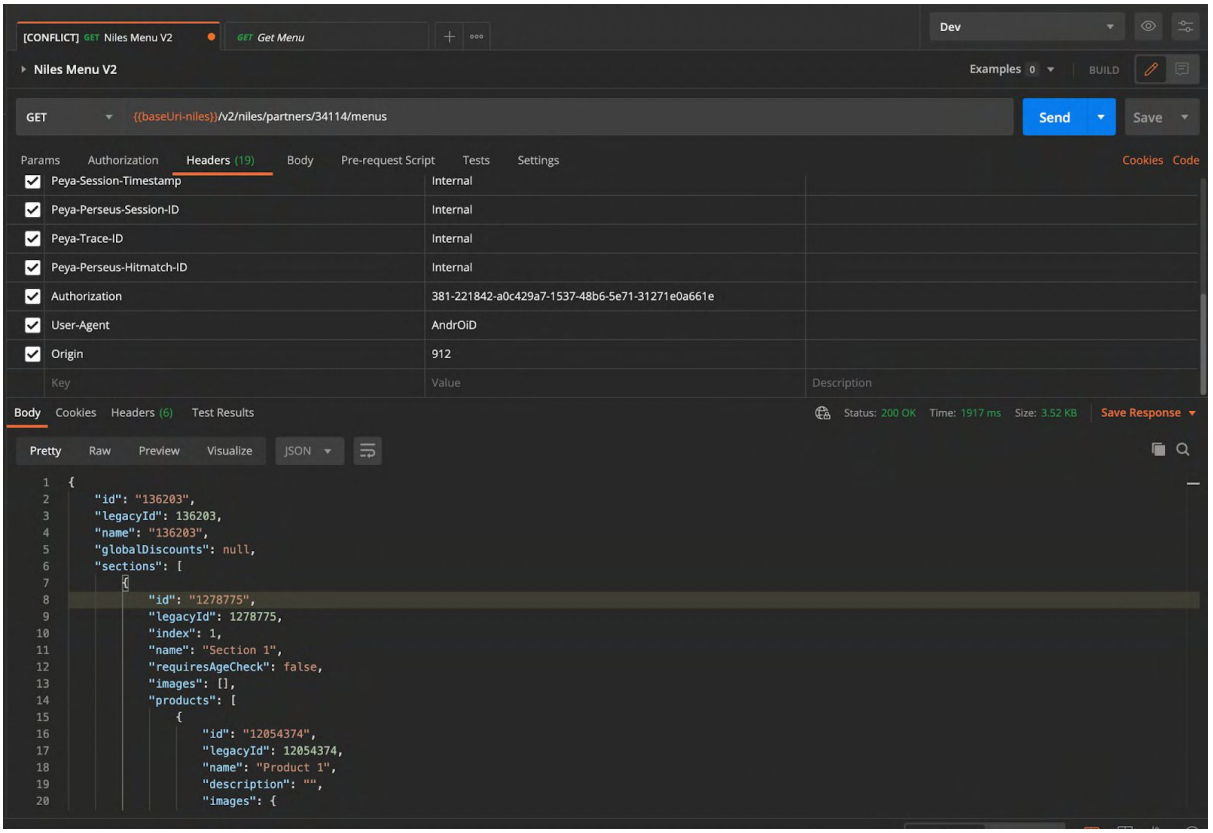

Figura 4.21 - Petición exitosa sobre el recurso de menú

A partir de la última figura, resulta claro que el patrón *Bulkhead* permite aislar los fallos, de manera de poder seguir ofreciendo la funcionalidad deseada. Para esto es necesario entender cuáles son los objetivos prioritarios en un microservicio, además de contar con las herramientas necesarias para poder tener múltiples instancias en diferentes nombres de dominio.

4.6. Trabajo Experimental y Resultados Obtenidos para combinación de patrones

4.6.1. Trabajo Experimental

Para realizar el experimento, se utilizó la misma estrategia utilizada en los resultados experimentales del patrón *Retry*, donde se modificó *Favourites-service*, para que uno de cada diez requerimientos falle al azar.

El proceso experimental sigue los pasos detallados en la sección 4.1.Diseño Experimental, el cual se realiza sobre la versión de Niles que no contiene ningún patrón aplicado y sobre la versión contiene la combinación de los patrones mencionados en la sección 3.3.6.1 La combinación Timeout, Retry y Circuit Breaker.

4.6.2. Resultados en ausencia de combinación de patrones

El primer punto crítico a analizar es que al no implementar el patrón *Timeout*, no existe una finalización de la comunicación desde Niles hacia ningún servicio. Esta situación conduce a

que, ante cualquier petición de menor importancia como puede ser la que se realiza hacia *Favourites-service*, se penalice al tiempo total de respuesta (ver Figura 4.22).

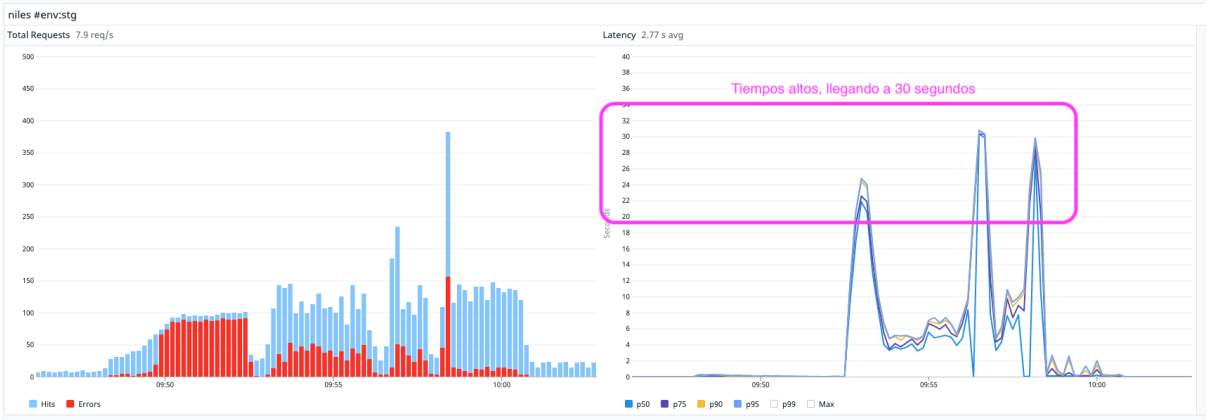

Figura 4.22 - Tráfico a Niles (izq) y tiempos de respuesta del menú (der)

Como siguiente punto crítico, al no contar con la implementación del patrón *Retry*, toda falla transitoria se traduce en un fallo. Ante un problema de intermitencias de red, o una respuesta de tiempos altos, la petición fallará.
Por último, se debe tener en cuenta que no existe un mecanismo que evite seguir realizando requerimientos hacia un servicio, incluso cuando éste tenga altas probabilidades de falla. Esta situación lleva a que, al degradar *Favourites-service* con una ráfaga de peticiones, Niles continúe solicitando información, incluso en etapa de incidentes. En la Figura 4.23 se observa cómo, en una etapa de constantes fallos en el servicio de favoritos (rectángulo de la derecha), el consumidor continúa realizando peticiones (rectángulo de la izquierda).

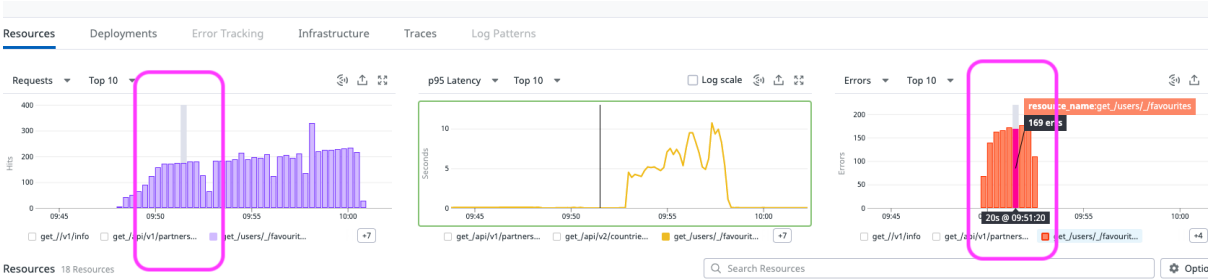

Figura 4.23 - Peticiones constantes sobre un servicio que se encuentra en etapa de fallos

4.6.3.Resultados en presencia de combinación de patrones

Al aplicar la combinación de patrones en Niles, se analizan los puntos críticos mencionados en las pruebas sin patrones. En la Figura 4.24 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 derecha de la imagen, donde el tiempo más alto registrado es de 6.5 segundos, lo que resulta mucho menor a los observados en la Figura 4.22.

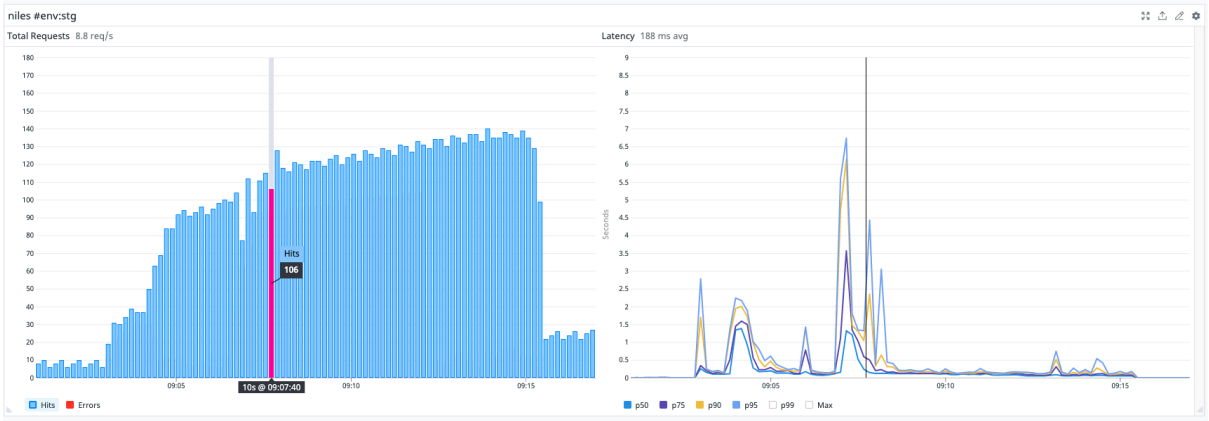

Figura 4.24 - Patrón *Timeout*: tiempo de respuesta con pico de 6.5 segundos

Ante la falla del intento anterior, se reintenta en 3 oportunidades más para lograr dar respuesta al requerimiento. Sin embargo, por la configuración empleada, todas fallarán, dando lugar a la ejecución de Circuit Breaker.

En la Figura 4.25 se puede ver cómo impacta la inclusión del 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 utilizando autocannon.

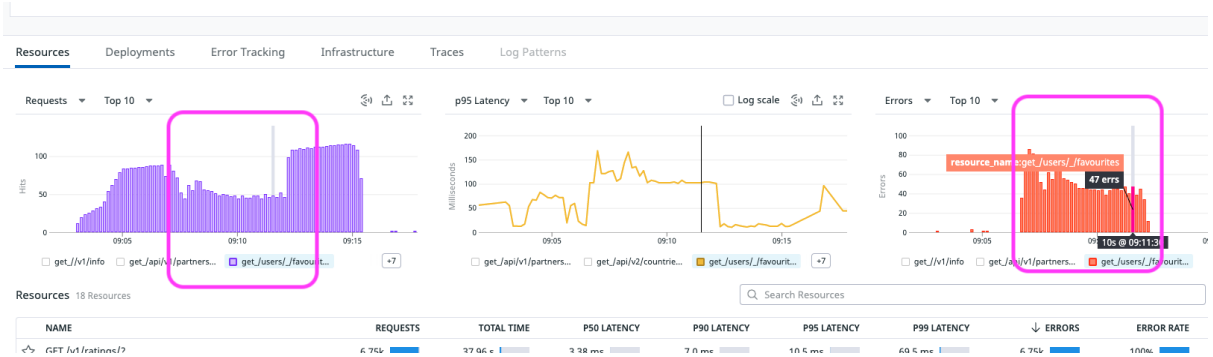

Figura 4.25 - Circuit breaker en estado *abierto,* evitando peticiones hacia *Favourites-service*

Combinando patrones de diseño *Timeout, Retry* y *Circuit Breaker* se mejora en los siguientes puntos:

- Encolamiento de hilos en largas espera debido a latencia lenta en servicios consumidos.
- Penalización en cuanto a tiempos sobre la funcionalidad principal, evitando por ejemplo largas esperas por un producto favorito, cuando lo indispensable es la información básica del menú
- Reintentos en fallas transitorias por lentitud en red, o fallos de la misma.
- Ahorro de recursos, evitando realizar peticiones hacia un servicio que se encuentra en estado de alerta y las probabilidades de que un requerimiento falle son altas.

Capítulo 5

Conclusiones e ideas para trabajos futuros

En los últimos años se ha incrementado la utilización de arquitecturas de microservicios en la industria del desarrollo de software, para dar respuesta a algunas de las limitaciones de los sistemas monolíticos. Sin embargo, estas arquitecturas también presentan una serie de desventajas que son propias de la complejidad de los sistemas distribuidos, tales como la propagación de fallos en cascada cuando un microservicio experimenta errores (si no se aplica una estrategia de contención adecuada), los inconvenientes para mantener la consistencia y las dificultades de monitorización del estado de los componentes que conforman la arquitectura.

La resiliencia es uno de los aspectos no funcionales más importantes, especialmente en grandes compañías. El manejo de las fallas se torna una cuestión fundamental, ya que su impacto tiene relación directa con el negocio de una empresa. Es por ello que entra en juego el concepto de patrones de diseño para la resiliencia entre microservicios, que se aplican en la implementación de estrategias para lidiar con estas fallas y mitigar sus efectos negativos. Por este motivo, en esta tesina se propuso como objetivo analizar el comportamiento de una serie de patrones utilizados para proveer resiliencia frente a diversos fallos capaces de afectar el funcionamiento del ecosistema de microservicios de la empresa PedidosYa.

A lo largo de este trabajo, se estudió parte de la arquitectura de microservicios de PedidosYa, comprendiendo diferentes escenarios típicos de fallos que afectan a estas arquitecturas. En particular, se analizó el microservicio Niles (que es el encargado de retornar el menú de un restaurante), detallando su operación y los servicios de los que depende para cumplir su funcionalidad. Se hizo foco en el tratamiento de errores vinculados con Niles, y se estudiaron los patrones que son frecuentemente utilizados para la resiliencia, abarcando su definición, conceptos asociados, su funcionamiento y su implementación dentro de PedidosYa. Posteriormente, se obtuvieron resultados experimentales que permitieron analizar y evaluar el impacto de la aplicación de los patrones en cuestión. En particular, se analizó el comportamiento de Niles en presencia de fallos tanto, con y sin la aplicación de los patrones estudiados, de manera de poder cuantificar su incidencia.

Luego de haber analizado los resultados en diferentes escenarios, se pudieron obtener las siguientes conclusiones del trabajo experimental:

- El uso del patrón *Timeout* permitió que el tiempo de respuesta consumido para retornar el menú se mantenga en un rango aceptable (mientras se respete una priorización adecuada), disminuyéndolo en un factor de aproximadamente 35x (de 6000 a 170 milisegundos). Además, al finalizar la comunicación en un tiempo acotado, se evita la acumulación de requerimientos pendientes, lo que se traduce en un mejor uso de los recursos.
- El uso del patrón *Retry* permitió que una mayor cantidad de las peticiones entrantes puedan ser resueltas exitosamente. En este caso, la cantidad de requerimientos con errores se redujo aproximadamente 15x (822 a 54). La aplicación de *Retry* muestra que, en algunos contextos, un simple reintento puede completar exitosamente una funcionalidad que impacta en las ganancias de la compañía (por ejemplo, la facturación de una orden).
- El uso del patrón *Circuit Breaker* facilitó la provisión de un menú disminuido (sin funcionalidad de favoritos) ante las elevadas probabilidades de falla de determinados recursos involucrados. Esto condujo a un ahorro de recursos, tales como uso de la red y elementos de procesamiento, evitando realizar peticiones hacia recursos temporalmente averiados. Incluso, en este escenario también se redujo el tiempo de respuesta del menú en un factor de 10x (de 5000 a 500 milisegundos), lo cual representa un valor aceptable.
- El uso del patrón *Bulkhead* permitió el aislamiento de los fallos que ocurren en el ámbito de una funcionalidad no se propaguen, afectando al funcionamiento de otros servicios. De esta manera, se da respuesta a la priorización de funcionalidades de un sistema.
- Es posible diseñar una combinación de patrones, para dar respuesta frente a escenarios complejos. Si bien esto aumenta la complejidad de la implementación, el beneficio directo es el aumento en la robustez del sistema, ya que se contempla la prevención y mitigación de una mayor cantidad de casos de fallas.

En forma más general y, a partir de las lecciones aprendidas durante el desarrollo de la tesina, se puede concluir que:

- La implementación de patrones de resiliencia en arquitecturas orientadas a microservicios resulta un pilar fundamental para mejorar el rendimiento y proporcionar robustez a la arquitectura.
- Cada una de las interacciones entre los componentes que conforman el ecosistema de microservicios debe ser analizada de manera particular. En ese sentido, resulta vital poder contar con información y métricas de cada componente con el que se realizará una etapa de integración, ya que esto permite establecer diferentes configuraciones en los patrones de resiliencia a implementar.
- Las librerías que proveen implementaciones de patrones de resiliencia son una excelente herramienta para establecer buenas prácticas para el desarrollo de microservicios sólidos y escalables.

Habiendo analizado el comportamiento de Niles y sus microservicios asociados ante un conjunto de escenarios típicos de fallas, tanto en ausencia como en presencia de distintos patrones de resiliencia, se considera que se ha cumplido con el objetivo planteado originalmente en esta tesina. La implementación de cada uno de estos patrones de resiliencia en el microservicio Niles se encuentran en producción en el ecosistema de

PedidosYa, logrando ser uno de los componentes más utilizados y robustos dentro de la compañía.

Como trabajos futuros, se pueden mencionar las siguientes ideas de investigación y desarrollo:

- Replicar el estudio realizado, considerando otros patrones habituales en la provisión de resiliencia en arquitecturas basadas en microservicios, tales como *Throttling*[36] y *Queue-Based Load Leveling*[37].
- Implementar patrones como *Throttling* o *Bulkhead* dentro de la librería *peya-ktor-utils*.
- Impulsar la implementación de los patrones estudiados en toda la compañía, estableciendo estructuras de proyectos independientemente de los lenguajes de programación utilizados, para hacer obligatorio el uso de los patrones desde la etapa de diseño, permitiendo realizar configuraciones específicas para cada caso de uso.
- Impulsar la práctica de *Chaos Engineering*[38] dentro de la compañía.

[36] https://learn.microsoft.com/en-us/azure/architecture/patterns/throttling
[37] https://learn.microsoft.com/en-us/azure/architecture/patterns/queue-based-load-leveling
[38] https://principlesofchaos.org/

Referencias

[1] - *2015 10th Computing Colombian Conference, September 21-25, 2015*. New Jersey: IEEE, 2015.

[2] Richardson, Chris. «Escaping monolithic hell». En *Microservices patterns: with examples in Java*. Shelter Island, New York: Manning Publications, 2019.

[3] martinfowler.com. «Microservices». https://martinfowler.com/articles/microservices.html.

[4] ION. «How Do You Manage Non-Functional Requirements?» Accedido 11 de octubre de 2022. https://iongroup.com/blog/markets/how-do-you-manage-non-functional-requirements/.

[5] «Estas son las pérdidas tras la caída de Facebook, Instagram y WhatsApp | Noticias | Agencia Peruana de Noticias Andina». https://andina.pe/agencia/noticia-estas-son-las-perdidas-tras-caida-facebook-instagram-y-whatsapp-864197.aspx.

[6] Amazon Web Services, Inc. «¿Qué son los microservicios? | AWS». https://aws.amazon.com/es/microservices/.

[7] martinfowler.com. «Microservices». https://martinfowler.com/articles/microservices.html.

[8] IONOS Digital Guide. «Microservice Architectures: More than the Sum of Their Parts?». https://www.ionos.com/digitalguide/websites/web-development/microservice-architecture/.

[9] dzone.com. «What Are Microservices, Actually? - DZone Microservices». https://dzone.com/articles/what-are-microservices-actually.

[10] Verdier, Diego, y Gonzalo Rodríguez. «Implementación de Patrones de Microservicios». Universidad de la República, 2020.

[11] Richardson, Chris. «Escaping monolithic hell: Benefits of the microservice architecture». En *Microservices patterns: with examples in Java*. Shelter Island, New York: Manning Publications, 2019.

[12] Richardson, Chris. «Escaping monolithic hell: Drawbacks of the microservice architecture». En *Microservices patterns: with examples in Java*. Shelter Island, New York: Manning Publications, 2019.

[13] «¿Qué es la resiliencia?» *SACAViX Tech Blog* (blog), 16 de agosto de 2021. https://sacavix.com/2021/08/16/que-es-la-resiliencia/.

[14] «Fallacies of distributed systems». https://blogs.oracle.com/developers/post/fallacies-of-distributed-systems.

[15] Raj, Pethuru, Anupama Raman, y Harihara Subramanian. *Architectural Patterns: Uncover Essential Patterns in the Most Indispensable Realm of Enterprise Architecture*. Birmingham Mumbai: Packt, 2017.

[16] Balmaceda, Tomas. «PedidosYa: el secreto de esta empresa uruguaya que vale u$s 2000 M y los argentinos aman». https://www.cronista.com/infotechnology/it-business/pedidosya-el-secreto-de-esta-empresa-uruguaya-que-vale-us-2000-m-y-los-argentinos-aman/.

[17] Hofmann, Michael, Erin Schnabel, Katherine Stanley, International Business Machines Corporation, y International Technical Support Organization. «From Development to Production: Deployment Patterns». En *Microservices Best Practices for Java*, 2016.

[18] Nygard, Michael T. «Stability Patterns: Use Timeouts». En *Release it! design and deploy production-ready software*, Second edition. The pragmatic programmers. Raleigh, North Carolina: Pragmatic Bookshelf, 2018.

[19] Hofmann, Michael, Erin Schnabel, Katherine Stanley, International Business Machines Corporation, y International Technical Support Organization. «Microservice Communication: Fault Tolerance». En *Microservices Best Practices for Java*, 2016.

[20] Turnbull, James. *The Art of Monitoring*, 2016.

[21] «Implement Retries with Exponential Backoff». https://docs.microsoft.com/en-us/dotnet/architecture/microservices/implement-resilient-applications/implement-retries-exponential-backoff.

[22] «Fusibles | ¿Qué Son, Para Qué Sirven, Cómo Funcionan, Tipos?», 6 de agosto de 2021. https://sdindustrial.com.mx/blog/fusibles/.

[23] Nygard, Michael T. «Stability Patterns: Circuit Breaker». En *Release it! design and deploy production-ready software*, Second edition. The pragmatic programmers. Raleigh, North Carolina: Pragmatic Bookshelf, 2018.

[24] Hofmann, Michael, Erin Schnabel, Katherine Stanley, International Business Machines Corporation, y International Technical Support Organization. «Microservice Communication: Fault Tolerance, Circuit Breakers». En *Microservices Best Practices for Java*, 2016.

[25] microservices.io. «Microservices Pattern: Circuit Breaker». http://microservices.io/patterns/reliability/circuit-breaker.html.

[26] Nygard, Michael T. «Stability Patterns: Bulkheads». En *Release it! design and deploy production-ready software*, Second edition. The pragmatic programmers. Raleigh, North Carolina: Pragmatic Bookshelf, 2018.