UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

MEJORA A SISTEMA DE DETECCIÓN DE SISMOS VÍA TWITTER, TWICALLI

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

JHON MAYCOL BRYAN SALGADO URIBE

PROFESORA GUÍA:
BÁRBARA POBLETE LABRA
PROFESORA CO-GUÍA:
JAZMINE MALDONADO FLORES

MIEMBROS DE LA COMISIÓN:
EDUARDO GODOY VEGA
FEDERICO OLMEDO BERÓN

Trabajo financiado por FONDECYT 1191604

SANTIAGO DE CHILE
2021

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN
POR: JHON MAYCOL BRYAN SALGADO URIBE
FECHA: 2021
PROF. GUÍA: BÁRBARA POBLETE LABRA

MEJORA A SISTEMA DE DETECCIÓN DE SISMOS VÍA TWITTER, TWICALLI

Twicalli es un sistema de detección de sismos que funciona utilizando la información publicada por usuarios en la plataforma social Twitter. Permite visualizar de manera interactiva en su página web la frecuencia con la que se publican mensajes relacionados a sismos, mostrando su localización en mapas e incluyendo una muestra de los mensajes, en tiempo cercano al real. Está en funcionamiento desde el 2018 y es utilizado por el Centro Sismológico Nacional de Chile y la Oficina Nacional de Emergencia del Ministerio del Interior, como complemento de los sistemas de medición sismológica. Aporta datos que permiten a los expertos comprender la extensión del impacto de un sismo y las potenciales repercusiones en la población.

Para conseguir los datos utiliza un módulo llamada detector de ráfagas, el cual obtiene mensajes en tiempo real directamente desde Twitter. Aplica filtros para separar aquellos mensajes que contienen palabras relacionadas a terremotos, procesándolos para obtener más información, como ubicaciones mencionadas, y finalmente los almacena en una base de datos desde donde los lee la aplicación web. El detector debe su nombre a que, a través del análisis de estadísticas y un modelo probabilístico, es capaz de detectar incrementos infrecuentes en el arribo de mensajes que contienen información relevante por unidad de tiempo. A estos incrementos infrecuentes se les llama ráfagas.

Debido a sus capacidades, el detector tiene un gran potencial para ser utilizado en análisis y estudios de todo tipo de eventos, como pandemias, actividad volcánica, eventos deportivos, etc. Sin embargo, debido a lo complejo que resulta el comprender su código fuente, modificarlo e incluso ejecutarlo, estas aplicaciones no han sido posibles.

En esta memoria se diseña e implementa una nueva versión del detector, que tiene todas las características deseables del original, pero carece de los problemas de mantenibilidad y extensibilidad.

El resultado es un nuevo detector más simple y conciso, que contiene todas las funcionalidades requeridas en tan solo un 5 % del tamaño del detector original (tomando en cuenta la cantidad de líneas de código). Además de esto, con una pequeña configuración puede adaptarse para detectar cualquier tipo de evento, incluyendo los sismos, por lo que es capaz de proveer los datos que necesita Twicalli para continuar con su funcionamiento.

Además de lo mencionado, se crean dos bibliotecas gratuitas y de código abierto para el lenguaje de programación Go, con funcionalidades que no son específicas al detector de eventos y por tanto pueden extraerse de la aplicación principal. La primera contiene métodos para el procesamiento de texto, incluyendo la posibilidad de filtrar palabras vacías en más de 28 idiomas. La segunda contiene métodos para detectar nombres de países dentro de textos en más de 120 lenguajes, entregando información como código y coordenadas del país detectado.

Dedicado a todas las personas que me han tendido una mano en mi camino.

Agradecimientos

Gracias a mis padres por haberme apoyado siempre y por todos los sacrificios que hicieron por mí. Espero ser capaz de recompensarles todo algún día.

Gracias a mis amigos por siempre estar ahí para darme una mano cuando la necesito, para compartir mis triunfos y para incentivarme a mejorar.

Gracias a mis compañeros de trabajo, de los que he aprendido mucho. Especialmente Felipe, que además de mi amigo ha sido mi mayor mentor.

Finalmente gracias a Catalina, mi pareja, que ha estado conmigo durante 3 años compartiendo penas y alegrías. Le debo mucho de lo que soy en este momento.

TABLA DE CONTENIDO

1. Introducción **1**
1.1. Contexto . . . 1
1.2. Problema . . . 1
1.3. Objetivos . . . 3
1.3.1. Objetivo general . . . 3
1.3.2. Objetivos específicos . . . 3
1.4. Plan de Trabajo . . . 3
1.5. Estructura de la memoria . . . 4

2. Antecedentes Previos **6**
2.1. Conceptos . . . 6
2.1.1. JSON . . . 6
2.1.2. BSON . . . 6
2.2. Tecnologías . . . 6
2.2.1. Git . . . 6
2.2.2. Github . . . 7
2.2.3. MongoDB . . . 7
2.2.4. Go . . . 7
2.2.5. Docker . . . 8
2.2.6. Docker Compose . . . 9
2.3. API Twitter . . . 9
2.4. Twicalli . . . 9

3. Situación actual **11**
3.1. Detector . . . 11
3.1.1. Agente Oyente . . . 11
3.1.2. Agente Almacenador . . . 12
3.1.3. Agente Empaquetador . . . 12
3.1.4. Agente Detector de Ráfagas . . . 12
3.1.5. Agente Descriptor . . . 12
3.1.6. Agente N-gramas . . . 13
3.1.7. Agente Oyente DB . . . 13
3.1.8. Configuración . . . 13
3.2. Algoritmo de detección . . . 13
3.3. Problemas del detector . . . 15

4. Solución **18**
4.1. Análisis inicial 18
4.2. Diseño del Nuevo Detector 19
4.2.1. Estructura 19
4.2.2. Tecnologías 20
4.3. Implementación 22
4.3.1. Procesador de texto 23
4.3.2. Detector de países 24
4.3.3. Nuevo detector de eventos 26

5. Validaciones **34**
5.1. Operación con Twicalli 35
5.1.1. Completitud de los datos 35
5.1.2. Velocidad 35
5.1.3. Volumen de los datos 36
5.2. Adaptabilidad a nuevos casos de uso 37
5.3. Mantenibilidad y extensibilidad 38
5.3.1. Ratio de comentarios por líneas de código 40
5.3.2. Reproducibilidad del ambiente de desarrollo 41

6. Conclusiones y trabajo futuro **43**

Bibliografía **46**

Appendices **47**

A. Twicalli **48**

B. Documentos que se almacenan en la base de datos. **49**

Índice de Tablas

5.1. Tiempo de procesamiento de tweets recibidos en un minuto 36
5.2. Cantidad de tweets relevantes almacenados en un día por cada detector. . . . 37
5.3. Ratio de comentarios por líneas de código para ambos detectores. 41

Índice de Ilustraciones

2.1. Vista principal de la página web de Twicalli 10

3.1. Gráfico que muestra cantidad de tweets relacionados a sismos por unidad de tiempo en página web de Twicalli. 17

4.1. Tiempo de detección de 2500 tweets dependiendo del largo del prefijo que se utiliza para descartar. 26

4.2. Arquitectura con contenedores Docker 33

5.1. Tweets de gol por minuto en partido de fútbol de Chile contra Uruguay . . . 39

Capítulo 1

Introducción

1.1. Contexto

Twicalli es una plataforma web[1] que permite visualizar la frecuencia con la que usuarios en Twitter[2] publican contenido relacionado a sismos, junto con la fecha, hora y ubicación desde la cual provienen las publicaciones (también llamadas *tweets*), en tiempo cercano al real. Por lo tanto, si bien no es una fuente oficial, sirve como complemento de los sistemas de medición sismológica, aportando datos como la extensión del impacto de un sismo y las potenciales repercusiones en la población. Está en funcionamiento desde el 2018 y es utilizado por el Centro Sismológico Nacional de Chile y la Oficina Nacional de Emergencia del Ministerio del Interior (ONEMI).

Para obtener los tweets y detectar los sismos a través de Twitter se utiliza una aplicación llamada Burst Detector (desde ahora detector de ráfagas o simplemente detector). Esta aplicación fue concebida inicialmente con propósitos de investigación, pues se buscaba desarrollar un sistema robusto de detección de eventos a través de Twitter que estuviera al nivel del estado del arte, quedando sus resultados registrados en un paper [3]. Se encarga principalmente de obtener el flujo de datos desde Twitter, procesarlo para obtener un poco más de información y detectar eventos, los cuales se definen como aumentos inusuales en la frecuencia de aparición de determinadas palabras en las publicaciones de los usuarios, en ventanas discretas de tiempo. En el caso particular de Twicalli, las palabras de interés son aquellas relacionadas con sismos. Luego, el detector almacena la información sobre los eventos detectados en una base de datos, la cual es utilizada por Twicalli para mostrar la información en su sitio web.

1.2. Problema

Los principales problemas se encuentran en el detector y vienen dados por el hecho de que su objetivo principal era apoyar una investigación y no exclusivamente ser el proveedor de

[1] Sitio web: http://twicalli.cl/

[2] Twitter es una plataforma y red social que permite a usuarios de todo el mundo crea su perfil y publicar mensajes de texto cortos.

datos de Twicalli u otras plataformas.

Entre los inconvenientes que presenta se pueden mencionar los siguientes:

- Posee varias características que no tienen uso práctico. Entre ellas está la implementación de dos algoritmos de detección de eventos desarrollados en otras partes del mundo, que solo se utilizaron para comparar la efectividad del detector al momento de escribir el paper. También posee varias características experimentales que al no dar buenos resultados para la investigación o bien no están en uso, pero su código sigue presente en la aplicación, o están en funcionamiento, consumiendo recursos, aunque se tenga evidencia empírica de que no representan una mejora para el detector. Un ejemplo de esto es un análisis de sentimientos que realiza sobre el texto de los tweets, que en la práctica no impactó en la precisión de las detecciones.

- Tiene muchas funcionalidades que resultan interesantes por separado, por ejemplo la recopilación de tweets, la capacidad de detectar países dentro de un texto, el procesamiento de éste para poder analizar cada palabra por separado y calcular estadísticas, etc. (para la lista detallada ver la sección "Situación Actual"). Sin embargo posee una arquitectura monolítica, es decir, sus funcionalidades están completamente acopladas en una sola aplicación haciendo imposible la reutilización de alguna de sus partes para otra investigación o proyecto.

- La curva de aprendizaje para comprender el código fuente es elevada. Tanto los nombres de variables como los comentarios usan lenguaje técnico muy específico en el contexto de la investigación. Además de eso, requiere una extensa configuración inicial y algunas de sus dependencias (bibliotecas externas), requieren pedir expresa autorización a los autores para poder utilizarlas, por lo que es difícil reproducir el detector en un ambiente de desarrollo local.

- La detección que lleva a cabo la aplicación no se alinea a la perfección con el objetivo de Twicalli, pues realmente lo que hace es analizar las estadísticas de cada una de las palabras que aparecen en las publicaciones. Revisa entre éstas, cuáles han tenido un incremento en su frecuencia en un intervalo de tiempo determinado, funcionando más como un detector de tendencias que un detector de un evento en específico como requiere Twicalli.

Estos problemas impactan directamente en la mantenibilidad y extensibilidad de la aplicación. Prueba de ello es que desde su creación en el 2018, el detector ha despertado interés en varios investigadores e instituciones, ya sea porque les interesa reutilizar algunas de las características mencionadas en el segundo punto o porque desean detectar otro tipo de eventos. Sin embargo, hasta la fecha de escritura de esta memoria (2021), esos proyectos no se han logrado concretar debido a lo complejo que resulta modificar la aplicación. Un caso real que se aborda en la sección de validación es el Servicio Hidrográfico y Oceanográfico de la Armada de Chile (SHOA), el cual requiere un detector de eventos relacionados a fenómenos marítimos.

1.3. Objetivos

1.3.1. Objetivo general

El objetivo general de esta memoria es hacer un nuevo detector desde cero, que sea mantenible, extensible y fácilmente reutilizable. Debe contar una estructura que permita también la utilización de manera independiente de ciertas funcionalidades que sean útiles en otros proyectos. También debe ser capaz de reemplazar al detector anterior como proveedor de datos de Twicalli, a pesar de que realizar el intercambio esta fuera del alcance de esta memoria, debido a los cambios que se deben realizar en la aplicación web.

1.3.2. Objetivos específicos

1. Comprender por completo el detector actual, para saber qué funcionalidades son indispensables para el funcionamiento de Twicalli, cuáles fueron solamente útiles para la investigación y cuáles pueden ser útiles para otros proyectos. En resumen, tener información suficiente para decidir qué mantener del anterior detector y qué no.
2. Definir el conjunto de tecnologías con las que se implementará el nuevo proyecto y el diseño que debe tener la estructura de éste para impulsar el objetivo general. Es decir, una estructura que permita la reutilización de componentes, separando adecuadamente las funcionalidades y un conjunto de tecnologías que facilite el ingreso de nuevos miembros a la mantención o modificación de la aplicación.
3. Generar igual o mayor cantidad de datos utilizando el nuevo detector (tweets guardados), que contengan su información relevante por completo (fecha, ubicación, lenguaje, etc) y que el procesamiento siga funcionando en tiempo cercano al real. Cumpliendo este objetivo se garantiza que el nuevo detector podrá actuar como proveedor de datos para Twicalli.
4. Adaptar el detector a un nuevo caso de uso, con un mínimo de esfuerzo. En particular, utilizar el detector para recopilar tweets con palabras de interés provistas por el SHOA.
5. Realizar una comparación de mantenibilidad y extensibilidad, en la cual el nuevo detector resulte mejor en 2 aspectos: ratio de comentarios por líneas de código y reproducibilidad del ambiente de desarrollo, es decir, qué tan complejo es ejecutar la aplicación en un ambiente local para un nuevo colaborador, basándose en la cantidad de pasos que requiere realizar.

1.4. Plan de Trabajo

1. Para comprender el detector actual:
    - Llevar a cabo un análisis completo del código de la aplicación, documentando cada una de sus partes.
    - Analizar modelo de datos, para ver qué información es relevante.
    - Reunirse con el desarrollador del detector para resolver dudas.

2. Detección de procesos reutilizables:

- Identificar de entre las funcionalidades que se implementarán en el nuevo detector, aquellas que no son necesariamente particulares a Twicalli y por tanto se pueden generalizar.
- Discutir y validar la utilidad que podría tener el separar estos procesos o funcionalidades con investigadores e instituciones interesadas. En particular con representantes del Servicio Hidrográfico y Oceanográfico de la Armada de Chile

3. Diseño de la solución:
    - Investigar sobre arquitecturas orientadas a microservicios.
    - Decidir qué tecnologías, protocolos de comunicación, lenguajes y motor de base de datos se utilizará.
    - Investigar sobre la implementación de módulos o bibliotecas para el lenguaje elegido.

4. Implementación:
    - Implementar el nuevo detector a partir de las conclusiones obtenidas en los puntos anteriores.
    - Mejorar la calidad y mantenibilidad del código mediante la creación de tests unitarios y documentación.
    - Añadir al proyecto el uso de Git para el versionamiento del código y Docker[3] para crear el ambiente de desarrollo.
5. Validación:
    - Ejecutar el nuevo detector para recopilar tweets y estadísticas por varios días y comparar los cambios en la calidad, cantidad y velocidad de generación de datos con respecto al detector en uso actualmente.
    - Validar la factibilidad de adaptar el detector al caso de uso del SHOA, utilizándolo para recopilar información relevante para esta organización.
    - Evaluar la mantenibilidad y extensibilidad del código en base a métricas como el ratio de comentarios y la reproducibilidad del ambiente de desarrollo.

1.5. Estructura de la memoria

En el capítulo 2 se describen los conceptos, herramientas y softwares generales que son necesarios para entender la memoria.

En el capítulo 3 se describe el estado actual de Twicalli y del detector, explicando su estructura, cada una de sus funcionalidades y profundizando en los problemas que presenta. Este capítulo corresponde al resultado de haber ejecutado los pasos 1 y 2 del plan de trabajo, por lo tanto también presenta un análisis sobre las funcionalidades que podrían ser reutilizadas.

[3] Docker es una herramienta que permite automatizar el despliegue de aplicaciones mediante contenedores. Un contenedor es similar a una máquina virtual que tiene instalado todo lo necesario para que la aplicación se pueda ejecutar.

En el capítulo 4 se presenta la solución, es decir, el diseño e implementación del nuevo detector, resultado de aplicar los pasos 3 y 4 del plan de trabajo. En este capítulo se detalla el conjunto de tecnologías utilizadas y la motivación tras su elección, el diseño de la nueva aplicación, mencionando también algunas ideas que fueron descartadas en el proceso, y describiendo la implementación.

En el capítulo 5 se encuentran las validaciones, describiendo las métricas y procedimientos que se utilizan para evaluar si el nuevo detector cumple los requisitos establecidos en la sección de objetivos y detallando el resultado de aplicarlos. Estas validaciones apuntan mayormente a comparar ciertos aspectos del detector desarrollado en esta memoria con el detector actualmente en funcionamiento.

Finalmente en el capítulo 6 se encuentra un resumen del trabajo realizado, analizando qué objetivos se cumplieron, cuáles no, los aprendizajes obtenidos y la propuesta de trabajos futuros que mejorarían la solución.

Capítulo 2

Antecedentes Previos

En este capítulo se describen los conceptos, herramientas y softwares generales que son necesarios para entender el trabajo realizado y algunas decisiones de diseño.

2.1. Conceptos

2.1.1. JSON

JSON[1], acrónimo para *JavaScript Object Notation*, es un formato ligero, basado en texto, de intercambio de datos, soportado por la mayoría de lenguajes de programación existentes. Sigue una estructura llave-valor similar a un diccionario, es legible para los seres humanos y es actualmente el estándar para intercambio de datos en la web.

2.1.2. BSON

BSON[2] significa "Binary JSON", es una representación binaria para almacenar datos en formato JSON. Debe su origen a que trabajar con representaciones binarias es mucho más eficiente en tiempo y memoria que trabajar con texto.

2.2. Tecnologías

2.2.1. Git

Git[3], es un sistema de control de versiones. Entre otras funciones, permite crear un historial con distintos estados del código a medida que un proyecto crece. De esa manera, si algún cambio nuevo afecta de manera negativa, se pueda volver rápidamente a una versión anterior que este funcionando.

[1]https://www.json.org/json-en.html
[2]https://www.mongodb.com/json-and-bson
[3]https://git-scm.com/

Además de esto, apoya al trabajo en equipo, pues permite crear ramas o versiones paralelas del código, en las cuales cada colaborador puede trabajar de manera individual para luego combinar sus cambios con la rama o versión principal.

2.2.2. Github

Github[4] es una plataforma donde se pueden almacenar de manera gratuita respaldos de todas las ramas y versiones de un proyecto (mayormente creadas con Git). De esa manera cualquier persona con acceso a internet y los permisos adecuados puede acceder a él, facilitando el trabajo en equipo de forma remota.

2.2.3. MongoDB

MongoDB[5] es una base de datos no relacional (NoSQL). En lugar de tener tablas para estructurar los datos como lo haría un esquema relacional, tiene colecciones de documentos (donde los documentos de una colección serían el equivalente a las filas de una tabla).

Los documentos no siguen una estructura fija, es decir que pueden tener un número variable de campos. Además admiten tipos de datos complejos, como listas y objetos, pudiendo tener incluso otros documentos anidados. Estos documentos se almacenan en formato BSON.

En lugar de SQL, en MongoDB se consulta con una sintaxis similar a JavaScript. Además de eso, al igual que las bases de datos relacionales, MongoDB permite realizar consultas basadas en numerosas condiciones, ordenar y limitar los resultados, indexar para crear consultas más eficientes, agregación,etc

2.2.4. Go

También conocido como Golang[6], Go es un lenguaje de programación de código abierto.

Está basado en C, por lo que tanto su sintaxis como su rendimiento son bastante similares, con la diferencia de que Go implementa un recolector de basura automático, que libera la memoria que no se esté utilizando, mientras que en C eso debe manejarlo el programador. Debido a esto, C suele tener tiempos de ejecución menores mientras que con Go se suelen crear códigos más simples.

Una de las mejores características de Go es que maneja la concurrencia de manera nativa. Para ejecutar un proceso de manera concurrente basta con agregar la palabra reservada `go` antes de la ejecución de una instrucción. Esto hará que la instrucción comience a ejecutarse en una rutina diferente (también llamada *goroutine*), mientras que el código que sigue quedará ejecutándose en la rutina principal.

Go internamente se encarga de manejar la cola de procesos y la asignación de el o los procesadores disponibles a las diferentes rutinas. Además, posee unas estructuras llamadas

[4] https://github.com/
[5] https://www.mongodb.com
[6] https://golang.org/

canales, que hacen la comunicación entre rutinas muy simple, pues con una sola instrucción (`canal <- datos a enviar`), se puede enviar información desde una rutina a otra, siendo la recepción igual de sencilla (`datos a recibir <- canal`).

Uno de los puntos remarcables de Go y su comunidad es que uno de sus focos principales es la legibilidad del código, debido a que promueve la filosofía "clear is better than clever", lo que se traduce como "claro es mejor que inteligente", y hace alusión a que un código que se entienda más fácil es mejor que un código escrito de manera más compleja e inteligente para que realice las mismas acciones en menos líneas.

Una característica de Go que apunta a ese objetivo, es que hay muy pocas formas de abreviar código, lo que se llama comúnmente azúcar sintáctico, por lo que en la mayoría de los casos cada línea representa exactamente una acción (declaración de una variable, inicio de un bucle, revisión de una condición, etc), haciendo que no se necesite un análisis complejo de cada una de ellas para comprender lo que hacen.

2.2.5. Docker

Docker[7] es una herramienta que permite encapsular proyectos y replicar ambientes de desarrollo y producción de manera exacta.

Para comprender su funcionamiento se deben conocer 2 conceptos:

- Imagen: Es una especie de plantilla que contiene el conjunto de elementos y tecnologías necesarias para ejecutar algo en específico. Por ejemplo, una imagen podría contener un sistema operativo Ubuntu, un servidor Apache y una aplicación PHP instalada.
- Contenedor: Es el resultado de ejecutar (o levantar como se conoce comúnmente), una imagen, lo que da como resultado algo similar a una máquina virtual, que tiene instalado todo el contenido de esta, y que usualmente se administra de manera externa, pero también se se puede acceder al interior, explorar sus sistema de archivos, ejecutar comandos, etc. A diferencia de una máquina virtual, los contenedores utilizan el kernel de la máquina en lugar de simular todo, por lo que son más eficientes.

Docker permite crear imágenes personalizadas y/o utilizar imágenes públicas. Debido a que su uso se ha extendido bastante, muchas compañías a cargo de los motores de bases de datos y lenguajes de programación más importantes han creado imágenes oficiales.

Por ejemplo, para ejecutar una base de datos MongoDB en ambiente local, basta con indicar el nombre de la imagen al momento de crear el contenedor (para lo que se utiliza un comando de docker), y luego de la descarga e instalación se tendrá un contenedor que ya está ejecutando la base de datos. Cabe destacar que al hacer esto en ningún momento se instala MongoDB en la máquina local, solamente se instala en la imagen.

Para crear imágenes personalizadas se utiliza un archivo llamado "Dockerfile". En este se especifica una imagen base (usualmente alguna distribución de Linux) y una lista de

[7] https://www.docker.com

instrucciones, que van desde descargar tecnologías hasta copiar archivos desde la máquina local al contenedor.

2.2.6. Docker Compose

Docker Compose[8] es una herramienta de Docker que sirve para definir, administrar y ejecutar aplicaciones que posean varios contenedores. Esto se realiza en a través de un archivo llamado usualmente "docker-compose.yml". En este archivo se definen los contenedores, por ejemplo, se puede crear uno para una base de datos y uno para una aplicación web, la cantidad de instancias de cada uno, las dependencias entre estos, etc.

Al guardar este archivo y ejecutar un solo comando, se construirán e iniciaran ambos contenedores y automáticamente se creará un red para que puedan comunicarse, por lo que la aplicación dentro de un contenedor puede conectarse sin problemas a la base de datos dentro del otro.

2.3. API Twitter

La API de Twitter[9], es un servicio ofrecido por la compañía que permite acceder a una gran cantidad de datos generados por sus millones de usuarios.

Permite entre otras cosas descargar información de usuarios, media, tendencias y tweets.

La descarga de tweets tiene dos modalidades principales. La primera y más común son las operaciones asíncronas, en las cuales se envía una solicitud con ciertos parámetros para filtrar y se recibe como respuesta un conjunto de tweets que cumplan con los criterios de búsqueda.

La segunda modalidad es una conexión persistente en la cual Twitter envía en tiempo real un porcentaje de los tweets que se van generando. Esta segunda modalidad ofrece a su vez dos tipos de datos, la primera opción es pedir una muestra de los tweets a nivel mundial y la segunda una muestra filtrada, ya sea por geolocalización o por palabras claves contenidas en el texto.

2.4. Twicalli

Twicalli está construida con el lenguaje de programación Python y el framework Flask. Posee una conexión con la base de datos MySQL del detector de eventos para obtener los tweets que muestra a través de la página web.

Actualmente está completamente operativa[10] y posee las siguientes características:

- Muestra en tiempo cercano al real los tweets relacionados con sismos y terremotos.

[8] https://docs.docker.com/compose/
[9] https://developer.twitter.com/en/docs/twitter-api/v1
[10] Accesible a través de https://twicalli.cl/aplicacion

- Grafica la cantidad de tweets relevantes por unidad de tiempo, detectando y destacando los aumentos inusuales de estos.
- Muestra un mapa de calor y un mapa con la geolocalización de los tweets que pudieron ser localizados. Ambos mapas son interactivos.
- Permite a los usuarios filtrar los datos por fecha y hora, permitiendo ver tanto información actual como pasada.

Su vista principal con el gráfico y los mapas recién mencionados puede observarse en la figura 2.1.

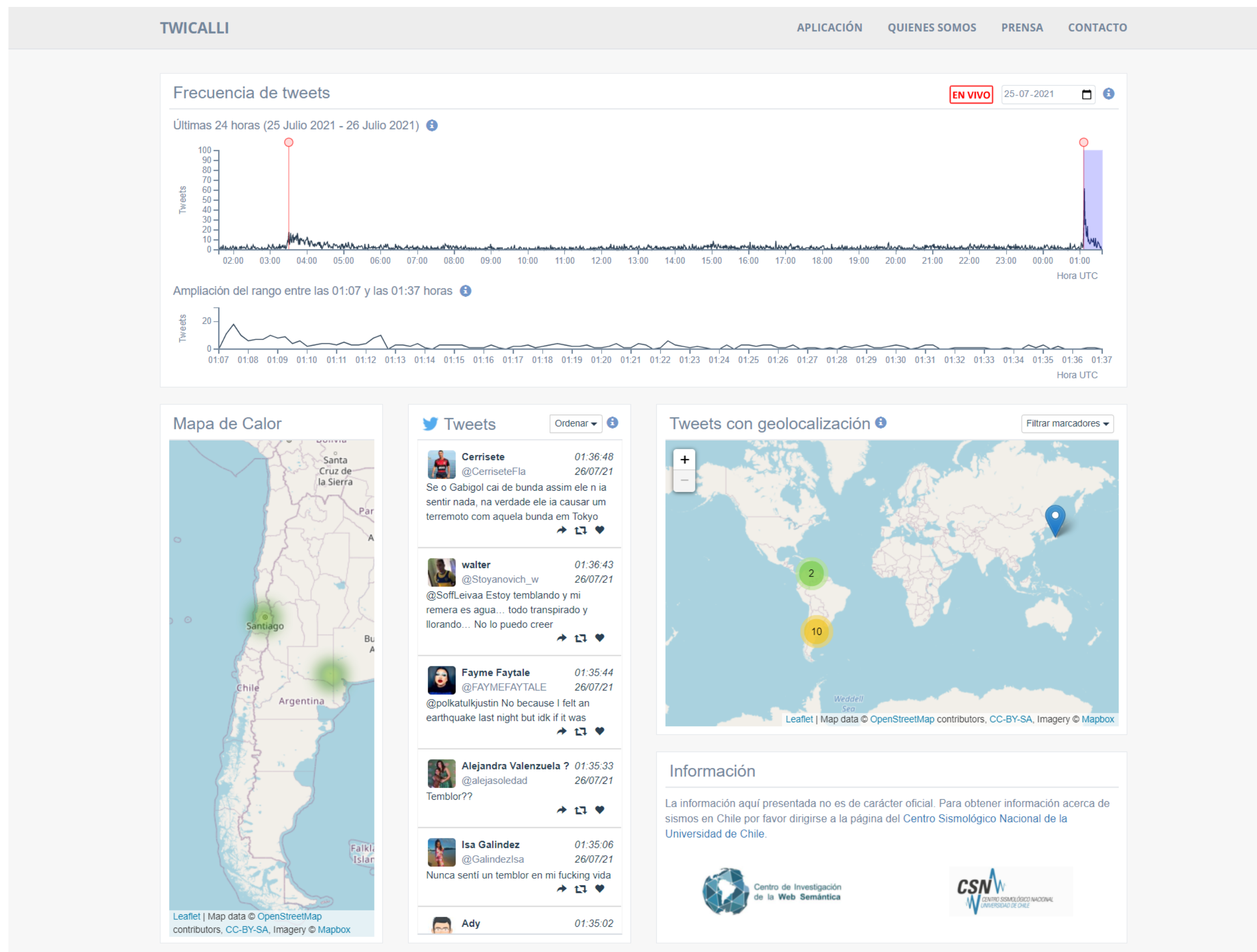

Figura 2.1: Vista principal de la página web de Twicalli

Capítulo 3

Situación actual

3.1. Detector

El detector de ráfagas está hecho con Java como lenguaje de programación y MySQL como base de datos. Su efectividad, específicamente en el caso de los sismos, ya ha sido probada y comparada con el estado de arte, validando que es una solución competitiva, en términos de precisión[1] y recuperación[2], con respecto a las soluciones líderes [3].

Debido a que Java es un lenguaje de programación orientado a objetos, los procesos principales se dividieron en clases llamadas agentes. Estos se instancian en la clase principal dentro de hilos (threads) diferentes y se utilizan colas como medio de comunicación entre ellos.

A continuación se presenta una breve descripción de la función de cada agente.

3.1.1. Agente Oyente

Se conecta a la API de Twitter para descargar tweets y transformarlos en objetos de java. Además, detecta la ubicación de los tweets, revisando si está presente en los metadatos del mismo, en los datos del usuario o si el lugar está mencionado en el texto. Si alguna de estas revisiones tiene éxito almacena la ubicación dentro de los tweets, incluyendo coordenadas geográficas, y los inserta en una cola.

Cabe destacar que Twitter tiene disponibles 2 tipos de endpoints para descargar tweets en tiempo real[3]. Uno llamado flujo de muestra que envía a través de un canal de comunicación un pequeño porcentaje de todos los tweets generados en el mundo y solo permite filtrar los tweets por lenguaje. Por otro lado esta el flujo de filtrado, el cual permite, como indica su nombre, aplicar diferentes tipos de filtros, entre los cuales se encuentra un filtro por palabras

[1]Proporción de eventos identificados como sismos que efectivamente correspondían a sismos

[2]Proporción de sismos identificados correctamente con respecto al total de sismos

[3]Documentación: https://developer.twitter.com/en/docs/twitter-api/v1

claves, que al aplicarse enviará solo tweets que contengan las palabras indicadas a través del filtro, y un filtro de ubicación al que se le puede pasar un set de coordenadas y solo enviará tweets que hayan sido emitidos dentro de éstas.

Existe otro proceso que puede llevar a cabo este agente y es un análisis de sentimiento sobre el texto de los tweets, sin embargo no se desea su continuación pues no impacta de manera significativa en la precisión del detector.

3.1.2. Agente Almacenador

Este agente almacena en una base de datos todos los tweets dejados en la cola por el Agente Oyente y adicionalmente los reenvía a la cola del agente empaquetador.

3.1.3. Agente Empaquetador

Este agente lee continuamente los tweets dejados en la cola por el agente almacenador y con estos lleva a cabo un proceso llamado generación de señales.

Una señal, en este contexto, es cada porción de información relevante presente en el tweet, por lo que además de considerarse como señales cada una de las palabras contenidas en su texto (salvo las definidas como "palabras vacías", las cuales son ignoradas), también se considera señal al idioma del texto, y a las ubicaciones relacionadas con el tweet.

El agente define una ventana de tiempo (fijando un tiempo inicial y uno final), genera las señales para cada tweet cuya fecha de creación este dentro de la ventana y las almacena en una lista. Cada vez que recibe un tweet cuya fecha de creación es posterior al final de la ventana, envía la lista de señales (junto con el tiempo inicial de la ventana, para poder identificarla) a la cola del siguiente agente, y crea una nueva ventana para continuar con el proceso.

3.1.4. Agente Detector de Ráfagas

Este agente recibe los paquetes de señales por ventana enviados por la cola por el agente empaquetador e inserta cada una de las señales como llaves en un diccionario, donde el valor son las estadísticas de la señal en la ventana actual (ventana 2), sus estadísticas en la ventana anterior (ventana 1), y la diferencia entre ambas ventanas. Si la llave ya estaba en el diccionario, solo actualiza sus estadísticas en la ventana actual. Al terminar de insertar las señales en el diccionario, el agente detecta como ráfagas aquellas cuya variación de frecuencia entre ventanas genera valores atípicos, por lo que guarda estas señales en la base de datos (con toda la información que tenían en el diccionario), y las envía mediante una cola al próximo agente.

3.1.5. Agente Descriptor

Este agente recibe mediante la cola el diccionario que contiene solamente las señales que causaron ráfagas (y sus estadísticas), y las describe como eventos, indicando la fecha y hora de de inicio (tiempo inicial de la ventana 1), fecha y hora de término (tiempo final de la

ventana 2), el tiempo medio entre ambos, el tipo de señal (palabra dentro del texto, hashtag, ubicación, lenguaje, etc), la señal como tal y el ranking de la señal dentro de la ventana (de mayor a menor variación en la frecuencia con respecto a la ventana anterior), y lo almacena en la base de datos.

3.1.6. Agente N-gramas

Este agente complementa al empaquetador, agregando n-gramas a las señales generadas. Es decir que además de guardar como señales cada una de las palabras contenidas en el texto del tweet, también almacena conjuntos de palabras que se usan juntas con frecuencia (como "Estados" y "Unidos", por ejemplo). Sin embargo, formar los n-gramas y utilizarlos para la detección era bastante más costoso, por lo que quedó como una característica opcional y actualmente está en desuso.

3.1.7. Agente Oyente DB

Este agente obtiene tweets desde la base de datos. Su objetivo es reemplazar al agente oyente en el proceso cuando se quieran replicar experimentos o llevarlos a cabo sin necesidad de funcionar en tiempo real. Se creó debido a que, como se mencionó anteriormente, en sus inicios el detector se implementó con fines de investigación y era necesario repetir experimentos para ajustar los parámetros iniciales de este (como el tamaño de las ventanas, por ejemplo).

3.1.8. Configuración

El comportamiento del detector es en gran parte manejado a través de un archivo de configuración, pudiendo definir el tipo de filtros que usará el agente Oyente en las consultas a la API de Twitter, los valores que utilizará para filtrar, el tamaño de las ventanas y el tipo de señales que registrará, entre muchas otras opciones. En el caso de Twicalli se descarga una muestra de los tweets localizados en Chile y una muestra de los tweets mundiales que contengan palabras claves relacionadas con terremotos. El tamaño de la ventana es de 5 minutos y las señales a registrar son las palabras en el texto, los sentimientos, el lenguaje del tweet, la ubicación de los usuarios, la ubicación de los tweets y las ubicaciones mencionadas en el texto.

3.2. Algoritmo de detección

Como se mencionó anteriormente, el detector fue desarrollado originalmente para realizar y validar una investigación, cuyo procedimiento y resultados se pueden ver a detalle en su paper[3]. Sin embargo, a continuación se presenta una breve descripción de uno de los puntos más importantes del sistema de detección, el cual corresponde al algoritmo que indica bajo qué condiciones se observa una ráfaga o evento.

El procedimiento es el siguiente:

1. Se define una ventana de tiempo W. Esto no es más que un intervalo entre dos instantes de tiempo, medido usualmente en segundos.

2. Se calcula la cantidad de tweets que se recopilaron dentro de la ventana de tiempo (representada por M_W), y entre los tweets obtenidos, cuantas veces están presentes las palabras y lugares relacionados al evento que se quiere detectar. La aparición de estos términos es llamada simplemente frecuencia y denotada por $freq$.
3. Se aplica logaritmo natural a la frecuencia, pues durante la investigación se observa que este (denotado como $log\text{-}freq$), sigue una distribución normal y por tanto se puede utilizar para realizar un análisis estadístico.
4. Se define la tasa de llegada relativa λ, como la proporción entre el logaritmo de la frecuencia y la cantidad total de tweets de la ventana (ecuación 3.1). Este es uno de los puntos claves, pues permite detectar cambios poco comunes en la frecuencia de los tweets relevantes para el estudio sin que la medición se vea afectada por la variación en el volumen total de tweets. Es muy común que algunas horas del día tengan más actividad que otras, más aún cuando se analiza una sola ubicación, pues la actividad en horas de la madrugada es mucho menor que durante la tarde, y sumado a eso también se tiene el hecho de que la cantidad de tweets aumenta día a día debido al aumento progresivo de usuarios que tiene Twitter. Por lo tanto, sin esta característica, podrían detectarse falsamente eventos solo porque la actividad total en Twitter aumenta en ciertos momentos y con ello también la probabilidad de que aparezcan tweets con términos relevantes.

$$\lambda = \frac{\text{log-freq}}{M_W} \tag{3.1}$$

5. Usando el modelo probabilístico que se acaba de definir, se analiza la variación que tiene el z-score de la distribución log-normal en cada ventana con respecto a las anteriores. Para calcular el z-score se utiliza la ecuación 3.2, donde μ es el promedio y σ la desviación estándar de λ. Este valor representa cuantas desviaciones estándar excede λ su propio promedio y por tanto cuan improbable es.

$$Z(\lambda) = \frac{\lambda - \mu}{\sigma} \tag{3.2}$$

6. Debido a que el flujo de datos de Twitter es bastante variable en el tiempo, se define un modelo adaptativo en el cual las estadísticas necesarias para calcular el z-score (promedio y desviación estándar), se aproximan siguiendo las ecuaciones 3.3 y 3.4, donde λ_n es la tasa de llegada relativa para la ventana n.

$$\mu_n = \frac{\mu_{n-1}(n-1) + \lambda_n}{n} \tag{3.3}$$

$$\sigma_n^2 = \frac{\sigma_{n-1}^2(n-1) + (\lambda_n - \mu_n)^2}{n} \tag{3.4}$$

7. Finalmente se establece que un z-score mayor que 1 es lo suficientemente infrecuente como para ocurrir sin que el incremento dependa de un evento en el mundo (el evento que se quiere detectar), por lo tanto esa será la condición de detección.

Pese a qué modificar o mejorar el algoritmo de detección no forma parte de los objetivos de esta memoria, se menciona pues servirá para comprender de mejor manera algunos de los problemas que enfrenta el detector actual.

3.3. Problemas del detector

Uno de los principales problemas del detector es que los agentes están completamente acoplados, no pueden funcionar de manera independiente. Esto es inconveniente pues varios de los procesos que lleva a cabo el detector serían útiles en otras aplicaciones que utilizan Twitter como fuente de datos. En concreto hay 3 procesos que funcionarían especialmente bien de manera independiente:

- La capacidad de recopilación de tweets. Tener una gran cantidad de tweets almacenados es muy útil a la hora de realizar una nueva investigación, pues pueden servir para validar el funcionamiento de un nuevo sistema o algoritmo, como fue el caso del paper creado a partir del detector, en el cual se utilizaron conjuntos de datos bastante grandes para calcular la precisión de detección y compararla con otros algoritmos en el estado del arte. Si bien el detector actual permite recopilar una gran cantidad de tweets, estos datos están sesgados debido al filtrado que se realiza y pueden no servir para cualquier tipo de investigación. Además, realiza un procesamiento extenso y costoso, que no sería necesario si el objetivo es solo recopilar tweets.
- El módulo de procesamiento de texto. Este módulo permite separar un texto en palabras, las cuales se pueden utilizar como piezas de información individuales que se pueden analizar por separado, filtrando además las palabras vacías, que no aportan información valiosa para la investigación. Esta funcionalidad no sólo es valiosa para analizar tweets sino que también para trabajar con cualquier tipo de texto.
- Detección de países mencionados en un texto. Este proceso es muy útil, pues es común que ni los tweets ni los usuarios tengan información de su ubicación en sus metadatos y con este proceso se agrega una oportunidad más de localizarlos, lo cual puede ser necesario para proyectos que estudien fenómenos en países en concreto. Además, no es una funcionalidad específica para tweets, pudiendo aplicarse a cualquier tipo de texto.

Sumado a esto, el detector tiene algunas características que hacen que la aplicación sea más compleja de lo necesario. Ejemplo concreto es el agente Oyente DB que se utilizó solo en la etapa de investigación para leer conjuntos de datos descargados en lugar de obtenerlos desde Twitter.

Además, están presentes dos características que no dieron buenos resultados. El primero es, como ya se mencionó anteriormente, el análisis de sentimientos, pues no aporta en la precisión de detección pero aumenta tanto la complejidad como los recursos computacionales que se necesitan para ejecutar la aplicación. Por otro lado, la generación de N-gramas esta completamente en desuso, también debido al alto costo computacional y bajo aporte en la eficacia del detector.

Finalmente, como se mencionó en la descripción del agente empaquetador, el sistema contabiliza cada entidad que considera señal (palabras, ubicaciones, usuarios, lenguajes, etc), destacando en cada ventana de tiempo las señales que tuvieron incrementos infrecuentes (utilizando el algoritmo de detección antes descrito). Por lo que a la vez que esta recopilando información de un evento en específico como los sismos, también está calculando las estadísticas para cada palabra, usuario, lugar y lenguaje que aparezca en los tweets, aunque estás no tengan relación alguna con el evento en particular.

Otro inconveniente es que replicar el ambiente de desarrollo es complejo, pues se deben instalar las tecnologías (Java, su kit de desarrollo, drivers, etc) y numerosas bibliotecas, siendo algunas de uso restringido y requiriendo permisos de los autores para poder utilizarse. Además se debe completar un archivo de configuración que contiene 77 variables y para el cual no hay documentación o guía, por lo que no es simple saber qué tipo de dato poner en cada una de ellas.

Todos los problemas mencionados en los puntos anteriores, sumados a que la documentación es insuficiente, causan que sea complejo mantener y extender la aplicación. Por ejemplo, si se quiere llevar a cabo una modificación en una parte específica del proceso, primero se debe llevar a cabo un análisis exhaustivo del detector, tanto para saber dónde modificar, como para no afectar al resto de procesos.

El último problema, que no se resuelve completamente en esta memoria, pero es importante mencionar, es que la detección de sismos que se está aplicando actualmente para Twicalli no sigue por completo el procedimiento desarrollado en el paper y resumido en la sección anterior. Esto debido a que, a diferencia de la investigación, donde se usaron conjuntos de datos extensos para realizar las mediciones, en la práctica los tweets que se reciben en tiempo real no son la totalidad de los tweets generados en Twitter, sino que un bajo porcentaje, por lo que los datos para realizar la detección son pequeños y por tanto muy sensible a cambios.

Por ejemplo, el flujo de datos filtrados que trae solo tweets generados en Chile, suele traer al rededor de 100 tweets en una ventana de 5 minutos, siendo muy frecuente que lleguen de 0 a 2 tweets que posean palabras relevantes para la detección. Esto no ocurre solo porque haya poca actividad, si no que para que Twitter pueda entregar los tweets a través del flujo filtrado por lugar, estos deben poseer localización GPS, lo cual no es muy común. Tomando una muestra, de 88719 tweets relacionados a sismos recopilados en un día, en inglés y en español, solo 3016 poseían localización GPS.

Para solucionar este problema parcialmente, el detector suma los tweets obtenidos desde el flujo filtrado por palabras claves (que solo trae tweets con piezas de información relevantes para la detección), y el flujo filtrado por lugar, para de esta manera aumentar la cantidad de datos. Sin embargo, esto no se alinea con el procedimiento de detección, pues una de las características que posee el algoritmo es que calcula la proporción entre tweets relevantes y la cantidad total de tweets, para que la detección no sea sensible a la fluctuación en la actividad de las personas. No obstante, el flujo de palabras clave entrega tweets provenientes de todo el mundo, por lo tanto el total de tweets de la ventana debiese ser el total mundial, no el total filtrado por ubicación, siendo en este caso particular Chile. Esto causaría, por ejemplo, que en horas donde normalmente en Chile no hay actividad en Twitter (como las horas donde la gente suele dormir), la proporción de tweets con palabras clave, que se mantienen mayormente constantes si no hay presencia de eventos, aumente en comparación con el total, generando falsos positivos.

Debido a esto, lo que está haciendo Twicalli actualmente es utilizar la recopilación y procesamiento de tweets del detector para permitir visualizar la información, sin utilizar directamente las detecciones de este. En lugar de eso posee un gráfico que muestra cantidad de tweets relacionados a sismos por unidad de tiempo, destacando los momentos en los que

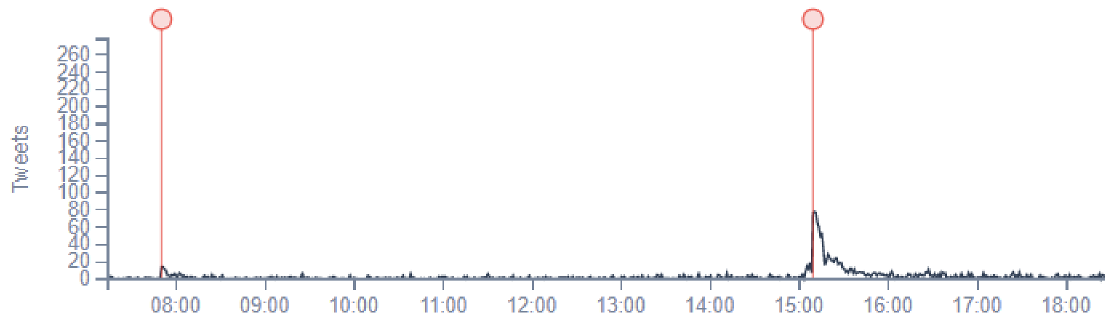

Figura 3.1: Gráfico que muestra cantidad de tweets relacionados a sismos por unidad de tiempo en página web de Twicalli.

estos aumentan de manera anormal, como se muestra en la figura 3.1, permitiendo a los usuarios conocer con una simple inspección visual los momentos en que más probablemente hubo un sismo, de acuerdo a las publicaciones en Twitter, y complementando esta información con los tweets reales emitidos en cada momento y los mapas que muestran de donde provienen.

Capítulo 4

Solución

4.1. Análisis inicial

Para poder diseñar una solución, primero se realiza un estudio exhaustivo del detector actual, el cual está plasmado en gran medida en el capítulo anterior.

Se identifican 3 procesos que se requieren realizar de manera independiente: recopilación de tweets (sin involucrar detección), procesamiento de texto y detección de lugares a partir de un texto.

Con el objetivo de separar debidamente los procesos y dejarlos fácilmente reutilizables por otras posibles aplicaciones, se plantea la idea de una arquitectura de microservicios, donde cada servicio estaría encargado de un proceso que se quiera llevar de manera individual, pudiendo poner incluso a cada uno de los agentes en un microservicio diferente y comunicándolos vía API REST, utilizando protocolo HTTP. Sin embargo, esta idea se descarta pues, pese a que los microservicios comunicados de esta manera pueden reutilizarse fácilmente (bastaría con que una nueva aplicación comience a enviar peticiones a la API de un microservicio para utilizarlo), realmente no hay mucha utilidad en hacerlo, pues las capacidades de estos son bastante específicas al caso de uso de Twicalli (salvo excepciones como el procesador de texto y el detector de países), y su desventaja mas importante es que la comunicación vía HTTP aumentaría la latencia, siendo más lento el mover los tweets entre servicios que tener los procesos centralizados como el detector actual.

Se explora también la posibilidad de utilizar colas o plataformas de mensajería como Rabbitmq y Apache Kafka como medio de comunicación entre microservicios en lugar de APIs, debido a que son mas rápidas para mover grandes cantidades de datos. Sin embargo, integrar nuevas aplicaciones que necesiten reutilizar los servicios sería mas complejo, pues en lugar de enviar una simple petición HTTP, que es una característica que la mayoría de los lenguajes de programación traen integrada y suele ser bastante estándar, una aplicación que quiera conectar con un microservicio tendría que implementar un publicador para poder enviar datos y un suscriptor para leer las respuestas, adaptándose al formato de los mensajes que se hayan adoptado en la cola. Además se debe agregar un sistema en las plataformas de

mensajería para que las respuestas de los microservicios solo lleguen al cliente que los solicitó.

Finalmente la idea de los microservicios se descarta por completo, pues al hablar con los interesados en investigar o utilizar un detector de característica similares (miembros del Instituto Milenio Fundamentos de los Datos y del Servicio Hidrográfico y Oceanográfico de la Armada de Chile), se llega a la conclusión de que no requieren reutilizar características particulares del detector, si no más bien un detector al completo que sea 100 % adaptable a sus necesidades.

Por lo tanto se sigue un modelo de paquetes o bibliotecas individuales que se detalla en la sección de diseño.

4.2. Diseño del Nuevo Detector

4.2.1. Estructura

Con el objetivo de separar debidamente las funcionalidades y permitir la reutilización de estas, se establece una estructura compuesta por una aplicación principal y 2 bibliotecas.

La primera biblioteca a desarrollar es la que agrupa las funcionalidades de procesamiento de texto, permitiendo quitar caracteres especiales, filtrar palabras vacías en más de 20 lenguajes distintos y obtener el set de palabras únicas contenidas en un texto, incluyendo la posibilidad de obtener cuantas veces aparece cada una. Esta biblioteca forma parte de un proyecto completamente independiente, teniendo documentación y versionamiento propios, además de tests unitarios.

La segunda biblioteca corresponde al detector de países, el cual analiza un texto en búsqueda de la mención de un país en más de 100 lenguajes disponibles, y al detectarlo retorna su información (nombre en inglés, código ISO alpha-2 y coordenadas). Adicionalmente, tiene la capacidad de detectar las comunas de Chile. Al igual que la biblioteca anterior, es un proyecto independiente que posee tests unitarios, documentación y versionamiento propios.

La aplicación principal cuenta a su vez con 2 módulos que agrupan sus funcionalidades internas. El primero sirve para manejar la base de datos, estableciendo la conexión, definiendo el modelo de datos e implementando las operaciones de lectura y escritura. El segundo se encarga de establecer la conexión con Twitter, descargar los tweets en tiempo real y procesarlos, para luego enviarlos a ser guardados al primer modulo. Finalmente, se tiene un archivo principal que lee las variables definidas para configurar el comportamiento del detector y utiliza ambos módulos para inicializar las conexiones tanto con Twitter como con la base de datos, e iniciar el procesamiento de tweets. Esta aplicación es completamente configurable, permitiendo decidir qué filtros se deben utilizar para descargar tweets (lenguajes, palabras clave y coordenadas geográficas), y si se desea activar la detección de eventos. De esta manera puede actuar como detector o como un simple recopilador de tweets, según se requiera.

Los beneficios de usar esta estructura es que es mucho más versátil, pues para reutilizar las funcionalidades de las dos bibliotecas basta con importarlas al proyecto donde se requieren

y de esa manera no existe latencia alguna, pues no hay que mover paquetes entre servicios para utilizarlas, es solo código que quedará disponible para su uso en la misma máquina que la aplicación principal. Por otro lado, si se quiere utilizar la arquitectura de microservicios descrita en el análisis inicial, bastaría con levantar un servicio que importe alguna de las bibliotecas y resuelva peticiones con el método de comunicación que se prefiera.

El inconveniente de este modelo es que las bibliotecas solo pueden ser importadas por proyectos que utilicen su mismo lenguaje de programación. Sin embargo, siempre está la posibilidad de levantar un servicio que hable un lenguaje común como JSON, a través de peticiones HTTP provenientes de cualquier aplicación, e internamente utilice las bibliotecas para resolverlas.

4.2.2. Tecnologías

Lenguaje de programación

Para desarrollar el nuevo Detector se ha elegido Go (también conocido como Golang), como lenguaje de programación principal. En los párrafos siguientes se presentan los motivos.

Debido a la naturaleza del detector, es requisito poder trabajar de manera concurrente o paralela, pues se deben consumir y procesar al mismo tiempo varios flujos de datos distintos, y este procesamiento debe ser en tiempo cercano al real. Esto es totalmente factible utilizando Go, pues como se mencionó en la sección 2.2.4, soporta esta característica de manera nativa.

Además, la inclinación de Go por la legibilidad se alinea perfectamente con uno de los objetivos de la memoria, pues un código legible es mucho más mantenible y extensible que uno que no lo es.

En cuanto a la comparativa de Go y Java, en un paper del 2011 de Hundt[2] se comparan los tiempos de ejecución y otras características de varios lenguajes de programación al ejecutar un algoritmo de detección de ciclos en grafos. En ese paper Go no destaca realmente, siendo 5.5 veces más lento que el lenguaje más rápido (C++), y siendo mejor que algunas versiones de java y peor que otras.

Sin embargo, ese mismo año Russ Cox publica una entrada en el blog oficial de Go[1], en la cual analiza el paper antes mencionado. Cox utiliza una herramienta de Go llamada Profiling para detectar los cuellos de botella del código utilizado por el autor del paper. Con tan solo cambiar una estructura de datos por otra más simple, muestra una reducción del tiempo de ejecución a menos de la mitad. Luego, aplicando mejoras como un caché (el código de Java contaba con uno, pero el de Go no), llega a reducir hasta 11 veces el tiempo de ejecución de Go, superando por mucho a Java.

Sin embargo, el objetivo de esta comparación no es argumentar que el nuevo detector será mejor solo por estar implementado en Go, sino más bien justificar que el hecho de utilizar Go en lugar de Java no empeorará por si mismo, al menos de manera significativa, el rendimiento del detector, haciendo de Golang un lenguaje adecuado para la tarea.

Base de datos

Como sistema de almacenamiento se ha elegido a MongoDB, una base de datos no relacional (NoSQL).

El motivo principal tras está decisión es que Twitter entrega los tweets en formato JSON. Debido a la total compatibilidad del formato JSON y el formato BSON, que es el que utiliza MongoDB, el procesamiento que se debe realizar sobre un tweet antes de guardarse en la base de datos es prácticamente nulo. Esto debido a que no se necesita normalizar ni verificar que todos los campos tengan valores con tipos de datos específicos o no nulos.

Además del argumento recién mencionado, a continuación se presenta una comparativa de las ventajas y desventajas de usar una base de datos relacional contra una no relacional y cuales de estas aplican al caso de uso del detector.

- La mayoría de bases de datos SQL soportan transacciones, es decir que pueden realizar varias operaciones de forma atómica, de manera que si falla alguna, ninguna de las operaciones queda efectiva. MongoDB por defecto no lo hace. Sin embargo, en este caso no se requiere el uso de transacciones, pues solamente se están guardando tweets y si uno llega a fallar no debiese afectar al resto.
- MongoDB al tener documentos no estructurados es muy versátil. Por otro lado, las bases de datos relacionales necesitan definir una estructura fija y si de pronto un objeto contiene un campo nuevo que se quiera guardar, al modelo se le debe agregar una nueva columna, afectando a todas las entradas anteriores. En el caso de los tweets, son estructuras bastante dinámicas, pueden tener o no localización, contenido multimedia, hashtags, otros tweets citados, etc. Por lo que coinciden mejor con la descripción de los documentos de MongoDB que con las tablas relacionales.
- En cuanto a las consultas, pese a que cambia la sintaxis, MongoDB ofrece prácticamente la misma capacidad de búsqueda que cualquier otra base de datos. Sin embargo, esto tiene como consecuencia que Twicalli tenga que adaptarse para obtener los datos, pues actualmente los obtiene a través de consultas SQL a la base de datos del detector.

Versionamiento

Para versionar los proyectos tanto de las bibliotecas como de la aplicación principal se utiliza Git. Además se utiliza GitHub para almacenar los repositorios remotos.

Encapsulamiento

Para encapsular la solución y crear el ambiente de desarrollo se utiliza Docker y Docker Compose.

Las ventajas de utilizar esta tecnología son las siguientes:

- Al utilizar Docker no se requiere tener instalado ningún lenguaje de programación o base de datos, solo docker, por lo que realizar el ajuste inicial es muy simple, basta con ejecutar un comando para montar el ambiente local y correr la aplicación.

- Al usar los mismos archivos de configuración se garantiza que todos colaboradores que los utilicen tendrán exactamente el mismo ambiente, por lo que se elimina el riesgo de errores por incompatibilidad de versiones o de sistemas operativos.
- El proyecto queda completamente encapsulado, por lo que, por ejemplo si se necesita actualizar la versión de Go o de MongoDB, basta con actualizar la imagen base del contenedor, no la versión que tiene instalada la máquina local, evitando afectar a otros posibles proyectos.

4.3. Implementación

Go, al igual que el lenguaje de programación C, no posee clases ni objetos, pero tiene estructuras (structs), que sirven para definir tipos de datos compuestos a los que se les pueden asignar métodos, funcionando de manera similar a una clase. Se menciona esto porque para implementar cada paquete se sigue un esquema similar a las clases, donde se define una estructura principal, una o más funciones que permiten crearlas (constructores), y un conjunto de métodos a los que se puede acceder utilizando la estructura.

Otra característica de Go que impacta en la implementación es que los programas o la aplicaciones realizadas con este lenguaje están organizados en paquetes y módulos. Un paquete es una colección de archivos fuente en el mismo directorio que se compilan juntos. Las funciones, tipos, variables y constantes definidas en un archivo fuente son visibles para todos los demás archivos dentro del mismo paquete. A su vez, un módulo es una colección de paquetes Go relacionados que se publican juntos. Un proyecto puede tener uno o más módulos.

Hasta ahora se ha hablado de bibliotecas, pues es el termino común que se utiliza para referirse un conjunto de funcionalidades, usualmente relacionadas entre sí, que pueden importarse desde otros proyectos. En el caso de Go este concepto no existe. En lugar de ello se utilizan los módulos mencionados anteriormente, los cuales pueden publicarse en repositorios remotos, quedando accesibles al público. Para ello, el proyecto debe contar con un archivo llamado `go.mod`, en el cual se especifica el nombre del módulo.

El estándar es utilizar como nombre la URL del repositorio remoto donde se encuentra el modulo. Por ejemplo, el nombre del que contiene al procesador de texto es `https://github.com/JhonSalgado/text-processor`, luego, para obtenerlo desde otro proyecto se debe ejecutar el comando `go get https://github.com/JhonSalgado/text-processor`, lo que descargará y compilará el paquete. Para utilizarlo en el código se agrega la siguiente línea `import github.com/JhonSalgado/text-processor/processor` (nombre_modulo/nombre_paquete). El nombre del paquete usualmente coincide con el de la carpeta en la cual están contenidos los archivos fuente y debe ser un texto simple, sin mayúsculas, sin espacios y sin guiones.

A continuación se detalla el proceso de implementación del detector y los módulos independientes.

4.3.1. Procesador de texto

Este módulo agrupa las funcionalidades de procesamiento de texto, entre las cuales se encuentra eliminar caracteres especiales, obtener el set de palabras únicas y filtrar palabras vacías, para lo cual incluye un set precargado de palabras vacías de 28 lenguajes distintos y da la posibilidad al usuario de agregar las suyas.

Para utilizar los métodos que contiene, se debe crear una estructura llamada `TextProcessor`, para lo cual se proveen 2 constructores. El primero llamado `GetTextProcessor` no recibe ningún argumento y retorna un procesador que no aplicará ningún filtro a las palabras del texto. El segundo, llamado `GetTextProcessorWithStopWordsFilter`, recibe como parámetro una estructura llamada filtro que tiene 3 campos, una lista de lenguajes, una lista personalizada de palabras vacías y un booleano que indica si se quieren utilizar solo las palabras personalizadas. Retorna un procesador que si no se activa la opción de solo palabras personalizadas, cargará las palabras vacías de los lenguajes que se indicaron, que estén incluidos en el paquete. Si la lista está vacía se cargan todos los lenguajes disponibles. De indicarse que solo se utilicen palabras vacías personalizadas, no se cargarán las palabras incluidas en el paquete.

El procesador tiene tres métodos públicos (aquellos visibles para el usuario del paquete):

`CleanText`: Recibe un string, lo transforma a minúsculas y elimina todos los caracteres especiales, excepto los espacios de longitud 1. Para ello utiliza una expresión regular que detecta todos los caracteres que no sean letras Unicode o números y los reemplaza por espacios, para luego reducir aquellos espacios de largo mayor a uno a espacios de largo uno. Los espacios se necesitan para distinguir donde comienza una palabra y termina otra, por eso no se remueven.

`GetWordsSetWithOcurrence`: recibe un texto y devuelve un mapa llave-valor donde las llaves son las palabras únicas y los valores son la cantidad de veces que ocurrieron en el texto. Para ello primero limpia el texto con `CleanText`, luego utiliza un método privado llamado `getWords` que divide el string por los espacios y entrega una lista con las palabras, y finalmente utiliza un map (implementación de hashmaps de Go), para insertar las palabras de la lista como llaves, sumando uno al valor de la llave cada vez que lo hace. En caso de tener palabras vacías cargadas, revisará si las palabras están incluidas en la lista a filtrar antes de ingresarlas al mapa.

`GetWordsSet`: Recibe un texto y devuelve una lista con todas las palabras únicas en él. Para ello llama a `GetWordsSetWithOcurrence` y entrega solamente las llaves, esto debido a que para comprobar qué palabras están repetidas se llevaría a cabo el mismo proceso con el map, duplicando gran parte del código.

Este paquete está completamente documentado y testeado (con 100 % de cobertura), y se encuentra disponible de forma gratuita para toda la comunidad a través de su repositorio público en github.com[1]. Además se incluye un flujo para que cada vez que alguien contribuya al repositorio y quiera traspasar sus cambios a la rama principal, se ejecuten automáticamente los tests en una máquina provista por Github, rechazando inmediatamente los cambios en

[1]https://github.com/JhonSalgado/text-processor

caso de que algún test falle.

4.3.2. Detector de países

Este paquete sirve para detectar menciones de países y comunas chilenas en textos, en más de 100 lenguajes disponibles. Al conseguir una detección retorna el nombre del país en inglés, su código ISO alpha-2 (2 letras), y sus coordenadas geográficas. En el caso de las comunas retorna su nombre completo y coordenadas. Está información la obtiene de un grupo de archivos estáticos que son los mismos que utiliza el detector que esta en funcionamiento actualmente.

La información y formatos de los archivos son los siguientes:

- Traducciones: Tiene 5 columnas separadas por tabuladores, que corresponden respectivamente a: código alfa-2 del país, nombre del país en inglés, nombre traducido, código ISO 639-1 del idioma al que está traducido, el nombre del idioma en inglés.
- Países: Tiene 4 columnas separadas por tabulación, que corresponden respectivamente a: código de país alfa-2, latitud, longitud, nombre del país en inglés.
- Municipios: Cada archivo en esa carpeta debe tener 4 columnas separadas por tabulación, que son respectivamente: nombre de detección, latitud, longitud, nombre completo. La primera columna es la que se usa para detectar por lo que recomiendo agregar una fila por cada nombre alternativo que tenga el municipio, incluyendo su nombre completo.

Este módulo funciona de manera análoga al paquete de procesamiento de texto, por lo que para acceder a los métodos se debe crear una estructura llamada detector. Para ello existen 2 constructores: un constructor que no recibe parámetros y retorna un detector solamente de países y un constructor que recibe un código de país y retorna un detector de países que también tiene cargadas las municipalidades del país al que corresponde el código (por ahora solo están disponibles las de Chile pero se espera que el proyecto se amplíe en el futuro).

El detector tiene dos métodos principales llamados `Detect` y `DetectInAnyLang`, ambos reciben un texto y devuelven un booleano para indicar si se detectó un país junto con la información de ese país. Si el detector tiene municipios cargados y el país detectado corresponde al país al que pertenecen, o si no se encontró ningún país, también intentará detectar el municipio. Si se encuentra, se entregará junto con la información del país.

La principal diferencia entre ambos métodos, como lo indican sus nombres, es que `DetectInAnyLang` verificará si el texto contiene el nombre de un país traducido en hasta 140 idiomas diferentes, mientras que `Detect` recibe una lista de idiomas (sus códigos ISO 639-1), y solo usará esas traducciones para detectar. En caso de que algún código de idioma proporcionado no sea válido o no sea compatible, el método devolverá un error como tercer resultado.

El algoritmo utilizado para detectar países fue renovado completamente con respecto al detector original, pues este último sólo revisa si el texto contiene alguna traducción del país, dando como resultado que, por ejemplo, en el texto "blusa" se detecte a Estados Unidos

porque “blusa” contiene la palabra “usa”. Para mejorar esto, en lugar de revisar si el texto contiene al país, se revisa si contiene una expresión regular, para asegurarse de que el nombre del país empieza justo después de un espacio o del inicio del texto y termina justo antes de un espacio o el final del texto, no en medio de una palabra. Para evitar complicar la expresión regular para que detecte casos como “#usa” o “usa!”, antes de realizar la detección se utiliza el paquete procesador de texto para quitar los caracteres especiales, quedando ambos ejemplos simplemente como “usa”.

Sin embargo, si bien con la expresión regular se evitan falsos positivos también se aumenta el costo computacional, pues se deben realizar más comparaciones para comprobar todos los casos en los que la expresión da positivo. Para compensar ese tiempo extra se implementa una mejora basada en la premisa del algoritmo de búsqueda de texto Knuth–Morris–Pratt. Este algoritmo se basa en comprobar si es posible que un string esté contenido en un texto revisando primero si un prefijo del string está contenido. Si bien el algoritmo completo es más complejo, para este caso de uso basta con implementar una versión simplificada del mismo para obtener resultados satisfactorios.

La variante que se implementa es la siguiente: antes de comprobar si la expresión regular de un país está contenida en el texto, se revisa si un prefijo de largo fijo de la expresión está contenido. Para elegir el largo del prefijo se realiza un experimento, detectando países en los textos de 2500 tweets en inglés. Se escogió este número pues ha sido el máximo de tweets que se han registrado en un minuto, al no aplicar ningún filtro al endpoint que entrega el 1 % de los tweets mundiales.

Los resultados se pueden ver en la figura 4.1. En ellos se observa que el mejor resultado se da con el prefijo de largo 4 (12.46 segundos), lo cual es cerca de un 24 % menor que al no usar prefijos (16.27 segundos), por lo que se fija en ese largo.

Al igual que el procesador de texto, este paquete esta documentado y testeado con el 100 % de las líneas de código cubiertas, y se encuentra a disposición de la comunidad en github.com[2]. También incluye un flujo de integración continua para que se ejecuten automáticamente los tests cuando alguien proponga un cambio.

Debido a que ambos paquetes están pensados para ser importados desde cualquier proyecto, el hecho de depender de archivos estáticos es inconveniente, pues estos no se importan por defecto junto con el código y de hacerlo se desconocería su ubicación final, pues dependería de la ubicación en la cual el usuario tenga su proyecto (esta ubicación se necesita para abrir los archivos desde el código y cargar su información).

Para solucionar esto, se añade a cada proyecto un programa ejecutable que transforma los archivos de texto a archivos Go, dejando la información que estos contienen inmediatamente en estructuras de datos convenientes para realizar el procesamiento y haciendo que al importar y compilar el paquete, este ya tenga cargada la información. La ejecución de este programa se debe realizar solo una vez antes de publicar el paquete, luego todo el que lo obtenga tendrá ya creados los archivos Go. También se debe ejecutar en caso de hacer cambios a los archivos

[2]https://github.com/JhonSalgado/country-detector

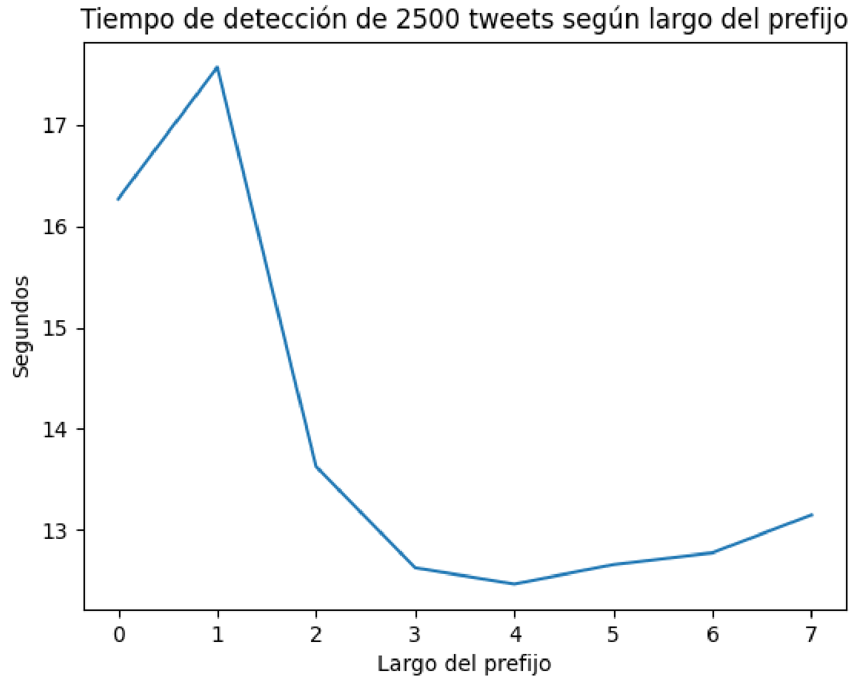

Figura 4.1: Tiempo de detección de 2500 tweets dependiendo del largo del prefijo que se utiliza para descartar.

estáticos, como agregar nuevas traducciones, nuevos lenguajes, etc. para que los cambios se hagan efectivos y queden en los archivos Go. Cabe destacar que esta funcionalidad solo afecta a la mantención y evolución de los proyectos, no afecta en nada a los usuarios de los paquetes, los cuales no necesitan realizar ninguna acción adicional para utilizarlos más que importarlos en su proyecto.

4.3.3. Nuevo detector de eventos

Esta es la aplicación principal que se encarga de la obtención, procesamiento y almacenamiento de tweets. Contiene 2 paquetes privados (no importables fuera del proyecto), que se describen a continuación:

DB

Este paquete posee métodos para iniciar la conexión con la base de datos, realizar operaciones de lectura y escritura, y define los modelos Tweet y Window como estructuras de Go, que se utilizarán para dar forma a los documentos que se creen en la base de datos. Para esto utiliza el controlador de MongoDB oficial para Go (creado por la misma compañía a cargo de MongoDB), el cual se importa de forma equivalente a los módulos antes descritos.

Sigue un patrón de diseño similar a los módulos anteriormente descritos. Para acceder a los métodos de lectura y escritura se debe crear una estructura llamada *DBHandler*, y para obtenerla existe un único constructor que recibe como parámetros un cliente para conectarse al servidor o contenedor de MongoDB y el nombre de la base de datos donde se guardarán

los tweets y las estadísticas.

Para obtener el cliente, en el mismo paquete se proporcionan 2 métodos. El primero sirve para obtenerlo a partir de un string de conexión, que es lo que usualmente proporcionan los servicios de bases de datos para conectarse y contienen la información del domino o dirección IP, el puerto y las credenciales de usuario para autenticarse en la base de datos. El otro método en lugar de recibir el string de conexión, recibe servidor y credenciales como parámetros separados y los utiliza para obtener el cliente, está pensado para conectarse a bases de datos que se levanten de forma local (se entregarán más detalles de esto cuando se hable sobre docker).

La estructura *DBHandler* contiene el cliente de la base de datos y referencias a las colecciones "tweets" y "windows", en las que se guardarán los tweets y las estadísticas por ventanas de tiempo respectivamente. Estas colecciones están dentro de la base de datos creada con el nombre indicado previamente en el constructor de la estructura. Posee 2 métodos principales, *SaveTweets* y *SaveWindows*, que como sus nombres indican, sirven para guardar tweets y ventanas respectivamente. Ambos pueden guardar varios documentos a la vez pues reciben listas de estructuras como argumento y utilizan el método *InsertMany* provisto por el paquete controlador de MongoDB.

Twitter

Este paquete es bastante más complejo que el anterior, esta dividido físicamente en 2 archivos, uno en el que se proveen los métodos para conectarse con Twitter y otro para recibir y procesar los Tweets.

De manera análoga a los paquetes anteriores, este paquete también tiene una estructura principal, llamada *TwitterHandler*, que se debe crear para acceder a los métodos principales. Esta estructura es bastante más extensa que las otras que se han comentado.

Posee los siguientes campos: Un cliente para comunicarse con Twitter, la lista de lenguajes, la lista de palabras claves y las coordenadas que se usarán para filtrar los tweets, 3 punteros a canales para recibir tweets en tiempo real, uno para recibir los tweets filtrados por palabras claves, llamdo *keywordStream*, otro para recibir los tweets filtrados por ubicación llamado *boundingBoxStream* y otro para recibir tweets sin filtrar llamado *sampleStream*, además de un cuarto puntero llamado *currentStream* que se utilizará para indicarle a las rutinas que estén trabajando en paralelo cual es el flujo de datos que deben procesar, por lo que siempre apuntará a uno de los 3 antes mencionados.

Además, posee un puntero a una estructura de tipo *DBHandler* que utilizará para guardar los tweets, un entero para indicar el tamaño de las ventanas que se debe utilizar, un canal llamado *Timer* que se utilizará para avisarle a las rutinas que trabajen en paralelo cada que vez que empieza y termina una ventana de tiempo, un bool para indicar si se debe realizar la detección de eventos, un canal para que las rutinas que reciben los tweets envíen las estadísticas de las ventanas en caso de requerir detección y un mapa para almacenar los parámetros que sirven para calcular la distribución normal de las estadísticas de cada flujo de datos (promedio, desviación estándar y número de ventana).

Para crearlo existe un constructor que recibe como parámetros las credenciales de desarrollador provistas por Twitter, y entrega un *TwitterHandler* que inicialmente solo tiene configurado el cliente.

La estructura posee 3 métodos para conectarse a los flujos de datos de Twitter.

ConnectToSampleStream: Se conecta a un endpoint de Twitter que entrega en tiempo real una muestra del 1 % de los tweets del mundo. Si la lista de lenguajes a filtrar no está vacía, la pasa como argumento a la conexión, de esta manera solo se reciben tweets que estén en alguno de los idiomas listados. Almacena el canal por el que se recibirán los mensajes en el campo *sampleStream* del *TwitterHandler*.

ConnectToBoundingBoxStream: Recibe como parámetro una lista de 4 coordenadas que definen un cuadro delimitador (bounding box), y se conecta a un endpoint de Twitter que entrega en tiempo real una muestra de los tweets emitidos dentro de esas coordenadas. Almacena el canal por el que se recibirán estos mensajes en el campo *boundingBoxStream*. Al igual que en el caso anterior, si hay lenguajes en la lista también aplicará ese filtro.

ConnectToKeywordStream: Recibe como parámetro una lista de palabras clave, y se conecta a un endpoint de Twitter que entrega en tiempo real una muestra de los tweets que contienen esas palabras. Almacena el canal por el que se recibirán estos mensajes en el campo *keywordStream*. Al igual que en los casos anteriores, enviará el filtro de lenguajes si aplica.

Cabe destacar de que no es necesario conectarse a los 3 flujos de datos mencionados más arriba, pues a través de una archivo de configuración, del que se hablará más adelante en esta misma sección, se pueden seleccionar los filtros que se desean utilizar y por tanto los flujos a los que se conectará el detector.

Además de los ya mencionados, se tienen los siguientes métodos para recibir y procesar tweets:

ProcessStreams: Este es el método principal que se debe llamar para iniciar todo el procesamiento, se ejecuta en la rutina principal. Envía a cada uno de los flujos de datos conectados (sampleStream, keywordStream y/o boundingBoxStream), a procesarse a una rutina diferente, llamando uno de los métodos descritos más abajo (*processStreamOnline* o *processStreamByWindows*).

Antes de iniciar el procesamiento de los flujos, crea un canal para notificarle a la rutina el inicio y el termino de cada ventana y lo guarda en el campo *Timer* del *TwitterHandler*, además indicando en el campo *currentStream*, cual será el flujo que procesará la rutina.

Luego crea un objeto llamado *ticker* que recibe como argumento una duración de tiempo y cada vez que transcurra ese tiempo enviará un mensaje a través de un canal que tiene almacenado en uno de sus campos. El tiempo que se le pasa al ticker son los segundos especificados en el tamaño de la ventana del *TwitterHandler*.

Cada vez que el *ticker* genere un mensaje en su canal, este método lo comunicará a las

rutinas para que empaqueten los tweets que han recibido desde el último mensaje y envíen sus estadísticas en caso de estar habilitada la detección, las cuales esperará a través del canal de estadísticas del *TwitterHandler*.

Al recibir las estadísticas desde todas las rutinas, las pasa como argumento a un método llamado *detectBursts*, que se encarga de realizar la detección, ejecutándolo también en una rutina diferente.

processStreamByWindows: Escucha permanentemente 2 canales, el *timer* y el flujo de tweets al que hace referencia *currentStream*. Cada tweet que recibe lo añade a una lista, hasta que llega un mensaje desde el *timer*, momento en el que pasa la lista como argumento a otro método llamado *processTweets* y la vacía, para comenzar a añadir los nuevos tweets que lleguen y repetir el proceso.

processStreamOnline: Este método es muy similar al anterior, pero en lugar añadir los tweets a una lista, cada vez que llega uno nuevo llama a *processTweets* para procesarlo y guardarlo inmediatamente, y suma 1 a un contador. Cuando llega un mensaje a través del *timer*, envía la cantidad almacenada en el contador a través del canal de estadísticas y lo reinicia. Este método esta pensado específicamente para el procesamiento del *keywordStream*, pues al ser filtrado por palabras clave, siempre la proporción de tweets relevantes (que contienen palabras de interés), con respecto a los tweets totales de la ventana va a ser del 100 %, por lo que no aporta por si mismo información para realizar detecciones y por tanto no es necesario realizar un análisis por ventana.

processTweets: Este método recibe una lista de tweets y si el *TwitterHandler* fue creado con palabras clave revisa si los textos de estos contienen alguna. Si la revisión es favorable, se considera que el tweet es relevante, se transforma al modelo Tweet definido en el paquete DB, se almacena en este la keyword que contiene el texto y se realiza la detección de país (utilizando el paquete detector de países), sobre el texto del tweet y sobre la ubicación del usuario emisor de este (el cual es un campo de texto libre, por lo que no siempre tiene información relevante). Cuando hay detecciones exitosas se almacena la información del lugar detectado en el mismo tweet.

Cuando ya se revisaron todos los tweets, se reportan las estadísticas a través del canal de estadísticas, enviando la cantidad de tweets relevantes y la cantidad de tweets totales de la lista (los cuales corresponden a los tweets recibidos en una ventana de tiempo). Finalmente se guardan los tweets relevantes en la base de datos.

En caso de que el *TwitterHandler* no se haya creado con palabras clave, simplemente se consideran todos los tweets como relevantes y el proceso que sigue es exactamente el mismo.

Otro caso es particular es cuando este método es llamado por la rutina que esta recibiendo los tweets filtrados por palabras clave, pues esa rutina envía sus estadísticas desde otro método, por lo tanto en este se omite ese paso. La detección de palabras clave se realiza igualmente en este flujo, para saber exactamente cual es la palabra clave que contiene el texto y de esta manera facilitar las consultas a la base de datos en el futuro, cuando se quiera conocer esta información.

detectBursts: Este método implementa el algoritmo de detección descrito en la sección 3. Este realiza pequeñas variaciones dependiendo del flujo al que pertenezcan las estadísticas. En el caso del flujo filtrado por palabras clave no realiza ninguna detección, pues como se mencionó anteriormente, en este siempre los tweets relevantes van a ser la misma cantidad que los tweets totales. En el caso del flujo filtrado por ubicación, simplemente realiza la detección utilizando sus estadísticas de la forma que indica el algoritmo.

Finalmente en el caso del flujo de muestra sin filtrar, antes de realizar la detección se le suma la cantidad de tweets obtenidos por palabras clave, tanto a los tweets relevantes como a los tweets totales. Esto debido a que tanto el flujo de muestra como el filtrado por palabras son tweets provenientes de todo el mundo, por lo que sus cantidades debiesen fluctuar de la misma forma cuando varía la actividad de los usuarios, a menos de que haya algún evento que aumente los tweets que contienen las palabras, el cual es justamente el caso que se busca.

Para terminar guarda la información de la ventana para cada uno de los flujos en la base de datos. Los documentos resultantes tendrán la siguiente información: fecha de inicio y de fin de la ventana, tweets relevantes, tweets totales, z-score calculado, bool para indicar si ocurrió un evento en esta ventana y el flujo al que corresponden estas estadísticas. Esto quiere decir que si utilizamos los 3 flujos se crearán 3 documentos distintos en cada ventana de tiempo.

Main

Este paquete contiene solamente un archivo, el cual es el ejecutable principal. Se encarga de leer la configuración inicial a partir de variables de ambiente.

Las variables de ambiente utilizadas se describen a continuación:

- ACCESS_TOKEN, ACCESS_TOKEN_SECRET, CONSUMER_KEY y CONSUMER_SECRET: Corresponden a las credenciales de desarrollador que otorga Twitter para poder utilizar sus APIs.
- DB_CONNECTION_STRING: String para conectarse y autenticarse en la base de datos. Si no se cuenta con uno se puede dejar en blanco y usar el segundo método de autenticación descrito más abajo.
- DB_HOST, DB_USERNAME y DB_PASSWORD: Información y credenciales para conectarse a la base de datos. Este método es el recomendado cuando se trabaje en ambiente local, pues al estar utilizando docker, en DB_HOST basta con pasarle el nombre del contenedor de la base de datos y las credenciales que se prefieran. Al ejecutar la aplicación por primera vez, el usuario será creado automáticamente según las credenciales especificadas, mismas que utilizará la aplicación para conectarse a la base de datos.
- DB_NAME: Este valor es independiente del método de autenticación e indica el nombre de la base de datos donde se guardarán los tweets y las estadísticas.
- WINDOW_SIZE: Tamaño de la ventana en segundos, define cada cuanto tiempo se analizaran los tweets.
- LANGS: lista de códigos ISO 639-1 (2 letras) de los lenguajes por los cuales se desea filtrar los tweets.

- ANALYZE_STATS: Indica si se desea analizar las estadísticas para analizar los tweets.
- KEYWORDS_FILE: Archivo que contiene la lista de palabras clave.
- BOUNDING_BOX: Coordenadas geográficas que delimitan una zona del mundo.
- USE_SAMPLE_STREAM: Indica si se desea utilizar el flujo de muestra sin filtrar.

De acuerdo a esto, el ejecutable, a través del paquete DB, crea un cliente para la base de datos, utilizando string o datos de conexión dependiendo de cual se haya provisto, dando prioridad al string. Luego utiliza el constructor para inicializar un *DBHandler*, pasándole el cliente y el nombre de la base de datos a utilizar. Si no se especifica un nombre en DB_NAME, se utiliza por defecto "twitter_data".

Utilizando el constructor del paquete Twitter y las credenciales de desarrollador, se crea un *TwitterHandler*. Si se provee una valor para KEYWORDS_FILE y se encuentra el archivo con las palabras clave, se leen y se utilizan para llamar al método *ConnectToKeywordStream*. Si se encuentra un valor para BOUNDING_BOX se utilizan las coordenadas para llamar a *ConnectToBoundingBoxStream*, Si el valor para USE_SAMPLE_STREAM es "true", se llama al método *ConnectToSampleStream*. Además se le pasa al *TwitterHandler* el tamaño de la ventana indicado en WINDOW_SIZE (por defecto 5 minutos si no se especifica), la lista de lenguajes especificados en LANGS (si no existe simplemente no se aplica el filtro de lenguaje), un booleano obtenido de ANALYZE_STATS para indicarle si se deben analizar las estadísticas para detectar eventos y el *DBHandler* para el guardado de tweets.

Con el *TwitterHandler* completamente configurado, se llama al método *ProcessStreams* para dar inicio al proceso de recopilación y/o detección.

Contenedores

Como se describe en la sección de tecnologías, se utiliza Docker para crear y replicar el ambiente de desarrollo y ejecución del detector de manera local.

Para la base de datos no se necesita crear una imagen, simplemente se utiliza la última disponible para MongoDB.

Para el detector se crean dos imágenes, una de desarrollo y una que podría ser utilizada en un ambiente de producción.

La imagen de desarrollo se crea a partir de la última imagen disponible de Go. Esta contiene una distribución Linux y el mismo lenguaje Go. Luego, en su Dockerfile, se establecen los siguientes pasos:

1. Crear un directorio donde se moverá el código fuente.
2. Copiar los archivos que contienen las dependencias del proyecto (paquetes externos). En Go estos archivos son go.mod y go.sum.
3. Instalar las dependencias. En Go se hace con `go mod download`.
4. Copiar el código de la aplicación y los archivos estáticos necesarios hacia el directorio establecido en el primer paso.

5. Establecer el comando que se ejecutará al levantar el contenedor generado por esta imagen. En este caso `go run ./src/main.go`. El archivo main.go, es el archivo ejecutable contenido el paquete main descrito en la sección 4.3.3.

Al ejecutar la imagen generada a partir de este archivo, se creará un contenedor que tiene instalado Go y todas las dependencias del proyecto, y ejecutará automáticamente la aplicación al iniciarse (con el comando especificado en el último paso).

Para la imagen de producción se realiza una construcción multietapa, creando primero una imagen intermedia exactamente igual a la imagen de desarrollo. Se utiliza esa imagen para crear el binario de la aplicación, con el comando `go build ./src/main.go`, luego se crea una nueva imagen muy ligera, que contiene solo el sistema operativo Alpine Linux, se copia el binario a esta imagen, y se establece el comando de ejecución como `./main`. Esto se puede hacer debido a que el binario producto de compilar una aplicación Go se puede ejecutar nativamente en Linux, sin necesidad de instalar compiladores adicionales.

Al utilizar este archivo para construir la imagen, esta corresponderá solamente a la última que definimos (que solo tiene Linux y el binario), mientras que la imagen intermedia será eliminada tras utilizarse para compilar.

La principal diferencia es el tamaño de las imágenes, pues mientras que la de desarrollo pesa 1.65 GB, la de producción pesa solamente 23.24 MB. Esto debido a que no tiene instalado Go, ni cuenta con los archivos fuente de la aplicación. Sin embargo, esto último hace que la imagen de producción no sea adecuada para desarrollo, pues al no tener nada instalado no se puede, por ejemplo, ingresar dentro del contenedor generado y utilizar comandos de Go para ejecutar los tests.

Finalmente se crea el archivo `docker-compose.yml` para manejar los contenedores. En este se indica cuales serán los servicios a levantar, en este caso el detector y la base de datos, a partir de que imagen se crearán los contenedores, siendo para el detector una de las que acabamos de crear y para la base de datos la imagen oficial de MongoDB y se establecen las variables de ambiente. Si las variables se encuentran en algún archivo, se le puede indicar a docker que las obtenga desde ahí al momento de crear los contenedores. También se pueden añadir configuraciones adicionales, como el puerto local que utilizaran los contenedores y también si se quiere ligar la salida estándar de la aplicación a la consola donde se crea el contenedor, para de esta manera poder observar los mensajes con información o errores que ésta emite.

Al tener este archivo se pueden utilizar los comandos de `docker-compose`. Con `docker-compose build` se crearán las imágenes, pero en este caso la base de datos usa una imagen predefinida así que solo se crea la imagen del detector a partir del Dockerfile que hayamos especificado (desarrollo o producción). Por último con el comando `docker-compose up` se crean contenedores a partir de las imágenes, lo cual creará automáticamente una red para conectarlos e iniciará la ejecución de la aplicación.

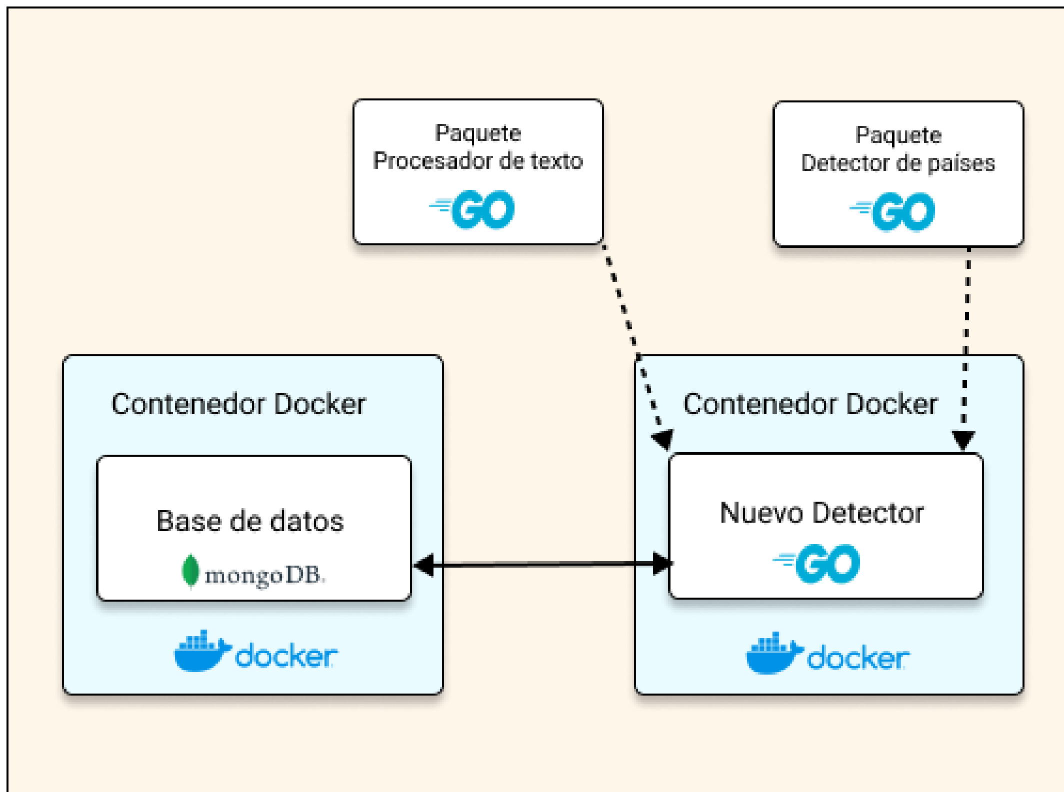

Figura 4.2: Arquitectura con contenedores Docker

En la imagen 4.2 se puede observar la arquitectura de la aplicación al crear los contenedores. El Nuevo Detector contiene los paquetes Main, DB y Twitter y a su vez importa los paquetes externos Procesador de Texto y Detector de Países, los cuales son instalados también dentro del contenedor.

Capítulo 5

Validaciones

Para validar si la solución descrita en la sección anterior satisface los objetivos, se evaluarán 3 aspectos:

- Factibilidad de operación con Twicalli: Capacidad del nuevo detector de reemplazar al original, proporcionando los datos que Twicalli necesita para su funcionamiento.

  Para ello se definen 3 criterios de aceptación sobre los datos generados:

  - Completitud: Los tweets almacenados por el detector deben contener toda la información que requiere Twicalli (texto del tweet, fecha de creación, información del usuario, lenguaje y localización).
  - Velocidad: Se requiere que el detector procese los tweets en tiempo cercano al real, en particular que la tasa de procesamiento de tweets por minuto sea mayor a la tasa de descarga de tweets por minuto.
  - Volumen: La cantidad de tweets guardados por unidad de tiempo (por ejemplo de un día), debe seguir el orden de magnitud de los tweets almacenados por el detector original. Por lo tanto, si se consultan estas cantidades para un día en que ambos detectores hayan estado funcionando, la cantidad total almacenada por el nuevo detector debe ser igual o mayor.
- Adaptabilidad de este a nuevos casos de uso: El detector debe ser capaz de orientarse a dos nuevos casos de uso: eventos marítimos y goles en un partido de fútbol, solo modificando la configuración.
- Mantenibilidad y extensibilidad:
  - Documentación: El nuevo detector debe tener un mayor ratio de comentarios por líneas de código que el original.
  - Reproducibilidad: El nuevo detector debe ser ejecutado en una menor cantidad de pasos que el original.

Cada uno de estos aspectos junto con sus criterios de aceptación se evaluarán y profundizarán en las siguientes secciones.

5.1. Operación con Twicalli

Lo indispensable para que Twicalli pueda operar son los tweets que entrega procesados el detector, por lo que todos los criterios que se establecen para la comparación están relacionados a este concepto. Estos se explican a continuación:

5.1.1. Completitud de los datos

La completitud corresponde a si se provee la totalidad de información requerida para cada tweet o no.

Al recibir un tweet directo desde Twitter, este trae bastante información. Sin embargo lo más importante es: texto del tweet, fecha de creación, información del usuario, lenguaje y localización GPS, la cual no siempre está presente, pero en caso de estarlo incluye nombre del país, código ISO, coordenadas geográficas o cuadro delimitador y en algunos casos también el nombre de una ciudad o comuna. Estos campos son imprescindibles para que Twicalli pueda mostrar los datos en la página web.

Además, el detector actual analiza el texto del tweet y el texto contenido en la localización del usuario (que puede tener cualquier valor), buscando la mención de un país. Si lo detecta agrega la misma información que traería una localización GPS: nombre, código y coordenadas. Esto para lograr localizar una mayor cantidad de tweets y permitir visualizarlos en los mapas disponibles en la página web de Twicalli.

El nuevo detector también realiza el proceso de detección de países, con el paquete mencionado en la sección 4.3.2. En el cuerpo principal del tweet añade los códigos de los países detectados en el texto, el usuario y el GPS, si es que existen, para que luego realizar las consultas a la base de datos sea más directo. Luego se agregan 3 objetos que contienen el resto de información de las ubicaciones (nombre completo y coordenadas), estos se llaman `placeText`, `placeUser` y `placeGPS`, haciendo referencia en su nombre a como se obtuvieron. Una versión simplificada del documento resultante que se almacena en la base de datos se puede ver en el anexo 1 (no se muestran los campos no relevantes). Además, como revisa en cada tweet que recibe si el texto contiene una palabra clave, añade la palabra detectada al cuerpo del tweet (en el campo `keyword`), para que de esta manera se puedan realizar análisis más específicos, como por ejemplo ver que palabra clave del conjunto de palabras clave es la que genera más ruido (tweets que contienen una palabra clave, pero utilizada en otro contexto, y por tanto no están relacionados con el evento que se quiere detectar).

Considerando que el nuevo detector almacena la misma información de los tweets que se almacena actualmente, añadiendo solo pequeños cambios para permitir más tipos de consultas, se puede establecer bajo este criterio que los datos generados por el detector si poseen completitud, es decir, que tienen toda la información necesaria.

5.1.2. Velocidad

Para cumplir con este criterio se requiere que el detector sea capaz de procesar y almacenar una mayor cantidad de tweets de los que recibe por minuto, pues de esta manera se garantiza

Tweets recibidos por minuto	Tiempo de Procesamiento y guardado (segundos)
3141	14.12
3425	15.03
3286	14.41
3355	14.56
3214	14.26

Tabla 5.1: Tiempo de procesamiento de tweets recibidos en un minuto

que podrá funcionar en tiempo cercano al real como hace Twicalli con el detector actual.

Para realizar esta prueba se ejecuta el detector sin ningún tipo de filtro, para que obtenga desde Twitter la mayor cantidad de mensajes posibles. Se registra cuantos tweets se reciben por minuto, los cuales luego pasan en conjunto por el procesamiento y guardado, tomando el tiempo de este proceso justo después de que los tweets se almacenan en la base de datos. Los resultados pueden verse en la tabla 5.1.

Se puede observar que en promedio se reciben 3282 tweets, los cuales en promedio tardan 14.48 segundos en procesarse. Utilizando regla de 3 se obtiene que en un minuto el detector es capaz de procesar y guardar aproximadamente 13599 tweets, lo cual es cerca de 4 veces la cantidad que se recibe por minuto.

Si bien estos tiempos pueden variar dependiendo de las especificaciones de la máquina donde se ejecuten, es un margen lo suficientemente amplio como para concluir que cumple con este criterio.

5.1.3. Volumen de los datos

Este criterio busca medir la cantidad de tweets que se almacenan en la base de datos por unidad de tiempo, pero a diferencia del anterior, no se enfoca en la capacidad de procesamiento y guardado del detector, si no que en la capacidad de recolección, que depende de la correcta obtención y filtrado de datos desde Twitter.

Para medir el volumen simplemente se toma el total de tweets almacenados en un día en que ambos detectores se hayan estado ejecutando con los mismos filtros. En este caso se utiliza el día 20 de Julio de 2021, de 00:00 a 23:59, hora UTC.

Cabe destacar que el detector original guarda todos los tweets que descarga, posean estos o no palabras relevantes para Twicalli, mientras que el detector nuevo solo almacena los tweets que poseen palabras clave, pues Twicalli solo muestra estos en su plataforma de visualización. Por lo tanto, solo se cuenta el total de tweets relevantes almacenados, donde se considera relevante a todo tweet que posea una palabra clave.

Los resultados de la medición se muestran en la tabla 5.2, donde se observa que el nuevo detector recopila 11228 tweets más que el original, lo que equivale a un aumento de un 44.98 %.

	Tweets Relevantes
Detector original	24980
Nuevo detector	36208

Tabla 5.2: Cantidad de tweets relevantes almacenados en un día por cada detector.

Esto se debe a que el detector original solo utiliza el flujo de datos filtrado por palabras clave y el flujo de datos filtrado por ubicación, mientras que el nuevo detector también procesa el flujo de muestra sin filtrar, cuyo volumen es mucho mayor que el de los flujos anteriores, almacenando también los tweets relevantes que provienen de este flujo.

Este flujo se usa por 2 razones: La primera es que el algoritmo de detección calcula la proporción entre el logaritmo de la frecuencia de tweets relevantes y el total de tweets de una ventana. Con este flujo se puede obtener una mejor aproximación del total de la ventana que con el flujo filtrado por ubicación. La segunda razón es que si el detector posee la capacidad de procesar un volumen mayor de datos, no hay razón para no hacerlo, pues el objetivo de Twicalli es detectar sismos a nivel global, no solo en Chile, y el flujo de muestra, al igual que el flujo filtrado por palabras clave, trae tweets de todo el mundo.

También se observa que, según los resultados obtenidos en la sección 5.1.2, procesar los 36208 tweets del día toma un máximo aproximado de 9 minutos. Por lo tanto, es perfectamente factible utilizar los tres flujos a la vez para generar más datos.

5.2. Adaptabilidad a nuevos casos de uso

Esta validación busca medir la capacidad del nuevo detector de adaptarse a nuevos casos de usos y la dificultad de hacerlo.

Para realizar esta medición se toma una solicitud del SHOA, el cual en primera instancia no necesita detección de eventos, sino comprobar si los usuarios en Twitter utilizan los conceptos en los que están interesados, entre los que se encuentran marea, linea de playa, derrame de petróleo, marea roja, carta náutica, entre otros.

Como se busca ejecutar el detector por un tiempo prolongado, se decide crear una máquina virtual en Google Cloud Platform, con sistema operativo Ubuntu, para ejecutar la aplicación. Para ello, se sube a la máquina el binario que se obtiene al compilar el código y el archivo de texto con las palabras en las cuales el SHOA está interesado. En este caso la única diferencia en la configuración con el detector de sismos es que se utiliza otra lista de palabras clave (dadas por el archivo mencionado anteriormente), que la detección de eventos esta desactivada y que no se utiliza el flujo de muestra sin filtrar, pues este trae tweets de todo el mundo y el SHOA solo está interesado en lo que ocurre en Chile.

Esto se logra configurando en particular las siguientes variables de ambiente:

- KEYWORDS_FILE="./static/keywords_shoa.txt" (la ubicación del archivo con las palabras clave del SHOA).

- USE_SAMPLE_STREAM=false
- ANALYZE_STATS=false

El resto de variables utilizan valores muy similares a los del detector de sismos:

- LANGS=es (código ISO 639-1 del lenguaje español)
- BOUNDING_BOX=-76.8507235,-55.1671700,-66.6756380,-17.5227345 (cuadro delimitador de Chile).

Las credenciales de Twitter utilizadas son las mismas, pero no se muestran aquí debido a su naturaleza confidencial.

Para almacenar los tweets recopilados por el detector, se utiliza otro servicio en la nube llamado Mongo Atlas, el cual permite crear y administrar bases de datos MongoDB, además de poseer varias características como escalabilidad, seguridad, etc. que no son realmente relevantes en este caso. Para conectarse a sus servidores provee un string de conexión, el cual se debe agregar a la variable de ambiente DB_CONNECTION_STRING, en la máquina virtual creada en Google Cloud donde se encuentra la aplicación.

El detector se ejecuta de manera satisfactoria por una semana, recopilando 45.657 tweets, los cuales quedan a disposición del SHOA para su posterior análisis. Sin embargo, una vista previa de los resultados revela que, en general, los usuarios no publican frecuentemente sobre eventos marítimos.

Muchos de los tweets observados tienen las palabras claves utilizadas en otro contexto, siendo "marea" la mas frecuente (casi el 50 % de los tweets), pues esta se utiliza también para referirse a alteraciones del sentido del equilibrio de una persona.

Otra aplicación del detector a un nuevo caso de uso, es la recopilación de tweets que contienen palabras relacionadas a goles, durante un partido de fútbol de la selección chilena contra la selección de Uruguay, el cual terminó en empate 1-1 con goles en el minuto 26 y 66.

Para ello, la única configuración que cambia con respecto a la realizada para el SHOA es la variable KEYWORDS_FILE, la cual recibe esta vez la ubicación del archivo que contiene las palabras claves relacionadas con goles (gol, autogol, golazo, etc).

En el gráfico 5.1 se puede ver la cantidad de tweets relacionados a goles, emitidos en Chile y Uruguay, por cada minuto de partido, observando claramente un aumento de estos en el momento en que se produjeron las anotaciones.

5.3. Mantenibilidad y extensibilidad

La mantenibilidad, en el contexto de computación, es la facilidad de hacer que un software permanezca en un estado funcional, o de ser restaurado a este.

Por otro lado la extensibilidad es la facilidad de evolucionar un software, usualmente para

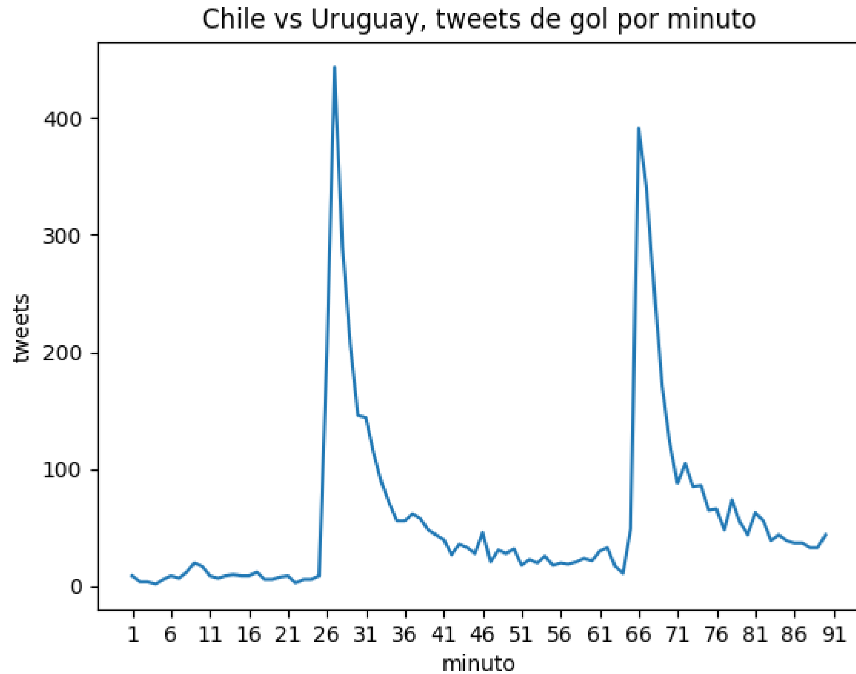

Figura 5.1: Tweets de gol por minuto en partido de fútbol de Chile contra Uruguay

agregar nuevas características.

Algunos factores que afectan a la mantenibilidad y extensibilidad son la presencia de test, la facilidad para reproducir errores, la documentación, la duplicidad de código y la facilidad con la que se integran nuevos colaboradores, pues las personas que mantienen y extienden un software no siempre participan en la creación del mismo desde el inicio.

Para mejorar ambos aspectos, tanto la aplicación principal como los paquetes creados de manera independiente cuentan con documentación dentro y fuera del código. Dentro en forma de comentarios y fuera en forma de breves manuales de uso. Además, los paquetes tienen tests unitarios que dan cobertura al 100 % de las líneas de código.

En la implementación se siguen estándares y buenas prácticas como evitar código duplicado, el uso de nombres descriptivos para variables y funciones, el principio de separación de intereses, entre otros.

Sin embargo, no se utilizan todas estas características para evaluar al nuevo detector. Debido a que mantenibilidad y extensibilidad son conceptos subjetivos por definición, pues decidir si algo es fácil o difícil es subjetivo en si mismo, se establecen dos métricas concretas para comparar al nuevo detector con el original.

La primera guarda relación con la documentación y corresponde a la proporción entre comentarios y líneas de código. La segunda tiene relación con la facilidad para recrear el ambiente de desarrollo y corresponde a la comparación del número de pasos necesarios para

llegar desde un estado donde no se tiene ni el código fuente ni tecnologías instaladas, a tener el detector en ejecución (en ambiente local).

5.3.1. Ratio de comentarios por líneas de código

Esta métrica se define como la proporción entre líneas de código y líneas de comentarios y se simboliza con una R. La fórmula para calcularla es la siguiente:

$$R = \frac{lineas_comentarios * 100}{lineas_codigo}$$

Se considera como línea de comentarios a toda línea en que se utilice el carácter designado por el lenguaje para crear comentarios y que tenga contenido.

En el detector original, que está programado en Java, se usa este formato de comentarios:

```
/**
   * Descripción función.
   *
   * @param descripción parámetro.
   * @return descripción valor de retorno.
*/
```

En este caso solo se cuentan 3 líneas de comentarios, pues la primera, la tercera y la última línea no tienen contenido.

En el caso del nuevo detector, que está programado en Go, los comentarios siguen el siguiente formato:

```
# Comentario 1.
# Comentario 2.
```

En este caso, al ser comentarios que se definen por línea y no por bloque, no hay líneas de comentarios que no tengan contenido.

Otra diferencia que vale la pena destacar, pero que no será utilizada en la medición, es que en Go la descripción de los métodos se da con formato de párrafo, por lo que las descripciones de los parámetros se separan simplemente con comas. En el formato de Java se crea al menos una línea por cada parámetro que reciba la función a describir.

Se considera como línea de código a toda línea no vacía que no sea un comentario. Esto aplica de igual forma a ambos lenguajes.

Con las definiciones dadas, se realiza el conteo cuyos resultados obtenidos se observan en la tabla 5.3. Como se puede observar, el nuevo detector tiene un ratio de comentarios un 15 % más alto que el detector original. Además, del valor de R se puede interpretar que en

	Líneas de código	Líneas de comentario	R
Detector original	13907	2834	20.37
Nuevo detector	750	176	23.47

Tabla 5.3: Ratio de comentarios por líneas de código para ambos detectores.

el nuevo detector cada aproximadamente 4 líneas de código hay un comentario.

Como observación adicional extraída de la tabla, se puede notar que el nuevo detector tiene solo un 5.39 % del tamaño del detector original (en líneas de código), por lo que la cantidad de código que se debe mantener es mucho menor. Esto debido principalmente a que el nuevo detector solo implementa las características que se utilizan o podrían utilizarse en el original, omitiendo todo el código hecho con fines de investigación.

5.3.2. Reproducibilidad del ambiente de desarrollo

Este criterio busca evaluar la complejidad de recrear el ambiente de desarrollo para un nuevo miembro del equipo de mantención y extensión de los detectores, que no posee ni el código fuente ni las tecnologías instaladas.

Para realizar la comparación se cuenta el número de pasos necesarios hasta que este nuevo colaborador pueda ejecutar ambos detectores.

Pasos requeridos para el detector original:

1. Descargar e instalar git.
2. Descargar el código. Debido a que este proyecto se encuentra en GitHub, basta con ejecutar el comando `git clone` junto con el nombre del repositorio.
3. Descargar e instalar un kit de desarrollo para Java.
4. Descargar e instalar MySQL.
5. Descargar e instalar las dependencias del proyecto (bibliotecas externas).
6. Contactar a los autores de las bibliotecas de uso restringido y pedir autorización para utilizarlas.
7. Completar el archivo de configuración, en el cual se le debe asignar un valor a 77 variables diferentes. Este es uno de los inconvenientes principales a la hora de ejecutar la aplicación, pues no existe documentación de como se debe llenar este archivo.
8. Iniciar el servicio MySQL.
9. Crear la base de datos.
10. Compilar el detector.
11. Ejecutar el archivo compilado.

Pasos requeridos para el nuevo detector:

1. Descargar e instalar git.

2. Descargar el código. Este proyecto al igual que el detector original se encuentra en GitHub, por lo que también basta con el comando `git clone`.
3. Descargar e instalar Docker.
4. Crear una copia del archivo `.env.template` y nombrarla `.env`.
5. Siguiendo el manual, asignarle un valor a cada variable del archivo `.env`, las cuales son 14 en total.
6. Ejecutar el comando `docker-compose up` en la raíz del proyecto. La primera vez que se ejecuta este comando, realiza varias acciones de manera automática: construye las imágenes siguiendo las especificaciones del `Dockerfile` presente en el proyecto, lee las variables de ambiente desde el archivo .env, levanta el contenedor de la base de datos y el contenedor del detector, crea un usuario en la base de datos y le pasa las credenciales al detector, crea la red para comunicar ambos contenedores y finalmente ejecuta el detector dentro de su contenedor. Las ejecuciones siguientes del comando solo levantarán los contenedores usando las imágenes ya creadas, les proporcionarán las variables de ambiente y ejecutarán el detector.

Como se puede observar, el detector actual no solo necesita de menos pasos (6 contra 11), si no que estos son más simples y acotados. En particular, la configuración inicial del detector original consiste en el llenado de 77 variables sin ninguna guía, mientras que en el nuevo de detector son sólo 14 y se cuenta con un manual para hacerlo.

Capítulo 6

Conclusiones y trabajo futuro

En esta memoria se crea una nueva versión del detector de eventos, una aplicación que recopila tweets relacionados a sismos, los procesa, analiza sus estadísticas para detectar aumentos inusuales en la cantidad recibida y los guarda en una base de datos, desde donde Twicalli, una aplicación web, los obtiene para mostrarlos de manera interactiva a través de gráficos y mapas.

El primer objetivo de la memoria es identificar cada característica del detector original que se debe mantener y cuáles no. Esto se cumple, resultando en un detector mucho más conciso, que como se vió en el capítulo 5, tiene un tamaño que representa solo el 5 % del tamaño del detector original, manteniendo todas las funcionalidades imprescindibles.

El segundo objetivo corresponde a diseñar una solución que permita la utilización de 3 características principales del detector de forma independiente, las cuales son el procesamiento de texto, la detección de países y la recopilación de tweets sin necesidad de pasar por el flujo de detección. Se logra dejar las primeras dos independientes, implementando cada una en un proyecto separado, en forma de paquetes de Go, que se pueden importar desde cualquier proyecto que desee utilizar sus funcionalidades de forma libre y gratuita. Para poder utilizar solo la recopilación de tweets, el detector se deja completamente configurable, permitiendo deshabilitar la recopilación de estadísticas y detección de eventos.

El tercer objetivo es que el nuevo detector genere tantos datos y tan completos como el detector original, en tiempo cercano al real. Esto para que Twicalli pueda seguir funcionando como lo ha hecho desde su creación. Este objetivo también se cumple, pues la generación de datos cumple con todos los criterios de aceptación establecidos en el capítulo 4. En particular, la información que se provee de cada tweet es ligeramente superior, el volumen de tweets es bastante más grande y se pueden procesar a una tasa mayor al ratio de llegada de los tweets en el peor caso (cuando no se aplica ningún filtro).

El cuarto objetivo es que el detector se pueda adaptar a nuevos casos de usos, para recopilar tweets y detectar eventos relacionados a otros conceptos además de los sismos. Este objetivo se cumple, pues se logra utilizar el detector para que recopile tweets relacionados a

conceptos principalmente marítimos requeridos por el Servicio Hidrográfico y Oceanográfico de la Armada de Chile. Sin embargo, se observa que, si bien el detector se puede utilizar para otro tipo de eventos, no siempre será útil utilizarlo, pues depende de los datos que se generan en Twitter. Si los usuarios no crean publicaciones relacionadas o suelen utilizar los conceptos bajo otros contextos que no son de interés, los datos obtenidos por el detector no serán de buena calidad.

El quinto y último objetivo es que el nuevo detector sea más mantenible y extensible, pues, por la falta de estas características, el detector original no se puede mejorar ni utilizar para nuevos casos de uso. Esto se logra a través del aumento de la documentación, la creación de manuales de uso, la utilización de tecnologías que faciliten el ingreso de nuevos colaboradores y la reducción del tamaño y complejidad del detector, removiendo características no necesarias.

Con cada uno de los objetivos específicos cumplidos, se concluye que el trabajo realizado en esta memoria logra cumplir el objetivo general, pues se crea un detector que carece de los problemas que tiene el detector original, pero mantiene todas las funcionalidades que le dan su utilidad. Esto se debe en gran medida a que se dedicó mucho tiempo al análisis de la situación actual y los problemas del detector antes de implementar la nueva versión.

Con el nuevo detector se abren muchas posibilidades, pues como se vió en el caso del SHOA, sus funcionalidades se pueden aplicar a nuevos casos de uso, permitiendo estudiar todo tipo de eventos, como erupciones volcánicas, pandemias o incluso eventos más triviales, como detectar cuándo se anotan goles en un partido.

Por otro lado, los paquetes creados de forma independiente representan un aporte para la comunidad de Go, pues son de código abierto y poseen algunas funcionalidades que no se pueden encontrar en otros paquetes gratuitos, debido también a que Go es un lenguaje relativamente nuevo.

La problemática que se busca resolver en esta memoria permite comprender la importancia de considerar la mantenibilidad, extensibilidad y el uso de buenas prácticas a la hora de diseñar e implementar un software, pues el detector original es una herramienta muy poderosa, con numerosas funcionalidades. Sin embargo, su utilidad se desvanece debido la dificultad de modificarlo y adaptarlo para aprovechar todas sus características y potenciales usos.

El trabajo realizado en esta memoria se puede mejorar de diversas formas. Por ejemplo, un aspecto que se puede mejorar mucho del detector, es justamente el algoritmo de detección, pues, si bien en Twicalli, con la visualización que provee es suficiente para que un usuario comprenda cuándo ha ocurrido un evento, podrían haber aplicaciones que necesiten conocer los eventos de manera automática, sin intervención humana, como por ejemplo un sistema que emita alertas cada vez que haya una detección.

Otro aspecto mejorable es la manera de proveer los datos recopilados, pues actualmente si una aplicación como Twicalli quiere obtener los tweets procesados, debe leer la base de datos en la que escribe el detector, teniendo que conocer el modelo de datos para poder realizar las consultas y teniendo que necesariamente utilizar controladores para conectarse con MongoDB. En lugar de eso, podría crearse una forma de acceso a los datos más estándar,

como una API, a la que se le envíen simplemente peticiones HTTP, ya que esta característica viene incluida en la mayoría de lenguajes, quitando la necesidad de instalar controladores externos.

Bibliografía

[1] Russ Cox. Profiling go programs. https://blog.golang.org/pprof, 2011.

[2] Robert Hundt. Loop recognition in c++/java/go/scala. https://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf, 2011.

[3] B. Poblete, J. Guzmán, J. Maldonado, and F. Tobar. Robust detection of extreme events using twitter: Worldwide earthquake monitoring. *IEEE Transactions on Multimedia*, 20(10):2551–2561, 2018.

Anexos

Anexo A

Twicalli

Anexo B

Documentos que se almacenan en la base de datos.


```
1  {
2      "_id" : ObjectId("60f604c2ec251190ca83d96a"),
3      "startDate" : ISODate("2021-07-19T22:58:28.404Z"),
4      "endDate" : ISODate("2021-07-19T23:03:28.404Z"),
5      "relevantTweets" : 4,
6      "totalTweets" : 6504,
7      "zScore" : 0.000152360916544377,
8      "isBurst" : false,
9      "streamType" : 2
10 }
```


Listing 1: Estructura de los documentos que guardan las estadísticas de cada flujo de datos por ventana de tiempo.

```
{
    "countryCodeText" : "cl",
    "countryCodeGPS" : "cl",
    "countryCodeUser" : "cl",
    "keyword" : "sismo",
    "createdAt" : "2021-07-20T03:10:52.000Z",
    "fullText" : "@GermanEliasP @SismoDetector Solo activación,
                 sin percepción del sismo en Concepción.",
    "placeText" : {
        "coordinates" : {
            "coordinates" : [
                -73.0444,
                -36.8201
            ],
            "type" : ""
        },
        "country" : "chile",
        "countryCode" : "cl",
        "fullName" : "concepción, chile"
    },
    "placeGPS" : {
        "coordinates" : null,
        "country" : "Chile",
        "countryCode" : "CL",
        "fullName" : "Concepción, Chile"
    },
    "placeUser" : {
        "coordinates" : {
            "coordinates" : [
                -71.542969,
                -35.675147
            ],
            "type" : ""
        },
        "countryCode" : "cl",
        "fullName" : ""
    },
    "user" : {
        "location" : "Chile",
        "name" : "Alvaro Molina Jara"
    }
}
```

Listing 2: Versión simplificada de los documentos guardados en la colección de tweets de la base de datos.