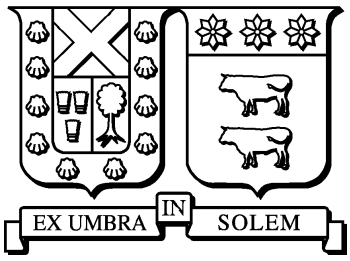
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA

DEPARTAMENTO DE INFORMÁTICA

SANTIAGO – CHILE



"MIGRACIÓN DE UN SISTEMA MONOLÍTICO A UNA ARQUITECTURA DE MICROSERVICIOS"

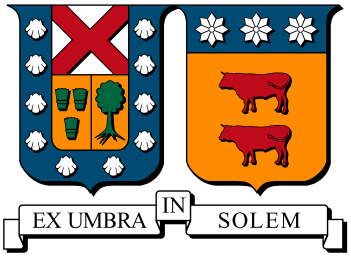
RAFIK MAS'AD NASRA

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL INFORMÁTICO

PROFESOR GUÍA: HERNÁN ASTUDILLO

JULIO 2016

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
SANTIAGO – CHILE



"MIGRACIÓN DE UN SISTEMA MONOLÍTICO A UNA ARQUITECTURA DE MICROSERVICIOS"

RAFIK MAS'AD NASRA

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE INGENIERO CIVIL INFORMÁTICO

PROFESOR GUÍA: HERNÁN ASTUDILLO

PROFESOR CORREFERENTE: RAUL MONGE

JULIO 2016

Agradecimientos

Quiero agradecer a todos aquellos que fueron parte de este proceso de seis años que culmina con la defensa de esta memoria: mi familia, profesores, ayudantes, amigos y compañeros de organización estudiantil. Gracias a todos ustedes me formé como el profesional que soy.

Resumen

El sistema a tratar en el presente trabajo, es una plataforma web para análisis de noticias mediante minería de texto. Este sistema presentaba falencias en mantenibilidad y resiliencia a fallos, por lo que se desea resolver esto sin realizar un impacto fuerte en el rendimiento del sistema. Se propone y se implementa una migración a una arquitectura de microservicios, donde se divide el sistema en siete servicios, se utiliza un único *back-end for front-end* acoplado a la interfaz y los servicios se descubren mediante el patrón de descubrimiento de servicios por el lado del servidor. Finalmente se realizan pruebas de rendimiento para medir el impacto de dicha migración dando que, además de solucionar los problemas descritos, mejoró los tiempos de respuesta en alguna de las secciones y por sobre todo, mejoró la escalabilidad de la aplicación.

Abstract

The system to be treated in this work is a web system of text mining for news analysis. This system was presenting shortcomings in maintainability and resilience problems so it need to be resolved this without a strong impact on the system performance; because this, is proposed and implemented a migration to a microservices architecture where the system is divided into seven services, a single back-end for front-end coupled to the interface and services are discovered by the server-side discovery pattern. Finally test are performed to measure the impact of the migration in question. Additionally to solve the problems described, the migration improved response times in some of the sections and above all, improved the scalability of the application.

Índice de Contenidos

Índice de cuadros

Índice de figuras

Introducción

JONC es un sistema que realiza minería de texto y diversos análisis de noticias que obtiene del sitio del diario de La Nación. Una de sus principales características es que siendo un sistema monolítico, tiene tres secciones que requieren alto cómputo.

Producto de las funcionalidades que implementa y del crecimiento en características que ha tenido, el sistema se estaba volviendo más difícil de mantener y las fallas que se producían en funcionalidades no esenciales estaban repercutiendo en el sistema en su conjunto.

En este trabajo se realizó una migración arquitectónica para solventar los problemas antes mencionados sin generar un impacto negativo considerable en rendimiento. Para cuantificar el impacto, finalizada la migración, se realizaron pruebas de rendimiento.

La estructura de la presente memoria es la siguiente:

- El capítulo 1 presenta el problema del sistema que se le realizara el cambio de arquitectura, las razones para hacerlo y los objetivos que se quieren cumplir con ello.
- El capítulo 2 realiza una descripción del sistema inicial y sus secciones.
- El capítulo 3 muestra el desarrollo que ha tenido en los últimos las diversas alternativas para solventar el problema descrito.
- El capítulo 4 presenta la propuesta de arquitectura a realizarse en el sistema
- El capítulo 5 detalla la forma en que se llevó a cabo la implementación de la arquitectura y su resultado.

- El capítulo 6 presenta las diferencias en rendimiento y escalabilidad del cambio arquitectónico.
- Finalmente, en la conclusión, se discutirá el cumplimiento de los objetivos de la presente memoria.

Capítulo 1

Definición del Problema

El sistema a migrar es una aplicación web que consta de tres secciones: noticias, tópicos y análisis de lenguaje. Las noticias que se utilizan de insumo para el análisis se obtienen de revisar diariamente las paginas del diario La Nación. A continuación se describirán los problemas abordados y el objetivo de la presente memoria.

1.1. Descripción

Producto de que, en las tres secciones mencionadas, se utilizan algoritmos de análisis inteligente de datos, que se caracterizan por su alto cómputo y consumo de memoria, el sistema se a vuelto paulatinamente más difícil de mantener y más propenso ha que producto de alguna falla puntual en alguna sección afecte a todo el sistema.

Para sistemas que tienen proyección en el tiempo es fundamental que los cambios de requerimientos se puedan implementar en el menor tiempo posible. Actualmente, ya que todo corre en el mismo sistema, cualquier característica nueva hay que revisar que no perjudique el resto del sistema. Además, para probar un cambio se debe correr todo el sistema, esperando que cargue todos los módulos y algoritmos, lo que es tedioso y lento.

Además, en el presente sistema existen funcionalidades no esenciales, pero que al fallar, lo

hace todo el sistema, o los tiempos de respuesta son bastante mayores a los que soportara un usuario. A modo de ejemplo, en la sección de revisión de una noticia del sitio, realizando una simulación de 500 usuarios entrando en 5 segundos, el sistema falla en un 71 % de las consultas; principalmente, de la búsqueda de noticias similares, que es una funcionalidad complementaria de la sección. Es deseable reducir estos porcentajes de falla y disminuir los tiempos de respuesta, aunque para ello se deban sacrificar, en algunas consultas, mostrar características secundarias del sistema.

1.2. Objetivos

1.2.1. Objetivo principal

El objetivo principal de la presente memoria es realizar una migración arquitectónica que solucione los problemas de mantenibilidad y resiliencia a fallos, sin generar un impacto negativo importante en rendimiento del sistema.

1.2.2. Objetivos secundarios

El objetivo secundario es cuantificar el impacto de la migración en términos de escalabilidad. En miras a que el sistema soporte un flujo mayor de usuarios, es de vital importancia que pueda, con la menor cantidad de maquinas, recibir la mayor cantidad de usuarios.

Capítulo 2

Descripción del sistema inicial

El sistema a migrar está programado en Python 2.7[1] utilizando el micro framework Flask [2]. Para almacenar los datos se utiliza la base de datos no relacional MongoDB en su versión 3.2. [3].

Como MongoDB es una base de datos documental no tiene modelo de datos definido. Existen tipos de documentos, que en el caso del presente sistema son noticias (News) y tópicos (Topics).

Los documentos de noticias (News) son los encargados de almacenar todas las noticias que se descargan del sitio de La Nación.

Mientras los documentos de tópicos (Topics) son los encargados de almacenar todas los tópicos que generan el algoritmo NMF que se implementó.

[1] Python 2. `https://docs.python.org/2/` (Consultado el 22 de Junio de 2016)
[2] Flask (A Python Microframework). `http://flask.pocoo.org/` (Consultado el 22 de Junio de 2016)
[3] MongoDB 3.2 `https://www.mongodb.com/mongodb-3.2` (Consultado el 22 de Junio de 2016)

2.1. Sección de noticias

En la sección de noticias se pueden visualizar todas las noticias, realizar una búsqueda de ellas y revisar cualquier noticia en particular. En el caso de la revisión de una noticia se muestra, además del detalle de la noticia, un porcentaje de objetividad y muestra las dos noticias más parecidas a la consultada.

Para la búsqueda se usan los índices de texto de MongoDB y una consulta directa. Para revisar la objetividad de la noticia, se hace uso de la librería TextBlob tanto para traducir al inglés como revisar objetividad. Mientras tanto para encontrar las noticias similares se realiza una búsqueda a los vectores TF-IDF [60] mediante diferencia de coseno.

En el caso de búsqueda de noticias similares, para evitar buscar entre todos los documentos, previamente se acota el espacio de búsqueda mediante un algoritmo propio (denominado *Topic Hashing*) que genera un *hash* por cada documento y luego busca solo en los documentos que tienen una *hash* compatible.

Se entrena el algoritmo antes descrito una vez al día.

2.2. Sección de tópicos

En la sección de tópicos se muestran los tópicos subyacentes en las noticias mediante el algoritmo de *topic modeling*, *Non-negative Matrix Factorization* (NMF) [34]. Resultado de esto, por cada tópico, se muestra una nube de palabras con los conceptos que componen el tópico y las noticias que le pertenecen.

Se generan los tópicos una vez al día.

2.3. Sección de análisis de lenguaje

Usando el algoritmo desarrollado por Google, Word2vec [28], basado en una red neuronal de dos capas, se puede realizar diversos análisis de lenguaje, en el contexto de los documentos que almacena el sistema. Estos análisis pueden ser desde calcular la similaridad entre dos palabras, cuál palabra de un conjunto no coincide con el resto y cómo la suma y resta de conceptos puede derivar en otro.

Se entrena la red una vez al día.

Capítulo 3

Estado del Arte

Existen múltiples casos en la industria de sistemas que, luego de su éxito inicial, necesitaron cambios profundos en miras a estar a la altura de los desafíos que se le fueron imponiendo. De esto son ejemplos emblemáticos Twitter con su rediseño, en primera instancia, y su posterior migración de Ruby on Rails a Scala [32]; o SoundCloud y su migración de un gran bloque monolítico a una arquitectura de microservicios [18].

Para realizar dichos cambios existen múltiples soluciones, algunas más estructurales como migraciones arquitectónicas y otras más superficiales como realizar una refactorización de algunas partes del sistema. En el caso del presente sistema, además de la refactorización, se esgrimen, como solución al problema planteado en el capitulo 1, principalmente migrar a microservicios o microkernel.

3.1. Refactorización mediante módulos

Algunos lenguajes integran la capacidad de descomponer de un sistema en módulos independientes que se levantan en procesos separados, incluso incluyendo su propio sistema de para manejar la compatibilidad con distintas versiones de los distintos módulos. Tanto Java en su versión 9 [29] como Erlang [13], tienen descomposición de módulos como parte de sus intérpretes.

Para el sistema descrito se esgrime como una alternativa reescribirlo en un lenguaje que lo soporte y realizar uso de dicha técnica.

Es justamente Erlang el lenguaje que más realce ha tenido en el último tiempo. Es, a modo de ejemplo, el lenguaje que esta detrás del éxito de WhatsApp [44] donde mediante una arquitectura monolítica y con tan solo 32 ingenieros lograron escalar a 450 millones de usuarios. Otras compañías también han optado por este lenguaje para algunos de sus componentes como es el caso de Facebook y su sistema de mensajería [44]. También los gigantes de los videojuegos como Activision para los servidores de los Call of Duty [22] y Riot para el sistema de mensajes de League Of Legends [31].

Durante los últimos años, en miras a agilizar el desarrollo en Erlang, nació Elixir [45], un lenguaje que corre directamente en la máquina virtual de Erlang (Erlang VM) que incluye varios cambios como incluir metaprogramación o los *pipelines*. Sobre este lenguaje es que se construye Phoenix [45], un framework web que incluye la modularización de Erlang y Elixir, concurrencia, un alto rendimiento (sus tiempos de respuesta se miden en microsegundos) y vistas que muestran datos en tiempo real por defecto.

3.2. Microkernel

Ante la problemática de requerir cambios constantes de requerimientos en un sistema nace como solución el patrón de arquitectura microkernel [54]. Este consta, como se ve en la figura 3.1, de un pequeño núcleo que implementa la base del sistema y luego extensiones que implementan las diversas características del sistema.

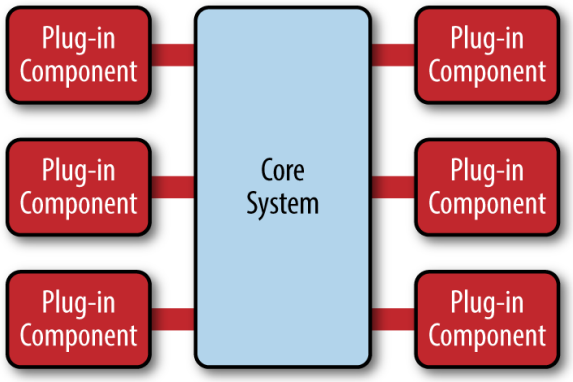


Figura 3.1: Diagrama de la arquitectura de microkernel [50].

Para integrar las distintas extensiones con el núcleo se puede realizar mediante vínculos a librerías remotas o comunicando procesos mediante RPC o mediante una API TCP/IP. En el caso que una de las extensiones sea un servicio externo se realiza una extensión que adapte el servicio a la interfaz del que ocupa el núcleo.

Existen múltiples aplicaciones de este patróná principalmente en la construcción de sistemas operativos. El kernel experimental de Microsoft, Singularity [38], el sistema operativo de celulares Symbian OS y el kernel que esta desarrollando la fundación GNU, Hurd[1], son ejemplos de su implementación. También se puede considerar como implementación de este patrón software como el editor de texto Eclipse[2] o Sublime Text[3] donde, en estos casos, existe un editor simple que se complementa con extensiones para mejorar sus funcionalidades.

3.3. Microservicios

Ante la necesidad de mantener sistemas cada vez más grandes nace como patrón de arquitectura los microservicios. “Los microservicios son servicios pequeños y autónomos que trabajan en conjunto” [42].

[1]GNU Hurd. `https://www.gnu.org/software/hurd/hurd.html` (Consultado el 22 de Junio de 2016)
[2]Eclipse. `https://eclipse.org/` (Consultado el 22 de Junio de 2016)
[3]Eclipse. `https://www.sublimetext.com/` (Consultado el 22 de Junio de 2016)

Por pequeños existen múltiples definiciones, pero probablemente la más adecuada es la del principio de responsabilidad limitada de Robert C. Martin: “Mantener juntas las cosas que cambian por la misma razón, y separá las que cambian por diferentes razones” [36].

Por autónomos fundamentalmente deben respetar la regla de oro [42] dada por Sam Newman: que se pueda editar y poner en producción un servicio sin tener que modificar ningún otro.

Antecedentes de desacoplar un sistema en pequeños servicios autónomos los podemos encontrar desde los noventa, pero no es hasta el 2011, en un workshop cerca de Venecia[25], en donde se empieza a generalizar esta noción, convirtiéndola en patrón. Es en el 2012 cuando el mismo grupo le pone definitivamente a este patrón el nombre de microservicios.

Desde el 2012 a la fecha el crecimiento del desarrollo en el área ha sido exponencial. Esto debido principalmente a que empresas del tamaño de Amazon [51], The Guardian, HP, Microsoft Azure, Netflix [20], Nirmata, eBay [55], Riot Games y SoundCloud [18] han migrado sus sistemas a esta arquitectura.

Pero no tan sólo que importantes empresas del rubro hayan adoptado este patrón de arquitectura ha contribuido a este crecimiento exponencial. Tecnologías como frameworks que fácilmente generan una API RESTful (por ejemplo LoopBack), sistemas de balanceo y descubrimiento de servicios (como Eureka) y los contenedores (Docker) tienen crédito en este crecimiento.

Al ser este tema nuevo y de rápido desarrollo en la industria, la investigación académica es incipiente.

3.3.1. Métodos de integración

Existen múltiples métodos para integrar servicios entre sí, desde RPC, pasando por Protocol Buffers, hasta el uso de una API RESTful. Siendo esta última, generalmente mediante HTTP y enviando documentos JSON, la que ha tomado más preponderancia en el último tiempo.

En esa línea, ha sido de vital importancia el desarrollo de frameworks especializados en

la creación de APIs RESTful como lo es LoopBack[6] o que populares frameworks como Ruby on Rails (en su versión 5)[8], Django[3] o Flask[4] tengan extensiones para adaptar su funcionamiento para entregar este tipo de interfaz a otras aplicaciones.

Además, no tan sólo frameworks han dado un aporte en esta materia. Sistemas que sin mayor desarrollo o configuración generan APIs listas para usar como es el caso de la base de datos CouchDB[2] de Apache.

3.3.2. Patrones de arquitectura asociados

Al momento de desarrollar un sistema con microservicios surgen múltiples problemas recurrentes en todos los desarrollos con microservicios. Es por ello que existen múltiples patrones de arquitectura asociados, que nos permiten construir e integrar nuestros microservicios con las estrategias que más se adecuen al problema y la necesidad organizacional.

Coordinación de los servicios

Cuando una acción implica que se deben generar consultas o acciones a múltiples servicios, existen dos estrategias posibles: orquestación y coreografía.

La orquestación de servicios implica que un servicio se encarga de llamar a los servicios asociados con a la acción ejecutada por el usuario. Dicho servicio se comunica por alguno de los métodos de integración antes descrito.

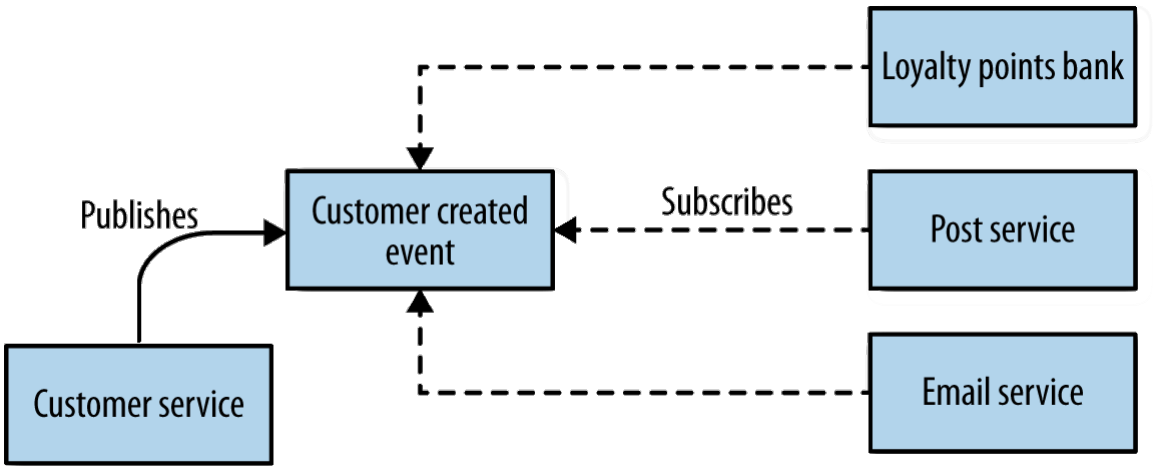


Figura 3.2: Ejemplo en un sistema de manejo de clientes mediante orquestación [42].

Por otro lado, mediante una estrategia de coreografía, en vez de dejar encargado un servicio de la comunicación con sus pares cada servicio, mediante por ejemplo un patrón de publish-subscribe, interactuando con los servicios que le interesa obtener la información. Así, cuando se genera un evento relevante, se publica un evento y los servicios que están suscritos al evento se dan por notificado y realizan las acciones pertinentes.

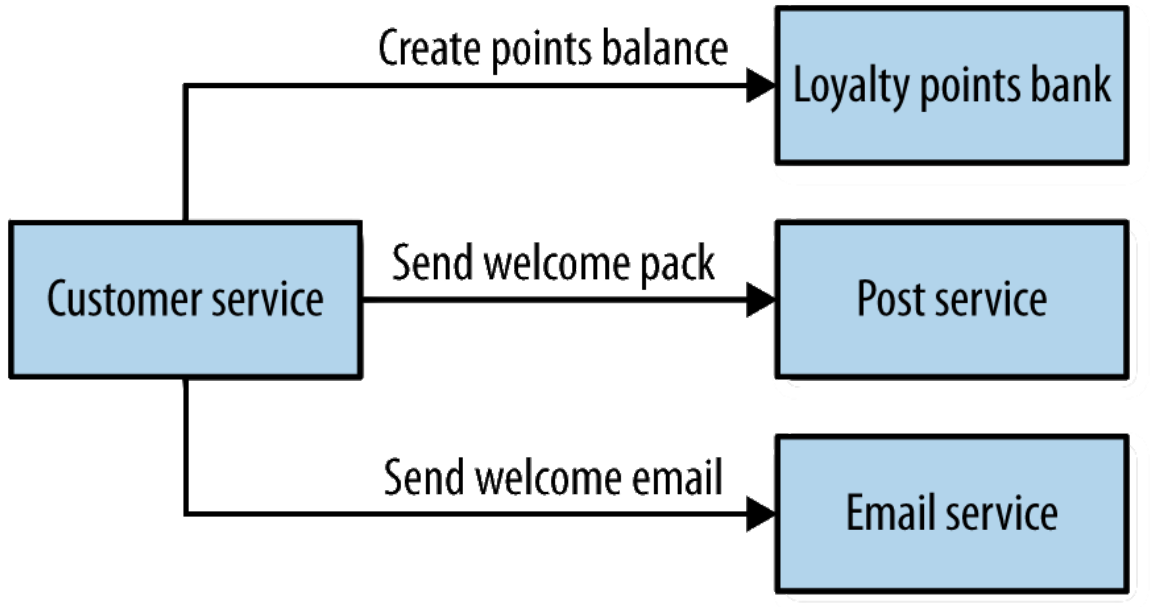


Figura 3.3: Ejemplo en un sistema de manejo de clientes mediante coreografía [42].

Descubrimiento de servicios

Cuando la interfaz realiza una consulta a la API del sistema que está compuesto por múltiples servicios, sobre todo si estos están replicados, es necesario tener un registro de todos los servicios, su estado y forma de acceder a éste.

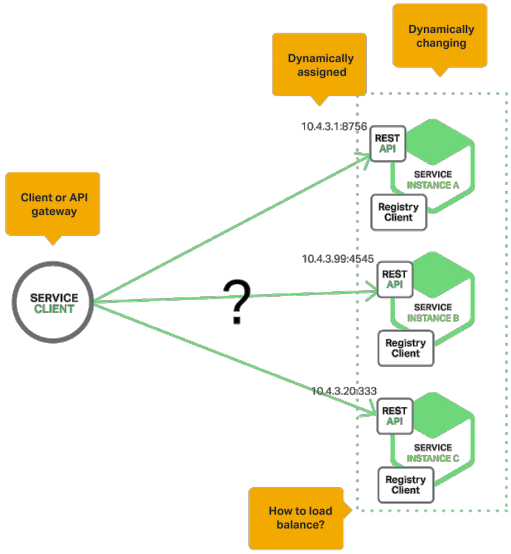


Figura 3.4: Diagrama que muestra el problema asociado a descubrir los servicios de un sistema con microservicios [52].

Existen dos patrones que vienen a solucionar este problema: descubrimiento por el lado del cliente y descubrimiento por el lado del servidor.

En ambos casos es fundamental mantener un registro de los distintos servicios y sus réplicas, para ello existen múltiples alternativas como Consul [30], Eureka [1], ZooKeeper [15], doozer [39] y etcd [21].

Que los servicios se descubran por el lado del cliente implica, que al hacer una petición un cliente, el sistema de registro de servicios le devuelve la dirección y el cliente directamente se conecta al servicio asociado. A modo de ejemplo, Netflix utiliza este patrón de arquitectura para descubrir sus servicios integrando como registro de servicios a Eureka [1] y Ribbon [12], un cliente IPC que llama a Eureka para saber dónde se encuentra el servicio que busca

antes de hacer la conexión.

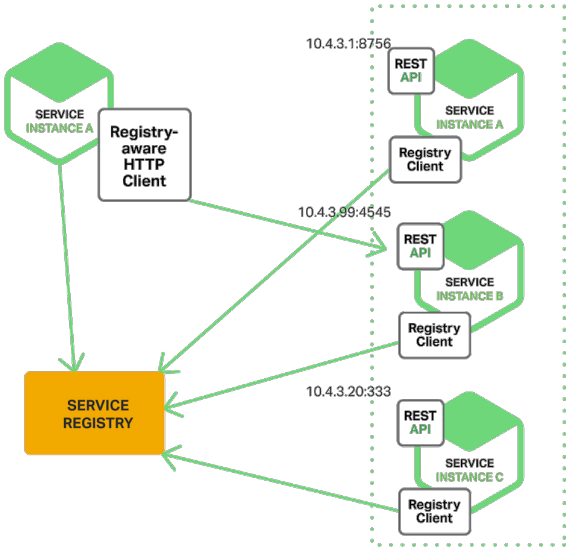


Figura 3.5: Diagrama que muestra el patrón de descubrimiento por el lado del cliente [52].

Mientras tanto que los servicios los descubran por el lado del servidor implica en la práctica que se introduzca un balanceador de carga que consulte al registro de servicios y redireccione con las tareas típicas de un balanceador de carga. Ejemplo de esto es el balanceador de carga de Amazon (AWS Elastic Load Balancer [10]), que permite implementar este patrón mientras a su vez crea de forma elástica nuevas instancias de los servicios, dependiendo de su demanda.

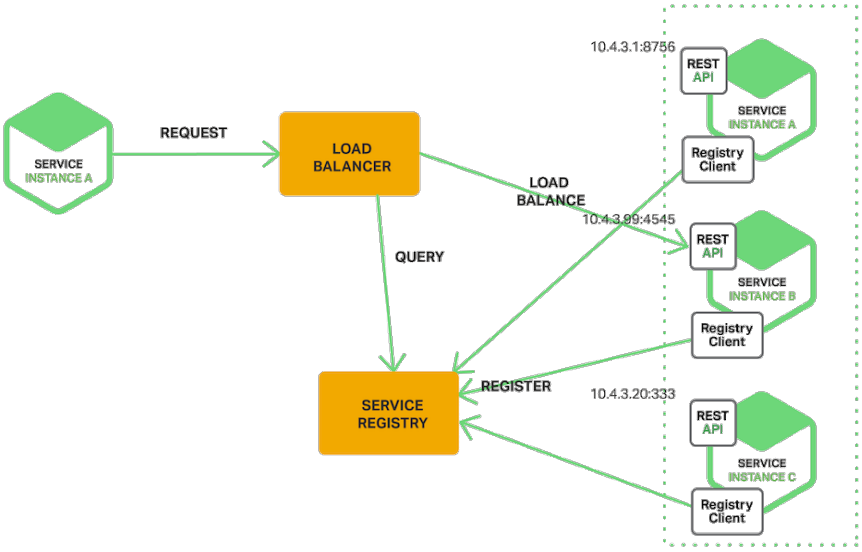


Figura 3.6: Diagrama que muestra el patrón de descubrimiento por el lado del servidor [52].

Interfaces de usuario

Otro problema recurrente a la hora de implementar un sistema mediante microservicios es la interfaz de usuario. Mientras nuestro sistema está separado en múltiples servicios para los usuarios del sistema no puede notar esa diferencia. Además pueden existir diversas interfaces para el mismo sistema (como un sitio web y su aplicaciones móviles), que pueden tener requerimientos distintos a la hora de acceder al sistema.

Como solución recurrente se ha utilizado *Backends For Frontends* [43] (en adelante BFF) para proveer a las interfaces de usuario de las funcionalidades que se requiere.

En el caso más simple, un BFF es un sistema que provee una API (mediante alguno de los protocolos antes descritos) a las interfaces del sistema. Este sistema se comunica con los servicios que requiera para proveer la API antes descrita y enviar la información que a través de ésta reciba.

Como pueden existir múltiples interfaces de usuario con necesidades particulares, por parte de dicha API se pueden crear múltiples BFF. Es de esperarse que la cantidad de BFF se reduzca, pero sin comprometer o limitar las funcionalidades que se puedan implementar.

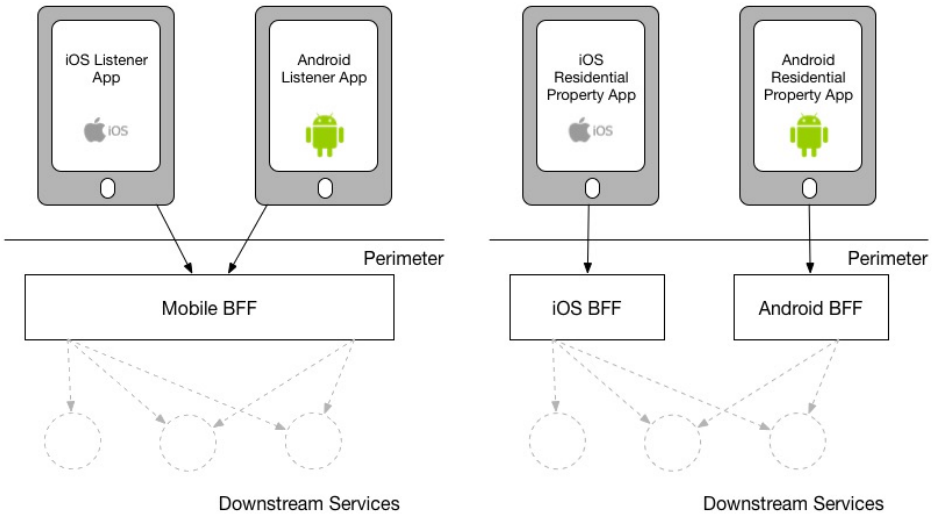


Figura 3.7: En el caso de la derecha dos BFF, uno para cada dispositivo móvil soportado por la plataforma (en este caso `http://realestate.com.au/`). A la izquierda un sólo BFF para ambos dispositivos soportados por la plataforma (en este caso SoundCloud) [43]

3.3.3. Puesta en producción

Para su rápida puesta en producción y facilidad a la hora de desarrollar sistemas distribuidos es fundamental el uso de contenedores que aislen y encapsulan los sistemas. Esto facilita el manejo automático de los sistemas y la puesta en producción continua.

En esa línea se ha investigado en el último tiempo sobre puesta en producción automática [19], manejo automático de contenedores [58], sanado automático de los contenedores en producción [48] y el uso de diversas tecnologías como Docker para llevar esto a cabo [56] [14].

Principalmente para facilitar el trabajo de los desarrolladores y administradores de sistema, se han creado tecnologías como Kubernetes [11] y Apache Mesos[9] que permiten de transparente abstraer los múltiples computadores y sus recursos donde corre, lo que permite desarrollar y poner en producción los sistemas pensando que esto se realiza en un único servidor y no en un clúster de éstos.

Los contenedores, sobre todo Docker, su principal exponente, han tenido múltiples críticas sobre todo en lo que respecta a seguridad y eficiencia [24]. Producto de esto han surgido los unikernels [53]: pequeños sistemas de propósito único que pueden correr en una máquina virtual o directamente en un computador sin un sistema operativo. Actualmente ya se logró tratar un unikernel como contenedor, integrando una imagen de MirageOS en Docker [40]. Por su parte Otto [7], el sucesor de Vagrant, de manera incipiente automáticamente crea un contenedor, detectando lo que contiene el proyecto y buscando la mejor configuración para éste.

Las herramientas antes mencionadas vienen a facilitar la puesta en producción de sistemas que contienen múltiples microservicios que a su vez cada uno de éstos está replicado en clusters en múltiples locaciones. Todas, sobre todo por la importancia que ha tomado este patrón de arquitectura, prometen soporte de primera clase para microservicios. Tanto los contenedores [56] como el manejo automático de clusters de contenedores [16], han sido adaptados e implementados pensando en este patrón de arquitectura.

En otra línea también se ha avanzado en la automatización de los controles de calidad [47] [26], la puesta en producción automática del código [5] y monitoreo de los servicios [51].

3.3.4. Implementando microservicios

Hoy en día son múltiples las experiencias a nivel mundial con microservicios, empresas de la magnitud de Amazon, Microsoft y Netflix.

Esto nos da una vasta experiencia en la implementación de sistemas con este patrón de arquitectura y los problemas que pueden surgir. Además ha dado cabida un progreso en tecnologías relacionadas de enorme impacto.

No sólo experiencias generales se han documentado en el último tiempo, además existen experiencias donde los desarrolladores detallan cómo fueron implementados distintas secciones de sistemas reales, dando así un importante conocimiento a nuevas implementaciones de microservicios. De lo anterior se puede destacar la migración realizada por la tienda de libros online Safari [17] y el sistema inglés de envío Shult[49].

Además de experiencias reales existen sistemas pequeños con fines pedagógicos para tener un primer apronte antes de implementar un sistema real. A modo de ejemplo se puede encontrar un sistema para administración de un cine realizado en Python mediante el framework Flask [35].

Una recomendación fundamental para construir un sistema mediante microservicios es siempre partir por un sistema monolítico, tanto por los costos asociados como por que es más simple separar un sistema en componentes que pensar dicho sistema desde un comienzo. Inclusive con lo antes descrito migrar a una arquitectura de microservicios puede ser un costo bastante elevado, en este contexto toma relevancia la propuesta del neomonolític [59].

El neomonolítico [59] es, de cierta forma, un caso particular de microservicios donde los servicios se conectan entre sí mediante RPC dentro de la misma red local. Entonces el conjunto de microservicios en el neomonolítico se puede ver como un único servicio no teniendo que lidiar con la la el descubrimiento de los servicios ni la coordinación de estos.

3.3.5. Automatización de la creación de microservicios

En la actualidad existen dos frameworks, relevantes, que generan automáticamente sistemas bajo la arquitectura de microservicios.

Synapse [59] es un sistema que mediante Ruby on Rails y una variedad importante de bases de datos relacionales y no relacionales generar código automático para crear múltiples microservicios que se comunican entre sí mediante un sistema de mensajes RabbitMQ.

Por otra parte, DEEP framework [33] mediante el uso de NodeJS y (actualmente), las tecnologías asociadas a la nube de Amazon, crea un sistema de microservicios abstrayendo al programador de la integración con la nube, automáticamente generando los códigos para el uso de balanceadores de carga, bases de datos replicados, almacenamiento y cómputo en la nube de AWS.

Cabe mencionar que lo antes descrito son tecnologías aún en desarrollo y altamente inmaduras, por lo que existe un amplio margen de mejora.

Capítulo 4

Propuesta de arquitectura

En el capítulo 3 se estudiaron diversas alternativas planteadas en el capítulo 1, las cuales se comentarán a continuación.

El lenguaje de programación Python no tiene módulos independientes como los tiene Erlang o Java; por lo que, para implementar esta alternativa, lo óptimo seria rescribir en el framework Phoenix. Esto supone un costo demasiado alto en horas de desarrollo.

En el caso de migrar la aplicación a una arquitectura de microkernel supone una mejora considerable en resiliencia a fallos y mantenibilidad, pero genera un único punto de fallo (el kernel) y la documentación existente del patrón está más enfocada en sistemas operativos que plataformas web.

Finalmente se decide realizar una migración a una arquitectura de microservicios, ya que, genera una mejora considerable en resiliencia a fallos y mantenibilidad, además de que hoy se cuenta con una basta documentación sobre su implementación, en sistemas web, producto de su desarrollo en los últimos años. También se espera tener mejoras considerables en escalabilidad producto de su capacidad de distribuir servicios de forma autónoma.

A continuación se realizará una explicación de cómo se llevará a cabo su implementación y los patrones de arquitectura a utilizarse.

4.1. Servicios del sistema

Como principio para la división del sistema se utilizó el primer principio de la filosofía de UNIX [37], que cada programa, en este caso cada servicio, realice solo una cosa (bien). Conforme a esto, el presente sistema, se divide en los siguientes siete servicios:

- Interfaz: mediante consulta a los otros servicios obtiene la información y la muestra a los usuarios finales.
- Noticias: encargado de descargar y entregar las noticias de La Nación al resto de los servicios.
- Planificador: encargado de enviar eventos para que los diversos servicios realicen acciones programadas.
- Análisis de sentimientos: devuelve la objetividad de un texto.
- Tópicos: se encarga de aplicar *topic models* a los documentos del servicio de noticias.
- Transformación de texto: conversión a vector TF-IDF de las noticias y búsqueda de noticias similares.
- Word2vec: uso de dicho conjunto de algoritmos para análisis de lenguaje.

4.2. Patrones a utilizar

Los servicios antes descritos se coordinarán mediante el patrón de descubrimiento de servicios por parte del servidor, como se describe en la subsección 3.3.2. Como se puede observar en la figura 4.1, esto implica que los servicios notificarán a un descubridor de servicios su dirección, funcionalidad y salud, y será éste el que se coordinará con el balanceador de carga además de manejar las rutas.

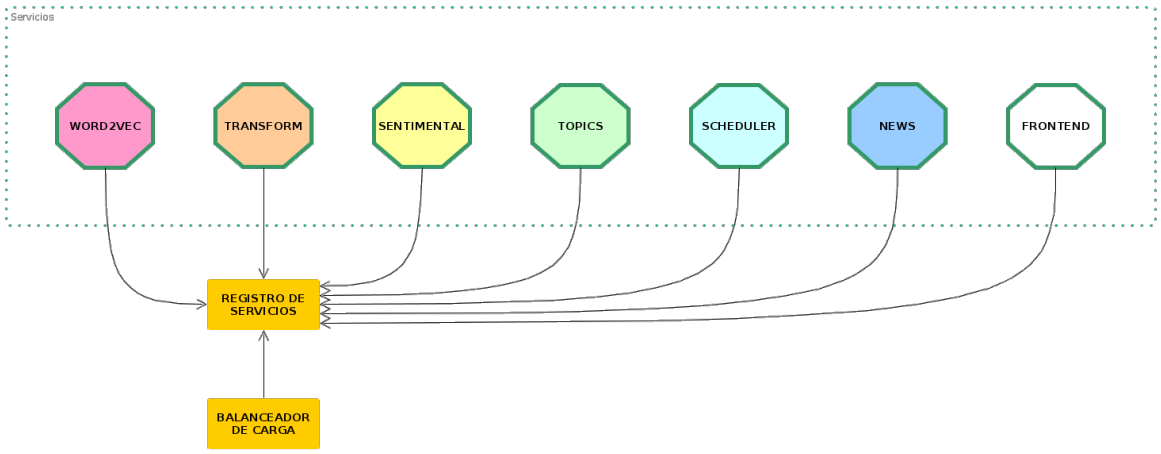


Figura 4.1: Diagrama de la arquitectura de la aplicación.

Respecto a la interfaz de usuario, se generará un único "*back-end for frontend*" (en adelante BFF) que juntará los distintos servicios en una única ruta como se describe en la subsección 3.3.2. Además sera el mismo servicio el que proveerá la interfaz. Como el sistema no proyecta tener múltiples interfaces, es más óptimo tener ambos servicios juntos.

Además los servicios se coordinarán utilizando el patrón de coreografía, como se describe en la subsección 3.3.2, para dotar de mayor independencia a estos y potenciar su autonomía. Lo anterior no quita que existirá un servicio planificador que enviará señales para que los otros servicios realicen ciertas tareas programadas a cierta hora.

4.3. Protocolos de comunicación

Para comunicarse los servicios entre sí, todos las consultas serán mediante una API HTTP y el formato de los datos enviados serán mediante el protocolo JSON.

4.4. Distribución del sistema

Para distribuir el sistema se mantiene en una única máquina el balanceador de carga, pero en todas se replica el registro de servicios conectándose entre ellos para su coordinación. Como se puede observar en la figura 4.2, en las máquinas que se distribuya el sistema se puede tener la combinación que se desee de réplicas de los servicios. Esto permite replicar los servicios que más se utilicén o que son más importantes.

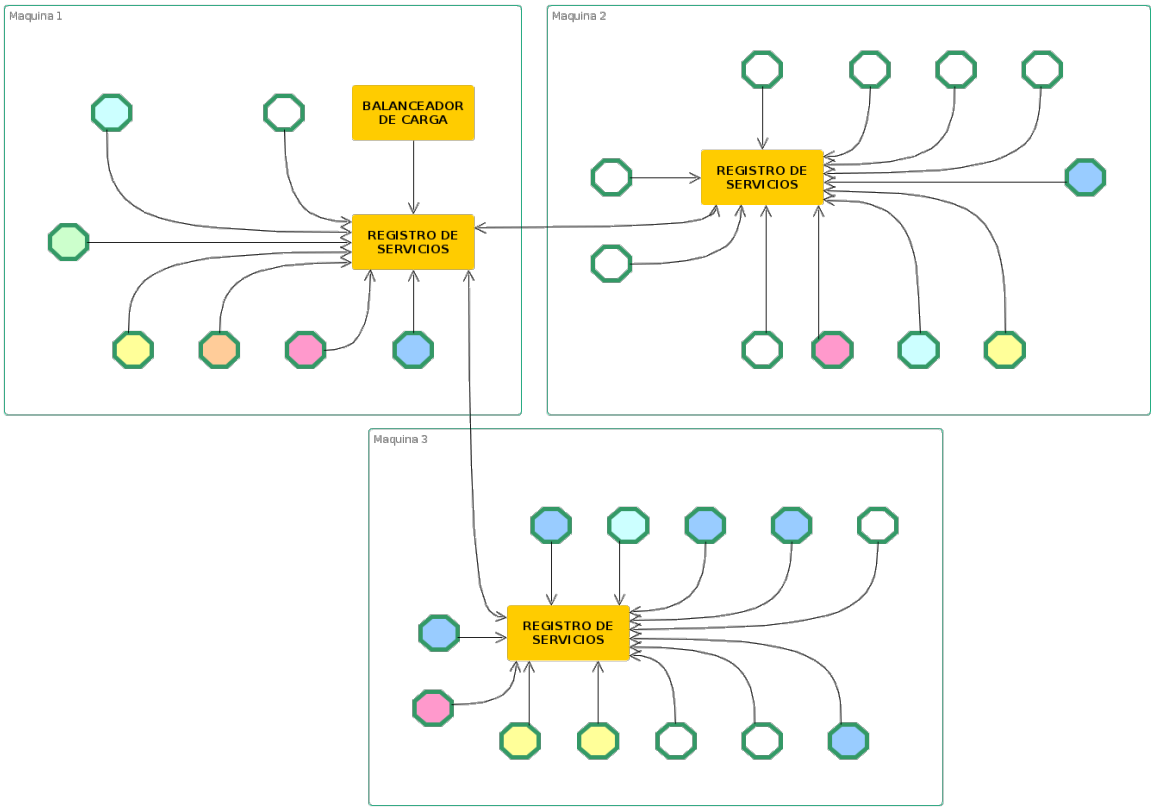


Figura 4.2: Diagrama de la arquitectura de la aplicación cuando se distribuye.

4.5. Tecnologías a utilizar

Se decide mantener Python 2.6 y Flask para reducir los tiempos de migración y poder medir solamente el cambio arquitectónico y no el rendimiento de un lenguaje o *framework*.

4.5.1. Descubridor de servicios

Para descubrir los servicios se estudián tres alternativas principalmente: Consul [30], Etcd [21] y escribir un sistema propio para realizar esta tarea. La ultima alternativa se descarta rápidamente ya que el reuso de estas soluciones permite aprovechar las funcionalidades que incluyen y ahorrar tiempo.

Se decide utilizar Consul ya que a diferencia de Etcd, que esta ligado fuertemente al sistema operativo CoreOS, es agnóstico a la tecnología y tiene un excelente sistema para revisar la salud de los servicios.

Además se revisó Eureka [1], pero éste está diseñado para funcionar con Ribbon [12] que implementa el patrón de descubrimiento por el lado del cliente y, como se explica en la subsección 4.2, se quiere descubrir los servicios por el lado del servidor.

4.5.2. Balanceador de carga

Para manejar las rutas de los distintos servicios y balancear la carga se estudian tres alternativas: HAproxy [57], Fabio [23] y consul-http-router [41]. Se descarta consul-http-router por que no es una alternativa madura para realizar la labor esperada.

Siendo HAproxy la alternativa más madura y potente, se decide utilizar Fabio ya que es más sencillo. Aunque no tiene la madurez de HAproxy, Fabio es de eBay, empresa que lo está utilizado en algunos sistemas en producción.

Capítulo 5

Implementación

5.1. Infraestructura base

Para implementar la infraestructura base se compila y ejecuta Consul en modo servidor, almacenando la información en la carpeta temporal del sistema con el comando que se observa en código 5.1.

```
consul agent -server -bootstrap -data-dir /tmp/consul -ui
```

Código 5.1: Comando para iniciar Consul.

Luego se compila Fabio, el cual queda en un único binario (ya que esta escrito en Go) y se ejecuta como se observa en el código 5.2.

```
./go/bin/fabio
```

Código 5.2: Comando para iniciar Fabio.

Para agregar un servicio se debe agregar a Consul se utiliza su sistema de etiquetas. Para esto se antepone a la ruta que responde dicho servicio el prefijo “urlprefix-/”. Fabio utiliza esta nomenclatura para reconocer los servicios y realizar el balanceo de carga.

Para realizar esto en Python basta con registrar el servicio con las etiquetas, como antes se describio, con las rutas que escucha el servicio como se puede observar en el código 5.3.

```
1  import consul
2
3  ADDR = "http://{0}:{1}/health".format(addr, PORT)
4  health_check = consul.Check.http(ADDR, "5s")
5  service_name = "{0}={1}:{2}".format(SERVICE_NAME, addr, PORT)
6
7  c = consul.Consul(host=SERVICE_REGISTER_ADDR)
8  c.agent.service.register(service_name, address=addr,
9                           port=PORT, tags=["urlprefix-/test"],
10                          check=health_check)
```

Código 5.3: Ingresar a Consul el servicio teniendo *addr* como la dirección propia del servicio y *PORT* el puerto donde corre. Fabio va a redireccionar las rutas cuyo padre sea *test* a dicho servicio (o sus replicas)

5.2. Migración de los servicios

Para la migración de las distintas partes del sistema en servicios separados fue bastante útil que el sistema original estuviera dividido en clases y librerías propias. Esto simplificó el proceso.

Las partes que tenían menor deuda tecnológica fueron los más fácil de migran.

5.2.1. Proceso de migración para servicios básico

En primera instancia, para realizar la migración del sistema, se migró un primer servicio, el de noticias. De forma iterativa incremental se creo un servicio que, al principio, levantará un servidor web y se coordine con la infraestructura base descrita en la sección 5.1 usando como

primer apronte el código 5.3. Luego, tomando el código de dicha sección, paulatinamente se le fueron agregando las diversas funcionalidades.

Para generar reuso interno en el proceso de migración, se tomó dicho servicio, se tomó la parte que se conecta con el resto de la infraestructura y algunas consultas de ejemplo, y se generó una plantilla a partir de él para su uso en el resto de los servicios. La plantilla levanta un servidor que maneja las rutas, se conecta con el descubridor de servicios y levanta el servidor.

Todos los servicios tendrán una ruta "health" que devuelve "OK", de ser correcta la llamada para que Consul pueda revisar la salud del servicio.

5.2.2. Proceso de migración para servicios de alto computo

Hay tres servicios que tienen un comportamiento similar, los tres que tienen que procesar datos periódicamente y sus consultas ejecutan algoritmos complejos que implican alto cómputo. Estos servicios son: transformación del texto, tópicos y análisis de lenguaje. Una vez al día descargan gran cantidad de información para procesarlos y generar un objeto que es capaz de resolver en poco tiempo las operaciones que se le consultan.

Para introducir estos servicios en el sistema, se migró en primera instancia el servicio de transformación del texto tomando la plantilla descrita en la subsección 5.2.1 y agregando una clase que maneje el procesamiento, en un proceso aparte, y el objeto resultante fue guardado en un archivo mediante la librería de Python Pickle [46], quedando disponible para otras réplicas del servicio o el mismo servicio al reiniciarse. Luego se recupera en el proceso principal para el uso sistemático por parte del servicio.

Se toma lo realizado en este servicio y se usa como plantilla para los otros dos servicios: tópicos y análisis de lenguaje.

5.2.3. Servicio de noticias

Su nombre en el sistema es "news". Este servicio es el encargado de descargas las noticias de la página de La Nación periódicamente, almacenarlas y mostrarlas.

El servicio tiene los siguientes *endpoints*:

- **/api/v1/news**: devuelve las noticias con paginación y posibilidad de búsqueda.
- **/api/v1/news/:id**: devuelve la noticia cuyo identificador es "id".
- **/api/v1/news/in**: devuelve las noticias de una lista determinada de noticias.
- **/api/v1/news/scrape**: inicia el scrapping de noticias al sitio `http://www.lanacion.cl/`.
- **/health**: devuelve "OK" si el servicio está arriba, error en caso contrario.

5.2.4. Servicio de transformación del texto

Su nombre en el sistema es "similarity". Este servicio periódicamente transforma las noticias que almacena el servicio de noticias en vectores TF-IDF [60] y genera *hash* por cada documento, mediante un algoritmo propio (denominado *Topic Hashing*), para acotar el espacio de búsqueda.

El servicio tiene los siguientes *endpoints*:

- **/api/v1/similarity/topic_hashing**: inicia el proceso de crear los vectores TF-IDF y la generación de un hash por documento. Llama al servicio de noticias en su *endpoint* "/api/v1/news" para obtener las noticias.
- **/api/v1/similarity**: busca noticias similares a un texto determinado.
- **/api/v1/similarity/tfidf**: devuelve todos los vectores TF-IDF generados.
- **/health**: devuelve "OK" si el servicio está arriba, error en caso contrario.

5.2.5. Servicio de análisis de sentimientos

Su nombre en el sistema es “sentimental”. Este servicio se encarga de tomar una cadena de texto y calcular su subjetividad.

El servicio tiene los siguientes *endpoints*:

- **/api/v1/sentimental/subjectivity**: toma una cadena de texto y llamando a servicios externos calcula la subjetividad del texto.
- **/health**: devuelve “OK” si el servicio esta arriba, error en caso contrario.

5.2.6. Servicio de tópicos

Su nombre en el sistema es “topics”. Periódicamente calcula, mediante el algoritmo NMF [34], los tópicos que componen las diversas noticias. Además devuelve información sobre los tópicos.

El servicio tiene los siguientes *endpoints*:

- **/api/v1/topics/generate**: inicia el proceso de crear los tópicos mediante el algoritmo NMF y almacenarlos. Consulta al servicio de “similarity” para obtener los vectores TF-IDF de cada noticia.
- **/api/v1/topics**: devuelve todos los tópicos con las palabras que lo componen.
- **/api/v1/topics/:id**: devuelve el tópico cuyo identificador es “id” con las palabras que lo componen y los identificadores de las noticias que son parte de dicho tópico.
- **/health**: devuelve “OK” si el servicio esta arriba, error en caso contrario.

5.2.7. Servicio de análisis de lenguaje

Su nombre en el sistema es “word2vec”. El presente servicio periódicamente calcula mediante el algoritmo Word2vec [28] una red neuronal de dos capas utilizando las noticias obtenidas. Devuelve los resultados de los tres análisis descritos en la subsección 2.3.

El servicio tiene los siguientes *endpoints*:

- **/api/v1/word2vec/generate**: inicia el proceso de generar la red neuronal y la almacena. Consulta al servicio de noticias para obtener los textos a analizar.
- **/api/v1/word2vec**: devuelve el espectro de palabras que son posibles de utilizar para realizar los análisis.
- **/api/v1/word2vec/similarity**: devuelve el porcentaje de similaridad entre dos palabras.
- **/api/v1/word2vec/most_similar**: devuelve una lista de palabras similares a un conjunto de conceptos que a su vez es distante a otro conjunto.
- **/api/v1/word2vec/doesnt_match**: devuelve, de una lista de palabras, la que no coincide.
- **/health**: devuelve “OK” si el servicio esta arriba, error en caso contrario.

5.2.8. Servicio planificador

Su nombre en el sistema es “scheduler”. Este servicio se encarga de enviar a los servicios de noticias, de tópicos, de transformación y de análisis de lenguaje que realicen una vez al día sus tareas antes mencionadas.

Solo tiene un *endpoint*: “/health”.

5.2.9. Servicio de interfaz

Su nombre en el sistema es “frontend”. Llama a todos los servicios menos, a “scheduler”, para obtener la información que muestra al usuario final. Se puede observar la interacción de las distintas secciones del sitio con el resto de los servicios en la figura 5.1.

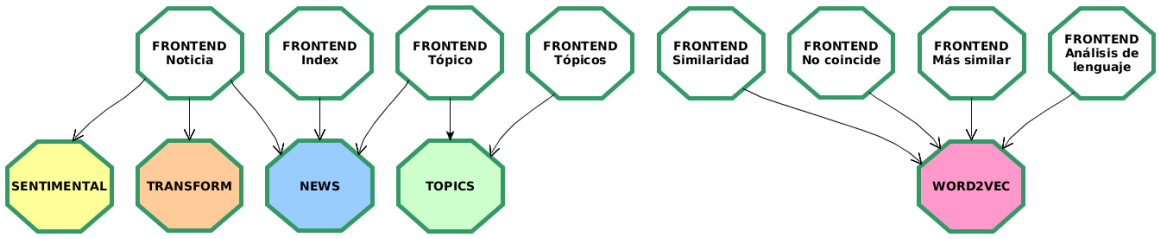


Figura 5.1: Diagrama de la interacción del servicio de interfaz y sus distintas vistas con el resto de los servicios.

Por diseño, este servicio también es el BFF del sistema como se explica en la sección 4.2.

Cabe mencionar que visualmente es exactamente la misma interfaz que la versión monolítica.

5.3. Distribución del sistema

Para distribuir el sistema en otras máquinas se debe instalar y ejecuta Consul mediante el comando que se muestra en el código 5.4.

```
1  ./consul agent -join=$IP -data-dir /tmp/consul
```

Código 5.4: Comando para iniciar Consul en una réplica con “$IP” como la IP del servidor que contiene el enrutador.

Luego de esto, todos los servicios que se inicien en la máquina, y se registren en Consul, serán considerados dentro del ecosistema de microservicios de la aplicación.

Ambos sistemas, tanto el monolítico como la versión en microservicios, se distribuyó hasta en 3 máquinas. Cada una es una máquina virtual de DigitalOcean, con 2GB de RAM y dos núcleos de procesamiento.

En la versión monolítica, cada máquina tiene una instancia de la aplicación. Ya que el sistema consume 1.1GB de RAM, no se logró una segunda réplica por máquina.

Las réplicas realizadas en la versión de microservicios se muestran en el cuadro 5.1.

Servicio	**Máquina 1**	**Máquina 2**	**Máquina 3**
frontend	1	8	3
news	1	3	6
sentimental	1	1	4
topics	1	4	0
word2vec	1	1	2
scheduler	1	1	1
similarity	1	3	0

Cuadro 5.1: Cantidad de replicas por máquina de los distintos servicios.

Los datos, tanto la base de datos como los archivos Pickle de Python, se replican en todas las máquinas.

Cabe mencionar que para medir la migración de la arquitectura, y no la calidad de Fabio como balanceador de carga, se utilizó la misma combinación (Consul en conjunto con Fabio) en el sistema monolítico para distribuirlo.

Capítulo 6

Resultados

En miras a revisar el cumplimiento de los objetivos planteados en la sección 1.2, en cuanto a cuantificar el impacto de la migración en términos de rendimiento y escalabilidad, se realizan los experimentos que se detallarán en el presente capítulo.

Las pruebas realizadas se aplicaron a las siguientes versiones del sistema:

- `MONOLITHIC`, sistema original sin replicar.
- `MONOLITHIC REPLICATE (2)`, sistema original replicado una vez (dos maquinás).
- `MONOLITHIC REPLICATE (3)`, sistema original replicado dos veces (tres maquinás).
- `MICROSERVICES`, sistema migrado a microservicios sin replicar. Solo utiliza la primera maquiná de la tabla 5.1.
- `MICROSERVICES REPLICATE (2)`, sistema migrado a microservicios replicado y distribuido en dos maquinás. Utiliza la primera y segunda maquiná de la tabla 5.1.
- `MICROSERVICES REPLICATE (3)`, sistema migrado a microservicios replicado y distribuido en tres maquinás. Utiliza las tres maquinás de la tabla 5.1.

Para realizar las pruebas se utilizó el software JMeter de Apache en su versión 2.13 corriendo

en un computador utilizando Archlinux con 4GB de RAM y un procesador Intel Core i7-3517U con dos núcleos de 1.90GHz. Se realizaron 7 pruebas a cada versión del sistema, cada una simulando 500 usuarios tratando de entrar en 5 segundos.

Cada una de las 7 pruebas fue a una parte distinta del sitio, las cuales fueron:

- **Index**: portada del sitio, contiene todas las noticias.
- **Noticia**: página de una noticia en particular.
- **Portada de tópicos**: portada de la sección de tópicos, contiene todas los tópicos.
- **Tópico**: página de un tópico en particular.
- **Análisis de lenguaje**: portada de la sección de análisis de lenguaje.
- **Similitud entre palabras**: comparación de la similitud de dos palabras.
- **Salud**: página simple que solo devuelve "OK", se usa para medir el tiempo de reacción del servicio de la interfaz solamente.

6.1. Análisis de resultados

En la presente subsección se mostraran y analizaran los resultados detallados obtenidos de las pruebas antes descritas en las diversas secciones.

6.1.1. Inicio

Ninguna de las versiones mostró errores al realizar las pruebas. Además, como se puede ver en las figuras 6.1 y 6.2, la versión `MONOLITHIC REPLICATE (2)` tiene mejor resultado que la versión `MICROSERVICES REPLICATE (2)` y a su vez `MONOLITHIC REPLICATE (3)`. En los otros casos las versiones realizadas con microservicios tienen mejor desempeño que su par monolítico.

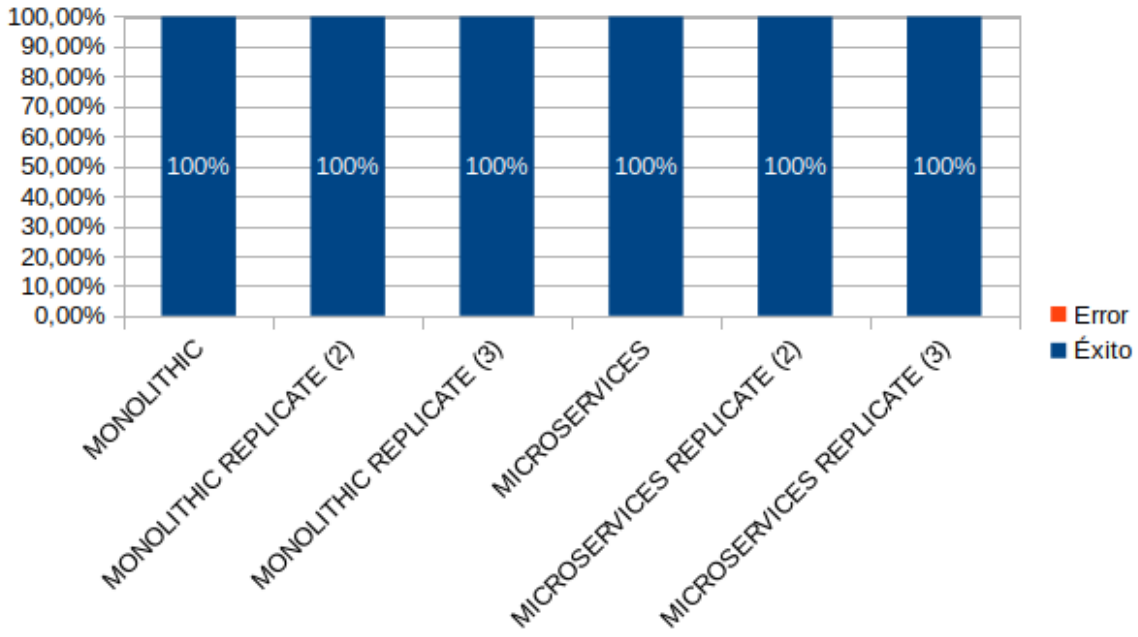


Figura 6.1: Porcentaje promedio de error de las pruebas en su respectiva versión del sistema en la página principal del sitio.

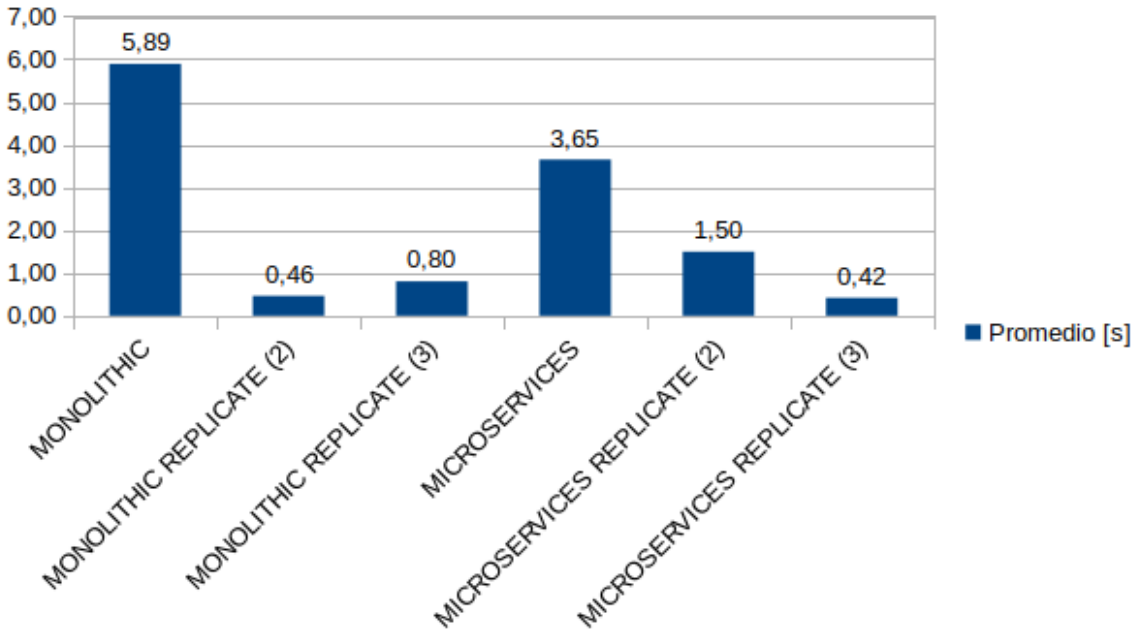


Figura 6.2: Tiempos promedio de las pruebas en su respectiva versión del sistema en la página principal del sitio.

A su vez, como se ve en la figura 6.3, tanto `MONOLITHIC REPLICATE (2)` como `MICROSERVICES REPLICATE (3)` responden prácticamente igual en todo momento, manteniendo prácticamente siempre un tiempo de respuesta menor a 0.5 segundos.

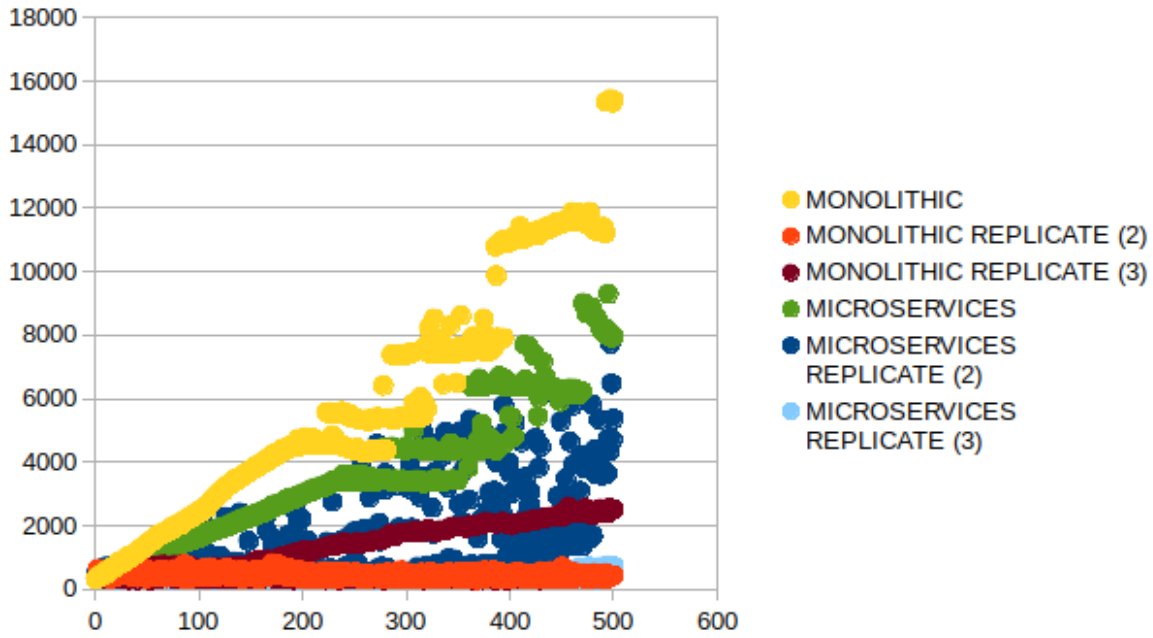


Figura 6.3: Detalle de las muestras en las pruebas realizadas a las distintas versiones del sistema en la página principal del sitio.

6.1.2. Noticias

En ambos casos, tanto en la versión monolítica como la versión migrada a microservicios, existe una tasa de error enorme en las versiones no distribuidas. Además, como se puede ver en las figuras 6.4 y 6.5, las versiones monolíticas tienen un error y tiempo de respuesta mayor a las versiones de microservicios.

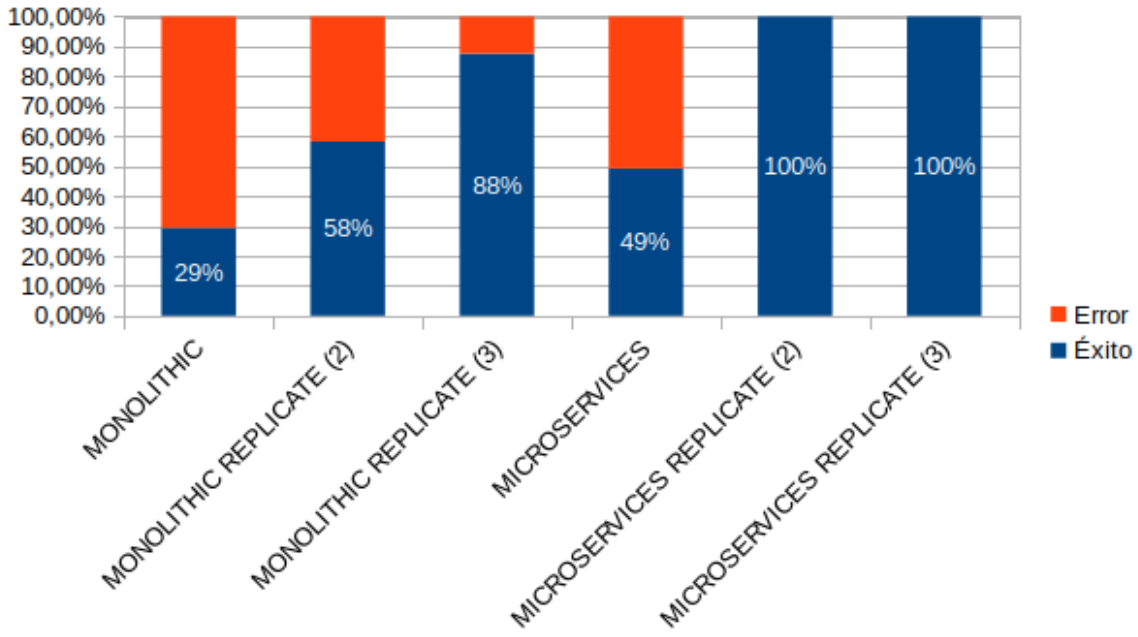


Figura 6.4: Figura el porcentaje promedio de error de las pruebas en su respectiva versión del sistema en la página de una noticia del sitio.

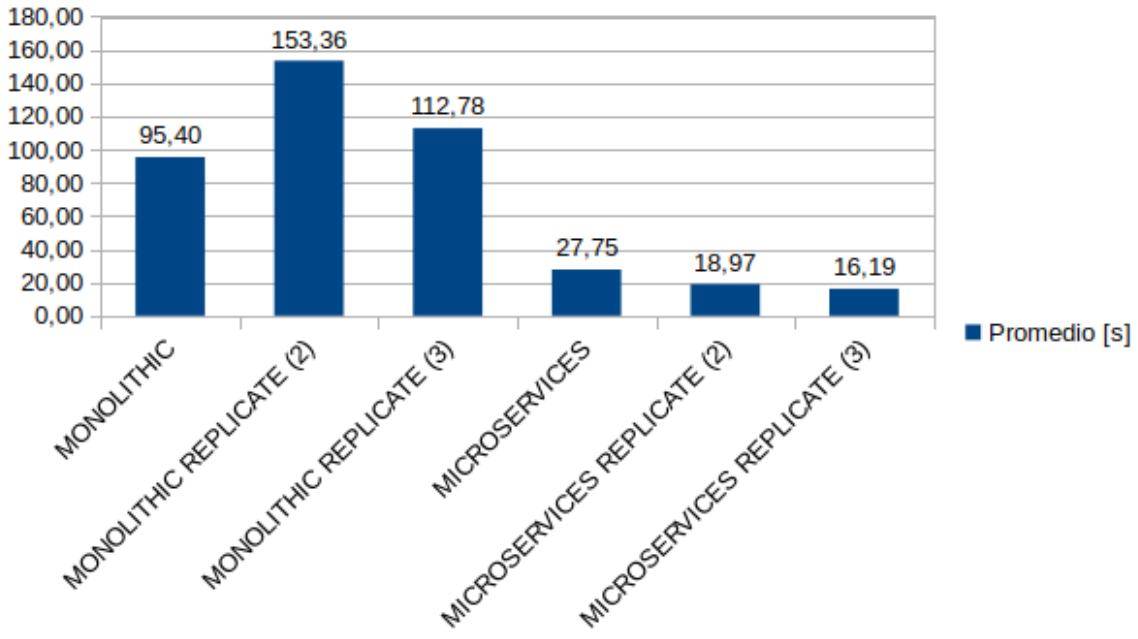


Figura 6.5: Tiempos promedio de respuesta de las pruebas en su respectiva versión del sistema en la página de una noticia del sitio.

A su vez, como se ve en la figura 6.6 todas las versiones del sistema crecen linealmente en sus tiempos de respuesta pero la versiones con microservicio tienen una pendiente menor. Los experimentos realizados en las versiones monolíticas mantienen el mismo tiempo de respuesta ya que son consultas que devuelven error, cuando dejan de devolver error vuelven a tener un comportamiento lineal con una enorme pendiente.

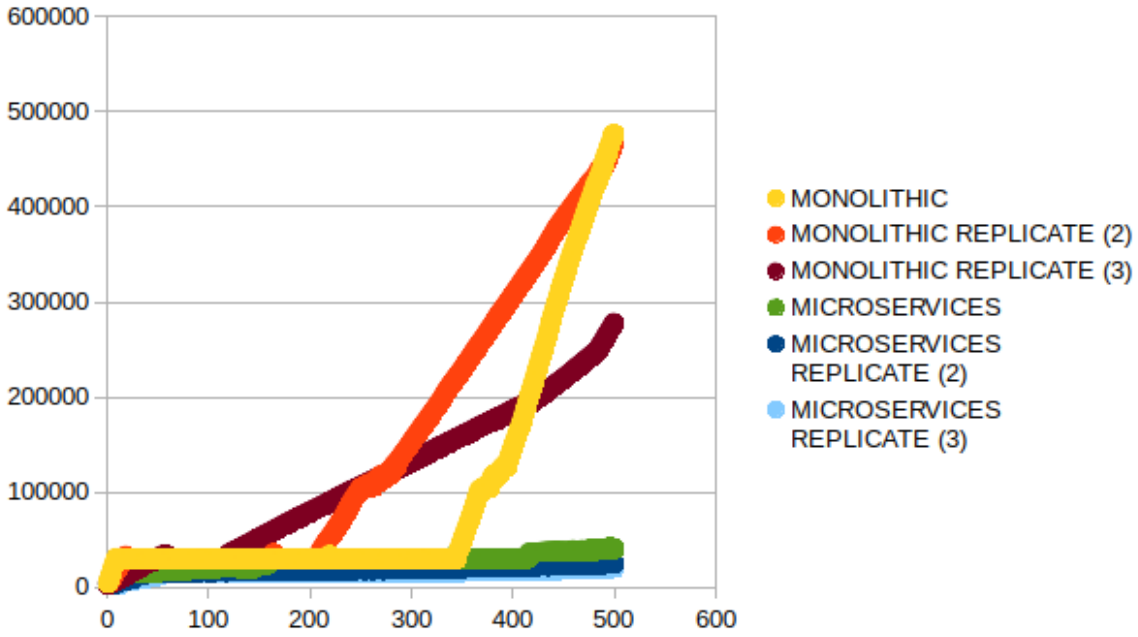


Figura 6.6: Detalle de las muestras en las pruebas realizadas a las distintas versiones del sistema en la página de una noticia del sitio.

Esta es la sección con el cómputo más alto del sistema. Aislar los servicios que requieren mayor tiempo de cómputo resultó altamente efectivo en este caso.

6.1.3. Inicio tópicos

Esta sección del sistema tiene una gran tasa de error en la versión `MICROSERVICES`, que desaparece al distribuir el sistema.

Respecto a los tiempos de respuesta, tanto la versión monolítica como la de microservicios

no distribuye bien. En ambos casos, tanto la versión monolítica como la migración a microservicios, sin considerar la versión `MICROSERVICES` producto de su porcentaje de error, los resultados se mantienen prácticamente igual.

Como se puede observar en las figuras 6.7 y 6.8, las versiones monolíticas tienen un error y tiempo de respuesta bastante menor a las versiones de microservicios.

El comportamiento antes descrito es producto de la latencia adicional que genera pasar 6 MB entre servicios para que le llegue al usuario.

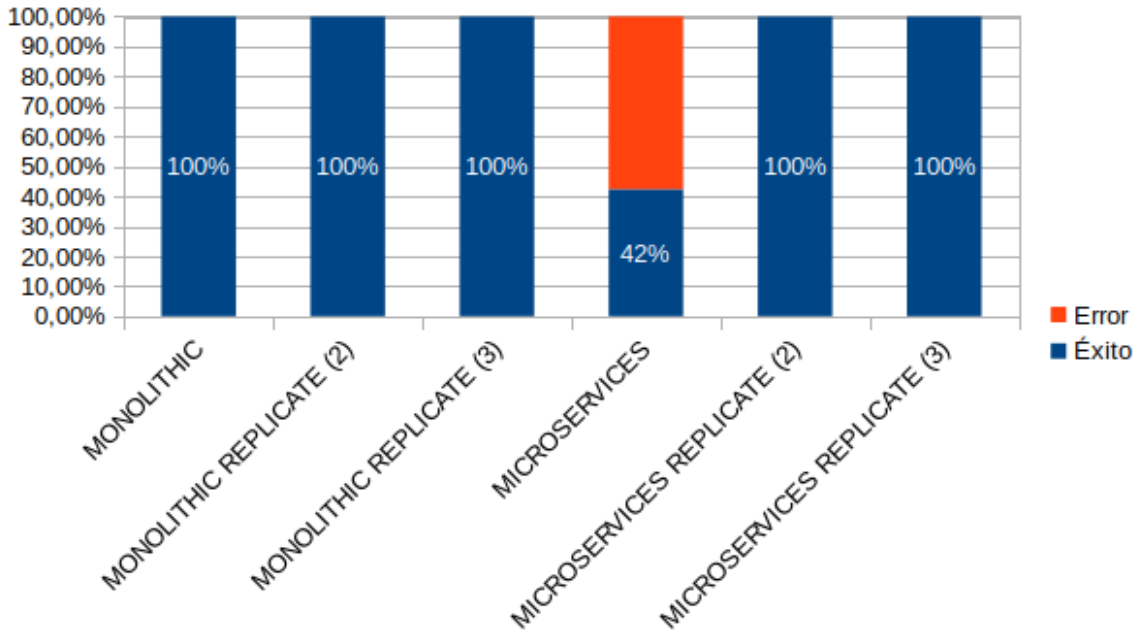


Figura 6.7: Figura el porcentaje promedio de error de las pruebas en su respectiva versión del sistema en la página inicial de los tópicos.

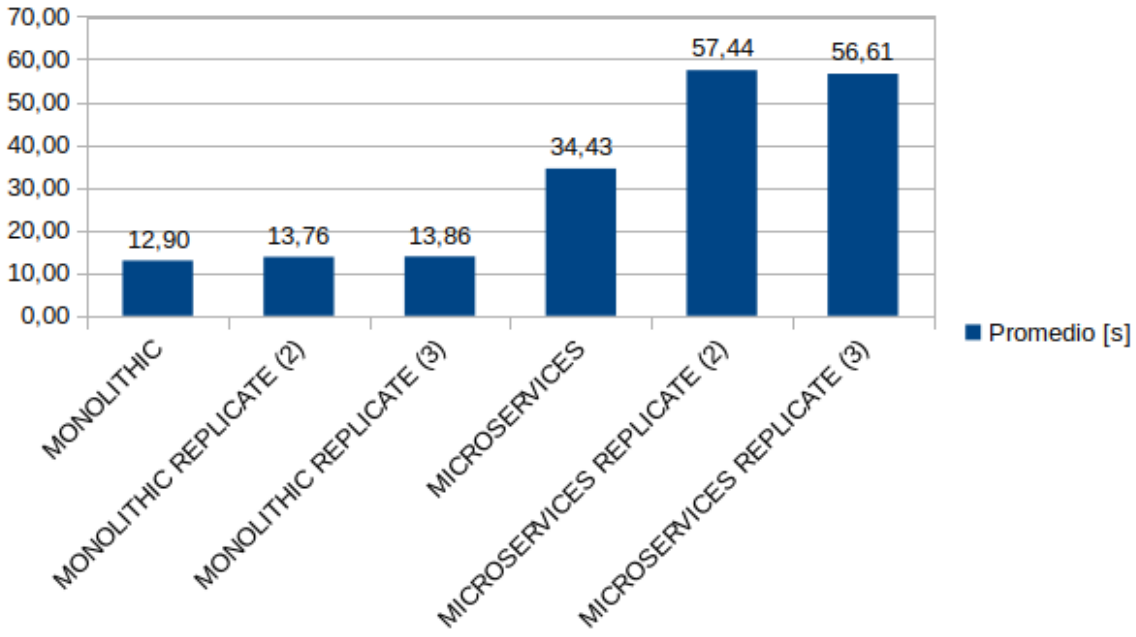


Figura 6.8: Tiempos promedio de respuesta de las pruebas en su respectiva versión del sistema en la página inicial de los tópicos.

A su vez, como se ve en la figura 6.9, las versiones con microservicios `MICROSERVICES REPLICATE (2)` como `MICROSERVICES REPLICATE (3)` tienen un comportamiento similar pero `MICROSERVICES REPLICATE (3)` tiene mejor desempeño que `MICROSERVICES REPLICATE (2)` cuando hay menos carga, pero pasado cierto punto `MICROSERVICES REPLICATE (2)` tiene mejor desempeño.

En contrapartida, las distintas versiones del sistema monolítico se comportan prácticamente igual.

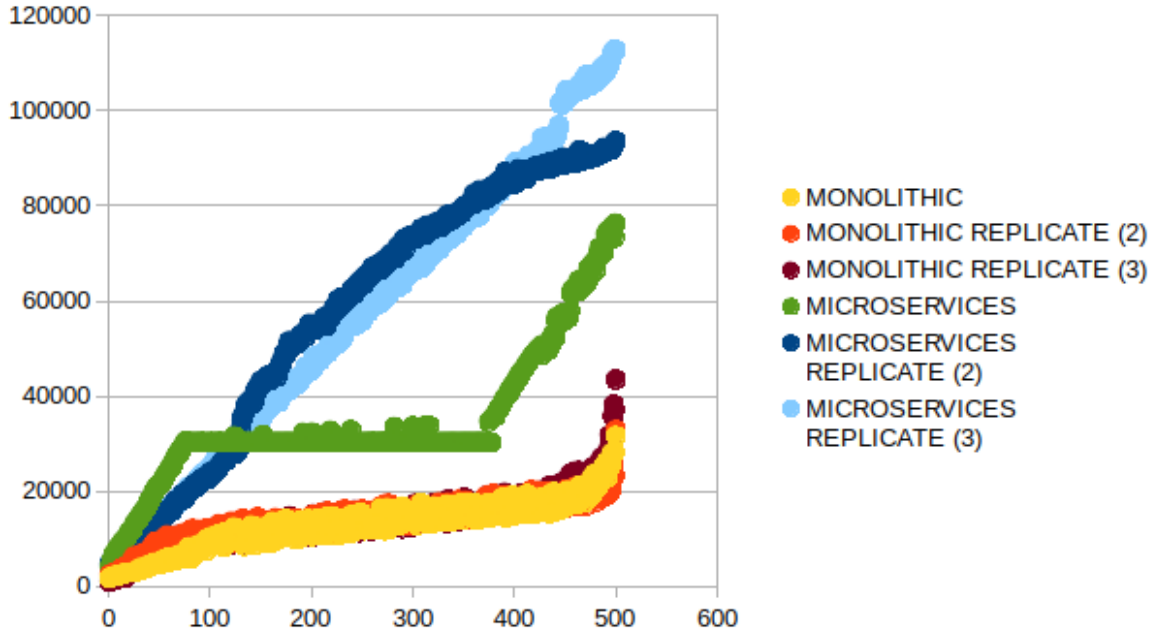


Figura 6.9: Detalle de las muestras en las pruebas realizadas a las distintas versiones del sistema en la página inicial de los tópicos.

6.1.4. Tópico

Al igual que la sección anterior existe una tasa considerable de error en la versión `MICROSERVICES`. En ambos casos, se distribuye bastante bien, ya que no comparte las características de alto peso de la prueba anterior.

Como se puede ver en las figuras 6.10 y 6.11, las versiones monolíticas tienen un error y tiempo de respuesta menor a las versiones de microservicios.

Aun así las versiones de microservicios escalan bastante bien. Los cuatro servicios adicionales de tópicos ayudan a eliminar las consultas que devuelven error y disminuir el tiempo promedio de respuesta de 19.77 a 4.64.

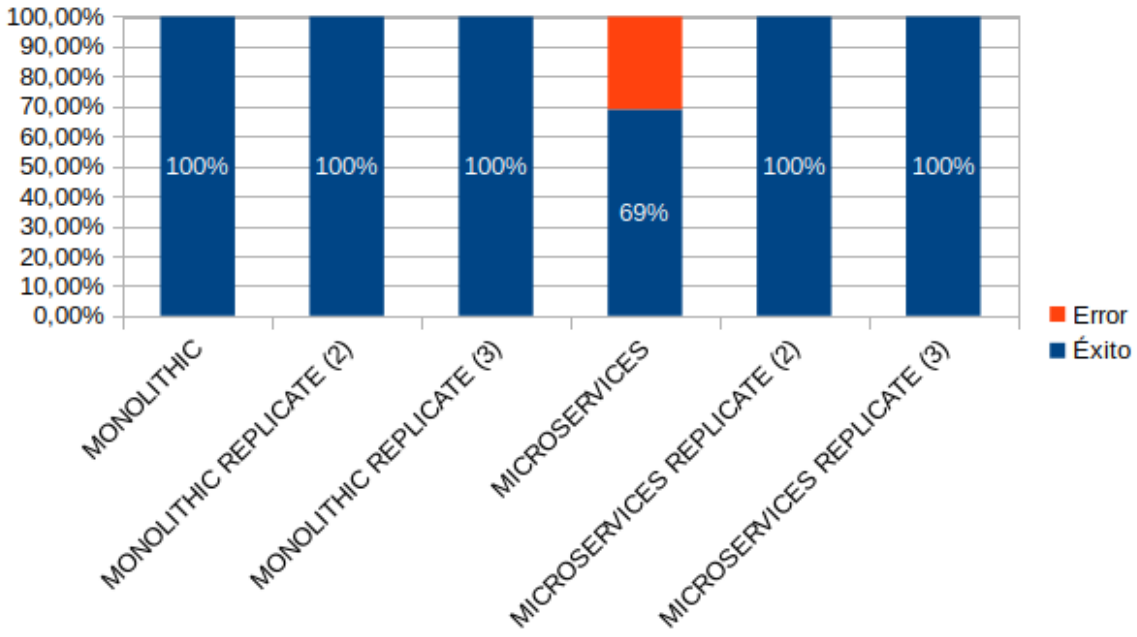


Figura 6.10: Figura el porcentaje promedio de error de las pruebas en su respectiva versión del sistema en la página de un tópico.

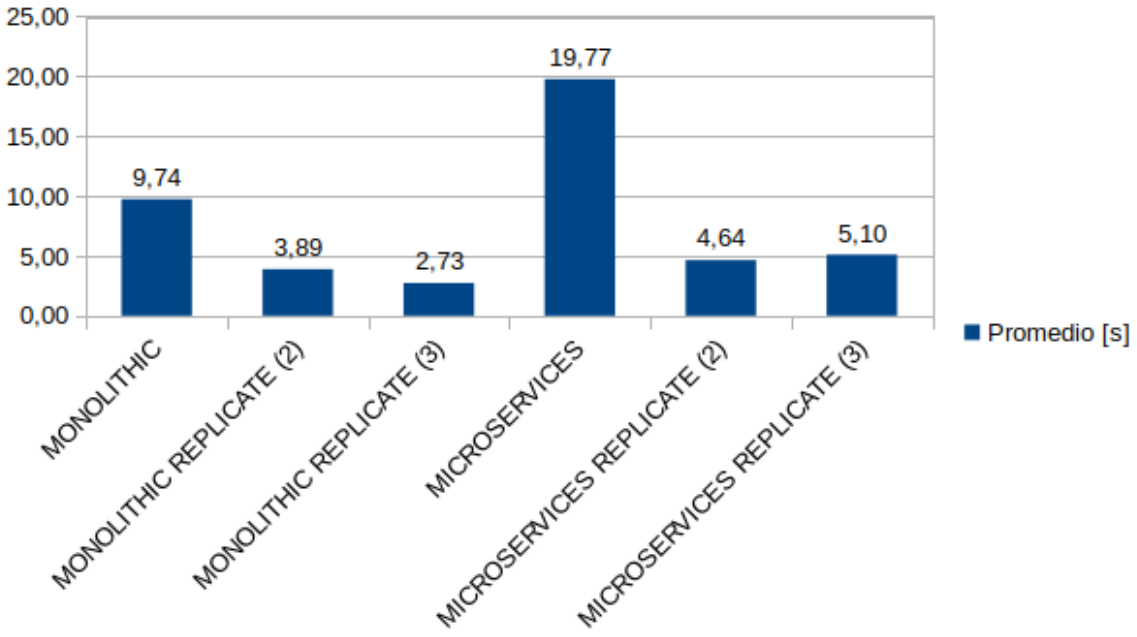


Figura 6.11: Tiempos promedio de respuesta de las pruebas en su respectiva versión del sistema en la página de un tópico.

A su vez, como se ve en la figura 6.12, todas las versiones mantienen el mismo comportamiento lineal. Tan solo cambia la pendiente.

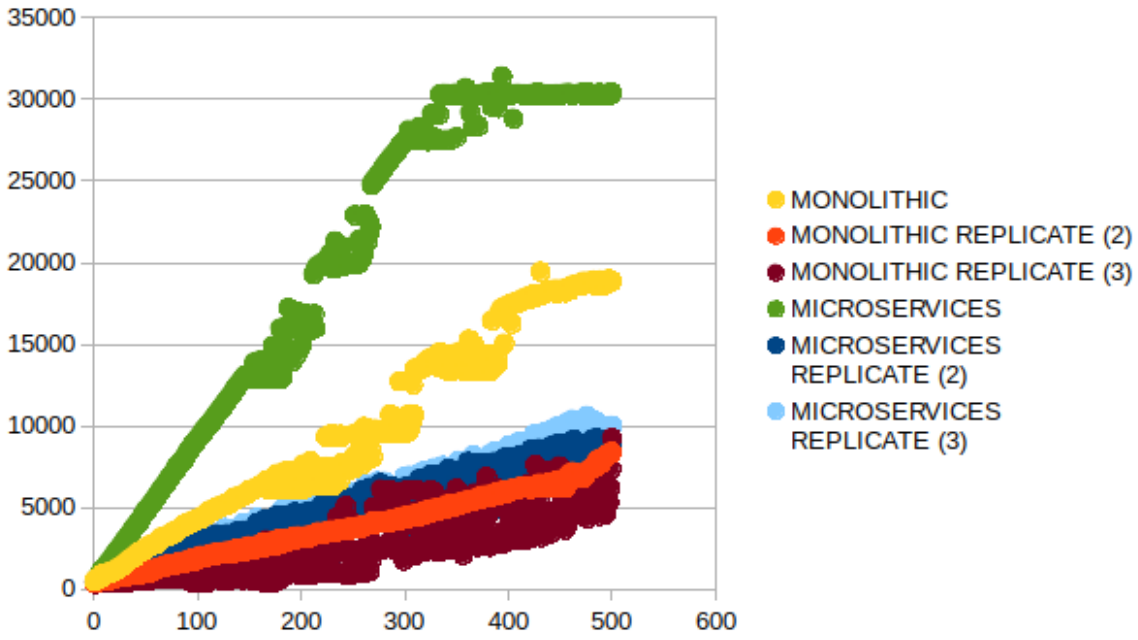


Figura 6.12: Detalle de las muestras en las pruebas realizadas a las distintas versiones del sistema en la página de un tópico.

6.1.5. Inicio de análisis de lenguaje

En esta sección en ninguna de las versiones las pruebas realizadas arroja errores, como se puede ver en la figura 6.13. Además, como se puede ver en la figura 6.14, las versiones monolíticas tienen tiempo de respuesta levemente menor a las versiones de microservicios.

No solo las versiones de microservicios tienen en todos los casos mejores resultados que las versiones monolíticas, además escalan bastante bien.

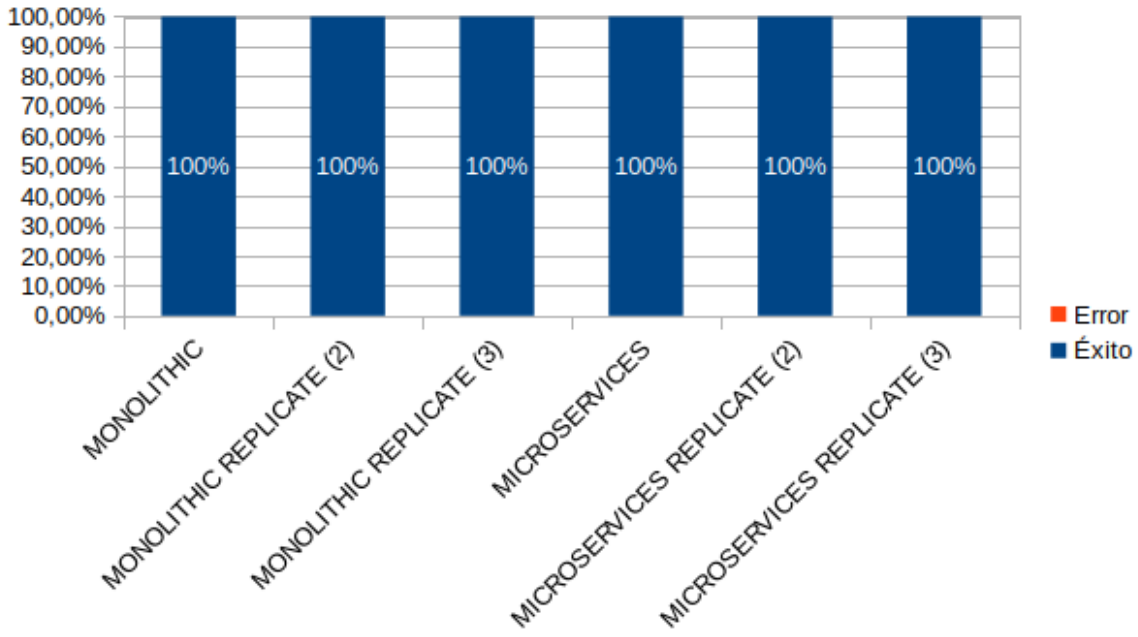


Figura 6.13: Figura el porcentaje promedio de error de las pruebas en su respectiva versión del sistema en la página de inició de la sección de análisis de lenguaje.

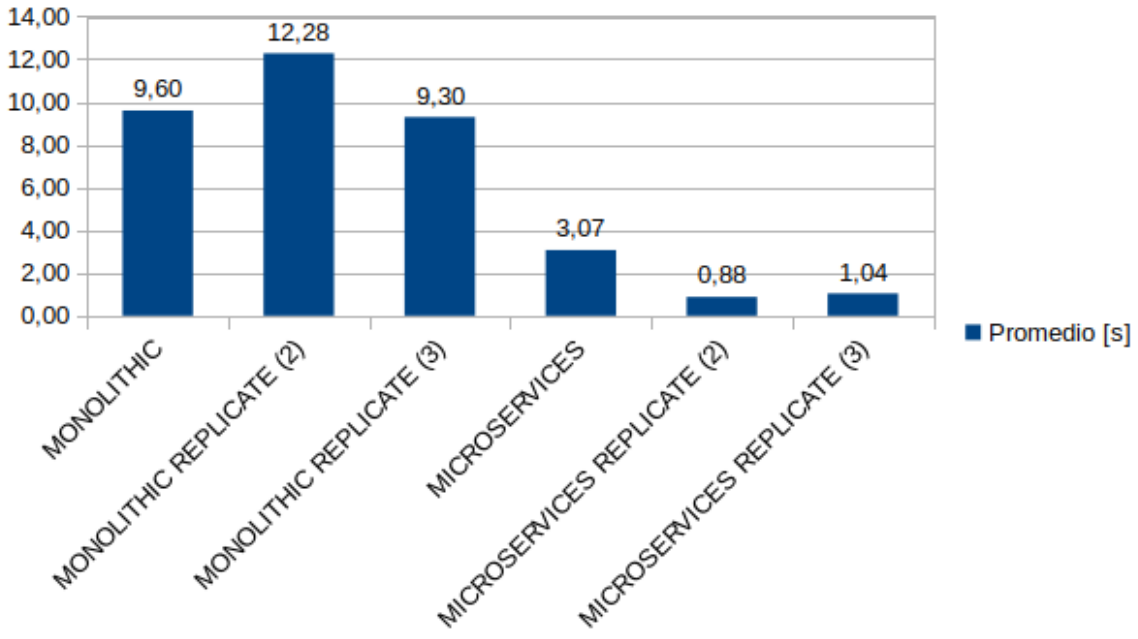


Figura 6.14: Tiempos promedio de respuesta de las pruebas en su respectiva versión del sistema en la página de inició de la sección de análisis de lenguaje.

A su vez, como se ve en la figura 6.15, todas las versiones mantienen el mismo comportamiento lineal excepto `MICROSERVICES REPLICATE (2)` y `MICROSERVICES REPLICATE (3)`, que su pendiente tiende a 0.

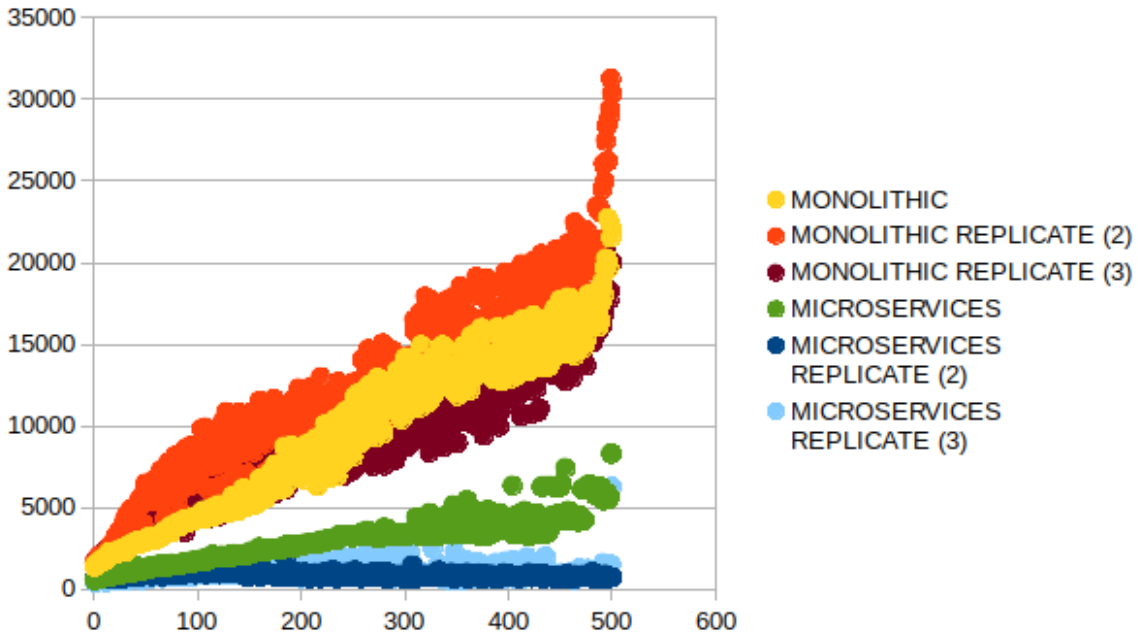


Figura 6.15: Detalle de las muestras en las pruebas realizadas a las distintas versiones del sistema en la página de inició de la sección de análisis de lenguaje.

6.1.6. Similitud entre palabras

En esta sección ninguna de las versiones las pruebas realizadas arroja errores, como se puede ver en la figura 6.16. Además, como se puede ver en la figura 6.17, las versiones monolíticas tienen tiempos de respuestas levemente menor a las versiones de microservicios.

Se puede observar como las versiones de microservicios son levemente más lentas por la latencia propia de la arquitectura, pero son irrelevantes las diferencias por los bajos tiempos de respuesta ya que no se alcanza a estresar los sistemas.

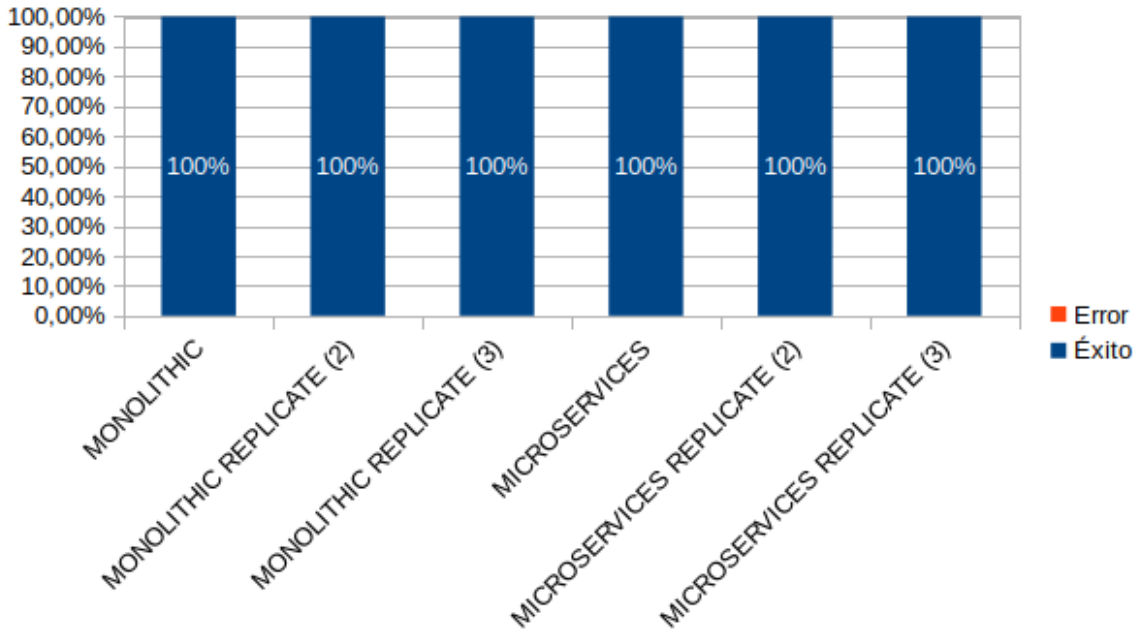


Figura 6.16: Figura el porcentaje promedio de error de las pruebas en su respectiva versión del sistema en la página de similitud entre dos palabras.

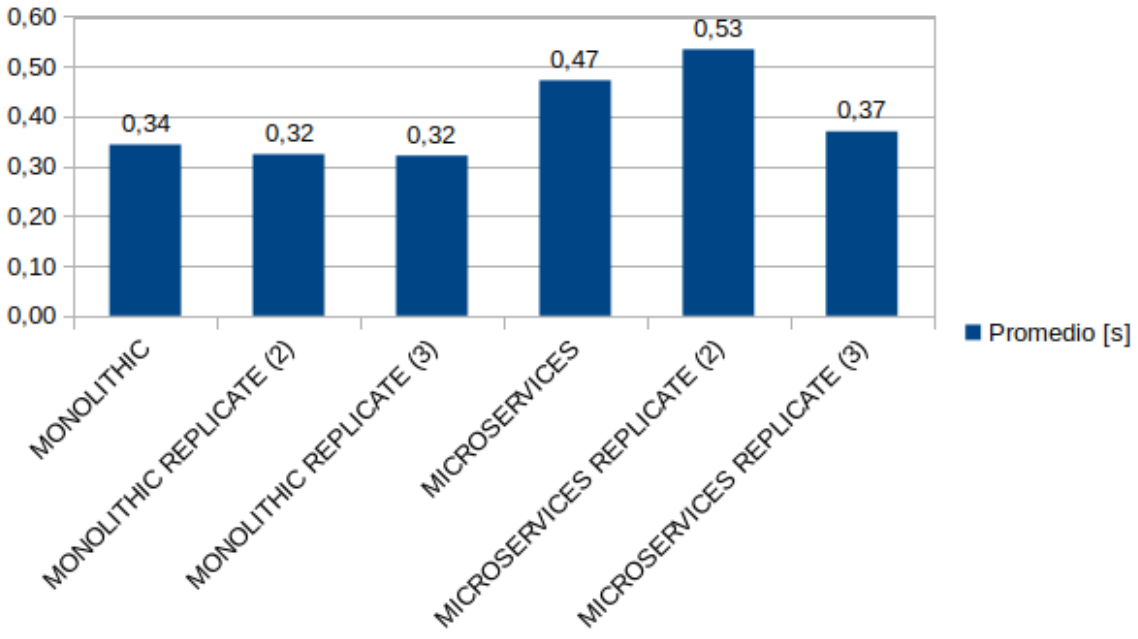


Figura 6.17: Tiempos promedio de respuesta de las pruebas en su respectiva versión del sistema en la página de similitud entre dos palabras.

A su vez, como se ve en la figura 6.18, todas las versiones mantienen el mismo tiempo con algunas excepciones no relevantes para el caso.

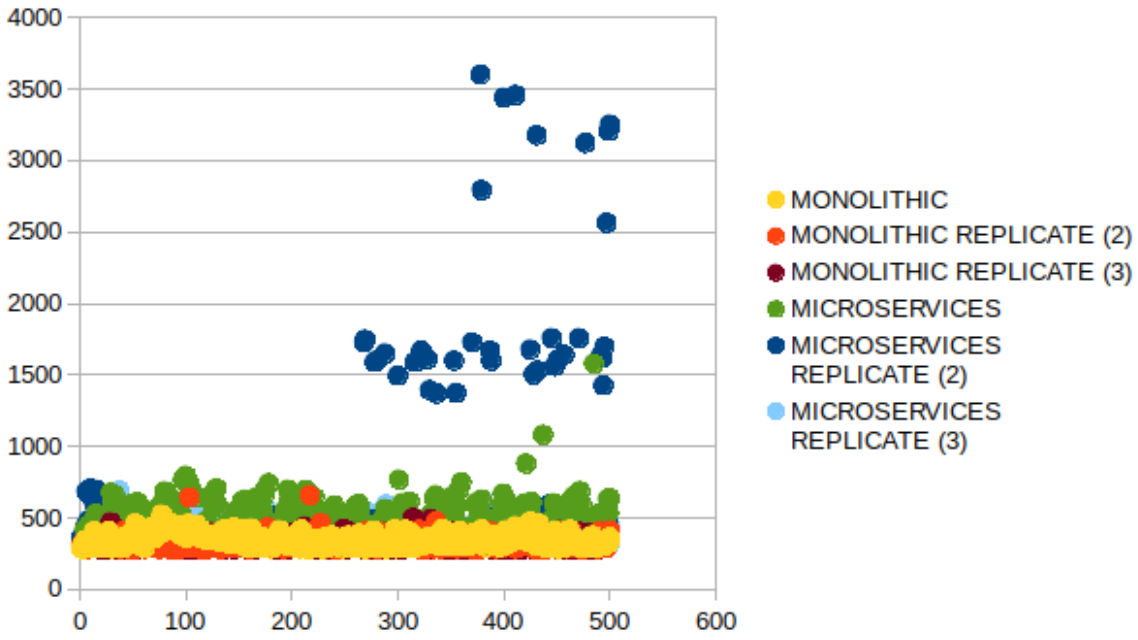


Figura 6.18: Detalle de las muestras en las pruebas realizadas a las distintas versiones del sistema en la página de similitud entre dos palabras.

6.1.7. Salud

En esta sección ninguna de las versiones en las pruebas realizadas arroja errores, como se puede ver en la figura 6.19. Además, como se puede ver en la figura 6.20, las versiones monolíticas tienen tiempo de respuesta levemente menor a las versiones de microservicios.

Se puede observar como las versiones de microservicios son levemente más lentas por la latencia propia de la arquitectura, pero son irrelevantes las diferencias por los bajos tiempos de respuesta ya que no se alcanza a estresar los sistemas.

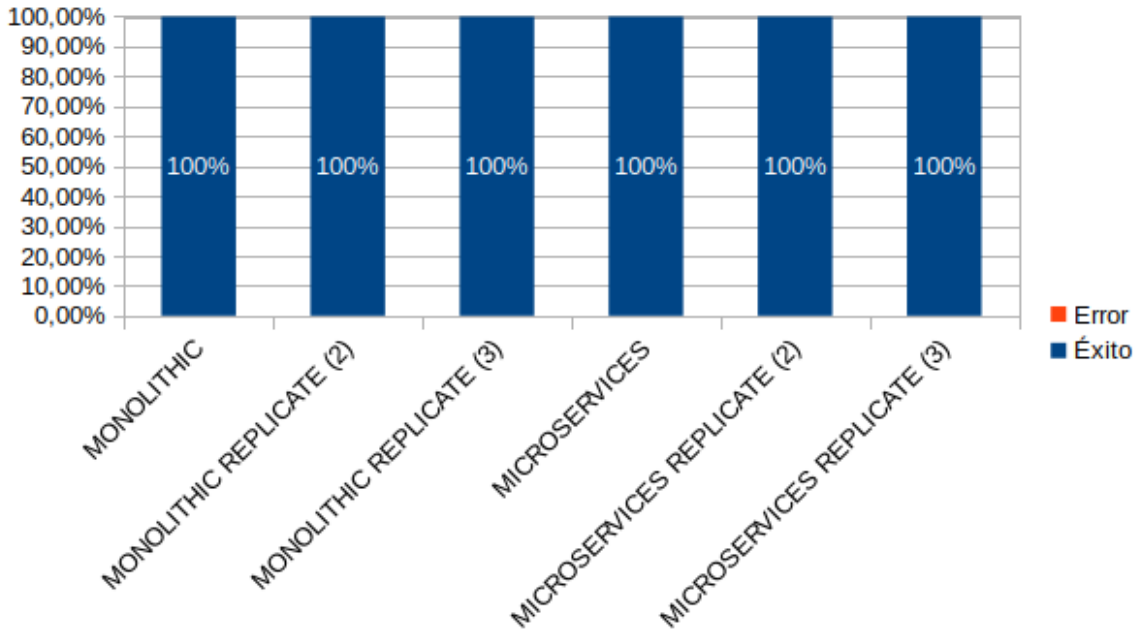


Figura 6.19: Figura el porcentaje promedio de error de las pruebas en su respectiva versión del sistema en la página que se usa para medir la salud del servicio de interfaz.

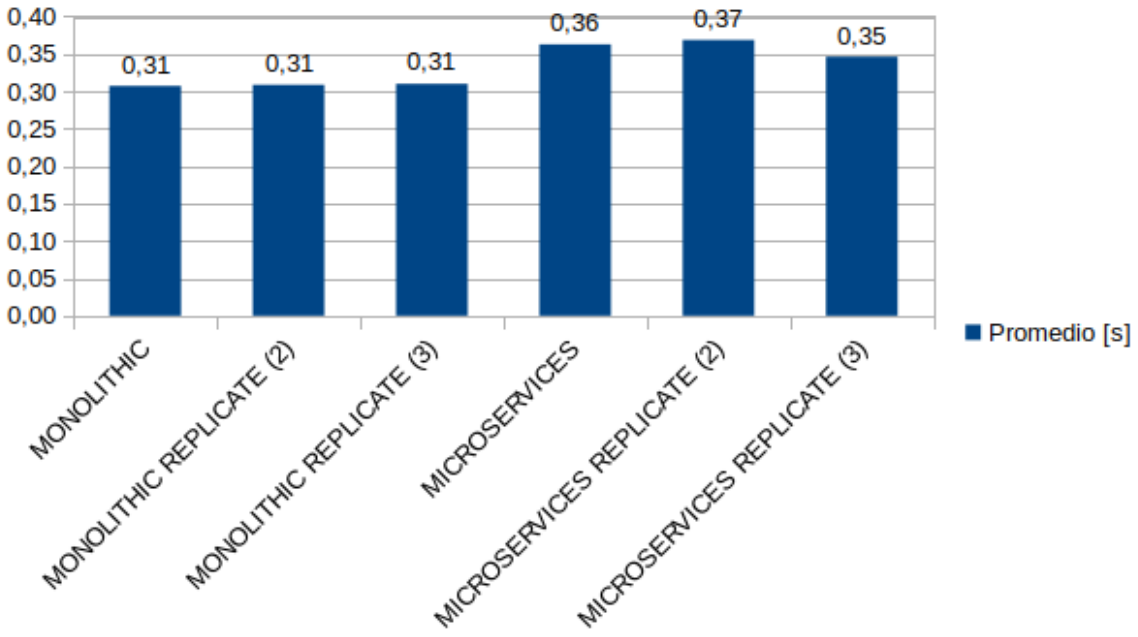


Figura 6.20: Tiempos promedio de respuesta de las pruebas en su respectiva versión del sistema en la página que se usa para medir la salud del servicio de interfaz.

A su vez, como se ve en la figura 6.21, todas las versiones mantienen el mismo tiempo con algunas excepciones no relevantes para el caso.

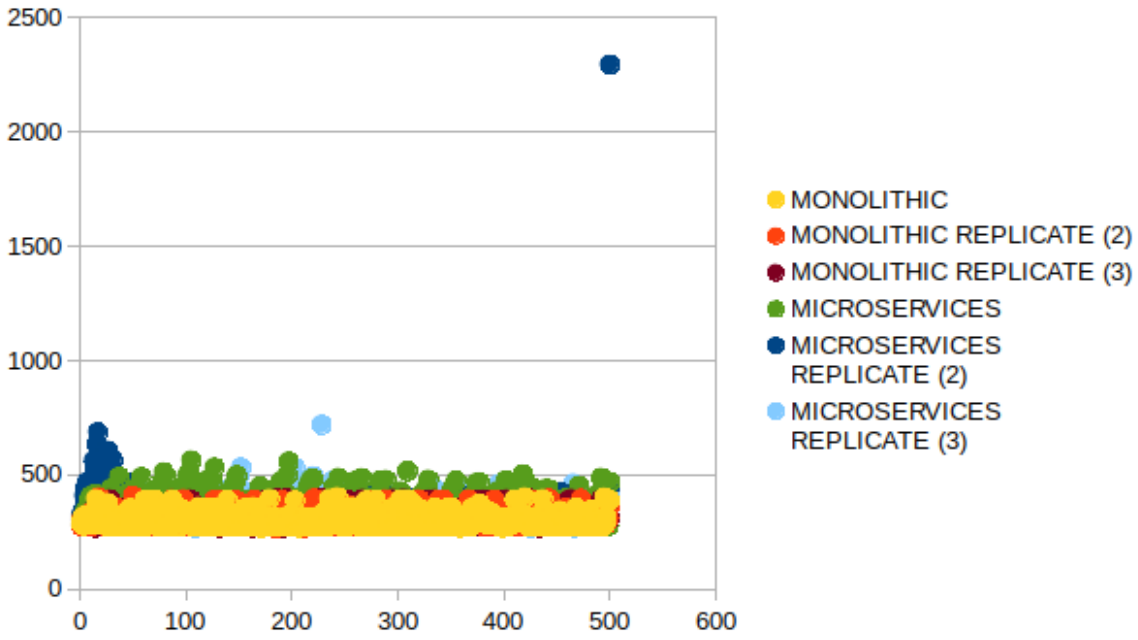


Figura 6.21: Detalle de las muestras en las pruebas realizadas a las distintas versiones del sistema en la página que se usa para medir la salud del servicio de interfaz.

6.2. Resultados generales

Realizando una ponderación de los resultados obtenidos, como se observa en la figura 6.22 y 6.23, la versión sin replicar de microservicios tiene, en promedio, mejor tiempo de respuesta pero mayor porcentaje de consultas erróneas que la versión sin replicar monolítica.

Al replicar y distribuir el sistema, se puede observar que `MICROSERVICES REPLICATE (2)` y `MICROSERVICES REPLICATE (3)` se reduce a 0 % el error y reduce levemente los tiempos de respuesta, superando con creces la versiones `MONOLITHIC REPLICATE (2)` y `MONOLITHIC REPLICATE (3)`.

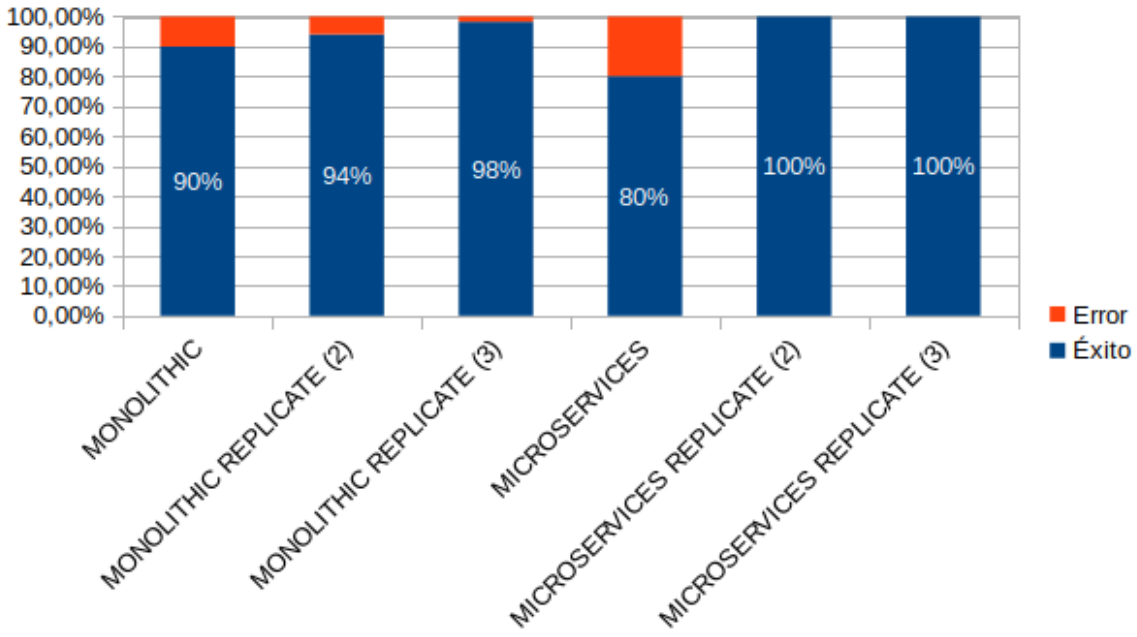


Figura 6.22: Porcentaje promedio de error de las pruebas en su respectiva versión del sistema.

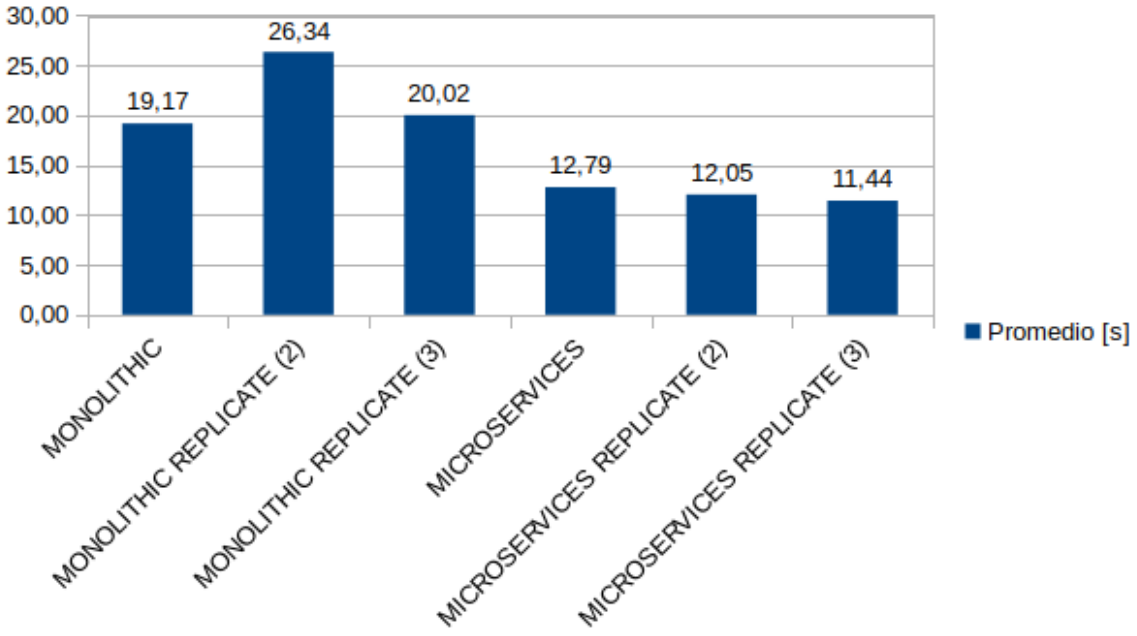


Figura 6.23: Tiempos promedio de respuesta de las pruebas en su respectiva versión del sistema.

De todas formas, quitando las secciones asociadas a tópicos, como muestra la figura 6.24 y 6.25, que son las que menos escalan en la versión de microservicios, todas las versiones migradas a microservicios superan a las versiones monolíticas.

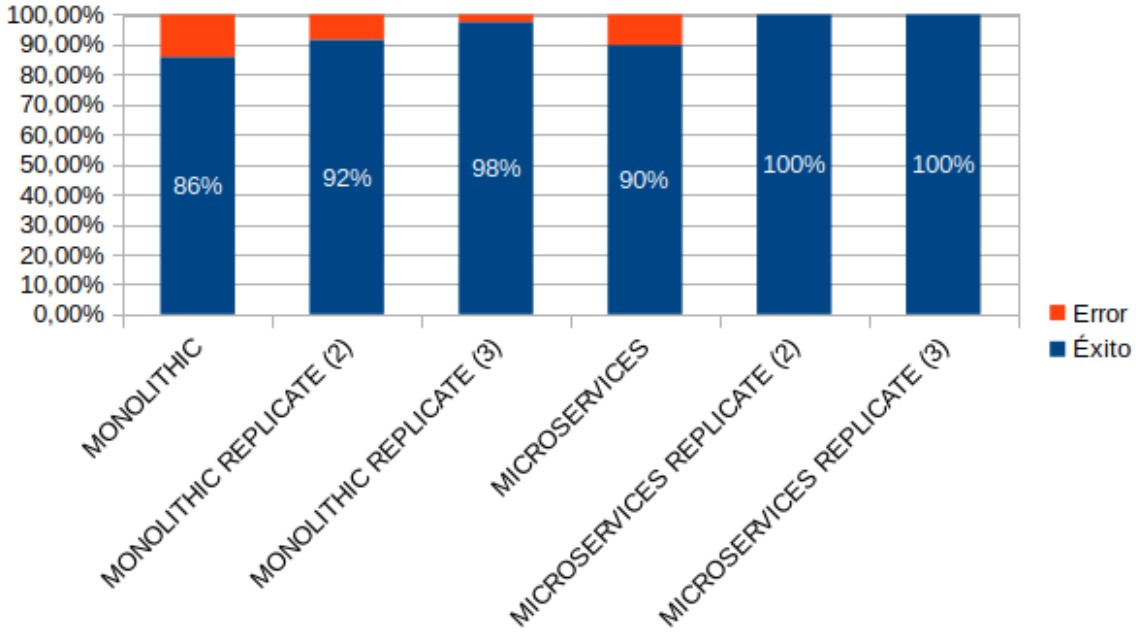


Figura 6.24: Porcentaje promedio de error de las pruebas en su respectiva versión del sistema sin la sección de tópicos.

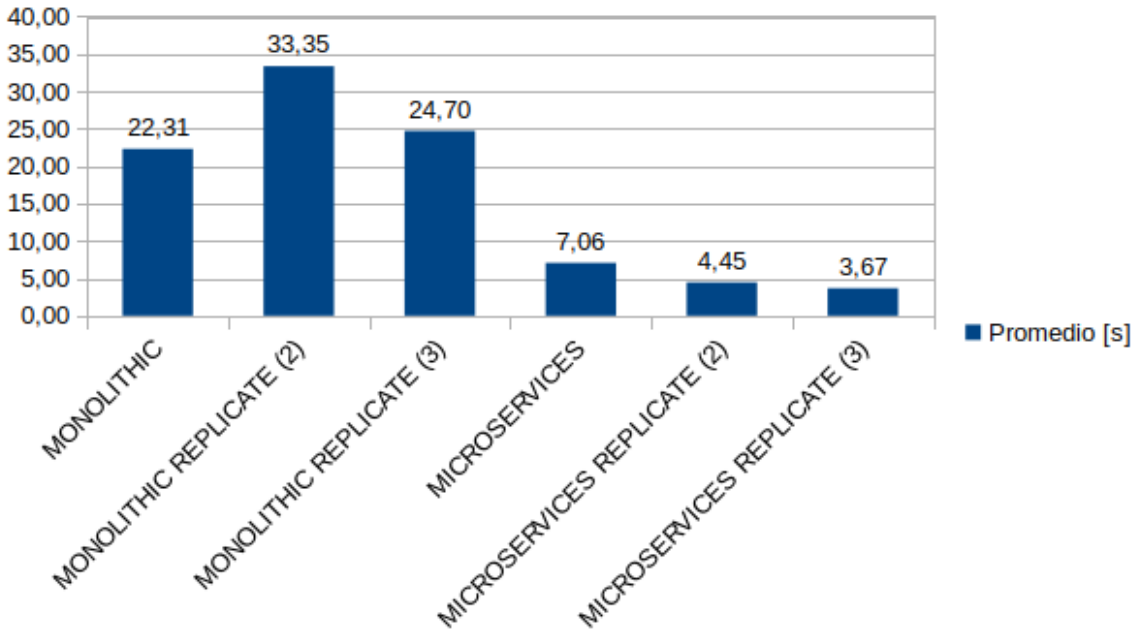


Figura 6.25: Tiempos promedio de respuesta en sus respectivas versión del sistema sin la sección de tópicos.

Capítulo 7

Conclusiones

El objetivo de este trabajo es realizar una migración arquitectónica que solucione los problemas de mantenibilidad y resiliencia a fallos del sistema descrito en el capitulo 2, sin generar un impacto negativo importante en rendimiento. Además de observar cómo impacta en la escalabilidad del sistema dicha migración.

Se observa el cumplimiento de este objetivo, ya que el impacto negativo de la migración en el sistema es bastante marginal, e incluso si no se considera una de las secciones del sistema (la de tópicos), la versión con microservicios supera en todos los aspectos a la versión monolítica.

Al distribuir rápidamente, supera en promedio las versiones con microservicios a sus pares monolíticas. Esto se debe a que se pueden replicar los servicios que mayor impacto tienen más veces, y por ello se puede hacer un mejor uso de la memoria. En el caso del presente experimento, por un pequeño margen no se pudo replicar dos veces por máquina la versión monolítica, mientras que los servicios de la versión realizada con microservicio se pudieron replicar a antojo.

Los microservicios, como se puede observar en la subsección 6.1.3, tienen un peor desempeño cuando hay un gran traspaso de datos. Se debe reducir esto a la hora de implementar microservicios.

Los microservicios, como se puede observar en la subsección 6.1.2, tienen un desempeño mejor cuando hay alto procesamiento ya que pueden aislar y distribuir en diversas máquinas las distintas partes de dicho procesamiento.

En servicios donde no hay alto procesamiento ni alto traspaso de datos entre los microservicios, y que no alcanzan a ser estresados como el descrito en la subsección 6.1.6, el sistema migrado a microservicios tienen un desempeño levemente peor producto de la latencia adicional que produce el traspaso de datos entre los servicios. Pero es tan baja (en algunos casos no alcanza los 0.1 segundos), que es despreciable.

Sobre el proceso de desarrollo, se observa que la migración realizada de la infraestructura base, como se describe en la sección 5.1, es bastante transparente, principalmente por que los módulos son bastante autónomos y se utilizó el mismo *framework* del sistema original para realizar la migración.

Es fundamental a la hora de trabajar con microservicios tener una buena infraestructura base que automatice la puesta en producción del servicio. Principalmente, al utilizar el patrón de descubrimiento de los servicios por parte del servidor, es fundamental tener una infraestructura que, iniciado el servicio, se pueda acceder a él automáticamente mediante su ruta. Para esto, en este caso, se utilizó una configuración de Consul como descubridor de servicios y Fabio como balanceador de carga.

Además, es importante generar principios y prácticas para el uso de microservicios, sobre todo cuando el trabajo es realizado por múltiples equipos. Así se genera uniformidad entre los servicios y se evita problemas de comunicación entre estos. Ayuda para esto generar una plantilla, ya que funciona como excelente documentación y genera reuso importante en elementos esenciales para todos los servicios.

El uso del monitoreo de salud de Consul nos permite automatizar la escalabilidad del sistema.

Respecto a la puesta en producción del sistema, es bastante costoso y tedioso, haciendo las pruebas más lenta. Esto debe ser solventado utilizando herramientas de continuous delivery o continuous integration, como las que provee GitLab [27] o Spinnaker [5].

7.1. Trabajo futuro

7.1.1. Pruebas empíricas de mantenibilidad

Para cuantificar el impacto en mantenibilidad de la migración de arquitectura, sería importante realizar pruebas empíricas de su impacto y deducir la curva de aprendizaje.

7.1.2. Integración continua

Utilizar integración continua y contenedores para agilizar el proceso de desarrollo y puesta en producción.

7.1.3. Carga en paralelo desde la interfaz

En este momento las llamadas a los diversos microservicios se hacen de forma secuencial, ya que el BFF y la interfaz están acoplados. Podría ser una fuente de mejora de desempeño que la carga se realice en paralelo, pero esto implica agregar una fuente de complejidad, ya que hay que manejar concurrencia. Esto se podría implementar con algún *framework* Javascript como React o AngularJS, o mediante las funcionalidades asíncronas de Python 3.5.

7.1.4. Estrategias para máquinas de aprendizaje

Estudiar e implementar formas más inteligentes de manejar el uso de máquinas de aprendizaje en microservicios; estrategias para su réplica y procesamiento en múltiples máquinas para poder distribuir la carga.

Bibliografía

[1] *Eureka at a glance*, 2014. https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance, visitado el 2016-05-31.

[2] *Apache CouchDB*, 2015. http://couchdb.apache.org/, visitado el 2016-05-31.

[3] *Django REST framework*, 2015. http://www.django-rest-framework.org/, visitado el 2016-05-31.

[4] *Flask-RESTful*, 2015. http://flask-restful-cn.readthedocs.io/en/0.3.4/, visitado el 2016-05-31.

[5] *Global Continuous Delivery with Spinnaker*, 2015. http://techblog.netflix.com/2015/11/global-continuous-delivery-with.html, visitado el 2016-05-31.

[6] *LoopBack*, 2015. http://loopback.io/, visitado el 2016-05-31.

[7] *Otto*, 2015. https://www.hashicorp.com/blog/otto.html, visitado el 2016-05-31.

[8] *Using Rails for API-only Applications*, 2015. http://edgeguides.rubyonrails.org/api_app.html, visitado el 2016-05-31.

[9] *Apache Mesos*, 2016. http://mesos.apache.org/, visitado el 2016-05-31.

[10] *AWS — Elastic Load Balancing*, 2016. https://aws.amazon.com/es/elasticloadbalancing/, visitado el 2016-05-31.

[11] *Kubernetes - Accelerate Your Delivery*, 2016. http://kubernetes.io/, visitado el 2016-05-31.

[12] *Ribbon*, 2016. https://github.com/Netflix/ribbon, visitado el 2016-05-31.

[13] AB, Ericsson: *Erlang – Modules*, 2016. http://erlang.org/doc/reference_manual/modules.html, visitado el 2016-06-31.

[14] Amaral, Marcelo, Jorda Polo, David Carrera, Iqbal Mohomed, Merve Unuvar y Malgorzata Steinder: *Performance Evaluation of Microservices Architectures using Containers*. En *Network Computing and Applications (NCA), 2015 IEEE 14th International Symposium on*, páginas 27–34. IEEE, 2015.

[15] Apache: *Apache ZooKeeper*, 2016. https://zookeeper.apache.org/, visitado el 2016-05-31.

[16] Bronchain, Thibault: *How Hypernetes Brings Multi-tenancy to Microservice Architectures*, 2015. http://thenewstack.io/hypernetes-brings-multi-tenancy-microservices/, visitado el 2016-05-31.

[17] Brookins, Andrew: *Teardown: Refactoring Search from Django App to Microservice*, 2015. https://www.safaribooksonline.com/blog/2015/11/03/teardown-refactoring-search-from-monolith-to-microservice/, visitado el 2016-05-31.

[18] Calçado, Phil: *How we ended up with microservices*, 2015. http://philcalcado.com/2015/09/08/how_we_ended_up_with_microservices.html, visitado el 2016-05-31.

[19] Ciuffoletti, Augusto: *Automated deployment of a microservice-based monitoring infrastructure*. Procedia Computer Science, 68:163–172, 2015.

[20] Coburn Watson, Scott Emmons y Brendan Gregg: *A Microscope on Microservices*, 2015. http://techblog.netflix.com/2015/02/a-microscope-on-microservices.html, visitado el 2016-05-31.

[21] CoreOS: *coreos/etcd: Distributed reliable key-value store for the most critical data of a distributed system*, 2016. https://github.com/coreos/etcd, visitado el 2016-05-31.

[22] Dowse, Malcolm: *Erlang and First-Person Shooters*, 2012. http://web.archive.org/web/20120913070310/http://www.erlang-factory.com/upload/presentations/395/ErlangandFirst-PersonShooters.pdf, visitado el 2016-06-31.

[23] eBay: *Fabio: A fast, modern, zero-conf load balancing HTTP(S) router for deploying microservices managed by consul*, 2016. https://github.com/eBay/fabio, visitado el 2016-05-31.

[24] Eskildsen, Simon Hørup: *Why Docker is Not Yet Succeeding Widely in Production*, 2015. http://sirupsen.com/production-docker, visitado el 2016-05-31.

[25] Fowler, Martin: *Microservices*, 2015. http://martinfowler.com/articles/microservices.html, visitado el 2016-05-31.

[26] Fowler, Martin: *Testing Strategies in a Microservice Architecture*, 2015. http://martinfowler.com/articles/microservice-testing/, visitado el 2016-05-31.

[27] GitLab: *GitLab Continuous Integration*, 2016. https://about.gitlab.com/gitlab-ci/, visitado el 2016-05-31.

[28] Goldberg, Yoav y Omer Levy: *word2vec explained: Deriving mikolov et al.'s negative-sampling word-embedding method*. arXiv preprint arXiv:1402.3722, 2014.

[29] Hanson, Jeff: *Modularity in Java 9: Stacking up with Project Jigsaw, Penrose, and OSGi*, 2016. http://www.javaworld.com/article/2878952/java-platform/modularity-in-java-9.html, visitado el 2016-06-31.

[30] HashiCorp: *Consul by HashiCorp*, 2016. https://www.consul.io/, visitado el 2016-05-31.

[31] Hoff, Todd: *How League Of Legends Scaled Chat To 70 Million Players - It Takes Lots Of Minions*, 2014. http://highscalability.com/blog/2014/10/13/how-league-of-legends-scaled-chat-to-70-million-players-it-t.html, visitado el 2016-06-31.

[32] Humble, Charles: *Twitter Shifting More Code to JVM, Citing Performance and Encapsulation As Primary Drivers*, 2011. https://www.infoq.com/articles/twitter-java-use, visitado el 2016-05-31.

[33] Inc, Mitoc Group: *DEEP Framework*, 2016. https://github.com/MitocGroup/deep-framework, visitado el 2016-05-31.

[34] Liu, Weixiang, Nanning Zheng y Qubo You: *Nonnegative matrix factorization and its applications in pattern recognition*. Chinese Science Bulletin, 51(1):7–18, 2006.

[35] Mansoor, Umer: *Cinema 3 - (Extremely Simplified) Example of Microservices in Python*, 2015. https://github.com/umermansoor/microservices, visitado el 2016-05-31.

[36] Martin, Robert Cecil: *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[37] McIlroy, M. D., E. N. Pinson y B. A. Tague: *UNIX Time-Sharing System: Foreword*. Bell System Technical Journal, 57(6):1899–1904, 1978, ISSN 1538-7305. http://dx.doi.org/10.1002/j.1538-7305.1978.tb02135.x.

[38] Microsoft: *Singularity - Microsoft*, 2016. https://www.microsoft.com/en-us/research/project/singularity/, visitado el 2016-06-31.

[39] Mizerany, Blake y Keith Rarick: *Doozer*, 2016. https://github.com/ha/doozerd, visitado el 2016-05-31.

[40] Mortier, Richard: *Unikernels, meet Docker!*, 2015. http://unikernel.org/blog/2015/unikernels-meet-docker, visitado el 2016-05-31.

[41] Nauts.io: *Consul HTTP Router*, 2015. https://github.com/nautsio/consul-http-router, visitado el 2016-05-31.

[42] Newman, Sam: *Building Microservices*. O'Reilly Media, Inc., 2015.

[43] Newman, Sam: *Pattern: Backends For Frontends*, 2015. http://samnewman.io/patterns/architectural/bff/, visitado el 2016-05-31.

[44] O'Connel, Ainsley: *Inside Erlang, The Rare Programming Language Behind WhatsApp's Success*, 2014. http://www.fastcompany.com/3026758/inside-erlang-the-rare-programming-language-behind-whatsapps-success, visitado el 2016-06-31.

[45] Plataformatec: *Elixir*, 2016. http://elixir-lang.org/, visitado el 2016-06-31.

[46] Python: *Pickle*, 2016. https://docs.python.org/2/library/pickle.html, visitado el 2016-05-31.

[47] Rahman, Mazedur y Jerry Gao: *A Reusable Automated Acceptance Testing Architecture for Microservices in Behavior-Driven Development*. En *Service-Oriented System Engineering (SOSE), 2015 IEEE Symposium on*, páginas 321–325. IEEE, 2015.

[48] Rajagopalan, Shriram y Hani Jamjoom: *App–Bisect: Autonomous Healing for Microservice-Based Apps*. En *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.

[49] Ranieri, Cosimo: *Teardown: Refactoring Search from Django App to Microservice*, 2015. http://techblog.shutl.com/2015/10/rebuilding-shutl-com-using-microservices-and-layer-7-load-balancing-2/, visitado el 2016-05-31.

[50] Richards, Mark: *Software architecture patterns, Microkernel Architecture*, 2015. https://www.oreilly.com/ideas/software-architecture-patterns/page/4/microkernel-architecture, visitado el 2016-06-31.

[51] Richardson, Chris: *Microservices: Decomposing Applications for Deployability and Scalability*, 2014. https://www.infoq.com/articles/microservices-intro, visitado el 2016-05-31.

[52] Richardson, Chris: *Service Discovery in a Microservices Architecture*, 2015. https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/, visitado el 2016-05-31.

[53] Schuller, Sinclair: *Why Docker is Not Yet Succeeding Widely in Production*, 2015. https://gigaom.com/2015/10/09/why-unikernels-will-kill-containers-in-five-years/, visitado el 2016-05-31.

[54] Sheta, Dharmesh: *Microkernel Architecture Pattern & Applying it to Software Systems*, 2009. http://viralpatel.net/blogs/microkernel-architecture-pattern-apply-software-systems/, visitado el 2016-06-31.

[55] Shoup, Randy y Dan Pritchett: *The eBay Architecture*, 2006. http://www.addsimplicity.com/downloads/eBaySDForum2006-11-29.pdf, visitado el 2016-05-31.

[56] Stubbs, Joe, Walter Moreira y Rion Dooley: *Distributed Systems of Microservices Using Docker and Serfnode*. En *Science Gateways (IWSG), 2015 7th International Workshop on*, páginas 34–39. IEEE, 2015.

[57] Tarreau, Willy: *HAproxy - The Reliable, High Performance TCP/HTTP Load Balancer*, 2016. http://www.haproxy.org/, visitado el 2016-05-31.

[58] Toffetti, Giovanni, Sandro Brunner, Martin Blöchlinger, Florian Dudouet y Andrew Edmonds: *An architecture for self-managing microservices*. En *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*, páginas 19–24. ACM, 2015.

[59] Viennot, Nicolas, Mathias Lécuyer, Jonathan Bell, Roxana Geambasu y Jason Nieh: *Synapse: a microservices architecture for heterogeneous-database web applications*. En *Proceedings of the Tenth European Conference on Computer Systems*, página 21. ACM, 2015.

[60] Wu, Ho Chung, Robert Wing Pong Luk, Kam Fai Wong y Kui Lam Kwok: *Interpreting tf-idf term weights as making relevance decisions*. ACM Transactions on Information Systems (TOIS), 26(3):13, 2008.

Anexos

Anexo A: Capturas de pantalla

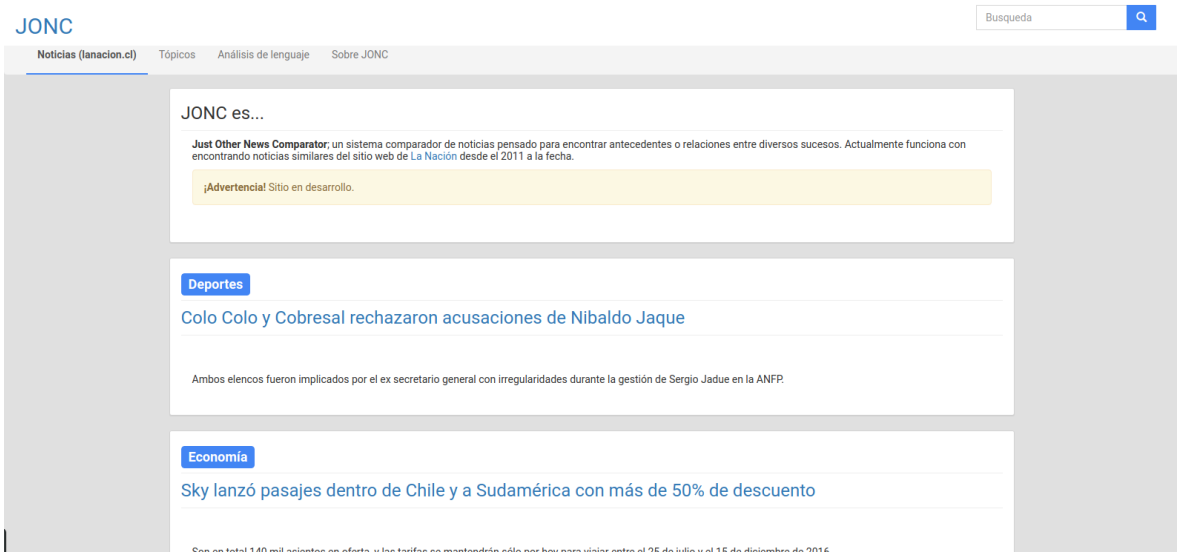


Figura 7.1: Captura de pantalla del inicio del sistema.

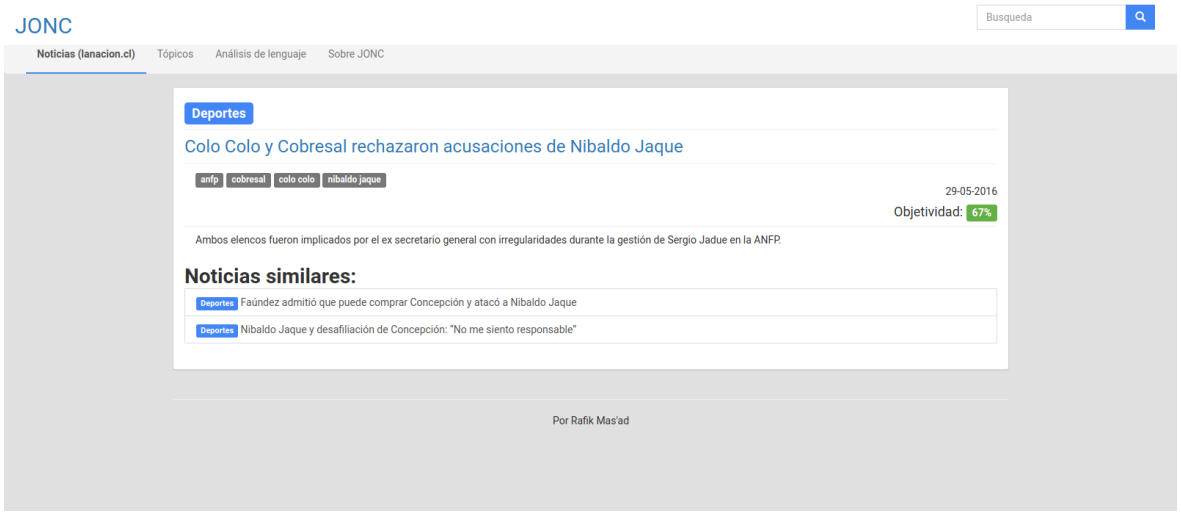


Figura 7.2: Captura de pantalla de la vista de una noticia.

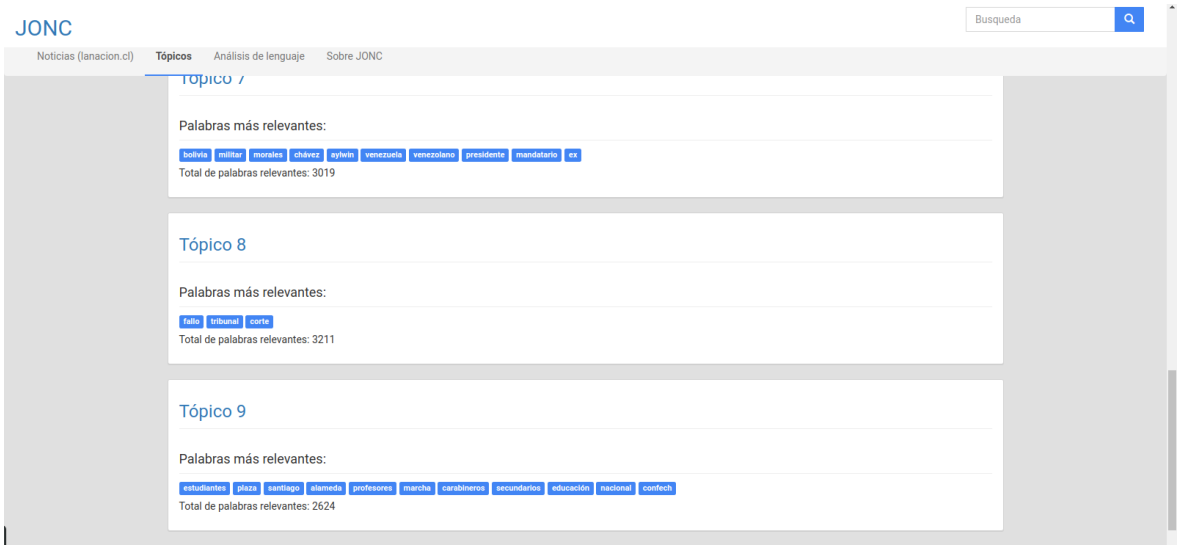


Figura 7.3: Captura de pantalla de la sección de tópicos.

Figura 7.4: Captura de pantalla de la vista de un tópico.

Figura 7.5: Captura de pantalla de la sección de análisis de lenguaje.

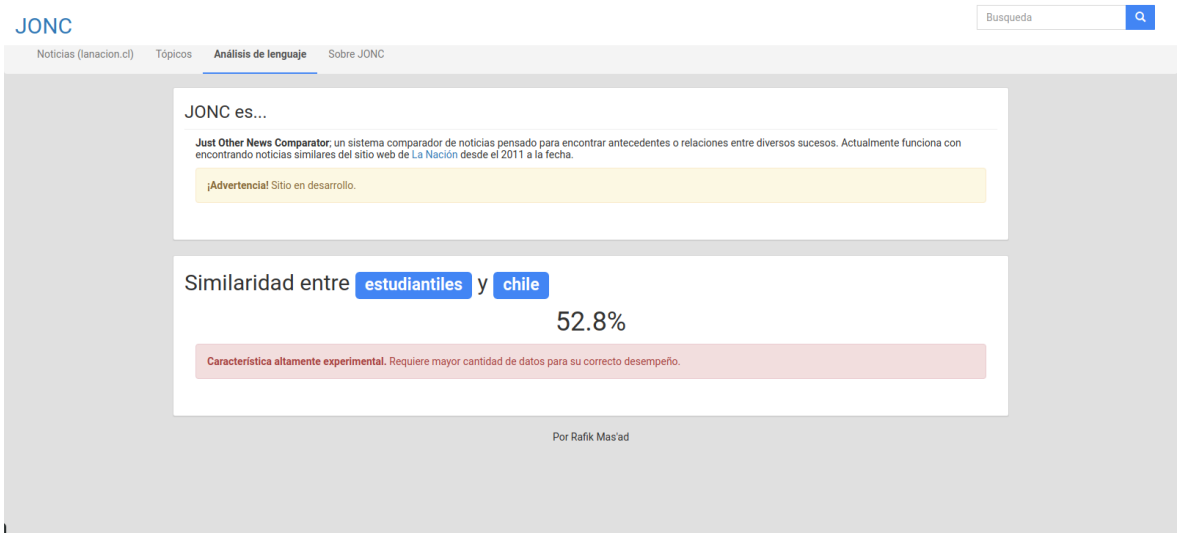


Figura 7.6: Captura de pantalla de comparación de similaridad entre dos palabras en la sección de análisis de lenguaje.

OK

Figura 7.7: Captura de pantalla de la vista que utiliza Consul para medir la salud del servicio.