

**UNIVERSIDAD DE MÁLAGA**

**ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA**

**INGENIERO EN INFORMÁTICA**

**IMPLEMENTACIÓN DE LA ARQUITECTURA KAPPA PARA EL CÁLCULO  
DE MÉTRICAS EN TIEMPO REAL SOBRE STREAMS DE DATOS**

**Realizado por  
ÁLVARO MARTÍN LOZANO**

**Dirigido por  
ANTONIO VALLECILLO MORENO  
DOLORES BURGUEÑO CABALLERO**

**Departamento  
LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN**

**MÁLAGA, diciembre de 2017**



UNIVERSIDAD DE MÁLAGA ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA  
INFORMÁTICA

INGENIERO EN INFORMÁTICA

Reunido el tribunal examinador en el día de la fecha, constituido por:

Presidente/a D/D<sup>a</sup> \_\_\_\_\_

Secretario/a D/D<sup>a</sup>. \_\_\_\_\_

Vocal D/D<sup>a</sup>. \_\_\_\_\_

para juzgar el proyecto Fin de Carrera titulado:

**IMPLEMENTACIÓN DE LA ARQUITECTURA KAPPA PARA EL CÁLCULO DE  
MÉTRICAS EN TIEMPO REAL SOBRE STREAMS DE DATOS**

realizado por D. Álvaro Martín Lozano

dirigido por D. Antonio Vallecillo Moreno y D<sup>a</sup> Dolores Burgueño Caballero

ACORDÓ POR \_\_\_\_\_ OTORGAR LA

CALIFICACIÓN DE \_\_\_\_\_

Y PARA QUE CONSTE, SE EXTIENDE FIRMADA POR LOS COMPARECIENTES DEL  
TRIBUNAL, LA PRESENTE DILIGENCIA.

Málaga a \_\_\_\_\_ de \_\_\_\_\_ del \_\_\_\_\_

El/La Presidente/a

El/La Secretario/a

El/La Vocal



# **Implementación de la Arquitectura Kappa para el cálculo de métricas en tiempo real sobre *streams* de datos**

---



# Índice de contenidos

<b>ÍNDICE DE CONTENIDOS .....</b>	<b>7</b>
<b>CAPÍTULO 1. INTRODUCCIÓN.....</b>	<b>11</b>
1.1. MOTIVACIÓN .....	11
1.2. OBJETIVOS.....	13
1.3. RESUMEN DE CAPÍTULOS.....	14
1.3.a. Capítulo 2: Introducción al procesamiento de eventos en tiempo real.....	14
1.3.b. Capítulo 3: Tecnologías utilizadas. ....	14
1.3.c. Capítulo 4: Diseño e implementación de la solución .....	15
1.3.d. Capítulo 5: Funcionamiento del sistema.....	15
1.3.e. Capítulo 6: Conclusiones y extensiones futuras. ....	15
<b>CAPÍTULO 2. INTRODUCCIÓN AL PROCESAMIENTO DE EVENTOS EN TIEMPO REAL .....</b>	<b>17</b>
2.1. ESCENARIOS DE ANÁLISIS EN TIEMPO REAL.....	17
2.1.a. Datos de monitorización para la operación de sistemas. ....	17
2.1.b. Analíticas empresariales y publicidad.....	18
2.1.c. Redes sociales .....	18
2.1.d. El internet de las cosas .....	19
2.2. CARACTERÍSTICAS PRINCIPALES DEL PROCESADO EN STREAMING .....	19
2.2.a. Flujo constante de datos nuevos y alta cardinalidad del conjunto de datos .....	19
2.2.b. Poca estructuración de los datos.....	21
2.3. ARQUITECTURAS BIG DATA PARA EL PROCESAMIENTO DE FLUJOS DE DATOS .....	21
2.3.a. Arquitectura Lambda.....	23
2.3.b. Arquitectura Kappa.....	27
2.4. EL MOMENTO TEMPORAL DE UN EVENTO .....	29
<b>CAPÍTULO 3. TECNOLOGÍAS Y LENGUAJES UTILIZADOS .....</b>	<b>31</b>
3.1. SCALA.....	31
3.2. TWITTER STREAM API .....	32
3.3. KAFKA .....	34
3.3.a. Conceptos y arquitectura de Kafka.....	34
3.3.b. APIs de Kafka .....	36
3.4. KAFKA STREAMS.....	38
3.4.a. Topología de procesado.....	38
3.4.b. Agregaciones temporales y ventanas.....	40
3.4.c. Paralelización y recálculo del cómputo .....	40
3.5. INFLUXDB .....	42
3.5.a. Estructura de datos.....	42
3.5.b. Escritura de datos .....	44
3.5.c. Lectura de datos.....	44
3.6. GRAFANA .....	45
3.7. DOCKER Y DOCKER-COMPOSE .....	46
3.7.a. Conceptos básicos de Docker y utilización .....	46
3.7.b. Docker-compose .....	49

<b>CAPÍTULO 4. DISEÑO E IMPLEMENTACIÓN DE LA SOLUCIÓN.....</b>	<b>51</b>
4.1. REQUISITOS .....	52
4.2. DISEÑO .....	53
4.2.a. Diagramas de casos de uso.....	53
4.2.b. Arquitectura.....	53
4.3. IMPLEMENTACIÓN .....	57
4.3.a. Decisiones de implementación .....	59
4.3.b. Inyector de Twitter.....	61
4.3.c. Definición de métricas.....	65
4.3.d. Agregador.....	69
4.3.e. Escritor a InfluxDB.....	74
4.3.f. Despliegue mediante Docker-compose .....	76
4.3.g. Otras consideraciones.....	76
<b>CAPÍTULO 5. FUNCIONAMIENTO DEL SOFTWARE .....</b>	<b>83</b>
5.1. DESPLIEGUE DEL SISTEMA.....	83
5.2. ACCESO A LA INTERFAZ GRÁFICA .....	86
5.3. CONFIGURACIÓN DE UNA NUEVA MÉTRICA .....	91
<b>CAPÍTULO 6. CONCLUSIONES Y EXTENSIONES FUTURAS .....</b>	<b>97</b>
CONCLUSIONES .....	97
6.1. ....	97
6.2. AMPLIACIONES FUTURAS.....	98
6.2.a. Diferentes fuentes de datos de entrada .....	98
6.2.b. Nuevos tipos de métricas.....	99
6.2.c. Lenguaje específico para la definición de métricas e interfaz gráfica.....	99
6.2.d. Despliegue en la nube y escalado .....	99
<b>REFERENCIAS.....</b>	<b>101</b>
<b>ANEXO 1. MENSAJE EJEMPLO DE TWITTER. CREACIÓN DE UN TWEET.....</b>	<b>103</b>
<b>ANEXO 2. MENSAJE EJEMPLO DE TWITTER. BORRADO DE UN TWEET .....</b>	<b>107</b>



# Índice de figuras

FIGURA 1 EVOLUCIÓN DE LA GENERACIÓN DE DATOS .....	11
FIGURA 3 ARQUITECTURA LAMBDA.....	25
FIGURA 4 RECÓMPUTO DE DATOS EN LA ARQUITECTURA KAPPA .....	28
FIGURA 5 ARQUITECTURA ASÍNCRONA DE LOS SISTEMAS BIG DATA .....	29
FIGURA 6 ARQUITECTURA MAP-REDUCE .....	32
FIGURA 7 DISTRIBUCIÓN DE BROKERS, PARTICIONES Y TOPICS EN KAFKA .....	34
FIGURA 8 PARTICIONES Y RÉPLICAS EN KAFKA .....	36
FIGURA 9 ELEMENTOS DE KAFKA .....	37
FIGURA 10 TOPOLOGÍA DE KAFKA STREAMS .....	39
FIGURA 11 ESCALADO DE UNA APLICACIÓN EN KAFKA STREAMS .....	41
FIGURA 12 ESCRITURA DE DATOS EN INFLUXDB .....	44
FIGURA 13 LECTURA DE DATOS EN INFLUXDB .....	45
FIGURA 14 DASHBOARD EJEMPLO DE GRAFANA .....	45
FIGURA 15 DIFERENCIAS ENTRE CONTAINERS DOCKER Y MÁQUINAS VIRTUALES .....	46
FIGURA 16 DOCKERFILE DE LA IMAGEN OFICIAL DEL REPOSITORIO PÚBLICO DE GRAFANA .....	47
FIGURA 17 FICHERO DE CONFIGURACIÓN DE DOCKER-COMPOSE .....	50
FIGURA 18 DIAGRAMA DE CASOS DE USO .....	54
FIGURA 19 ARQUITECTURA KAPPA .....	54
FIGURA 20 DIAGRAMA DE COMPONENTES .....	56
FIGURA 21 ARQUITECTURA DEL SISTEMA Y TECNOLOGÍAS .....	58
FIGURA 22 CLASES DEL INYECTOR DE TWITTER .....	63
FIGURA 23 CLASE PRINCIPAL DEL INYECTOR DE TWITTER .....	64
FIGURA 24 DIAGRAMA DE FLUJO DEL INYECTOR DE TWITTER.....	64
FIGURA 26 DIAGRAMA DE CLASES DEL RECOLECTOR DE DEFINICIONES DE MÉTRICAS .....	69
FIGURA 27 DIAGRAMA DE FLUJO DEL AGREGADOR .....	71
FIGURA 28 CÓDIGO DE LA TOPOLOGÍA DEL AGREGADOR .....	73
FIGURA 29 DIAGRAMA DE FLUJO DEL ESCRITOR A INFLUXDB .....	75
FIGURA 30 PUNTOS ESCRITOS EN INFLUXDB TRAS APLICAR UNA DEFINICIÓN DE MÉTRICA .....	76
FIGURA 31 FICHERO DE DESPLIEGUE DE DOCKER-COMPOSE .....	78
FIGURA 32 TEST IMPLEMENTADO PARA LA CLASE CACHEDMETRICDEFINITIONRETRIEVER .....	81
FIGURA 33 SCRIPT DE COMPILACIÓN Y GENERACIÓN DE IMÁGENES DOCKER .....	85
FIGURA 34 COMANDOS PARA ESPECIFICAR LAS CREDENCIALES DE ACCESO A LA BASE DE DATOS .....	85
FIGURA 35 CONTAINERS DE DOCKER EN EJECUCIÓN TRAS EL DESPLIEGUE .....	86
FIGURA 36 ACCESO A LA CONFIGURACIÓN DE LA BASE DE DATOS EN GRAFANA .....	87
FIGURA 37 CONFIGURACIÓN DE LA BASE DE DATOS INFLUXDB EN GRAFANA .....	89
FIGURA 38 CREACIÓN DE UN NUEVO DASHBOARD EN GRAFANA .....	90
FIGURA 39 CONFIGURACIÓN DE UNA CONSULTA EN UN GRÁFICO DE GRAFANA .....	90
FIGURA 40 GRÁFICA DEL NÚMERO DE MENSAJES RECIBIDOS CADA 30 SEGUNDOS.....	91
FIGURA 41 DEFINICIÓN DE MÉTRICA 'CONTIENE FELIZ POR IDIOMA' .....	92
FIGURA 42 CONFIGURACIÓN DE LA CONSULTA PARA LA GRÁFICA 'CONTIENE FELIZ POR IDIOMA' .....	93
FIGURA 43 CONFIGURACIÓN DE TÍTULO PARA LA GRÁFICA 'CONTIENE FELIZ POR IDIOMA' .....	94
FIGURA 44 CONFIGURACIÓN DE LEYENDA PARA LA GRÁFICA 'CONTIENE FELIZ POR IDIOMA' .....	94
FIGURA 45 GRÁFICA 'CONTIENE FELIZ POR IDIOMA' .....	95



# Capítulo 1. Introducción

## 1.1. Motivación

La era de los datos ha llegado. Vivimos en la época de las redes sociales, los dispositivos móviles y el internet de las cosas. La cantidad de datos generados se ha incrementado de manera exponencial en los últimos años. Se estima que desde los orígenes de la especie humana hasta el año 2003, la cantidad total de datos generada ronda los 5 Exabytes (aproximadamente 5.000 Terabytes). Solo en el período entre el 2003 y el 2007, se generaron 290 Exabytes, y contando únicamente el año pasado, la cifra asciende a 20 Zettabytes (aproximadamente 20.000 Exabytes). El crecimiento sigue la misma evolución según las últimas estimaciones por lo que en 2025 se espera que la cantidad de datos generada durante el año alcance los 160 Zettabytes, tal y como muestra la Figura 1. [1]

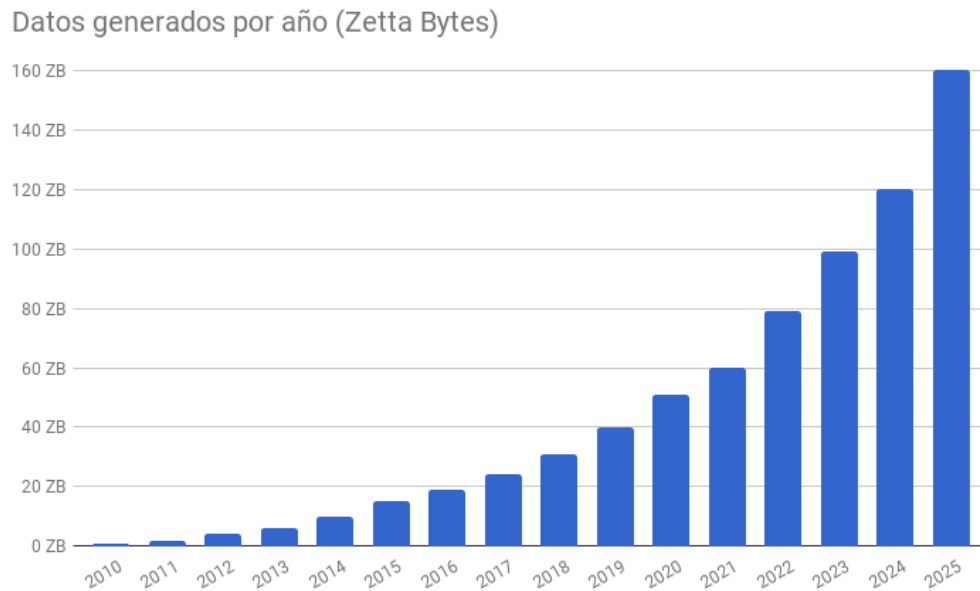


Figura 1 Evolución de la generación de datos

Debido a la explosión en cantidad de datos disponibles, empresas de todos los sectores se han ido transformando progresivamente para intentar sacar provecho de esta inmensa fuente de

información. Ejemplos como Google<sup>1</sup> o Facebook han pasado a ser líderes mundiales gracias a su habilidad para aprovechar la tendencia, centrando casi exclusivamente el volumen de su negocio en el análisis de datos.

La transformación del tejido empresarial viene de la mano con un cambio en las metodologías de análisis de datos. Con la gran cantidad de datos generada de manera continua, las tradicionales arquitecturas basadas en bases de datos relacionales han dejado de ser efectivas. La cantidad de datos a analizar supera con creces los límites de lo que podemos llegar a almacenar en soluciones de bases de datos tradicionales.

Es así como las tecnologías de Big Data entran en escena. Cuando la capacidad de captura, gestión y procesamiento de los datos no es suficiente mediante el uso de las metodologías tradicionales, aparecen las arquitecturas basadas en el procesamiento de eventos. Los datos dejan de ser modelados como conjuntos estáticos para convertirse en un flujo continuo e infinito que debe ser procesado conforme es recolectado.

Dentro de las tecnologías de procesamiento de flujos o *streams* de datos encontramos dos modelos principales:

- El **procesamiento por lotes**, inspirado en el procesamiento clásico de datos. Esta técnica es lenta pero precisa: necesita de un lote o *batch* completo para realizar el proceso. Al contar con todos los datos al completo, el resultado es exacto en la mayoría de los casos.
- El **procesamiento en tiempo real**. Como contraposición al procesamiento por lotes, no necesita de un *batch* para iniciar el proceso, sino que realiza el cálculo de manera continua sobre los datos al mismo ritmo que son recolectados. La velocidad de presentación de resultados es mucho más elevada, pero a cambio la exactitud de los cálculos se ve afectada.

En este proyecto se pretende exponer el estado del arte en el procesamiento de datos con metodologías Big Data, así como implementar una de las arquitecturas más vanguardistas en el

---

<sup>1</sup> [www.google.com](http://www.google.com)

procesamiento de datos en tiempo real, una arquitectura derivada de la ampliamente conocida arquitectura Lambda: la arquitectura Kappa.

## 1.2. Objetivos

El principal objetivo del proyecto es realizar una implementación de la arquitectura Kappa para el procesamiento de un *stream* de datos. La arquitectura Kappa da solución al problema del manejo de un flujo de eventos en tiempo real. En el proyecto se realizará un análisis detallado de esta arquitectura, explicando las causas que la convierten en una buena solución para el problema de generar métricas sobre los eventos de Twitter.

Antes de realizar la implementación se realizará un análisis del estado del arte en tecnologías y arquitecturas existentes para solucionar problemas relacionados con el procesamiento de flujos continuos de datos. Igualmente, se elegirán las tecnologías a utilizar, exponiendo los beneficios e inconvenientes que aportan sobre otras tecnologías.

La plataforma a implementar tomará como entrada un flujo continuo de datos, al que se le aplicará una serie de transformaciones para generar métricas. Estas métricas estarán disponibles para su visualización por parte del usuario en *dashboards* con gráficas temporales. Entre otras propiedades, se priorizará la velocidad de procesamiento para conseguir visualizar las métricas en tiempo real.

Como flujo de datos de entrada se tomarán datos de la plataforma Twitter<sup>2</sup>, que consisten en el flujo continuo de eventos publicados por los usuarios en tiempo real. Pese a esto, la solución del problema no estará ligada al uso exclusivo de esta fuente de datos. El sistema trabajará con datos no estructurados permitiendo así flexibilidad a la hora de ingerir datos de distintas fuentes [2].

Varios puntos a tener en cuenta son la escalabilidad, mantenibilidad y facilidad de despliegue de la solución. Dada la variedad de posibles escenarios donde se puede aplicar, existiendo una gran diversidad en el tamaño de los datos a procesar, la solución deberá ser

---

<sup>2</sup> [www.twitter.com](http://www.twitter.com)

explícitamente diseñada para una facilidad de escalabilidad que le permita adaptarse a los diferentes requisitos de forma sencilla.

Como principal caso de uso, el usuario deberá ser capaz de crear métricas nuevas sin necesidad de modificar el código o realizar un nuevo despliegue de la solución para que éstas tomen efecto. Para ello, se creará un lenguaje sencillo de definición de métricas que permitirá añadir nuevas métricas que tengan un efecto inmediato en las gráficas a visualizar. De igual manera, el usuario será capaz de componer diferentes gráficas de evolución de las métricas deseadas en una interfaz de usuario destinada a la visualización de series temporales.

### 1.3. Resumen de Capítulos

Este documento contiene los siguientes capítulos:

#### *1.3.a. Capítulo 2: Introducción al procesamiento de eventos en tiempo real.*

En este capítulo se explican los principios teóricos en los que el sistema está basado. Se expone el estado del arte en procesamiento de eventos, introduciendo el concepto de procesamiento de eventos, con especial énfasis en el procesamiento de eventos en tiempo real. Además, se detalla la arquitectura Kappa utilizada en la implementación de la solución, así como la arquitectura raíz de esta, la denominada arquitectura Lambda.

#### *1.3.b. Capítulo 3: Tecnologías utilizadas.*

Este capítulo realiza una descripción de las tecnologías utilizadas en el proyecto. Aquí se exponen tecnologías como Scala<sup>3</sup>, Apache Kafka<sup>4</sup>, Docker<sup>5</sup>, InfluxDB<sup>6</sup> y Grafana<sup>7</sup>. Se detalla la función que cumplen dentro del proyecto, se explica su funcionamiento de forma básica y se describen sus beneficios e inconvenientes en escenarios como el que el proyecto trata de abordar.

---

<sup>3</sup> [www.scala-lang.org](http://www.scala-lang.org)

<sup>4</sup> [kafka.apache.org](http://kafka.apache.org)

<sup>5</sup> [www.docker.com](http://www.docker.com)

<sup>6</sup> [www.influxdata.com](http://www.influxdata.com)

<sup>7</sup> [grafana.com](http://grafana.com)

### ***1.3.c. Capítulo 4: Diseño e implementación de la solución***

En este capítulo se detalla, con sus diagramas UML<sup>8</sup> correspondientes, el diseño de la aplicación implementada. Se hace especial inciso en los patrones, principios y metodologías utilizadas. De igual manera se explican los compromisos tomados, y los beneficios que éstos aportan potenciando algunas cualidades del diseño.

### ***1.3.d. Capítulo 5: Funcionamiento del sistema***

Este capítulo expone una guía detallada de usuario. Aborda el despliegue y uso de la aplicación incluyendo su despliegue desde un entorno vacío, con los prerequisites necesarios para su ejecución. Se incluirán diagramas e imágenes que guíen al usuario desde la compilación y despliegue de los componentes del sistema, hasta la ejecución de un caso de uso completo.

### ***1.3.e. Capítulo 6: Conclusiones y extensiones futuras.***

En este capítulo se presentan las conclusiones finales del proyecto, así como una serie de extensiones futuras a realizar para añadir nueva funcionalidad o mejorar la calidad del software.

---

<sup>8</sup> [www.uml.org](http://www.uml.org)





# ***Capítulo 2. Introducción al procesamiento de eventos en tiempo real***

## **2.1. Escenarios de análisis en tiempo real**

Con el creciente aumento en la generación de datos, en un mundo cada vez más conectado, hemos alcanzado volúmenes por unidad de tiempo que han cambiado totalmente la manera de afrontar el problema del procesamiento de los mismos.

Las arquitecturas clásicas estaban diseñadas para almacenar los datos en sistemas físicos, y proceder al análisis en una segunda fase de una manera síncrona con la interacción del usuario. Mediante una estructuración de los datos se conseguía optimizar el procesamiento posterior y así reducir su tiempo de ejecución. Este método sigue siendo válido en problemas en los que el conjunto de datos es relativamente estático. Esto es, problemas de gestión empresarial, sistemas de gestión de empleados, etc. Entornos en los que, si bien el conjunto de datos puede ser relativamente grande, su evolución en el tiempo es reducida, y el volumen de ingestión de nuevos datos puede ser manejado sin la necesidad de los sistemas distribuidos.

Pese a su gran utilidad, estas arquitecturas clásicas han quedado refutadas para dar solución a problemas donde los cambios en el contenido son tan frecuentes que necesitan de una paralelización en su procesamiento, y a la vez la urgencia en el resultado del cálculo es tal que, si se realiza con minutos u horas de retraso, su utilidad se ve seriamente afectada. Son diversas las fuentes de datos actuales que cumplen con estas características, de entre las cuales se exponen las siguientes:

### ***2.1.a. Datos de monitorización para la operación de sistemas.***

La monitorización de sistemas físicos ha cambiado drásticamente tras la adopción de los sistemas distribuidos en la nube. Hemos pasado de manejar un conjunto pequeño de grandes máquinas concentradas en unas decenas de *clusters*, a monitorizar centenas o miles de pequeñas o medianas máquinas distribuidas en la nube. Cada uno de estos componentes emite datos relevantes para conocer el estado al conjunto del sistema que está desplegado en ellos (uso de memoria, uso

de CPU, uso de disco, ancho de banda de red utilizado, etc.), a la vez que las máquinas físicas emiten datos relevantes de su estado físico (temperaturas, velocidades de los ventiladores, voltajes y amperajes, etc.). Todos estos datos pierden mucho valor si no son procesados en tiempo real, puesto que un problema serio puede ser evitado si se emite una alarma a tiempo.

### ***2.1.b. Analíticas empresariales y publicidad***

La automatización de sectores como el comercio, la banca, la fabricación, y cientos de sectores ha facilitado la recolección de datos de interés para las empresas. Centrándonos en el ejemplo del comercio electrónico, las empresas tienen a su disposición información detallada de compraventa de artículos en tiempo real, lo que ha abierto todo un mundo nuevo de analíticas empresariales.

A nivel de pequeña empresa o comercio local, estos análisis son suficientemente manejables como para poder realizarlos con las arquitecturas clásicas. Pero empresas como Amazon, Samsung, Nike, o similares con un alto volumen de ventas, genera una cantidad de datos que superan las capacidades de las metodologías clásicas.

Además, una mayor velocidad de reacción te puede asegurar enormes beneficios. Llegar el primero al mercado, o entender las necesidades de tus clientes de manera más rápida que tu competencia puede asegurarte el éxito de ventas. Es por esto que el análisis de datos en tiempo real es tan importante en este sector.

### ***2.1.c. Redes sociales***

Las redes sociales han sido una de las mayores impulsoras del uso de datos en el mundo. Con un simple vistazo a las estadísticas de datos generados por redes sociales como Twitter<sup>9</sup>, Facebook<sup>10</sup>, Snapchat<sup>11</sup>, Whatsapp<sup>12</sup>, Instagram<sup>13</sup>, etc. Nos indica la cantidad de datos que pueden ser utilizados para realizar estudios sociales. Cuando miramos la lista de *trending topic* de Twitter ya

---

<sup>9</sup> [twitter.com](https://twitter.com)

<sup>10</sup> [www.facebook.com](https://www.facebook.com)

<sup>11</sup> [www.snapchat.com](https://www.snapchat.com)

<sup>12</sup> [www.whatsapp.com](https://www.whatsapp.com)

<sup>13</sup> [www.instagram.com](https://www.instagram.com)

estamos consumiendo cálculos realizados en tiempo real, que nos dan una visión de qué está aconteciendo en las diferentes regiones del mundo.

En el ámbito periodístico e informativo, así como en el ámbito del entretenimiento estas métricas son bastante útiles para analizar el impacto o el seguimiento de programas televisivos, eventos de ocio o noticias, y adaptarlas para llegar a una mayor audiencia.

### ***2.1.d. El internet de las cosas***

El número y variedad de dispositivos que se conectan para compartir información está creciendo a una enorme velocidad. Hace un par de décadas, internet era un mundo exclusivo para los servidores y los ordenadores personales. Esto ha cambiado drásticamente con la llegada del *smartphone*, y está evolucionando con la inclusión de todo tipo de dispositivos: relojes, *gadgets* deportivos, redes de sensores, drones, electrodomésticos, coches, etc.

Es precisamente en los coches donde podemos observar un gran auge actualmente en las últimas fechas. El avance en el campo de la conducción autónoma se debe en parte al análisis en tiempo real de infinidad de datos. En estos datos no sólo incluimos los recolectados por los sensores propios del coche, sino una red de coches interconectados que comparten información entre ellos para realizar mapas en tiempo real del entorno en los que se encuentran. En el ejemplo de la conducción autónoma, el valor de los datos se pierde por completo si no se realiza un análisis en tiempo real, puesto que, para decidir una ruta óptima basada en el estado actual del tráfico, de poco valen datos atrasados.

## **2.2. Características principales del procesado en streaming**

En todos los ejemplos que hemos visto se generan datos en *streaming*, los cuales, además, urgen ser analizados. Todos ellos comparten una serie de características, e implican características diferenciales de diseño con respecto a los sistemas de procesamiento de datos tradicionales. Las principales se detallan a continuación.

### ***2.2.a. Flujo constante de datos nuevos y alta cardinalidad del conjunto de datos***

Los datos *streaming* siempre están fluyendo. Actualizaciones de valores en sensores, mensajes nuevos en redes sociales, nuevas ventas en una tienda online, etc. Además, suelen crecer

a un ritmo tal que llenarían la capacidad cualquier espacio físico manejable en caso de intentar almacenarlos. La principal consecuencia de esto es que el modelo del conjunto de datos a manejar se supone infinito. Esto conlleva a varias implicaciones a tener en cuenta en la implementación de un sistema de procesamiento de *streams*:

1. Los datos han de ser procesados rápidamente. Un retraso en el procesamiento conlleva un crecimiento en la cola de mensajes a procesar, lo que encarece considerablemente los costes, o incluso rompe la validez del sistema si el ritmo de procesado no equipara el ritmo de ingestión. Por esto, los algoritmos a implementar deben ser lo más ligeros y simples posibles.
2. El sistema tiene que estar constantemente en funcionamiento. Esto implica que un fallo en una parte del sistema debe estar lo más aislado posible, permitiendo al resto seguir con su función. Se deben implementar soluciones que conlleven a una alta disponibilidad del sistema y a la maximización de la resiliencia, o resistencia a errores. Esto se consigue mediante el aislamiento del entorno de ejecución de los diferentes componentes, así como la presencia de planes de extinción, o salvaguardia en caso de error. Un ejemplo de salvaguardia sería una vía alternativa de ejecución en paralelo que se activase en caso de error, o un sistema de respaldo que permita al sistema recuperarse sin pérdida de datos que hubiera podido tener durante el tiempo de inactividad.
3. La corrección y completitud de los cálculos no se puede alcanzar. Al estar frente a un modelo donde el conjunto de datos es infinito, no disponemos de la totalidad en ningún momento, y por tanto muchas métricas deben ser aproximadas. Un ejemplo sería el cálculo de percentiles. Pero incluso en casos en los que matemáticamente es posible, en la práctica, al estar tratando con sistemas distribuidos, estamos expuestos a retrasos en la entrega, o incluso fallos completos. En la mayoría de los casos no existe el momento en el tiempo en el que podamos asegurar que hemos recibido todos los datos dentro de un rango de tiempo, por lo que conlleva no poder garantizar la completa corrección de los cálculos.

### ***2.2.b. Poca estructuración de los datos***

Los datos se suelen obtener de fuentes diversas, cada fuente con su formato y estructura concreta. Los sistemas de origen, además, no suelen estar disponibles para modificación. Encontramos ejemplos en las redes sociales o las métricas de monitorización de sistemas.

Es por esto que los sistemas deben estar diseñados para trabajar con datos no estructurados. Las características a analizar deben ser extraídas realizando el mínimo acoplamiento a la estructura de la fuente de datos, y el sistema debe ser lo suficientemente flexible como para adaptarse a nuevas estructuras en el futuro sin necesidad de cambios mayores en el código.

## **2.3. Arquitecturas Big Data para el procesamiento de flujos de datos**

Cuando encontramos un problema de rendimiento en nuestro sistema, una de las soluciones disponibles es el escalado. Escalar significa aumentar la potencia disponible de nuestro sistema, y se puede realizar de dos maneras:

- **Escalado vertical.** El escalado vertical aumenta la potencia del sistema mediante el aumento de la capacidad individual de cada máquina. Escalaríamos verticalmente, por ejemplo, si cambiamos el procesador por un modelo más potente, aumentamos la cantidad o velocidad de memoria RAM, o la cantidad de disco.
- **Escalado horizontal.** Un escalado horizontal se refiere a un aumento de potencia mediante un aumento del número de máquinas dentro de un sistema distribuido. A diferencia del escalado vertical, y gracias a las máquinas virtuales, su coste es más bajo, y su ejecución más sencilla.

Las arquitecturas clásicas para el manejo de datos, las basadas en esquemas SQL, han fallado al escalar a tamaños de datos considerados *Big Data*. La principal causa del fracaso de las tecnologías relacionales es que no fueron diseñadas para trabajar en entornos distribuidos, su modelo de escalado se corresponde más con el escalado vertical que con el horizontal [3]. Con la reciente implantación de la nube, y la necesidad de escalar horizontalmente, los sistemas NoSQL han desbancado a los SQL para procesamiento masivo de datos.

Una nueva serie de tecnologías han aparecido en escena para solucionar este problema, las denominadas no relacionales o NoSQL. Su principal característica diferenciadora es la enorme capacidad de escalado horizontal ya que están diseñadas desde el primer momento para ser desplegadas como sistemas distribuidos. Para lograrlo, simplifican el complejo modelo de las bases de datos relacionales, introduciendo algunos compromisos. En estos sistemas existe un cambio de perspectiva en el diseño, la normalización de los datos no es una prioridad, ni tampoco la eficiencia en cuanto a volumen de almacenamiento utilizado. En cambio, se prioriza la eficiencia de consulta, diseñando los esquemas de almacenamiento para ajustarse a los casos de uso. Así pues, pasamos de diseños genéricos, basados en relaciones de entidades que dan cabida a cualquier tipo de consulta, a diseños específicos para consultas previamente conocidas, con pocas o ninguna relación entre entidades.

Si nos vamos a los orígenes teóricos de qué es un sistema que trabaja con datos, podemos modelarlo como una función de la siguiente manera:

$$\text{datos de salida} = \text{función}(\text{datos de entrada})$$

Los sistemas relacionales implementan este modelo almacenando los datos de entrada en estructuras de entidades y relaciones. Sobre esta estructura, se define un lenguaje con el cual se pueden implementar una gran variedad de funciones, dependiendo de los datos de salida que se deseen. Este lenguaje en la práctica se ejecuta de forma síncrona en el instante en el que el usuario realiza una consulta, dando lugar a una gran flexibilidad a cambio de un tiempo de respuesta relativamente alto.

En cambio, los sistemas NoSQL, en concreto los utilizados para el procesamiento de *streams*, dividen esta función en dos fases:

$$\text{vista agregada} = \text{función de agregación}(\text{datos de entrada})$$

$$\text{datos de salida} = \text{función de visualización}(\text{vista agregada})$$

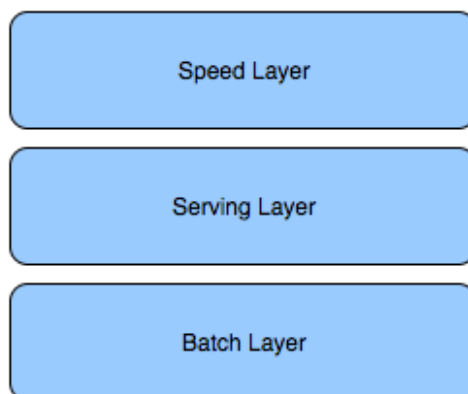
La función de agregación se realiza de manera continua sobre los datos, y su resultado se almacena en forma de vista. Esta vista está optimizada para las distintas funciones de visualización. El usuario, cuando realiza una consulta, especifica la función de visualización. Esta función no se

aplica sobre los datos originales de entrada, sino sobre una vista agregada. Una vista agregada corresponde con un conjunto de valores calculados en agrupaciones de varios valores originales. Por ejemplo, una agregación temporal define una ventana temporal, y agrupa todos los datos de esa ventana en un único valor, o conjunto de valores, siendo éstos una cuenta, una media, o cualquier otro valor estadístico del conjunto. Al tratar con un volumen inferior al original, las consultas sobre vistas agregadas son más rápidas que las consultas sobre los datos originales. Por otro lado, se pierde flexibilidad, ya que perdemos parte de la información detallada que nos proveen los datos originales.

### ***2.3.a. Arquitectura Lambda***

La Arquitectura Lambda [4] pretende dar solución al problema del análisis en tiempo real de un flujo de grandes cantidades de datos. Se trata de una arquitectura no relacional que se basa en la idea de las funciones de agregación que hemos visto en la sección anterior.

Esta arquitectura define una serie de capas de procesamiento enfocadas a dar cabida tanto a los requisitos de agilidad y velocidad de cálculo, como a los requisitos de alta precisión en los mismos. Las capas quedan definidas de la siguiente manera:



*Figura 2 Capas de la Arquitectura Lambda*

La *Batch Layer*, o capa de lotes, realiza la función de agregación. Se ejecuta periódicamente sobre todo el conjunto de los datos disponibles, dando como resultado el mejor de los resultados en cuanto a exactitud posibles en el sistema. Debido a la gran cantidad de datos que ingiere en cada ejecución, su procesamiento suele ser lento, lo que provoca que los datos de salida se encuentren con un retraso que suele rondar el orden de minutos u horas.

La *Serving Layer* se encarga de hacer disponible los cálculos de la *Batch Layer* al usuario. Esta capa almacena los resultados en una base de datos que soporta actualizaciones por lotes, y acceso aleatorio a los mismos. Con la llegada de un nuevo lote o *batch* se realiza la sobrescritura de los datos anteriores. De esta manera, esta capa se encarga de tener una vista actualizada del proceso de la *Batch Layer*.

Con el uso de estas dos capas únicamente, se podría dar solución a problemas en los que no se requiera de resultados rápidos, pero en cambio sí que se necesite un cálculo exacto. Por ejemplo, problemas de cálculo de tarificación en un sistema de pago por uso. En este caso, el cálculo de la cantidad a pagar debe ser lo más exacto posible, mientras que no es un problema el disponer de la cantidad con un cierto retraso, ya que la facturación se suele hacer de manera diferida.

La *Speed Layer* aparece para dar cabida al problema de cálculo en tiempo real. Esta capa es la encargada de servir los datos que se encuentran en el rango temporal entre la última actualización de la *Batch Layer* y la consulta realizada por el usuario. Esta capa se caracteriza por ser mucho menos precisa, a cambio de proveer con resultados en tiempo real. Los datos calculados en ella tienen fecha de caducidad, ya que se verán reemplazados con los más exactos resultados de la *Batch Layer* tan pronto como estén disponibles.

El usuario del sistema accede pues a una mezcla de los datos proporcionados por la *Serving Layer*, y la *Speed Layer*.



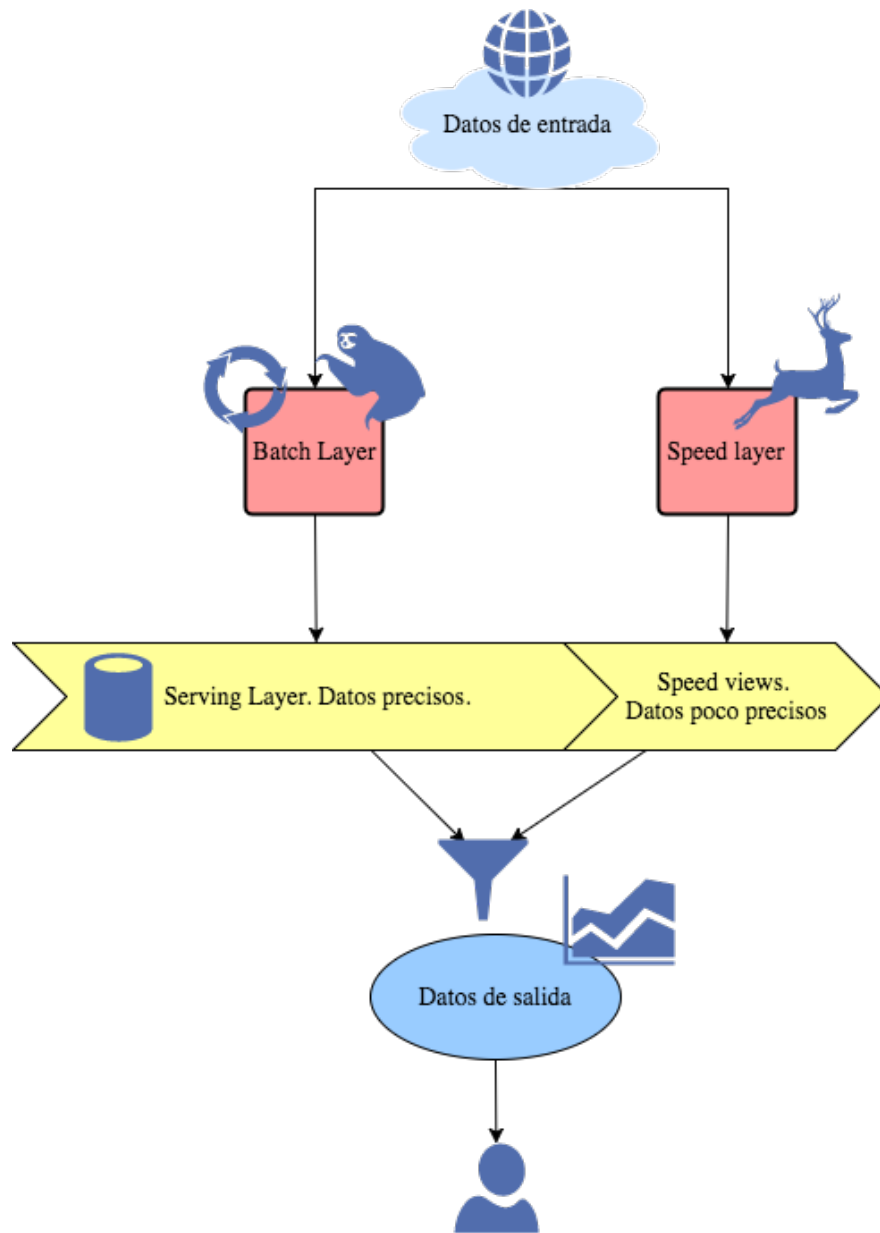


Figura 3 Arquitectura Lambda

Así pues, para dar solución a nuestra función sobre los datos, la Arquitectura Lambda propone la siguiente descomposición:

$$vista\ batch_t = agregación(datos_t)$$

$$vista\ speed_t = agregación(vista\ speed_{t-1}, datos_t - datos_{t-1})$$

$$datos\ de\ salida_t = combinación(vista\ batch_t, vista\ speed_t)$$

Esta arquitectura presenta una serie de ventajas que se enumeran a continuación:

- Disponemos de datos en tiempo real gracias a la *Speed Layer*, así como datos precisos gracias a la *Batch Layer*.
- Los posibles errores en el cómputo son fácilmente solucionables mediante el reproceso. Tanto la *Batch Layer* como la *Speed Layer* son datos en continua actualización. La *Speed Layer* se verá reemplazada por los cálculos de la *Batch Layer* tan pronto como ésta termine, y la *Batch Layer* está continuamente sobrescribiendo sus propios cálculos, con lo que encontrarse con un error en el código, el simple hecho de desplegar el código con la corrección hará que en la siguiente iteración dispongamos de los datos correctos.
- Las capas asíncronas de recolección y de procesado de datos, junto con la simplicidad de las bases de datos necesarias, hace que el sistema sea fácilmente escalable. Además, al calcular continuamente datos en dos vías distintas, disponemos de una tolerancia a fallos mayor, puesto que de fallar una de ellas, tendremos una salvaguardia.

Pero no son todo ventajas, es fácil encontrar una serie de inconvenientes que la hacen bastante complicada de desplegar en la práctica:

- La *Batch Layer* debe procesar una cantidad enorme de datos con periodicidad, esto implica un despliegue bastante costoso de infraestructura.
- Los mismos datos realizan cálculos similares en dos vías distintas del código. Esto conlleva mantener dos implementaciones de la capa de análisis, que suelen contener similares algoritmos. Además, normalmente las tecnologías y lenguajes utilizados en ambos son diferentes, lo que penaliza en gran medida la mantenibilidad del código.
- Los datos son recalculados continuamente. La *Batch Layer* ejecuta el conjunto total de datos en cada iteración, repitiendo los mismos cálculos una y otra vez. De igual manera, la *Speed Layer* ejecuta cálculos que tarde o temprano serán de nuevo calculados por la *Batch Layer*. Nos encontramos así con problemas de eficiencia en el uso de recursos.

Cuando los inconvenientes pesan más que las ventajas, nos encontramos con que la arquitectura no es un buen ajuste a nuestro problema. Pongamos el ejemplo del análisis de datos en tiempo real donde no es crítica su exactitud y podemos dar por bueno pequeños errores de cálculo. Para estos casos apareció, como alternativa y simplificación de la Arquitectura Lambda, la Arquitectura Kappa.

### ***2.3.b. Arquitectura Kappa***

La Arquitectura Kappa [5] [6] nace como una simplificación de la Arquitectura Lambda, en la que desaparece la *Batching Layer* para darle una mayor importancia a la *Speed Layer*. Es una arquitectura que se suele adoptar cuando es más importante el recibir los datos en tiempo real que recibir los valores exactos de los cálculos. Así pues, esta arquitectura se compone únicamente de una *Speed Layer* con algunas características diferenciales:

La *Speed Layer* en este caso debe estar preparada para poder recomputar los datos de entrada del pasado. Al carecer de *Batching Layer*, en caso de encontrarnos un error de cómputo, debemos mantener la capacidad de sobrescribir el cálculo. Este recómputo se realiza lanzando una segunda instancia del Agregador en paralelo con la ya existente, pero accediendo a datos de un instante temporal anterior.

Dado que necesitamos recómputo, los datos de entrada deben ser almacenados en un sistema de colas que permita la ejecución de diferentes consumidores no sincronizados. Así pues, necesitamos de dos consumidores simultáneos que lean mensajes correspondientes a distintos momentos en la línea temporal sobre la misma cola. La tecnología que más se adapta a este comportamiento son los sistemas de publicación y suscripción, donde el índice de lectura se mantiene en el cliente.

En el momento del recómputo, se inicia una segunda instancia que lee de la misma cola de datos de entrada, pero con un índice atrasado en el tiempo. Los datos de salida se redireccionan a una segunda vía, paralela a la original. De esta forma, el sistema puede seguir procesando los datos de la salida original, y a la vez procesar en paralelo en la segunda vía los datos procedentes del recómputo.

En cuanto el recómputo ha alcanzado en el tiempo a la vía principal, ésta se detiene, y se hacen los ajustes de redirección necesarios para pasar a tratar únicamente los datos de la vía del recómputo.

El diagrama de la Arquitectura Kappa, funcionando con un Agregador de recómputo queda como muestra la Figura 4.

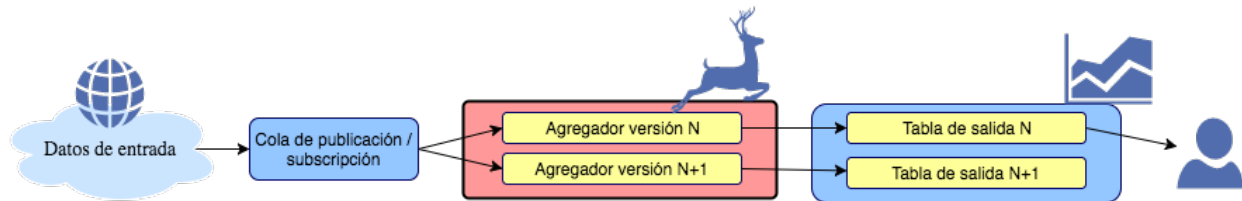


Figura 4 Recómputo de datos en la Arquitectura Kappa

La Arquitectura Kappa presenta sus ventajas e inconvenientes con respecto a la Arquitectura Lambda. Como ventajas destacan dos:

- El **ahorro de costes** al limitar el recálculo a los momentos estrictamente necesarios.
- La **simplificación de mantenimiento**, ya que no tenemos duplicidad de código en dos tecnologías distintas, sino un único código que se ejecuta sobre datos con momentos temporales distintos.

Y como inconvenientes podemos resaltar otro par:

- El recómputo requiere de una **ejecución excepcional** de nuestro sistema. Deja de ser algo que está cubierto dentro del funcionamiento normal, para ser un protocolo de actuación en caso de problemas.
- **Se pierde exactitud**. Al no poder computar de manera periódica el cálculo global, perdemos el efecto “coche escoba” que la *Batch Layer* realiza en la arquitectura Lambda.

La Arquitectura Kappa se ha convertido en un referente para problemas como el manejo de logs de aplicaciones, así como el cálculo de métricas de monitorización de las mismas.

## 2.4. El momento temporal de un evento

Cuando tratamos con datos en *streaming* el concepto de tiempo es muy importante. Los datos finales suelen presentarse mediante variaciones de valores en el tiempo, es por esto que es conveniente definir qué se entiende por tiempo de un evento en estos sistemas.

Las soluciones Big Data suelen tener una arquitectura asíncrona basada en colas. Mientras que las arquitecturas síncronas realizan llamadas directas a los componentes que bloquean su ejecución a la espera de una respuesta. Por otro lado, las arquitecturas síncronas se comunican mediante la publicación de mensajes a través de una cola. En la Figura 5 podemos ver una simplificación de la arquitectura asíncrona de colas a la que nos referimos.

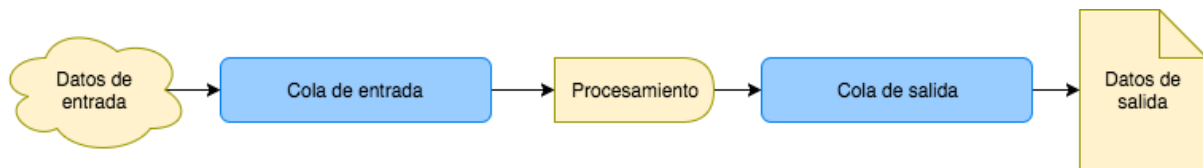


Figura 5 Arquitectura Asíncrona de los sistemas Big Data

Para determinar el tiempo de un evento tenemos 3 posibles puntos:

1. El tiempo de generación del evento en el **origen de los datos** de entrada. Es el tiempo más exacto para realizar cálculos, pero no siempre disponemos de él. Es posible que los datos de origen carezcan de indicación temporal, que la precisión del valor no sea suficientemente buena, o que lleguen eventos retrasados en el tiempo que nos obliguen a esperar para realizar el cálculo de la ventana temporal correspondiente.
2. El instante de tiempo de **inserción en la cola de entrada**. En este caso el valor está bajo nuestro control, por lo que podemos garantizar su presencia y exactitud. En caso de no disponer del valor en el origen es el valor más exacto que disponemos. No obstante, puede introducir errores en el caso en el que los eventos de origen lleguen con cierto retraso o no ordenados. Suele ser el tiempo que se utiliza en la mayoría de los sistemas que no precisan de una exactitud en sus cálculos.
3. El tiempo de **procesamiento del evento**. Este es el cálculo menos exacto de todos, pero tiene sus ventajas. En sistemas en los que no es crítica la corrección de los datos, pero

sí la velocidad de respuesta del sistema, es el tiempo óptimo a utilizar. Los casos anteriores requieren de un tiempo de espera prudente para asegurarnos de que hemos recibido todos los eventos correspondientes a una ventana temporal. En este caso, la ventana temporal se genera en tiempo de ejecución y puede dar salida a los resultados de una manera mucho más fluida y sin retrasos. En cambio, la necesidad de un reprocesado conllevaría un error quizás demasiado elevado al mover datos del pasado hacia el presente en la representación.

Otros momentos temporales como el tiempo de inserción en la cola de salida o el tiempo de presentación en los datos de salida no cabría contemplarlos, puesto que la agregación tiene en cuenta este valor temporal, y se produce en la fase de procesamiento.

En la Arquitectura Lambda se usan típicamente dos temporizaciones distintas. Para la *Batch Layer* es el valor temporal del origen de los datos el que se tiene en cuenta, en caso de existir, ya que es el valor que logra mayor exactitud en los cálculos. En cambio, en la *Speed Layer* suele optar por el tiempo de inserción en la cola de entrada, puesto que la velocidad es lo más importante para este cálculo. Es más, existe la posibilidad de usar el tiempo de procesamiento del evento, ya que en la *Speed Layer* no se van a realizar reprocesados, ya que de esto se encarga la *Batch Layer* en exclusiva.

En cambio, en la Arquitectura Kappa, la *Speed Layer* puede dedicarse ocasionalmente a recalcular datos del pasado, ya que no disponemos de la *Batch Layer*. Es por esto por lo que el tiempo de procesamiento del evento no es utilizable. De entre las dos opciones restantes, se utilizará el tiempo de inserción cuando sea más prioritaria la velocidad que la corrección, y se hará uso del tiempo en el origen cuando se requiera de un cómputo más exacto.

# *Capítulo 3. Tecnologías y lenguajes utilizados*

Este capítulo presenta las tecnologías utilizadas en la implementación de este proyecto fin de carrera. Presenta tanto una descripción general de cada una de ellas, como una explicación un poco más detallada de la función que cumplen dentro de la arquitectura del software implementado.

## **3.1. Scala**

Scala es un lenguaje de programación que une dos familias de lenguajes: la programación funcional y la programación orientada a objetos.

La adopción y el uso de Scala se ha incrementado en el mundo del procesamiento de datos desde la salida de Spark, uno de los principales framework de procesamiento paralelo utilizados en Big Data. Además de contar con Spark escrito en Scala, este lenguaje presenta una serie de características que facilitan el desarrollo de software.

El lenguaje incluye de forma nativa la **programación orientada a objetos**, de manera que la estructuración del código y la reutilización se convierte en tarea sencilla. Modela de forma directa el ‘objeto’ y permite la herencia múltiple, algo de lo que su principal competidor Java carece.

Además de la programación orientada a objetos, implementa de forma nativa la **programación funcional** permitiendo la definición de programas sin estado o *stateless*. Esto es muy importante ya que, al no disponer de estado, los sistemas distribuidos limitan el uso de técnicas de sincronización y de protección de acceso a datos como los *mutex* o los semáforos. Centrándonos en los sistemas Big Data, son sistemas que deben ser simples, como ya vimos en el Capítulo 2. La mayoría de los sistemas de procesamiento de datos implementan una arquitectura Map-Reduce [7], que define el procesado de los datos como una serie de funciones de transformación y reducción, que son fácilmente modelables en un lenguaje funcional.

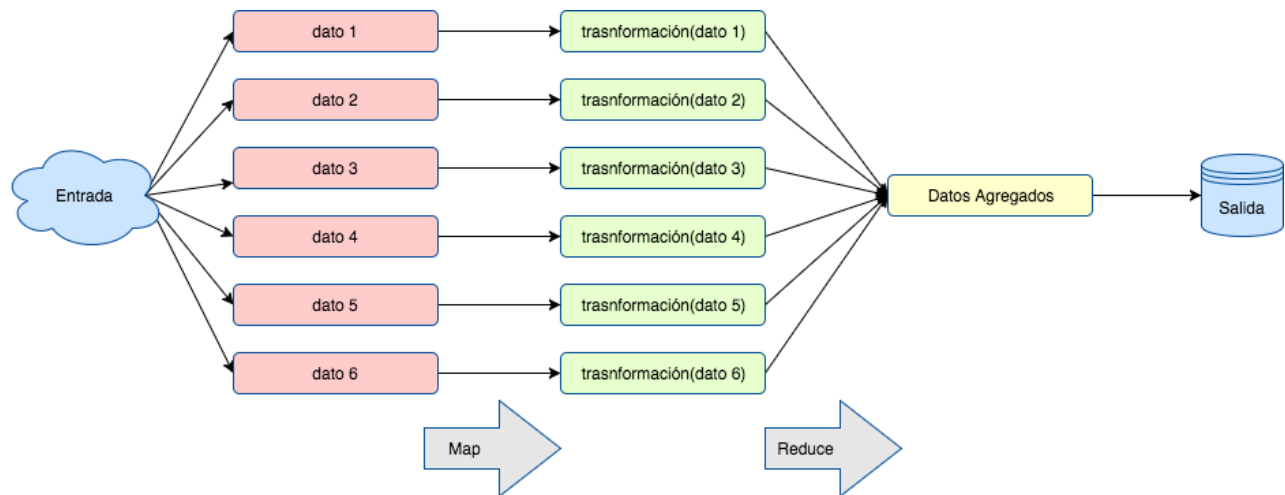


Figura 6 Arquitectura Map-Reduce

Scala es un lenguaje interpretado, que se ejecuta sobre la **Máquina Virtual de Java**. Por lo tanto, cuenta con todas las ventajas que Java posee en cuanto a portabilidad y soporte. Además, en Scala se puede hacer uso de las librerías existentes de Java, con lo que tenemos una gran serie de utilidades maduras a nuestra disposición.

Por último, la **sintaxis** de Scala es bastante simple. Se basa en Java y elimina de éste todos los caracteres y requerimientos que hacen de Java un lenguaje verboso. A parte de eliminar los punto y coma del final de línea, un ejemplo muy práctico está en la definición de constructores. Cuando se implementa el patrón de inyección de dependencias a través del constructor para incrementar la reusabilidad, Scala permite definir valores por defecto, lo que simplifica la interfaz de creación de los objetos, dejando la verbosidad para casos excepcionales.

## 3.2. Twitter Stream API

La fuente de datos que riega el sistema implementado es Twitter. Twitter es una red social en la que los usuarios pueden escribir ‘Tweets’, que son mensajes de texto cortos (280 caracteres a fecha de la publicación del presente documento) que pueden incluir referencias a otros usuarios, enlaces web, y *hashtags*. Su uso está muy distribuido, y a fecha actual el tráfico ronda los 6000 o 7000 mensajes por segundo [8].

Para consumir los datos de la red social, Twitter dispone de una API pública a la que podemos conectar nuestro sistema. Para los distintos recursos disponibles en la API dispone de



una versión gratuita y de una comercial. En este proyecto se ha decidido hacer uso de la versión gratuita, que tiene las siguientes características:

- Los datos se envían mediante una conexión HTTP que queda constantemente abierta. Actúa de manera similar que una descarga de un fichero de tamaño infinito.
- La autenticación se realiza mediante un *token* OAuth<sup>14</sup> generado desde una cuenta de Twitter. OAuth es un protocolo de autenticación y autorización basado en *tokens* orientado a aplicaciones web.
- Se limita el número de conexiones paralelas a una por cliente.
- El tráfico de datos enviado desde la API está limitado al 1% de todos los eventos generados en la red social. Esto es, a fecha de la implementación del proyecto, alrededor de 70 mensajes por segundo.

Una vez abierta la conexión, la API envía un stream continuo de mensajes en formato Java Script Object Notation (JSON). Los mensajes se categorizan en dos, atendiendo a su estructura:

- Mensajes de borrado de ‘tweets’. Indican que un usuario ha borrado un tweet de la plataforma.
- Mensajes de publicación de ‘tweets’. Indican que un usuario ha publicado un tweet en la plataforma.

Ejemplos de ambos mensajes se pueden encontrar en el Apéndice 1.

Para el uso de esta API, se ha usado la librería *Hosebird Client*<sup>15</sup>. Una librería java de código abierto que abstrae el manejo de la autenticación y el manejo de la conexión de la API.

---

<sup>14</sup> [oauth.net/2](https://oauth.net/2)

<sup>15</sup> <https://github.com/twitter/hbc>

### 3.3. Kafka

Kafka es una tecnología que ha crecido en popularidad en los últimos años. Se trata de un sistema distribuido de mensajería, que implementa la arquitectura pub-sub o publicación-subscripción. Cuenta con un diseño muy simple, el cual se detalla a continuación, que le permite escalar fácilmente y manejar grandes cantidades de ingestión de datos.

#### 3.3.a. Conceptos y arquitectura de Kafka

Kafka implementa un modelo de publicación subscripción. Un mensaje en Kafka está representado por una clave, un valor y un *timestamp*. Divide sus colas en *topics*, a los cuales pueden conectarse diversos publicadores y subscriptores. Cada uno de los *topics* es independiente de los demás, y se distribuye entre los nodos, denominados *brokers* en forma de particiones y réplicas. Cada *topic* tiene un número determinado de particiones que distribuyen el contenido del mismo, y define un nivel de replicación, que indica el número de réplicas por partición. Cada nodo recibe una serie de réplicas, dependiendo del factor de replicación elegido. Independientemente del factor, cada partición dispone de una única réplica maestra, que es la encargada de recibir las escrituras y las lecturas.

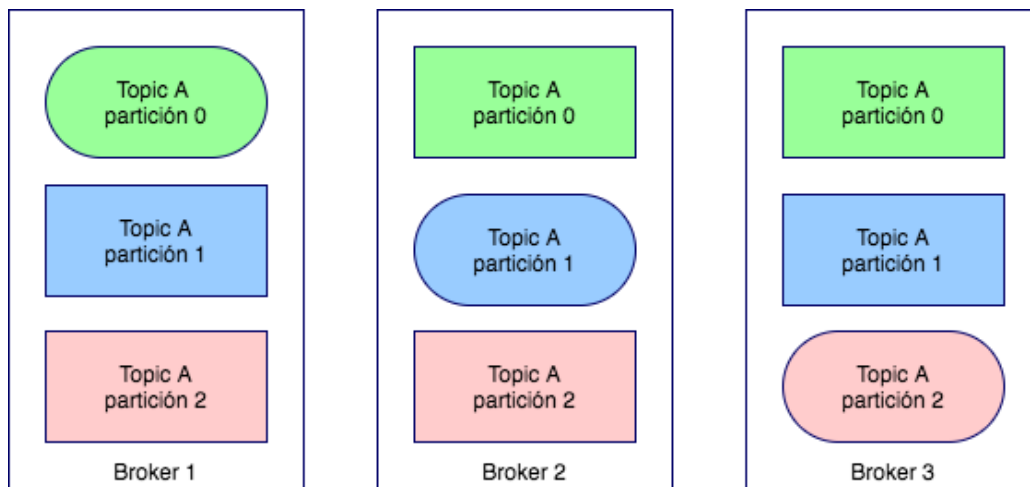


Figura 7 Distribución de brokers, particiones y topics en Kafka

En la Figura 7 podemos ver la distribución de un *topic* (Topic A) con tres particiones, y factor de replicación tres en cada partición. Las réplicas con forma redondeada representan la réplica maestra dentro de cada partición. Esta distribución de los datos sería la distribución óptima para un despliegue con tres nodos, ya que se distribuye la carga de cada *topic* en un nodo diferente,

y todos los *topics* tienen una copia en todos los nodos, lo que ayuda a recuperarse fácilmente en caso de fallo en uno de ellos. En caso de error de una de las réplicas maestras, el sistema elige la nueva maestra de entre las réplicas restantes que se encuentren sincronizadas.

Para entender cómo se escriben y se leen los datos debemos entender los conceptos de productor y grupo de consumidores:

- Un **productor** es un componente que escribe mensajes en uno o varios *topics* de Kafka. El productor decide, basado en la clave del mensaje, en qué partición escribe cada uno de ellos. Realiza la escritura en la réplica maestra, y ésta la distribuye a todas sus réplicas.
- Un **grupo de consumidores** representa al conjunto de lectores de un *topic*. Cada grupo distribuye entre sus consumidores las particiones, de manera que, en conjunto, se realiza la lectura de todas ellas. Al igual que la escritura, la lectura sólo se puede realizar en la réplica maestra de cada partición. Para la lectura, cada grupo de consumidores lee de las réplicas maestras de todas las particiones. Si la distribución de líderes de las particiones es equitativa, como vimos en la Figura 7, el ancho de banda es aprovechado al máximo por los consumidores. La lectura de una partición está limitada a un único consumidor por grupo. No obstante, es posible crear varios consumidores que lean de una misma partición, si están en distintos grupos de consumidores.

En la siguiente imagen se muestra el funcionamiento de las escrituras y las lecturas en las réplicas de un *topic*, sin tener en cuenta en qué *broker* se encuentran.

Como podemos observar en la Figura 8, las escrituras y las lecturas se realizan sobre la réplica maestra de cada partición. El productor escribe a cada una de las particiones, distribuyendo la carga y contenido de esta forma. Cada partición tiene asignada un único consumidor dentro de un grupo de consumidores, y en conjunto éstos leen el total del contenido del *topic*, ya que abarcan todas las particiones del mismo.

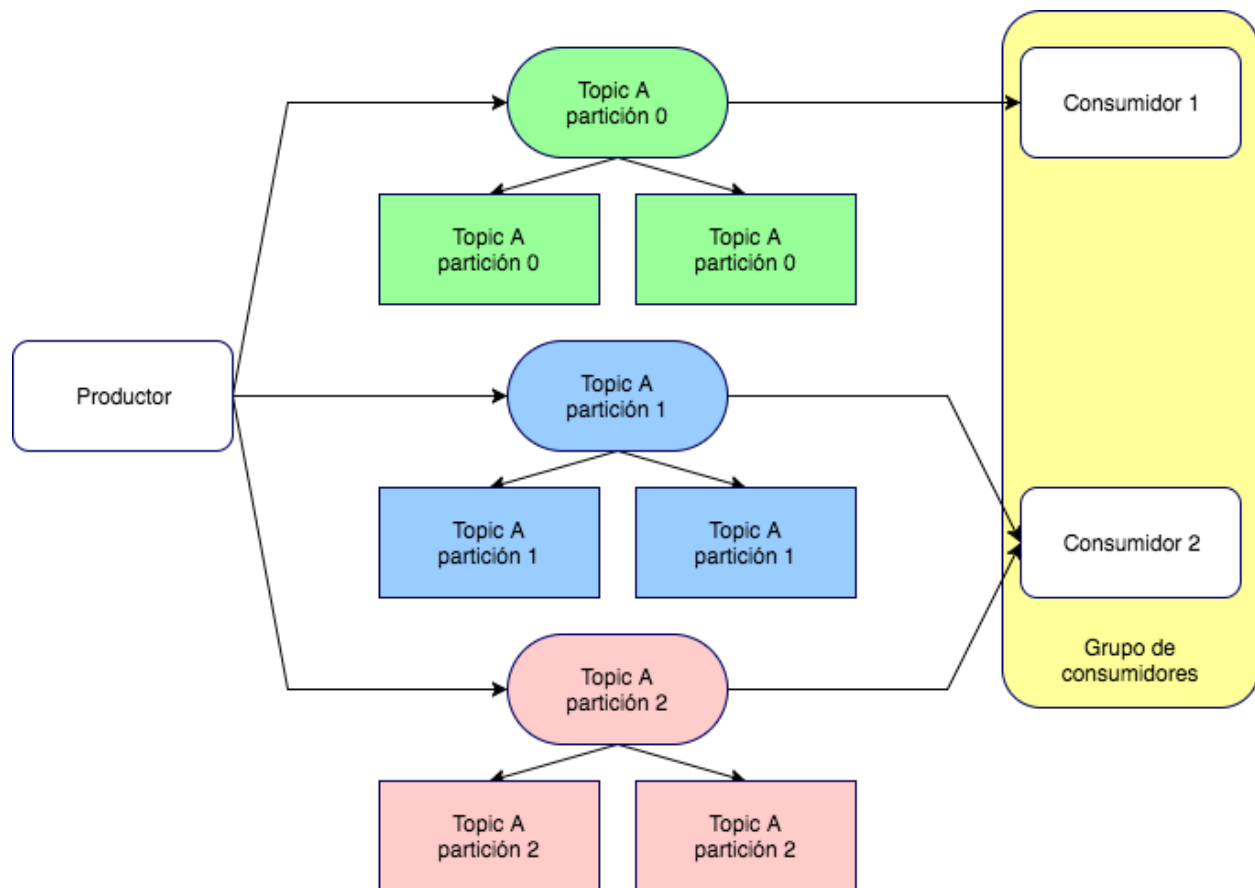


Figura 8 Particiones y réplicas en Kafka

Kafka necesita de otro sistema distribuido para su funcionamiento, y este es Zookeeper. Se trata de un sistema de distribución de configuración, que Kafka usa para distribuir información como cuál es la réplica líder, qué réplicas están sincronizadas, qué *broker* contiene cada réplica, etc.

### 3.3.b. APIs de Kafka

Kafka provee una serie de APIs para el usuario. Cada una de ellas va destinada a un caso de uso concreto, y tiene sus características:

- **Producer API.** Es la encargada de exponer la funcionalidad de los productores. Esta es la API que debemos usar en los componentes necesiten escribir mensajes a Kafka.
- **Consumer API.** Es la encargada de exponer la funcionalidad de los consumidores. Se usará en los componentes que necesiten leer mensajes de Kafka.

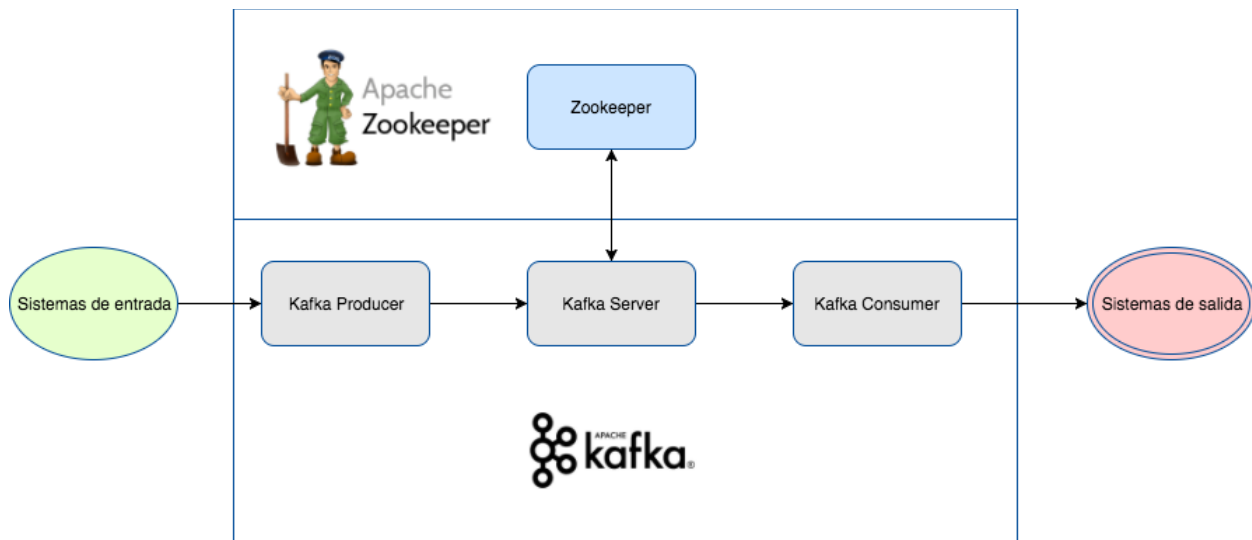


Figura 9 Elementos de Kafka

- **Streams API.** Es una API de reciente creación, disponible únicamente desde la versión 10.0 de Kafka. Al contrario que las dos anteriores, cuenta con una complejidad mayor y está destinada a aplicaciones que necesiten sincronizarse mediante el paso de mensajes. Hace uso tanto de los consumidores como de los productores, y modela el uso de Kafka como un flujo de datos distribuidos. Abstrae el manejo de particiones de manera que se gestionan de forma transparente al usuario, facilitando la escalabilidad de las aplicaciones. Se ha presentado como plataforma alternativa a las clásicas de procesamiento de streams de datos como Apache Spark Streaming<sup>16</sup>, Apache Storm<sup>17</sup> o Apache Flink<sup>18</sup>.
- **Connector API.** Está enfocada a usar Kafka como un gran sistema de distribución y caché intermedia de otros sistemas. Expone funcionalidad que facilita la conexión de diferentes sistemas tanto de entrada como de salida a Kafka.

En este proyecto se ha hecho uso de las tres primeras APIs, esto es, de Producer API, de Consumer API, y de Streams API.

<sup>16</sup> [spark.apache.org/streaming](http://spark.apache.org/streaming)

<sup>17</sup> [storm.apache.org](http://storm.apache.org)

<sup>18</sup> [flink.apache.org](http://flink.apache.org)

## 3.4. Kafka Streams

La Streams API es la utilizada en la parte de mayor complejidad del proyecto, y por tanto merece una sección para explicar su funcionamiento en detalle.

Kafka Streams simplifica el uso de los productores y consumidores de Kafka en aquellas aplicaciones que pretenden hacer una transformación sobre grandes volúmenes de datos.

### 3.4.a. Topología de procesado

Lo que en otros sistemas de procesamiento de datos se conoce como conjunto de transformaciones sobre los datos, Kafka Streams lo denomina topología. La topología se compone de una serie de pasos, aplicando ciertos procesadores. Para modelar el flujo de datos, la librería hace uso de dos términos: Tablas y Streams.

- Un **Stream** es una secuencia de cambios de valor asociado a cierta clave. Si un valor cambia, encontraremos dos elementos en nuestro Stream, uno con el valor anterior, y uno más reciente con el valor modificado.
- Una **Tabla** es la representación estática del instante actual de nuestro conjunto de claves y valores. Así pues, si un valor cambia en el tiempo, en la tabla únicamente tenemos un par clave-valor que representa el estado actual.

En el momento en el que introducimos agregaciones temporales, el resultado lo podemos modelar mediante una tabla sobre la que vamos actualizando el valor. Si en cambio queremos procesar evento a evento, nuestro modelo de datos será en este caso un Stream.

En la topología encontramos diferentes tipos de procesadores:

- Procesadores **fuentes de datos**. Son los orígenes de datos de la aplicación, no cuentan con ningún procesador por encima en la jerarquía. Su misión es leer datos de los *topics* y pasarlos a los consecutivos procesadores.

- Procesadores **finalizadores**. Son los sumideros de nuestros datos. Normalmente encontramos en ellos la escritura del resultado final del proceso de nuestra aplicación. Su misión es escribir los datos de salida del sistema al *topic* de salida.
- Procesadores **Stream-Stream**. Son procesadores que reciben un Stream, y realizan transformaciones evento a evento. Un ejemplo sería un filtrado, donde podemos aplicar el filtro a cada uno de los eventos de manera separada.
- Procesadores **Stream-Tabla**. Son procesadores que reciben un Stream, y realizan transformaciones de agrupación o agregación. Un ejemplo sería una función Reduce donde calculamos datos agregados en ventanas temporales.

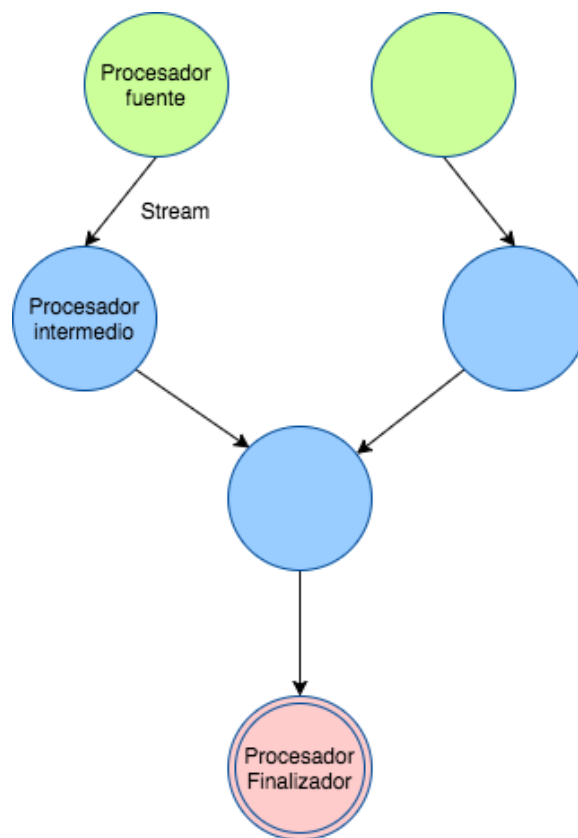


Figura 10 Topología de Kafka Streams

En la Figura 10 observamos una representación en forma de grafo de una topología. Los nodos representan procesadores, mientras que las aristas representan Streams. En color verde los procesadores fuente, en color rojo los procesadores finalizadores, y en color azul los procesadores intermedios, que pueden ser tanto Stream-Stream como Stream-Tabla.

### ***3.4.b. Agregaciones temporales y ventanas***

En todo flujo de datos de análisis que se precie encontramos agregaciones temporales, que son los cálculos que dan valor añadido al sistema de procesamiento de eventos. Kafka Streams modela las agregaciones en forma de Tabla. Como se explicó con anterioridad, una agregación no es más que el valor en un instante de tiempo en una ventana temporal de una métrica en concreto.

Como vimos en el Capítulo 2, el instante de tiempo es algo muy importante a tener en cuenta a la hora de realizar las agregaciones temporales en un sistema de eventos. Cómo tratar con eventos atrasados ha sido un tema muy discutido en los últimos años. La solución de Kafka Streams a esto viene dada también por el funcionamiento de la dualidad Tabla-Stream.

Una agregación de una ventana temporal se modela como una celda en una tabla, con cada nuevo evento del Stream que llega, se calcula el valor actualizado del mismo, de manera que la tabla representa en todo momento el valor más actualizado para el instante de tiempo actual de la métrica. Esto requiere de un estado, ya que debemos almacenar el valor actual de alguna manera para poder agregar un valor nuevo que llega del Stream.

En las agregaciones con ventanas temporales, que son las que nos interesan en este proyecto, es en la propia ventana donde decidimos el tiempo en que la ventana va a estar abierta a nuevos cambios. Una vez ese tiempo expira, si recibimos valores con timestamp correspondientes a la ventana serán descartados, por lo que perderemos datos. El tiempo de persistencia de la ventana queremos en nuestro sistema es una decisión de diseño. Una ventana grande implicaría un estado persistido o en memoria grande, mientras que una ventana pequeña implicaría que nuestro sistema perdería datos con mayor facilidad en caso de recibir eventos con un timestamp retrasado.

### ***3.4.c. Paralelización y recálculo del cómputo***

Kafka Streams define un número de hilos que ejecutará en paralelo la topología. El número de hilos dependerá del número de particiones de los *topics* de los procesos fuente. Si tenemos un único *topic* con tres particiones, el número de hilos que el sistema asignará a nuestro procesado será de tres.



Cada aplicación define un identificador (ID) de aplicación. Este ID se corresponde con el nombre del grupo de consumidores. Como vimos en la sección 3.3.a, el conjunto de los consumidores de un mismo grupo se distribuyen el total de las particiones de un *topic*.

Si ejecutamos dos instancias de la misma aplicación con distinto ID, ambas leerán del mismo *topic* en instantes de tiempos desfasados. Cada instancia, cuando se ejecute, leerá desde el principio del *topic*, lo que nos permite realizar recómputos.

En cambio, si ejecutamos dos instancias de la misma aplicación con el mismo ID, ambas se sincronizan para repartirse los hilos de ejecución, con lo que conseguimos escalar horizontalmente de manera sencilla.

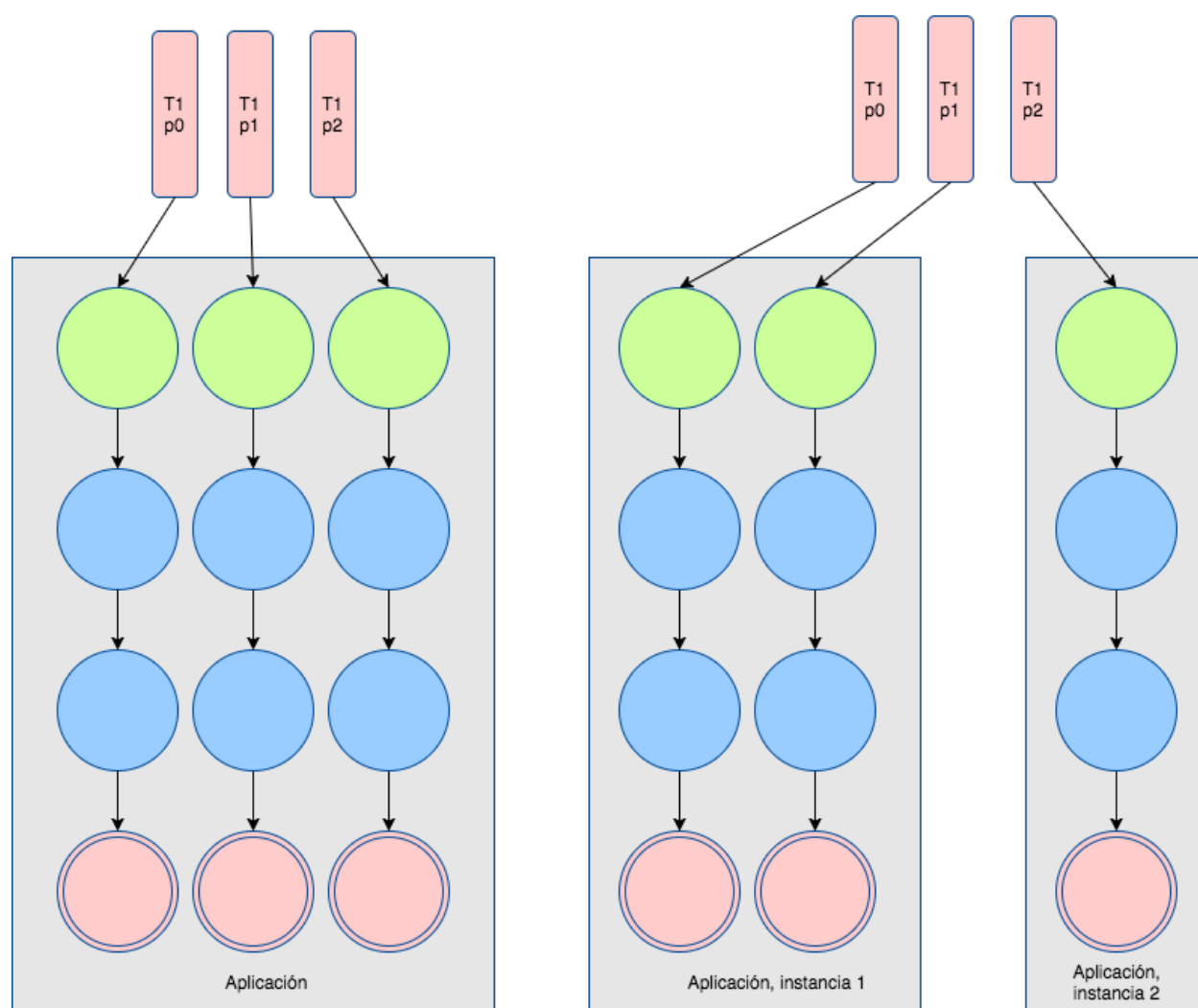


Figura 11 Escalado de una aplicación en Kafka Streams

En la Figura 11 podemos observar un escenario en el que se decide escalar la aplicación. La topología en este caso es lineal, un único procesador de entrada, un único procesador de salida, y dos procesadores intermedios. La existencia de varias líneas que conectan con cada uno de las particiones del *topic* (rectángulos de la parte superior) se corresponde con la ejecución de varios hilos. En el primer escenario nos encontramos con una única ejecución de la aplicación en una única máquina, en este caso todos los hilos se asignan a la misma ejecución. En el segundo escenario, introducimos una nueva instancia ejecutándose en una nueva máquina. Vemos como la asignación de los hilos cambia, pasando a ejecutar dos de ellos en la primera instancia, y el restante en la segunda.

## 3.5. InfluxDB

InfluxDB es una base de datos NoSQL de clave-valor fácilmente escalable. Está especialmente diseñada para almacenar series temporales y permitir un alto ancho de banda de escritura. Es ampliamente utilizada para resolver problemas de monitorización de sistemas y redes de sensores, aplicaciones de análisis de datos en tiempo real, y en definitiva cualquier entorno en el que se recolecten flujos de datos en forma de series temporales. Su función en estos sistemas es el de almacenar los resultados de los cálculos realizados en las fases anteriores. Por tanto, suele ser el penúltimo paso del recorrido de los datos en una aplicación de *streaming*, teniendo como principal, y a menudo único, cliente la interfaz de usuario donde se presentarán los resultados.

### 3.5.a. Estructura de datos

Los datos en InfluxDB se estructuran en una serie de capas que se detallan a continuación:

1. **Base de datos** o *database*. El primer nivel de división de los datos corresponde con la base de datos. En un mismo *cluster* de InfluxDB podemos definir varias bases de datos. Este primer nivel va dirigido a separar información de distintas aplicaciones.
2. **Serie** o *series*. Dentro de cada base de datos, la información se agrupa en series. A este nivel se define el periodo de retención, por lo que datos que tengan distintos períodos de retención deberán ser alojados en diferentes series. El número máximo de series por base de datos puede ser limitado en la configuración.

3. **Métrica** o *measurement*. En cada serie encontramos métricas. Estas serían las equivalentes a las tablas de las bases de datos relacionales. En la práctica, una métrica equivale a una serie temporal que se quiere monitorizar. Por ejemplo, en monitorización de sistemas, una métrica podría ser el espacio utilizado de disco duro ‘uso\_disco’.
4. **Timestamp**. Como no puede faltar en una base de datos de series temporales, nos encontramos con el valor que nos indica el instante de tiempo. El *timestamp* es la primera columna en esta visión tabular de la métrica, forma parte de la clave y puede ser usada para filtrar una consulta. La precisión por defecto es de nanosegundos.
5. **Etiquetas** o *tags*. Definen, junto con el *timestamp*, la clave de nuestros datos dentro de la métrica. Esto implica que una inserción sobre una base de datos, serie, métrica, *timestamp* y conjunto de etiquetas existentes, realizará una sobrescritura, ya que InfluxDB entiende que se corresponden a los mismos datos. Al formar parte de la clave, los datos se indexan por estos valores. La elección de qué valores son etiquetas es muy importante en el diseño, ya que, a la hora de filtrar, evitaremos un escaneado total de los valores si solo incluimos valores de la clave. Las etiquetas se almacenan en forma de tupla clave valor. En la analogía a las tablas de bases de datos relacionales, la clave de la tupla sería el nombre de la columna, mientras que el valor de la tupla sería el valor de la celda. Volviendo al ejemplo de la métrica, si establecemos como campo la IP de la máquina que estamos monitorizando, la clave de etiqueta dentro de la tupla sería ‘ip’, y el valor sería la IP concreta que cada máquina tenga.
6. **Campos** o *fields*. Los campos son el resto de valores que contienen información y que, normalmente, se corresponden con los valores resultado de una consulta, más que de los valores de filtrado de la misma. Al igual que las etiquetas se componen de una tupla clave valor, donde la clave sería el nombre de la columna en una representación tabular, y el valor el valor de la celda. En el ejemplo de la monitorización de sistemas, en nuestra métrica de uso de disco de los servidores, podríamos disponer del espacio usado como campo ‘espacio\_mb’, siendo la clave espacio, y el valor la cantidad de bytes ocupados.

A continuación, se muestra la representación tabular del ejemplo que hemos descrito:

timestamp	ip	valor
2017-11-18T00:00:00Z	1.1.1.1	100000
2017-11-18T00:00:00Z	2.2.2.2	105000
2017-11-18T00:05:00Z	1.1.1.1	100100
2017-11-18T00:05:00Z	2.2.2.2	105,150

### 3.5.b. Escritura de datos

Para interactuar con la base de datos, InfluxDB expone una API HTTP. En el caso de escritura, utiliza un protocolo denominado *line protocol* que define el cuerpo del mensaje POST que debemos mandar para escribir puntos en nuestra base de datos.

En el ejemplo que hemos visto anteriormente, para escribir los datos que encontramos en la tabla, dentro de la base de datos ‘monitoring’ mandaremos el mensaje que encontramos en la Figura 12.

```
curl -i -XPOST \
'http://influxDB:8086/write?db=monitoring' --data-binary \
'uso_disco,ip=1.1.1.1 valor=100000 1510963200000000000
uso_disco,ip=2.2.2.2 valor=100000 1510963200000000000
uso_disco,ip=1.1.1.1 valor=100000 1510963500000000000
uso_disco,ip=1.1.1.1 valor=100000 1510963500000000000'
```

Figura 12 Escritura de datos en InfluxDB

### 3.5.c. Lectura de datos

Para realizar consultas de los datos, InfluxDB también expone una API HTTP. A diferencia de la escritura, para la consulta InfluxDB sí que expone un lenguaje más estándar, ya que sigue la misma sintaxis SELECT que nos encontramos en SQL.

En el ejemplo anterior, para obtener los datos de la tabla correspondientes a la IP 1.1.1.1 el mensaje con *curl* nos quedaría como muestra la Figura 13.

```
curl -G 'http://influxDB:8086/query?pretty=true' \
--data-urlencode "db=monitoring" \
--data-urlencode \
"q=SELECT * FROM uso_disco WHERE `ip`='1.1.1.1'"
```

Figura 13 Lectura de datos en InfluxDB

## 3.6. Grafana

Grafana es una interfaz gráfica especializada en representación de gráficas y tablas de series temporales de código abierto. Acepta varias bases de datos como fuente, entre las que podemos encontrar InfluxDB. Su estructura es muy simple, y permite al usuario la creación de diversos *dashboards*, que no son más que tableros donde podemos configurar una serie de gráficas. Las gráficas están conectadas mediante consultas a diferentes fuentes de datos que podemos configurar.

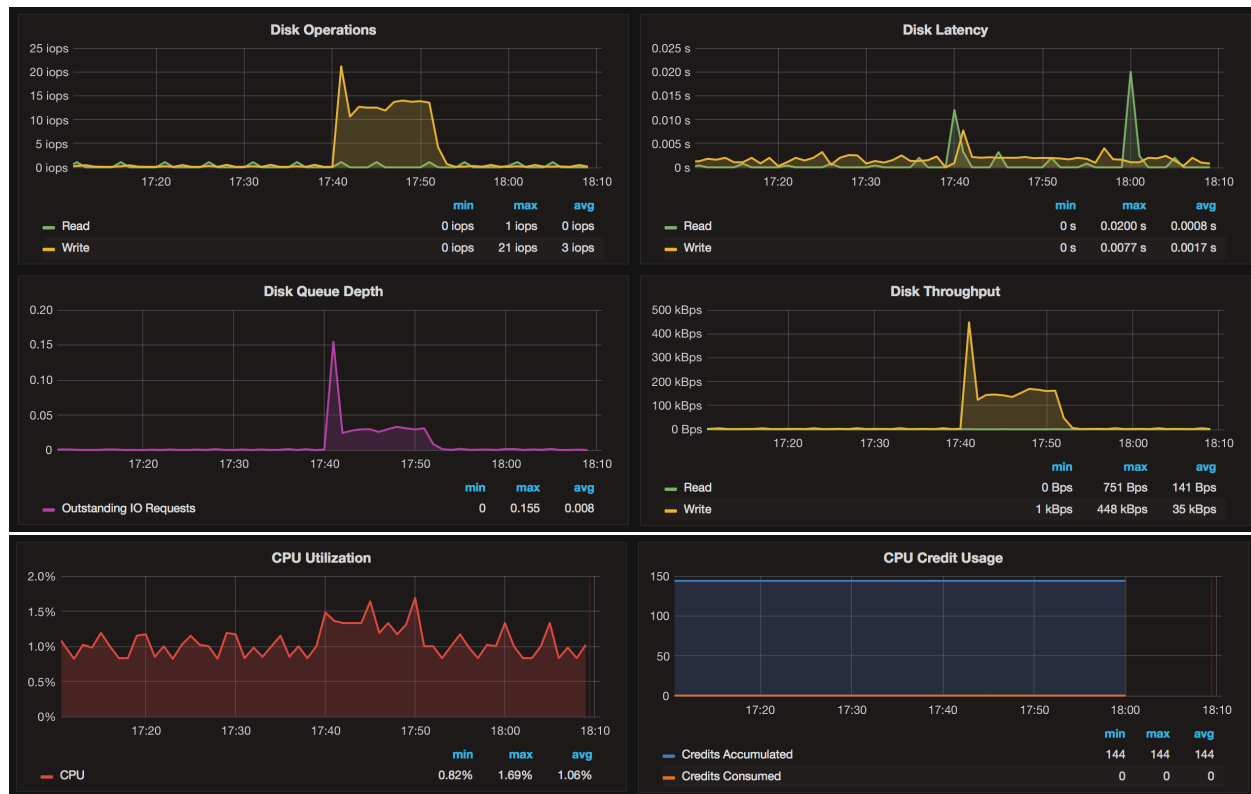


Figura 14 Dashboard ejemplo de Grafana

La capacidad de personalización de cada una de las gráficas y de los *dashboards* en general es bastante elevada, lo que ha hecho que se convierta en una de las herramientas más utilizadas como interfaz gráfica en sistemas de análisis de datos.

## 3.7. Docker y Docker-compose

### 3.7.a. Conceptos básicos de Docker y utilización

Docker es una tecnología en auge en los últimos años. Aprovecha el concepto de ‘cgroups’ de Linux para crear un entorno de virtualización ligera, donde las máquinas virtuales se comportan como procesos aislados dentro de la máquina. La gran ventaja con respecto a las máquinas virtuales tradicionales es su ligereza, ya que reutiliza la capa *kernel* del sistema operativo anfitrión.

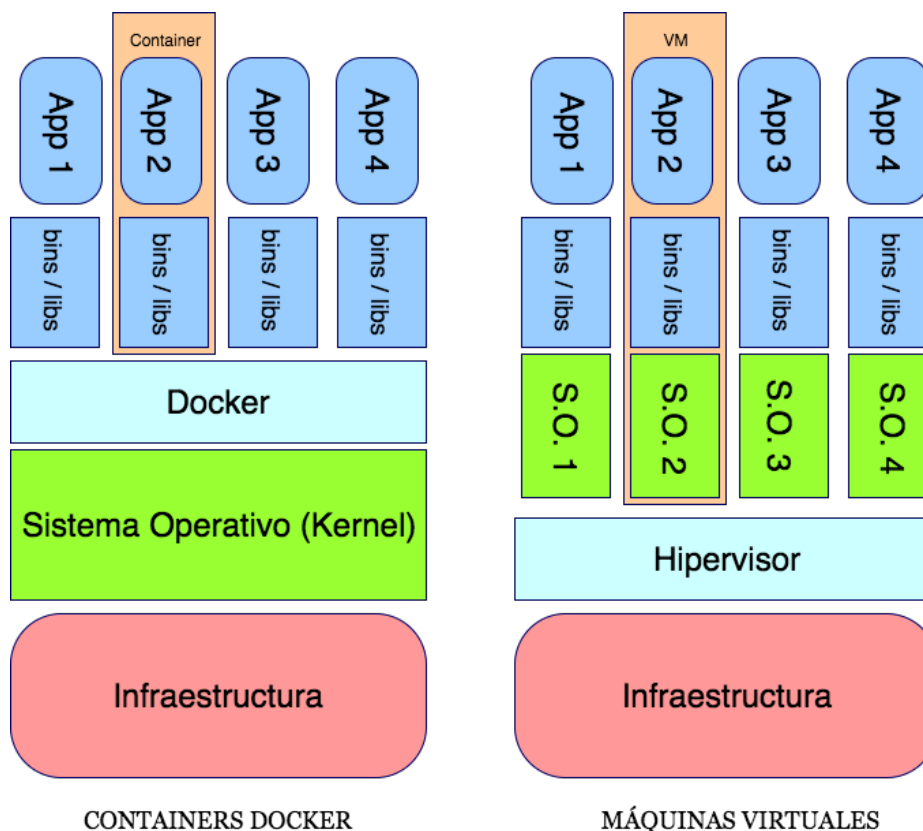


Figura 15 Diferencias entre containers Docker y Máquinas Virtuales

En Docker encontramos varios términos importantes:

- **Imagen Docker.** Es un *snapshot* del disco duro de una máquina virtual configurada. Al contrario que en las máquinas virtuales, la imagen Docker no se ejecuta. Existe versionado sobre las mismas, por lo que podemos tener varias versiones de una misma imagen.

- **Container Docker.** Es una ejecución de una imagen Docker. Es una instancia física de una imagen, con CPU, memoria, disco, y entornos de red asignados. Se pueden ejecutar varios *container* Docker de una misma imagen, y cada uno tendrá su ciclo de vida. Los *containers* pueden ser ejecutados, parados, reanudados o eliminados. Los *containers* tienen la característica de ser volátiles, esto significa que cuando un *container* es eliminado, su estado de disco desaparece con él.
- **Docker Registry,** o repositorio Docker. Es el lugar donde se almacenan las imágenes Docker. Podemos disponer de un repositorio local, o hacer uso del repositorio público de Docker, donde encontramos un catálogo bastante amplio de imágenes públicas, mantenidas por terceros, listas para ser utilizadas.

```

1 FROM debian:jessie
2
3 ARG DOWNLOAD_URL
4
5 RUN apt-get update && \
6     apt-get -y --no-install-recommends install libfontconfig curl ca-certificates && \
7     apt-get clean && \
8     curl ${DOWNLOAD_URL} > /tmp/grafana.deb && \
9     dpkg -i /tmp/grafana.deb && \
10    rm /tmp/grafana.deb && \
11    curl -L https://github.com/tianon/gosu/releases/download/1.7/gosu-amd64 > /usr/sbin/gosu && \
12    chmod +x /usr/sbin/gosu && \
13    apt-get autoremove -y && \
14    rm -rf /var/lib/apt/lists/*
15
16 VOLUME ["/var/lib/grafana", "/var/log/grafana", "/etc/grafana"]
17
18 EXPOSE 3000
19
20 COPY ./run.sh /run.sh
21
22 ENTRYPOINT ["/run.sh"]

```

Figura 16 Dockerfile de la imagen oficial del repositorio público de Grafana

- **Dockerfile.** Es la descripción en código de una imagen Docker. Las imágenes Docker se definen mediante capas de acciones sobre una imagen base. Tiene varias secciones que pueden ser consultadas en la documentación oficial, de las que destacan la primera línea que indica de qué imagen partimos, un conjunto de acciones o comandos que

alteran el estado de esta imagen de partida, y un punto de entrada, que es el comando que se ejecuta cuando lanzamos un *container* de esta imagen.

El funcionamiento de Docker a nivel de desarrollador está basado en el conocido sistema de control de versiones Git<sup>19</sup>. Podemos hacer *pull* de una imagen de un repositorio para descargarla, y podemos hacer *push* para publicar una nueva imagen o versión. Para ejecutar un container tenemos el comando *run*, donde especificamos una serie de parámetros, entre los que destacan:

- Imagen del container y versión
- Mapeo de puertos del host y del container. Debemos indicar qué puertos expone el container, y en qué puertos de la máquina anfitriona van a estar disponibles.
- Tipo de red a utilizar. Podemos usar una red virtual, o la red del anfitrión de manera directa.
- Enlazado de imágenes. Podemos enlazar el nuevo container con otros, de manera que su conexión es más sencilla. Este comando realiza una modificación del archivo hosts, además de crear una serie de variables de entorno, de manera que podemos conectarnos por nombre a los containers enlazados.
- Mapeado de volúmenes. Como hemos dicho anteriormente, los containers Docker no persisten su estado de disco una vez que finalizan. No obstante, podemos evitar perder los datos generados por un container mediante el mapeo de volúmenes internos del container a directorios externos del host.
- Comando de ejecución y argumentos. Por último, debemos especificar el comando a ejecutar, o los argumentos a pasar al comando de entrada especificado en el Dockerfile.

Las principales características de Docker son el aislamiento del entorno de ejecución de los componentes, la gran portabilidad, y la reutilización. Se ha convertido en un estándar para la ejecución de aplicaciones en la nube, ya que permite la migración de plataformas de forma sencilla.

---

<sup>19</sup> [git-scm.com](https://git-scm.com)



### 3.7.b. Docker-compose

Docker-compose<sup>20</sup> es una librería de utilidad que abre un abanico de operaciones sobre Docker. Permite realizar la definición de un despliegue de diversos containers Docker con un simple fichero de configuración YAML que facilita de gran manera la ejecución de un entorno compuesto.

En el fichero podemos definir qué containers componen nuestro sistema, cómo se enlazan entre ellos y las interdependencias de ejecución. Podemos mapear los puertos, darle nombres, definir volúmenes,

Una vez definido el fichero, podemos ejecutar, pausar o limpiar nuestro sistema compuesto de containers con un par de comandos:

- *docker-compose up*: Ejecuta los containers descritos en el fichero de configuración. Aparte de todos los comandos disponibles con *docker run* para cada uno de los containers ejecutados, disponemos una serie de comandos que nos permiten definir dependencias de ejecución, de manera que podemos orquestar la inicialización de los containers de manera sencilla.
- *docker-compose stop*. Detiene todos los containers.
- *docker-compose down*. Detiene los containers que se encuentren en ejecución, y los elimina del sistema.

La Figura 17 muestra un ejemplo de fichero de configuración de Docker-compose para la ejecución de Kafka y Zookeeper. En este caso, Kafka no se ejecuta desde una imagen almacenada en el repositorio, sino de un Dockerfile local que construye una imagen.

---

<sup>20</sup> docs.docker.com/compose

```
1  version: '2'
2  services:
3    zookeeper:
4      image: wurstmeister/zookeeper
5      ports:
6        - "2181:2181"
7    kafka:
8      build: .
9      ports:
10       - "9092"
11      environment:
12        KAFKA_ADVERTISED_HOST_NAME: 192.168.99.100
13        KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
14      volumes:
15       - /var/run/docker.sock:/var/run/docker.sock
```

*Figura 17 Fichero de configuración de Docker-compose*

# ***Capítulo 4. Diseño e implementación de la solución***

En este capítulo se abarca tanto el diseño como la implementación del sistema. Puesto que se ha seguido una metodología ágil de desarrollo, se ha decidido agrupar las dos categorías en una sola sección.

Cuando hablamos de metodologías ágiles de desarrollo nos referimos a aquellas que nacieron en contraposición de las metodologías en cascada. En las metodologías clásicas, las fases del desarrollo se dividen en capas de manera que, una fase daba comienzo cuando la anterior había finalizado. Estas capas dividen el proceso de desarrollo del software en (1) especificación y captura de requisitos, (2) análisis de casos de uso, (3) diseño de la arquitectura, (4) implementación y por último (5) testeo y validación.

Esta forma de organizar un proyecto cuenta con varios inconvenientes. Uno de ellos es la lentitud de adaptación de nuevos requisitos, o problemáticas del diseño original debido a que el proyecto se estructura en una única y gran iteración. Para tratar de solucionar estos problemas, surgieron las metodologías ágiles, que pretenden descomponer el software en pequeñas piezas sobre las que iterar más rápido.

En concreto, en este proyecto se ha utilizado la metodología Kanban [9], una modificación de la metodología Scrum [10] que elimina las iteraciones fijas en el tiempo y las planificaciones. En Kanban, se cambian los *sprints* constantes y con duración fija de Scrum por una lista de tareas ordenada por prioridad, y las *releases* o entregas se ejecutan cuando alguno de los componentes está listo para ser entregado. Esta simplificación tiene sus ventajas e inconvenientes, mientras la velocidad de desarrollo por lo general es más elevada al reducir la carga burocrática y de planificación, se carece de un ritmo constante de entrega, por lo que el cliente pierde en muchos casos la noción de la evolución del proyecto. Además, la falta de una planificación más detallada puede llevar a descoordinación entre los miembros del equipo, o al descubrimiento de problemas en la fase de desarrollo que pueden retrasar de forma imprevisible las entregas.

Dado que en este proyecto hay un único desarrollador, la metodología Kanban ofrece más ventajas que inconvenientes, y por tanto es la que se ha elegido utilizar.

## 4.1. Requisitos

Como todo proyecto, éste nace con unos requisitos tanto funcionales como no funcionales. A continuación, se citan los requisitos funcionales que el sistema ha de cumplir.

1. El usuario puede configurar agregaciones que se realizarán en tiempo real sobre un *stream* de datos de Twitter. Las métricas pueden ser definidas en cualquier momento sin necesidad de compilar o desplegar la solución, y éstas deben tener un efecto inmediato sobre los datos. Las métricas deben incluir al menos un filtrado del contenido del texto del Tweet, así como una agregación por alguno de sus campos.
2. El usuario debe ser capaz de visualizar los resultados de las métricas en gráficas temporales. Los resultados de las métricas se dispondrán en formato de serie temporal, con un punto por agregación temporal, y por cualquier otro campo del tweet especificada en la definición de la métrica.
3. El usuario debe ser capaz de generar nuevas gráficas temporales configurables.
4. El usuario debe ser capaz de agrupar las gráficas en *dashboards* configurables.

Además, se incluyen una serie de requisitos no funcionales:

1. Debe utilizar tecnologías que faciliten su despliegue, así como su portabilidad futura a la nube. El despliegue debe estar automatizado partiendo del código fuente.
2. Debe ser robusto frente a errores. El sistema debe seguir funcionando parcialmente si uno de los sistemas que lo componen deja de funcionar correctamente. Recuperarse de un error debe ser sencillo y debe estar incluido dentro del funcionamiento normal del sistema.
3. Debe ser sencillo habilitar una vía para recomputar datos del pasado sin afectar el correcto funcionamiento del sistema durante su ejecución natural.

4. Debe ser sencillo conectar nuevas entradas de datos.
5. El sistema ha de ser capaz de manejar el flujo de datos entrantes de la API de Twitter en tiempo real. Esto es, la velocidad de procesamiento ha de ser igual o mayor que la velocidad de inyección de datos.
6. El sistema al completo ha de ser capaz de ejecutarse en un ordenador portátil, sin comprometer ninguno de los requisitos.

## 4.2. Diseño

### 4.2.a. Diagramas de casos de uso

Para dar cabida a los requisitos funcionales, podemos extraer una serie de casos de uso, que se describen en la Figura 18.

Encontramos una relación uno a uno con los requisitos funcionales.

### 4.2.b. Arquitectura

Como se ha avanzado en capítulos anteriores, el sistema plantea resolver el problema del análisis en tiempo real de un *stream* de datos proveniente de Twitter mediante la implementación de la arquitectura Kappa.

Si recordamos lo dicho en el Capítulo 2, la arquitectura Kappa nace como una simplificación de la arquitectura Lambda. Volviendo al mismo esquema que se presentó en dicho capítulo, nos encontramos con la Figura 19. Así, pues el diseño del sistema a implementar debe seguir un esquema similar. Los datos de entrada en el caso que tratamos son los eventos de Twitter, recolectados mediante la Streaming API. Estos mensajes pasan a una primera cola, que podemos ver como cola de salvaguardia, puesto que podemos definir un periodo de retención para ser capaces de solventar problemas de ejecución de nuestro sistema. El sistema que colecte los datos y los almacene en esta cola deberá ser eficiente, seguro y simple. Su principal objetivo es el de ser capaz de ingerir todos los datos de entrada de forma rápida y segura. Una simplicidad en la implementación nos previene contra errores de codificación, o problemas de rendimiento. A su

vez, la cola de entrada debe ser paralelizable de manera que podamos conectar diferentes lectores en paralelo a la misma para lograr un mayor ancho de banda.

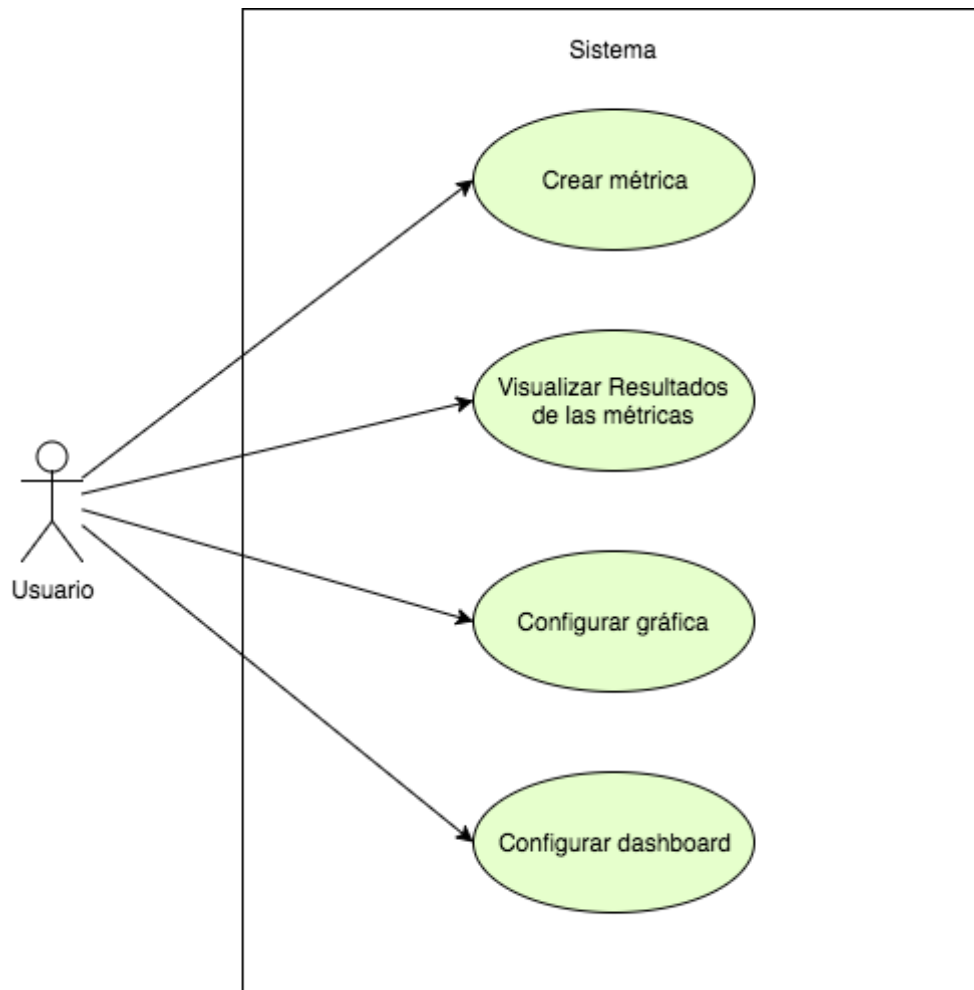


Figura 18 Diagrama de casos de uso

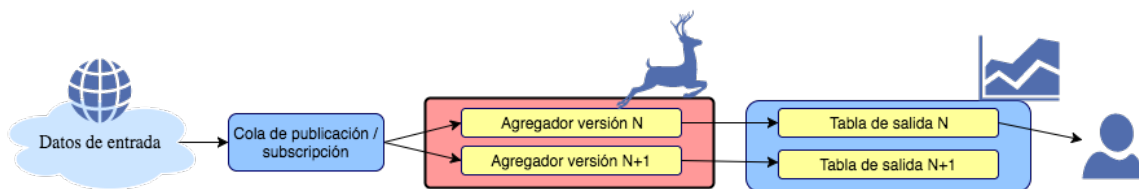


Figura 19 Arquitectura Kappa

Una vez los datos están en la cola de entrada, ya es posible realizar cálculos más complejos puesto que tenemos la seguridad de poder realizar recómputos en caso de error, o incluso escalar la aplicación de manera horizontal en caso de necesitar más capacidad de cómputo, ya que la cola de entrada permite una lectura en paralelo incrementando el ancho de banda.

El componente que lee de la cola de entrada se denomina Agregador, y su función es la de agregar, aplicando las definiciones de métricas, los datos provenientes de la cola de entrada, y escribirlos a una cola de salida. El Agregador ha de leer las métricas definidas por el usuario. Como se especifica en los requisitos funcionales, el sistema ha de estar en constante ejecución, sin necesidad de cambio alguno en el despliegue para actualizar el conjunto de definiciones. Es por esto que el Agregador, con cierta regularidad leerá las métricas definidas por el usuario, y las aplicará al *stream* de datos proveniente de la cola de entrada.

La cola de salida del Agregador es la cola de entrada para el escritor al sistema de representación final. El rol de este componente es el de leer los datos agregados, realizar las transformaciones necesarias para su escritura en el sistema de salida, y escribir los datos. Puesto que en los requisitos encontramos que el usuario visualizará el resultado en gráficas temporales, las agregaciones deberán ser escritas en una base de datos de series temporales.

Finalmente, para cumplir con los requisitos de visualización y edición de las gráficas temporales y *dashboards*, debemos presentar el resultado en una interfaz gráfica que permita acceder a la base de datos de series temporales, y mostrar gráficas. Las gráficas deben ser configurables por el usuario, por lo que las consultas a la base de datos deben estar abiertas a ser modificadas en cualquier momento. Además, las gráficas se compondrán en *dashboards* en esta interfaz, siendo la composición definida por el usuario.

La Figura 20 muestra el diagrama de componentes final del sistema. Puesto que el sistema está compuesto de pequeñas piezas que interactúan entre sí siguiendo la arquitectura de Microservicios [11], un diagrama de clases entraría en demasiado detalle. Los componentes de un sistema de Microservicios suelen ser bastante simples, por lo que un diagrama de componentes expresa de mejor manera el funcionamiento del sistema.

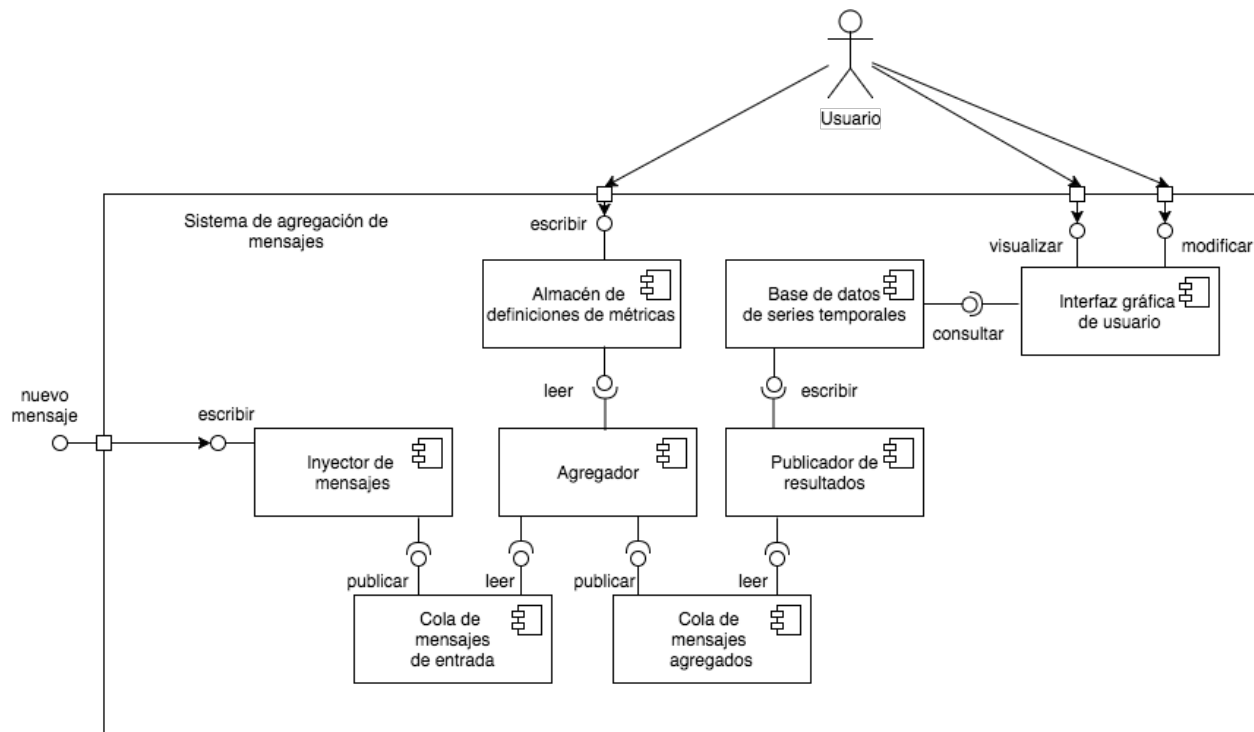


Figura 20 Diagrama de componentes

Para incrementar la robustez del sistema, se ha descompuesto en componentes independientes que se comunican de manera asíncrona. Cada componente es autocontenido y es capaz de ser desplegado de forma independiente de los demás. La cola que comunica los sistemas actúa como buffer y salvaguarda en caso de fallo de uno de los componentes. Si, por ejemplo, el Agregador fallara, el inyector seguiría funcionando y enviando mensajes a la cola. En el momento en el que el Agregador vuelve a funcionar, éste recupera su estado anterior y procesa todos los mensajes atrasados. De este modo, un fallo en uno de los componentes únicamente afecta en un retraso en la entrega de los datos, sin necesidad de actuar sobre el resto de los componentes.

Los mensajes de entrada vienen de Twitter. Twitter modela los mensajes como elementos JSON. El sistema estará únicamente acoplado a este formato, de manera que cualquier fuente de datos que escriba en la cola de entrada en formato JSON es capaz de ser integrado en el sistema de forma transparente. Las especificaciones de las métricas están definidas por el usuario, por lo que es aquí donde se puede hacer la discriminación de los mensajes a tener en cuenta en cada una de las métricas a calcular.



## 4.3. Implementación

La primera parte de la implementación es la elección de qué tecnologías deben implementar cada una de las piezas del diseño.

Los componentes que realizan toda la lógica del sistema son 3: El inyector de Tweets, el Agregador, y el escritor al sistema de salida. En la implementación se ha decidido por realizar aplicaciones separadas en Scala. Scala es un lenguaje funcional y orientado a objetos que se describió con más detalle en el Capítulo 3. Tiene las dos características que permiten una implementación limpia del funcionamiento que esperamos de estos sistemas. Al ser funcional, nos permite describir el flujo de datos con la terminología Map-Reduce de forma muy sencilla. Al ser orientado a objetos podemos aplicar todos los patrones de diseño y de *clean code* existentes para facilitar su extensibilidad y legibilidad.

Para las colas de publicación y subscripción se utiliza Apache Kafka, un software diseñado específicamente para la función de distribución de mensajes entre aplicaciones. Las tablas de salida serán en nuestro caso las métricas creadas en InfluxDB, y usaremos Grafana como interfaz gráfica de la misma.

Nos queda por definir cómo se va a realizar la agregación de las métricas, y cómo se van a especificar las mismas. Para esta tarea existen varias tecnologías que se adaptan bastante bien al problema, entre las que destacan Spark Streaming, Apache Flink, y Apache Storm. Estos sistemas son capaces de ingerir unas inmensas cantidades de datos, y escalar hasta cientos y miles de máquinas distribuidas. Sin embargo, todas ellas comparten otra característica no tan buena como la anterior, su despliegue es muy costoso, y requiere de mucha configuración para ponerlos a funcionar.

Como hemos podido ver en las tecnologías utilizadas, no tenemos ninguna de estas allí presentes, y es porque Kafka desarrolló recientemente una API que pretende luchar con las tecnologías de procesamiento de eventos actuales simplificando el desarrollo y el despliegue de la solución, reduciendo costes de infraestructura y mantenimiento. La API que permite implementar agregaciones sobre *streams* de datos se denomina Kafka Streams, y es la elegida en este proyecto.

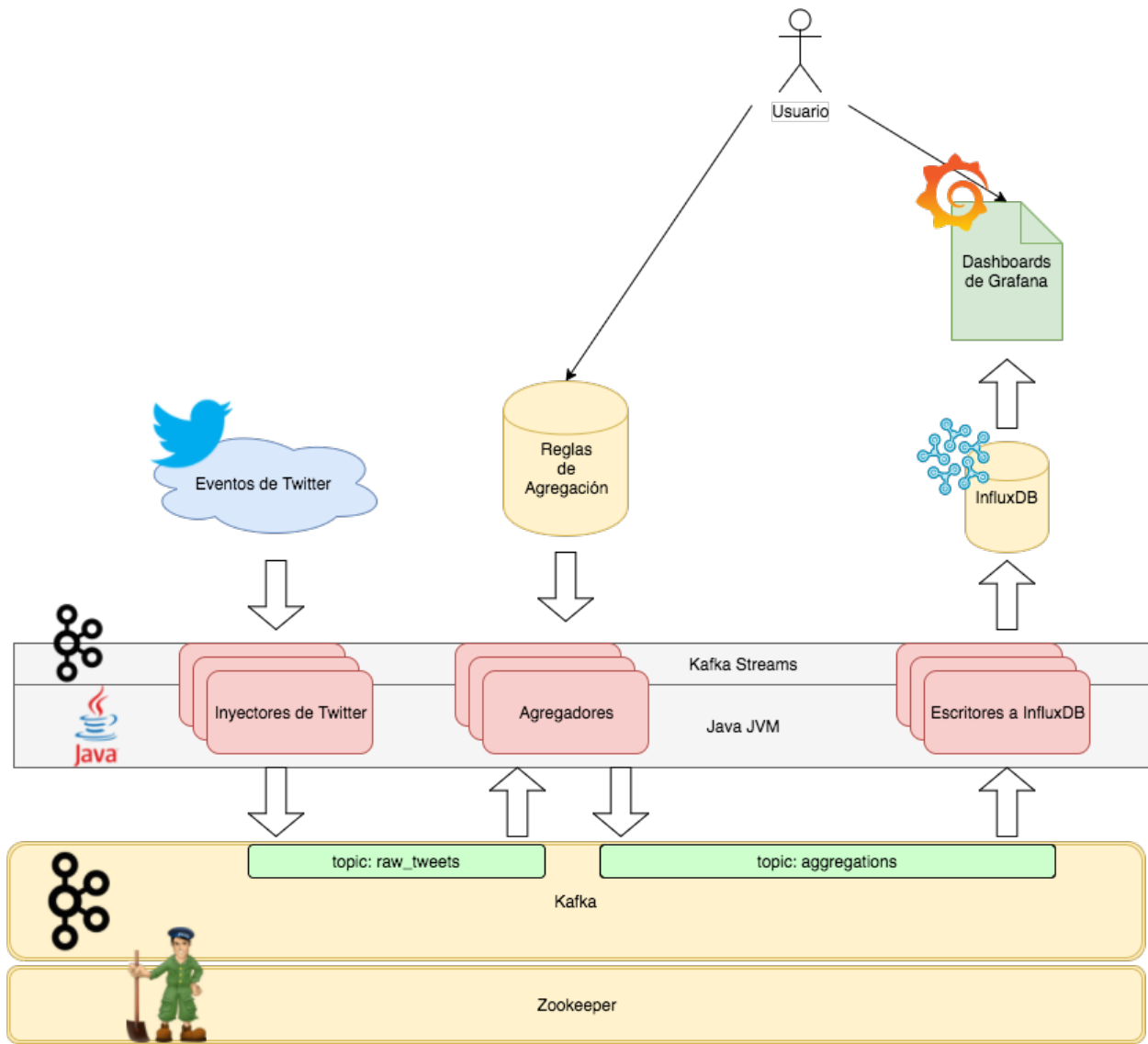


Figura 21 Arquitectura del sistema y tecnologías

Así pues, tras saber qué tecnología implementará qué rol en la arquitectura Kappa, es momento de presentar el diseño arquitectónico de la solución, con las tecnologías utilizadas. La Figura 21 muestra el diagrama.

La arquitectura se descompone en 3 componentes principales, los inyectores de Twitter, los agregadores, y los escritores a InfluxDB. Se utiliza notación plural, aunque la implementación sólo incluye uno de ellos, porque están diseñados para poder ser escalados de manera horizontal. Para escalar el proceso de agregación, por ejemplo, bastaría con ejecutar un Agregador más en el sistema.

Estos tres componentes se comunican entre sí de manera asíncrona, escribiendo mensajes en *topics* de Kafka. El usuario hace uso del sistema mediante la definición de nuevas reglas de agregación, y la creación de *dashboards* en Grafana.

### ***4.3.a. Decisiones de implementación***

#### ***4.3.a.I. Debido a requisitos funcionales***

Para asegurar los requisitos funcionales se han tomado una serie de decisiones:

La definición de métricas se realiza mediante una serie de ficheros JSON almacenados en disco. Puesto que esta implementación no está desplegada en la nube, se opta por la solución más sencilla que es el sistema local de ficheros. No obstante, la implementación del recolector de métricas está abstraída en una interfaz (*trait* en el caso de Scala), lo que permite su extensión en el futuro.

Estos ficheros son recolectados periódicamente por el sistema, actualizando el conjunto de métricas a calcular cada cierto tiempo. No se realiza de forma constante para no ralentizar el sistema con lecturas constantes a disco. Para evitar esto, se implementa una caché que tiene predefinido un tiempo de vida más o menos prudente (10 segundos), que consiga tanto evitar lecturas constantes, como permitir al usuario ver sus métricas calculadas de forma relativamente rápida.

Para permitir al usuario la visualización de datos en *dashboards* configurables, se ha optado por la tecnología Grafana. Como hemos visto en el Capítulo 3, es una muy buena opción para representar series temporales. Ofrece una gran cantidad de personalización y cumple con creces los objetivos planteados en los requisitos.

#### ***4.3.a.II. Debido a requisitos no funcionales***

En cuanto a los requisitos no funcionales, también se han tomado una serie de decisiones que ayudan a su cumplimiento:

Para facilitar el despliegue en la nube, se ha decidido compartimentar la aplicación en *containers* Docker. Puesto que Docker se ha convertido en un estándar de despliegue de servicios

en la nube, su migración a plataformas como Cloud Foundry<sup>21</sup>, Kubernetes<sup>22</sup>, etc. Es bastante sencilla. De igual manera, al aislar las dependencias con sistemas externos en un único container, la instalación es sencilla y requiere únicamente que la máquina de destino tenga una versión concreta del Kernel de Linux, así como Docker instalado en el sistema.

Para habilitar una vía de cómputo paralelo de datos atrasados no es necesario detener el funcionamiento actual del sistema. En caso de recómputo por error en el algoritmo podemos desplegar un Agregador en paralelo con el código corregido, y con un ID de aplicación distinto, que escriba a un *topic* de Kafka distinto, por ejemplo, ‘aggregations\_v2’. A la vez, ejecutaremos una nueva instancia del escritor de InfluxDB que lea de este *topic* ‘aggregations\_v2’ y escriba a una nueva serie de InfluxDB ‘tweeter\_aggregations\_v2’. En ese mismo momento, el usuario sería capaz de leer datos de ambos sistemas con una simple configuración en Grafana. En el momento en el que la nueva instancia pase a computar datos del presente, podemos eliminar las dos instancias anteriores.

Para asegurar que podemos procesar el volumen de entrada de datos a un ritmo suficientemente alto, así como garantizar una posible ampliación futura con más fuentes de datos se decide compartimentar la solución y realizar una comunicación asíncrona entre los componentes. De esta forma los mismos componentes dejan de tener dependencias fuertes entre ellos para convertirlas en dependencias débiles, al igual que sucede en la arquitectura de Microservicios. En vez de depender de una comunicación directa Agregador – Inyector, el Agregador pasa a depender de la presencia de mensajes en Kafka. Esta abstracción nos permite desplegar en paralelo diferentes inyectores que recolecten datos de distintas fuentes y los escriban en el mismo *topic*.

Este punto se une con otra decisión de implementación, el uso de datos no estructurados. Los agregadores dependen de métricas de agregación que están definidas como elementos dentro de un mensaje JSON. Siempre que la fuente de datos emita mensajes JSON, podremos definir

---

<sup>21</sup> [www.cloudfoundry.org](http://www.cloudfoundry.org)

<sup>22</sup> [kubernetes.io](http://kubernetes.io)

nuevas métricas que se adapten a dichos mensajes, y el Agregador seguirá funcionando correctamente sin necesidad de ningún cambio en el código fuente.

Puesto que entre los requisitos tenemos la simplicidad de despliegue, se ha decidido utilizar Docker, junto con Docker-compose. Docker permite aislar bastante bien el entorno de ejecución de cada componente, de manera que su despliegue no se ve alterado con las posibles modificaciones de herramientas instaladas en el sistema. Docker-compose permite ejecutar el despliegue de un sistema con múltiples componentes con un único comando, tras la definición de un fichero de configuración.

#### ***4.3.b. Inyector de Twitter***

El inyector de twitter es el componente que debe recibir los mensajes de la plataforma Twitter y escribirlos en Kafka. Como hemos podido ver en el Capítulo de las tecnologías utilizadas, hacemos uso de la librería de código abierto *twitter-hosebird*. Esta librería se conecta a la API de Twitter Stream y expone al usuario un buffer de cadenas de texto donde se van actualizando los mensajes.

Es la pieza más crítica del sistema, puesto que es la única que provoca un error irreparable en caso de fallo. Cuando los orígenes de datos son sistemas que se encuentran bajo nuestro control, sería sencillo situar un buffer en el origen, de manera que si existe un fallo de comunicación entre el origen de los datos y el receptor (inyector en este caso), se almacene de forma temporal en un fichero local, que será enviado desde el principio cuando esta conexión se vuelva a producir. En el caso en el que nos encontramos, Twitter no está bajo nuestro control, así que, en caso de un fallo de conexión, perderíamos mensajes que no podrían ser recuperados.

Por su papel tan importante de cara a la robustez del sistema, su función debe ser lo más simple posible para evitar tanto *bugs* como posibles retrasos o problemas de rendimiento. Debemos asegurarnos que los mensajes llegan a un componente que esté bajo nuestro control lo antes posible, y por esto, su función se limita a leer los tweets y escribirlos sin modificación alguna en Kafka. Una vez el mensaje ha llegado a Kafka, ya se encuentra bajo nuestro control y podemos realizar todas las modificaciones que estimemos oportunas. Puesto que la arquitectura permite el reprocesado de manera sencilla, y Kafka dispone de un tiempo de retención en sus mensajes, podemos despreocuparnos ante cualquier problema desde este punto en adelante.

El número de particiones asignadas al *topic* de Kafka nos indicará el número máximo de agregadores en paralelo que podremos estar ejecutando. Como hemos visto en el Capítulo 3, Kafka Streams ejecuta un hilo por cada partición de cada *topic* de entrada. En nuestro sistema sólo tenemos un *topic* de entrada, por lo que el número de hilos es directamente el número de particiones asignadas al mismo. No obstante, el sistema permite extensiones con más fuentes de datos. Su elección inicial no condiciona gravemente las capacidades de la aplicación, puesto que Kafka permite la creación de particiones en caliente.

Si echamos un vistazo a las clases dentro del código, veremos que no hay mucho código en ellas. La Figura 22 muestra el total de las clases que forman este componente.

Las clases bajo el paquete *config* se corresponden con las encargadas de la configuración de despliegue. Aquí encontramos las credenciales de acceso a Twitter, la URL de la API de Twitter Streams, etc.

El paquete *dest* incluye las clases encargadas de escribir en el destino, en este caso Kafka. Encontramos la interfaz (*trait* en Scala) `MessageWriter` que modela un escritor de mensajes, y su implementación, `KafkaMessageWriter`, que implementa la escritura de mensajes en este caso en Kafka.

La clase `RecordGenerator` se encarga de leer el mensaje de Twitter y transformarlo de la manera más simple posible a un ‘Record’, que es la estructura de datos que Kafka necesita para escribir. Este formato es una tupla clave-valor, con un *timestamp* generado de forma transparente. En este caso, la clave corresponde con un número entero aleatorio, y el valor corresponde la cadena de texto sin modificar del tweet. La decisión de la clave es importante, puesto que implica la decisión de qué partición será la que reciba la escritura. El valor aleatorio implica que un incremento en el número de particiones nos llevaría a un incremento automático del número de posibles agregadores paralelos, puesto que la redistribución en particiones sería transparente para el usuario.

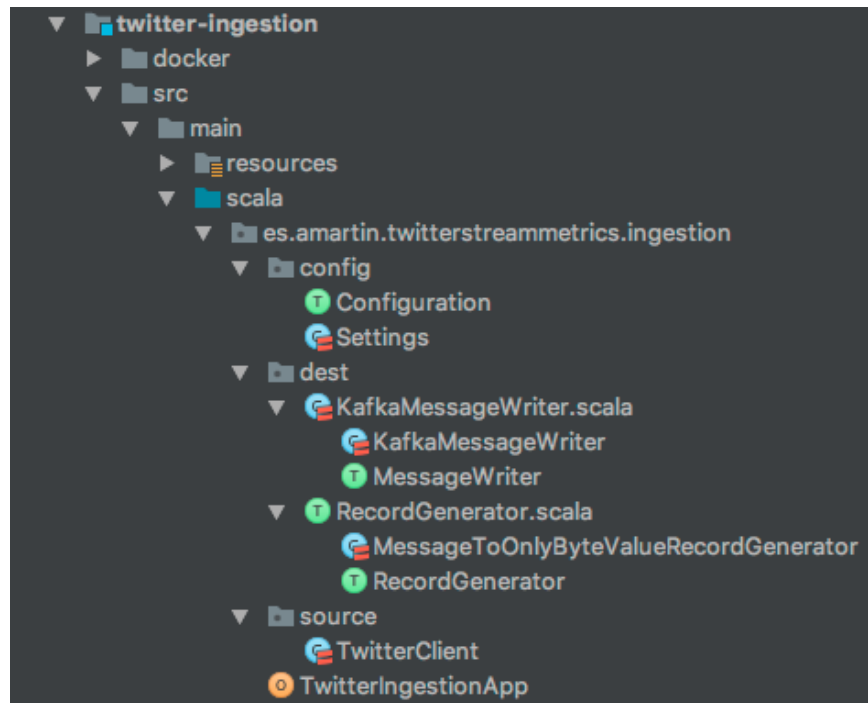


Figura 22 Clases del Inyector de Twitter

Al igual que en el caso anterior, se provee también de una interfaz para este RecordGenerator. Esta decisión aplica el principio ‘D’ de SOLID [12] [13]: “Se debe depender de abstracciones y no de implementaciones” que, además de potenciar su extensibilidad futura, facilita de gran manera la escritura de tests. En la sección 4.3.g se explican los principios SOLID en más detalle.

En el paquete *source* nos encontramos con ‘TwitterClient’, que no es más que el cliente de Twitter que hace uso de la librería Twitter-Hosebird para leer los mensajes.

Para finalizar, nos encontramos con el objeto ‘TwitterIngestionApp’, que es la entrada principal de nuestro programa, y que configura todos los elementos para el arranque. El programa principal se trata de un bucle infinito de lectura-escritura, puesto que nuestro sistema debe estar en constante ejecución.

Para completar la descripción de funcionamiento del inyector, se incluye en la Figura 24 el diagrama de secuencia de su funcionamiento, la cual muestra que el funcionamiento es bastante simple. El cliente de Twitter dispone de un buffer del que el inyector lee los mensajes, los transforma al formato requerido, y los escribe en Kafka.

```
object TwitterIngestionApp extends App with LazyLogging {
  implicit val settings: Configuration = new Settings(ConfigFactory.load())

  addShutdownHook {
    input.close()
  }

  val sourceOAuth = new OAuth1(
    settings.twitterConsumerKey,
    settings.twitterConsumerSecret,
    settings.twitterToken,
    settings.twitterTokenSecret
  )

  val input = new TwitterClient(sourceOAuth, new StatusesSampleEndpoint)

  val topic = settings.kafkaTopic
  val recordGenerator = new MessageToOnlyByteValueRecordGenerator(topic)
  val output = new KafkaMessageWriter(recordGenerator)

  input.connect()

  while (true) {
    output.write(input.nextTweet().get)
  }
}
```

Figura 23 Clase principal del Inyector de Twitter

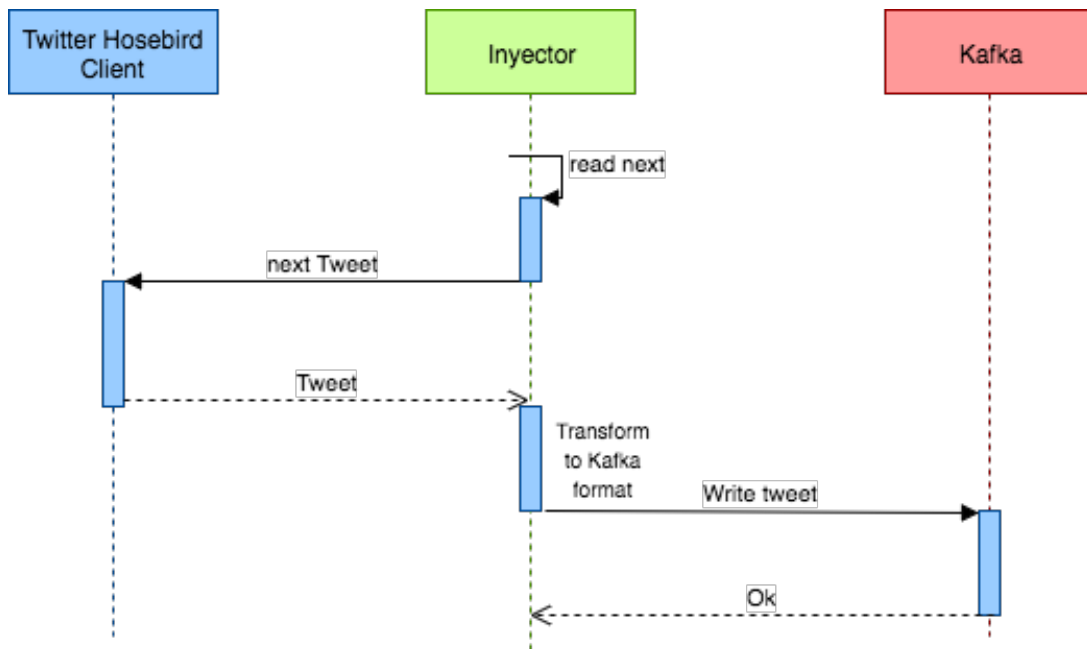


Figura 24 Diagrama de flujo del Inyector de Twitter



### 4.3.c. Definición de métricas

El usuario, además de acceso a los *dashboards*, tiene la posibilidad de definir las métricas. Para realizarlo de una forma estándar y sencilla, se ha decidido definir un lenguaje de dominio específico (DSL por sus siglas en inglés, Domain-Specific Language) basado en el formato JSON. La sintaxis de dicho lenguaje contiene los siguientes elementos:

- **textFilters**. La clave `textFilters` se encuentra en el nivel raíz del documento JSON. Su valor es un mapa de cadena de texto a lista de cadenas de texto.
  - La clave del mapa se corresponde con el *path* dentro del mensaje JSON de entrada que queremos observar para aplicar el filtro. Un *path* no es más que una cadena de elementos separados por punto, y nos permite recorrer el JSON de la misma manera que un *path* en un sistema operativo nos permite recorrer una estructura de directorios.
  - El valor del mapa corresponde con una lista de elementos que queremos filtrar. El filtro funciona con una composición en *OR*. Esto significa que, si nuestra lista contiene varios elementos, el filtro pasará en el caso de que alguno de los elementos está presente dentro del *path* proporcionado en la clave.
- **name**. La clave `name` es una cadena de texto que nos indica el nombre de la métrica. El nombre es importante para saber qué consulta realizar en Grafana, y forma parte de la clave de agregación en el Agregador, que veremos en la sección 4.3.d. El nombre debe ser único entre todas las definiciones de métrica, o estaremos incluyendo en la misma métrica valores procedentes de dos definiciones de métricas distintas.
- **groupBy**. La clave `groupBy` tiene como valor una lista de cadenas de texto. Las cadenas de texto del valor se corresponden con *paths* dentro del mensaje JSON de entrada. Cualquier *path* que indique una clave del JSON, cuyo valor sea un único elemento es apto para ser utilizado en la definición de métricas como valor `groupBy`. Estos valores son utilizados para realizar agrupaciones o agregaciones con nuestras métricas. El valor temporal de nuestras agregaciones está fijado a 30 segundos, pero es posible realizar agrupaciones espaciales mediante la declaración de estos elementos. Cuando definimos

una serie de *paths* por los que realizaremos agrupaciones, para cada valor de la clave del *path* correspondiente, se generará una métrica distinta. Esto es, si por ejemplo elegimos la localización de un mensaje como parte de nuestro *groupBy*, estaremos calculando una métrica distinta para cada uno de las localizaciones existentes en los datos de entrada.

- **IfHas.** La clave IfHas realiza un filtro dependiente de la presencia de una clave en el JSON de entrada. El valor de la clave IfHas es una cadena de caracteres que representa el *path* dentro del JSON de entrada que debe estar presente para aplicar la métrica. Por ejemplo, un valor de “user.lang” filtraría los mensajes que tuvieran un campo *user*, y dentro de éste un campo *lang*, no teniendo en cuenta para el cálculo de la métrica los mensajes que no cumplan esta propiedad.
- **IfHasNot.** La clave IfHasNot realiza un filtro dependiente de la no presencia de una clave en el JSON de entrada. Es el filtro inverso del filtro IfHas del punto anterior, por lo que tendrá en cuenta mensajes que no contengan el campo especificado en el valor de esta clave. Es muy útil para filtrar mensajes de borrado de Twitter, ya que su composición es muy diferente y limitada respecto a los mensajes normales de Twitter. Los mensajes de borrado pueden ser descartados para la agregación si incluimos un filtro IfHasNot con el valor “delete”, puesto que estos mensajes tienen este campo en la raíz del JSON.

En la Figura 25 encontramos un ejemplo de una definición de métrica. En ella se muestra un filtro sobre el campo ‘text’ situado en la raíz del JSON. Los valores a filtrar son la palabra “refugiado” en tres distintos idiomas, español, inglés y alemán. Los mensajes que contengan dentro de su cadena de texto del campo ‘text’ alguno de las palabras descritas en la definición, pasarán el filtro.

Como nombre de la métrica tenemos “containsRefugeeByTimezone”, algo a tener en cuenta para poder consultar los resultados de la métrica.

Como elementos de agregación encontramos “user.time\_zone”. Esto significa calcularemos una métrica distinta por cada elemento que encontremos en el campo time\_zone,

dentro del elemento user, que se encuentra en la raíz del JSON de entrada. En el caso de los mensajes de Twitter, este campo indica la zona horaria del usuario registrado que realiza el Tweet. Esto nos permitiría en Grafana filtrar, o agrupar los mensajes que contienen por ejemplo la zona horaria de Madrid.

Por último, queremos filtrar los mensajes que no contengan el campo “delete”, puesto que son los correspondientes a mensajes de borrado de Twitter y no contienen información relevante para nuestra métrica.

Para más información de cuál es el formato y qué campos están presentes en un mensaje de Twitter, el Anexo 1 incluye un ejemplo de mensaje de creación de nuevo Tweet, y el Anexo 2 muestra un ejemplo de borrado de un Tweet.

```
{
  "textFilters": {
    "text": ["refugee", "refugiado", "fluechtlinge", "flüchtlinge"]
  },
  "name": "containsRefugeeByTimezone",
  "groupBy": ["user.time_zone"],
  "ifHasNot": "delete"
}
```

*Figura 25 Definición de una Métrica*

En la versión finalmente implementada las posibilidades de generación de métricas están limitadas en cuanto a función de agregación (únicamente una cuenta), campos de filtrado (únicamente filtrado por texto) y composición del filtro (únicamente función OR). En el capítulo 6 veremos posibles extensiones futuras para poder potenciar la expresividad de estas métricas.

Una vez definidas las métricas, éstas son recogidas por el Agregador de forma periódica, con lecturas a un directorio local. Al igual que en el resto del proyecto, se ha seguido el principio de diseño de responsabilidad simple [12]. Cada clase se encarga de una única función, por lo que para tratar las métricas nos encontramos con tres interfaces dentro del código del Agregador, siguiendo el principio I de SOLID: “Muchas interfaces cliente específicas son mejores que una interfaz de propósito general”. Las interfaces son:

- La interfaz `FileRetriever`. Define los métodos que debe implementar una clase que lea ficheros. Debe poder listar los ficheros en un directorio, y debe poder extraer el contenido de un fichero.
- La interfaz `MetricDefinitionParser`. Define los métodos que debe implementar una clase que parsee archivos de métricas. Está formada de un único método *parse* que recibe una cadena de texto y devuelve una `MetricDefinition`.
- La interfaz `MetricDefinitionRetriever`. Define los métodos que debe implementar una clase que provea de la lista de métricas al sistema. Define un único método *get*, que no recibe parámetros y devuelve la lista de definiciones de métricas.

Para la implementación del sistema de recolección de métricas, además del principio de responsabilidad simple, se han implementado dos patrones o principios más:

- El principio de **composición sobre herencia**, que indica que es más favorable de cara a la reutilización del código utilizar una composición de clases que una herencia, siempre que sea posible.
- El patrón **decorador**. Dado que no queremos acceder al disco con mucha frecuencia, se han implementado dos `MetricDefinitionRetriever`: Uno accede directamente a disco y parsea los ficheros presentes en el directorio correspondiente, mientras que el otro actúa de decorador del primero añadiendo una caché.

Las clases que implementan dichos patrones son la clase `FileMetricDefinitionRetriever`, que implementa `MetricDefinitionRetriever` para acceder directamente al directorio correspondiente, parsear las métricas y devolver la lista. La segunda clase que implementa la interfaz `MetricDefinitionRetriever` es la `CachedMetricDefinitionRetriever`. Ésta hace uso, por composición, de cualquier clase que implemente `MetricDefinitionRetriever` (`FileMetricDefinitionRetriever` en la implementación final del sistema), y decora su funcionamiento con una caché, de manera que, si la caché aún es válida, se devolverá la lista cacheada, y si la caché es inválida, se llamará a la implementación del `MetricDefinitionRetriever` que se haya pasado en el constructor.

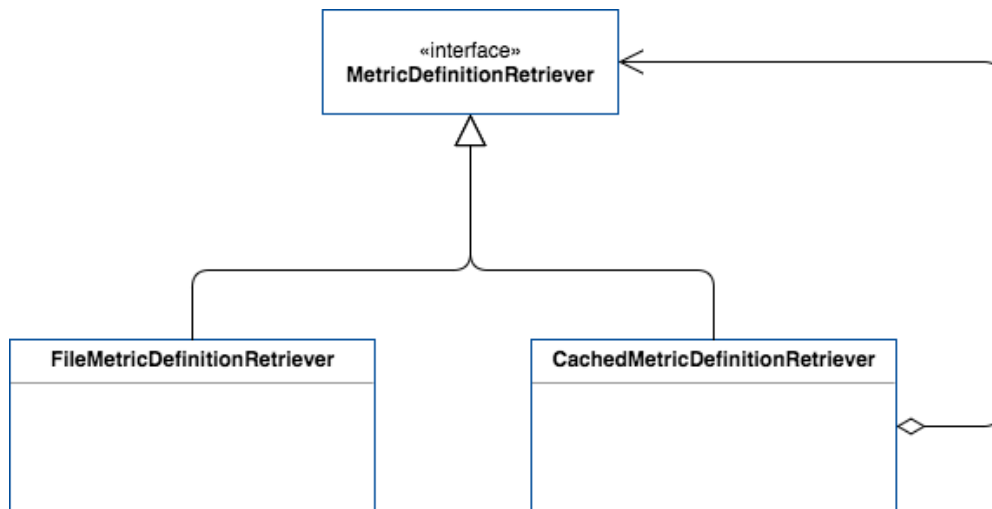


Figura 26 Diagrama de clases del recolector de definiciones de métricas

#### 4.3.d. Agregador

El Agregador es el componente más complejo del sistema. Es el encargado de recibir los mensajes de Kafka, leer las métricas creadas por el usuario, agregar los valores atendiendo a las métricas definidas en ventanas de 30 segundos, y enviar los datos al topic de agregaciones. Dado que el componente anterior debía ser lo más simple posible, éste es el que carga con toda la complejidad del sistema.

Como vimos en la sección 2.4 del Capítulo 2, existen varias temporizaciones que se pueden tener en cuenta a la hora de realizar agregaciones sobre un *stream* de datos. En este caso, la elección ha sido la de tomar el momento de la inserción en la cola de entrada. Es una elección que compromete la corrección de los datos a cambio de garantizar cierta velocidad de entrega. No obstante, esta decisión viene dada por los siguientes motivos:

1. El componente que escribe los datos a Kafka, es decir, el inyector de Twitter, no realiza cálculos complejos con los datos, sino que se limita a volcar los datos con las mínimas transformaciones posibles al *topic* de Kafka. Esto introduce un error mínimo, del rango de los milisegundos, que podemos aceptar como válido para el cálculo de métricas que queremos realizar.
2. Puesto que no controlamos el origen de los datos (Twitter en este caso), si este componente se desconecta, en cuanto recupere la conexión comenzará a recibir datos del instante temporal actual. Esto conlleva una pérdida de información, tal y como se

ha descrito en el apartado anterior. Por lo tanto, en caso de utilizar el instante temporal del origen no nos aporta la seguridad de poder reprocesar datos con cierto retraso, ya que estos no se producirán.

3. El tiempo de procesamiento del evento queda descartado, ya que los requisitos no funcionales incluyen la capacidad de realizar un recómputo de los datos. En este caso, al procesar datos que fueron escritos en Kafka con una diferencia temporal relativamente elevada, estaríamos introduciendo un error que deja de estar en el rango de lo aceptable.

El Agregador implementa el patrón Map-Reduce para realizar los cálculos de agregación. El diagrama de flujo de la lógica principal del Agregador se presenta en la Figura 27.

El paso 1 queda fuera del proceso Map-Reduce, ya que se encarga únicamente de refrescar la caché de métricas. El proceso lee los ficheros definidos por el usuario, presentes en un directorio local. En caso de fallo de lectura por mal formato, o por cualquier otro motivo, este proceso envía un log de *warning* indicando el fallo, y carga las métricas por defecto presentes en el código. En caso de éxito, las métricas por defecto son reemplazadas por las métricas definidas por las métricas cargadas desde disco.

El paso número 2 corresponde con una transformación Filter más una transformación Map. Para cada mensaje del flujo de entrada, se generan cero, una o varias tuplas clave-valor de la siguiente manera:

1. Se calcula el **filtro** de acuerdo a los valores encontrados en el campo 'textFilters' de la métrica. Si el filtro no pasa, no se genera tupla para esta métrica. Si el filtro es satisfactorio, seguimos con el siguiente paso.
2. Se crea una **clave** de la siguiente forma:
  - a. El **nombre** de la clave es el nombre definido en la etiqueta

- b. Se extraen los valores del JSON del Tweet referentes a los campos presentes en la etiqueta 'groupBy' de la métrica. Estos valores junto con los campos de la métrica pasan a formar el mapa *campos* de la clave.

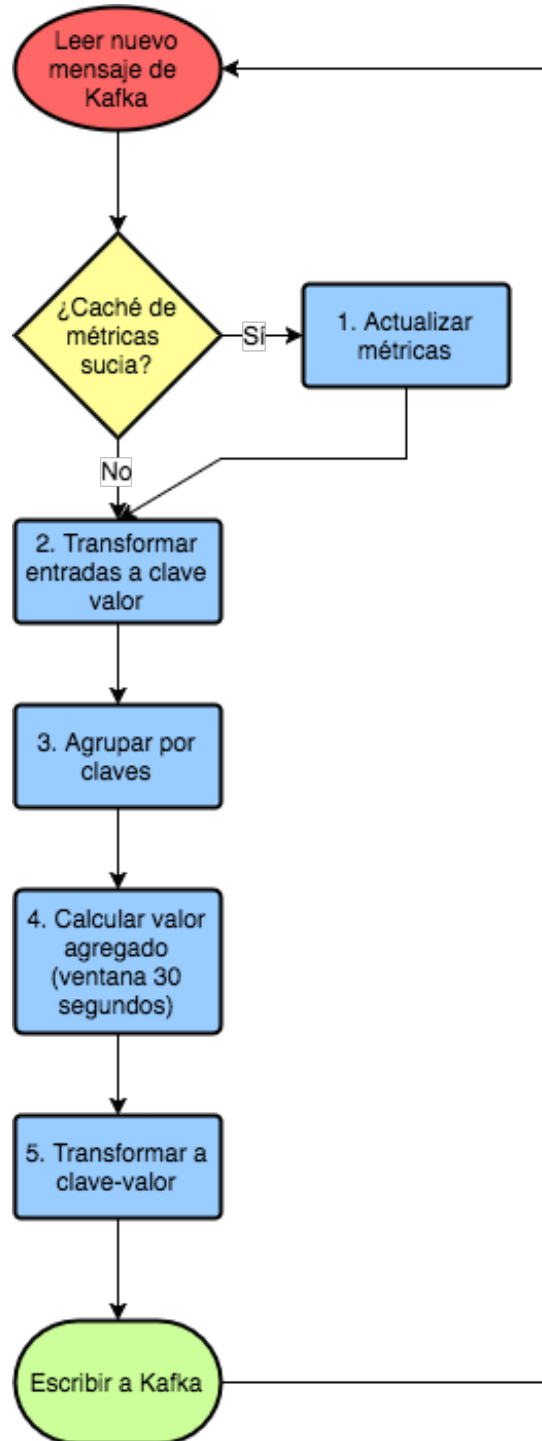


Figura 27 Diagrama de flujo del Agregador

- c. Se añade la **función de agregación**, que en este caso es *count*, al ser la única que se implementa en la solución.
3. Se crea un valor de la siguiente forma:
    - a. Se calcula el **valor** de la función de agregación para el mensaje. En este caso, al ser una función de *count*, el valor para cada mensaje es 1.
    - b. Se añade el campo de la **función de agregación** para poder conocer cómo se agregarán dos valores en el futuro. En este caso, la función es la suma puesto que tratamos con la función *count*.

Para cada mensaje de entrada, y para cada métrica:

El paso número 3 es una agrupación de los elementos que comparten una misma clave (group by key), de manera que todas las métricas que tienen la misma clave, quedan agrupadas en un mismo conjunto, listas para realizar el paso ‘Reduce’ del algoritmo. Esta agrupación realiza una repartición transparente al usuario, creando una escritura intermedia en un *topic* autogenerado. La repartición es necesaria ya que en el paso anterior (un *flat map*) hemos generado nuevas claves, que recordemos, son las que deciden qué partición es la encargada de manejar el mensaje y, por tanto, qué hilo ha de leerlo. Sin este paso intermedio no podríamos asegurar que todos los mensajes de una misma clave acaben en un mismo hilo y, por consiguiente, las agregaciones futuras llevarían a resultados erróneos.

El paso número 4 se corresponde con la fase ‘Reduce’ del algoritmo Map-Reduce. En este caso las reducciones se realizan en ventanas de 30 segundos. Dado que Kafka Streams modela las agregaciones por ventana temporal como un flujo de eventos de actualización, los cálculos son enviados de forma continua. Un nuevo valor sobre una clave que ya ha sido calculada previamente, indica que el valor de dicha casilla en la tabla ha cambiado. Valores nuevos sobre una ventana de tiempo son aceptados siempre que éste llegue dentro del período de validez de la ventana. Si un evento llega fuera de este tiempo, la ventana se considera cerrada, por lo que el evento es descartado. En este caso utilizamos el tiempo retención por defecto de una ventana, que es de 1 día. Esto significa que seguiríamos actualizando el valor del cálculo de una ventana con mensajes con hasta 1 día de retraso.



En el paso número 2 habíamos mapeado cada mensaje a una tupla clave-valor, donde el valor corresponde a su vez de un par de elementos: la función de agregación, y el valor numérico del cálculo de la métrica. En un paso Reduce se reciben dos valores y se combinan para producir un tercer valor. La función aplicada en este proceso es la siguiente:

$$reducción(A, B) = (A.función, A.función(A.valor, B.valor))$$

Dado que nos hemos asegurado de que únicamente procesaremos valores con la misma clave, y la función de agregación es parte de la misma, podemos estar seguros que  $A.función$  es igual a  $B.función$ , lo que nos permite utilizar de manera indistinta cualquiera de las dos.

El resultado de esta reducción se presenta en un formato de ventana interno de la librería Kafka Stream, por lo que es necesario un último paso, el paso número 5, antes de escribir el resultado a Kafka. Para cada valor, calculamos el instante de tiempo correspondiente con la ventana y escribimos el resultado al *topic* de agregaciones en Kafka.

El código de la clase ‘TwitterMetricKafkaStreamWriter’ recoge todos los pasos descritos en este proceso, como podemos ver en la Figura 28.

```
class TwitterMetricKafkaStreamWriter(
  metricDefinitionRetriever: MetricDefinitionRetriever,
  outputTopic: String,
  aggPeriod: Duration = 30.seconds
)
extends StreamWriter[Integer, String] with LazyLogging {

  override def write(stream: KStream[Integer, String]): Unit = {
    stream
      .flatMap[AggregationKey, AggregationValue](new InputToAggKeyAggValueMapper(metricDefinitionRetriever))
      .groupByKey(new JSONSerde[AggregationKey], new JSONSerde[AggregationValue])
      .reduce(new AggregationValueReducer, TimeWindows.of(aggPeriod.toMillis))
      .toStream
      .map[AggregationKey, AggregatedValue](new WindowedToAggregatedValueMapper)
      .to(new JSONSerde[AggregationKey], new JSONSerde[AggregatedValue], outputTopic)
  }
}
```

Figura 28 Código de la topología del Agregador

Para este componente, al igual que para el resto, se han tomado una serie de decisiones de diseño que siguen la práctica de *fail fast* [14], o “falla pronto”. Esta práctica es considerada una buena metodología para el diseño de software moderno. El sistema implementado está compuesto de microservicios que se comunican entre ellos de manera asíncrona. Cada uno de estos pequeños componentes están especialmente diseñados para aislarse de errores producidos en el resto.

Además, un fallo en el sistema no implica una pérdida de datos, sino un retraso en la entrega de los mismos.

En estos casos es preferible hacer que el sistema falle ante un error imprevisto, a que siga funcionando produciendo datos incompletos o erróneos. Para una persona que tiene que mantener la operatividad del sistema es más sencillo recuperarse de un error en un componente que deja de funcionar, que realizar un recómputo porque éste estaba funcionando con datos parciales o erróneos. Es por esto por lo que, en caso de error de conexión, error de parseo de alguna de las métricas definidas, o cualquier error en los datos de entrada, la excepción causa el aborto completo del Agregador.

Si el sistema se llegara a desplegar en un entorno en el que su funcionamiento fuese crítico, bastaría con crear alertas cuando un componente deja de funcionar para ser lo suficientemente ágil a la hora de reparar el error. Ya que tenemos un buffer que nos garantiza una cierta maniobrabilidad, esta práctica es la más aconsejable en estos casos.

### ***4.3.e. Escritor a InfluxDB***

El escritor a InfluxDB es el último componente que se conecta con Kafka. Este componente se encarga de leer las métricas ya agregadas del *topic* de agregaciones de Kafka, y escribirlas en InfluxDB.

Como observamos, la escritura se realiza en lotes por temas de eficiencia. La gestión del buffer la realiza de forma interna la *influxdb-java* que se utiliza para la escritura.

A la hora de escribir, debemos realizar cuidadosamente la asignación de cada elemento de la métrica a su correspondiente elemento en InfluxDB. Como hemos visto en el Capítulo 3, InfluxDB es una base de datos de series temporales. El conjunto de (*timestamp*, clave, *tags*) identifica unívocamente un valor de métrica. En el caso de las métricas realizadas en este sistema, el *timestamp* viene dado por el tiempo de la ventana agregada, la clave es el nombre de la métrica, y los *tags* son el conjunto de campos que hemos especificado en la sección *groupBy* de la definición de la métrica, junto con el valor concreto en el fichero JSON. Como último elemento del punto de InfluxDB, como campo tendríamos un único par clave-valor con *count* en la clave, y el valor del cálculo como valor.

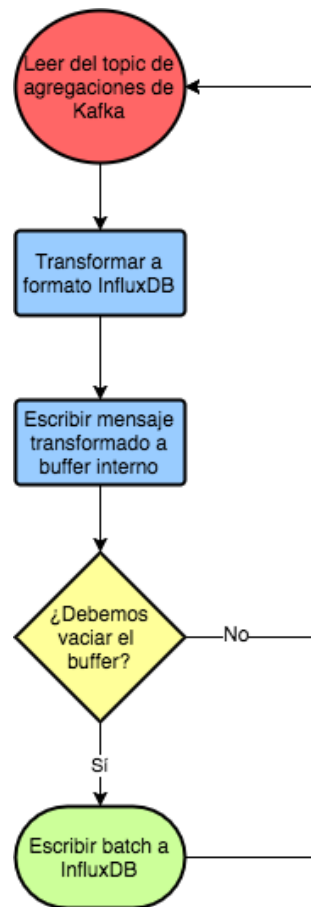


Figura 29 Diagrama de flujo del Escritor a InfluxDB

Tomemos como ejemplo la métrica de la Figura 25 de la sección 4.3.c. Imaginemos que la aplicamos a una serie de eventos que llegan en una ventana que va desde las 16:30:00 a las 16:30:30 de hoy. Imaginemos además que llegan 4 mensajes en esa ventana temporal con los valores dentro del JSON (expresando sólo el path de los elementos implicados por simplicidad):

- Mensaje 1:
  - “text”: “España recibe menos refugiados de los que debería”
  - “user.time\_zone”: “Madrid”
- Mensaje 2:
  - “text”: “Esta mañana he desayunado cereales, me siento espléndido”
  - “user.time\_zone”: “EST”

- Mensaje 3:
  - “text”: “Hoy he aprendido una nueva palabra en alemán: flütlinge”
  - “user.time\_zone”: “Berlin”
- Mensaje 4:
  - “text”: “Me siento como un refugiado en mi casa”
  - “user.time\_zone”: “Madrid”

Tras pasar por el Agregador, y llegar al escritor a InfluxDB, nos quedarían dos puntos, que se escribirían a InfluxDB como se muestra en la Figura 30.

```
timestamp: '16:30:00', name: 'containsRefugeeByTimezone', tags: ['user.time_zone':  
'Berlin'], fields: ['count': 1]  
timestamp: '16:30:00', name: 'containsRefugeeByTimezone', tags: ['user.time_zone':  
'Madrid'], fields: ['count': 2]
```

*Figura 30 Puntos escritos en InfluxDB tras aplicar una definición de métrica*

#### **4.3.f. Despliegue mediante Docker-compose**

Para desplegar la solución se ha optado por definir un despliegue con Docker-compose. Cada componente está integrado dentro de una imagen Docker, por lo que con Docker-compose podemos orquestar su despliegue de manera sencilla. La Figura 31 muestra el fichero YAML de despliegue de la solución.

Se definen todos los componentes que se han ido detallando a lo largo del capítulo, además de los componentes de los que depende la solución, como son InfluxDB, Grafana y Kafka.

#### **4.3.g. Otras consideraciones**

Además de los patrones concretos que se han expuesto, a lo largo del desarrollo del sistema se han seguido otra serie de principios, que se exponen a continuación:

Principios **SOLID** y **Clean Code** [12] [13]. Se han seguido algunos de los principios de Clean Code, que promueven aumentar la legibilidad del código. El código debe ser lo suficientemente claro para el lector como para no necesitar de comentarios, y si algo necesita de

ellos, es un signo de que debe ser simplificado. Por otro lado, los principios SOLID definen cinco pautas (uno por cada sigla) aplicables al software Orientado a Objetos:

```
version: '2'
services:
  kafka:
    image: spotify/kafka
    ports:
      - "9092:9092"
      - "2181:2181"
    container_name: kafka
  influxdb:
    image: influxdb
    ports:
      - "8086:8086"
      - "8083:8083"
    volumes:
      - $PWD/influxdb:/var/lib/influxdb
      - $PWD/influxdb.conf:/etc/influxdb/influxdb.conf:ro
    command: -config /etc/influxdb/influxdb.conf
    environment:
      - INFLUXDB_USER:$INFLUXDB_USER
      - INFLUXDB_USER_PASSWORD:$INFLUXDB_USER_PASSWORD
      - INFLUXDB_ADMIN_ENABLED=true
    container_name: influxdb
  grafana:
    image: grafana/grafana
    depends_on:
      - influxdb
    ports:
      - "3000:3000"
    volumes:
      - grafana:/var/lib/grafana
    links:
      - influxdb
    container_name: grafana
  twitter-ingestion:
    image: twitter-ingestion
    container_name: twitter-ingestion
    depends_on:
      - kafka
    volumes:
      - $PWD/../../secrets/reference.conf:/opt/twitter-ingestion/application.conf:ro
    links:
      - kafka:zookeeper
      - kafka
  aggregator:
    image: aggregator
    container_name: aggregator
    volumes:
      - $PWD/metrics:/opt/twitter-ingestion/metrics
    depends_on:
      - kafka
    links:
      - kafka:zookeeper
      - kafka
  toInflux:
    image: kafka-to-influxdb
    container_name: kafka-to-influxdb
```

```
depends_on:
- kafka
- influxdb
links:
- kafka:zookeeper
- kafka
- influxdb
volumes:
grafana:
```

*Figura 31 Fichero de despliegue de Docker-compose*

- S: Principio de responsabilidad simple. Un objeto sólo debe tener una responsabilidad. Esto implica que el software estará compuesto por muchas clases de extensión pequeña. Cada clase se dedica a hacer una única tarea, y la composición de todas ellas logra comportamientos más complejos. Se puede generalizar igualmente al nivel de componente software, donde tendríamos muchos componentes pequeños y simples que interactúan entre sí para solucionar un problema complejo.
- O: Principio de abierto/cerrado. Las entidades software deben estar abiertas para su extensión y cerradas para su modificación. Esto implica que no se deben realizar extensiones de funcionamiento mediante la modificación de los componentes existentes. Al nivel de clases, esto se logra mediante la inyección de dependencias a través del constructor. Si se quiere modificar el comportamiento de la clase, se creará una nueva clase y se inyectará en el constructor, de manera que se minimiza las modificaciones de las clases. A nivel de componente, las extensiones se deben realizar añadiendo nuevos componentes al sistema que interactúen con los ya existentes en vez de modificando los mismos.
- L: Principio de sustitución de Liskov. Los objetos de un programa deben ser reemplazables por instancias de sus subtipos sin alterar el correcto funcionamiento del programa. Esto nos fuerza a limitar la herencia a los casos que realmente son necesarios, y evitar herencias que rompen con la idea de similitud entre ambas.
- I: Principio de segregación de la interfaz. Muchas interfaces cliente específicas son mejores que una interfaz de propósito general. Esto permite la implementación del principio S de una manera más sencilla. Si tenemos interfaces pequeñas, podemos componer varias para generar un comportamiento más complejo. A su vez, este comportamiento puede ser implementado por distintos componentes.

- D: Principio de inversión de la dependencia. Se debe depender de abstracciones y no de implementaciones. Viene de la mano con el principio O. Si dependemos de abstracciones (interfaces o *traits* en el caso de este proyecto) su extensión será más sencilla sin necesidad de modificaciones.

***Test Driven Development.*** La metodología *Test Driven Development* (de sus siglas, TDD) aboga por centrar el diseño del software de manera que facilite la generación de tests. En su definición más estricta, los tests deben ser escritos antes que el código, de manera que la implementación está totalmente sugestionada por éstos.

La metodología TDD indica, además, que se deben crear el mayor número de test unitarios posibles, reduciendo el número de test de integración al mínimo posible. Los tests deben estar centrados en una unidad (*unit*) y el resto de dependencias deben ser sustituidas por versiones *mock* (versiones controladas de las mismas).

Todo caso de uso y todo comportamiento esperado debe tener un test para ello. Si algún comportamiento no está expresado en un test, se entiende que es un comportamiento de efecto secundario, o indeseado.

Se lleve a la práctica de manera estricta o algo más laxa, sus beneficios son varios:

- El código se simplifica de sobremanera, ya que sólo se implementa aquello que es necesario para cumplir con el test. Si no hay un test para una funcionalidad, ésta no debe estar presente puesto que se entiende que no es relevante.
- Los tests sirven como documentación del funcionamiento del sistema. Al escribir los tests en primer lugar, se crean de forma clara y concisa, y ayudan a un posible lector a entender perfectamente cuál es la funcionalidad esperada, tanto en casos normales como los casos extremos tenidos en cuenta.
- El código generado es fácil de extender. Puesto que TDD te fuerza a escribir el mayor número de test unitarios posibles, debes aislar la clase de dependencias externas. Para ello, las dependencias deben ser inyectadas de alguna manera que te permita cambiarlas por implementaciones *mock*.

- Al tener la gran mayoría del código cubierto con test, cualquier modificación es sencilla de realizar, puesto que puedes estar seguro que si algo deja de funcionar, algún test fallará en su ejecución. La implementación puede parecer tediosa al principio, pero incrementa en gran medida la mantenibilidad y extensibilidad del código.

Las características principales que un test debe tener son las siguientes:

- Un test debe ser simple, y testear una única característica del código.
- Los test deben ser legibles, son la fuente principal de documentación del código. El título del test debe ser el comportamiento que se pretende testear. Es buena práctica seguir la estructura *given/when/then*. Primero se prepara el escenario, segundo se ejecuta el caso a testear, y tercero se comprueban que el comportamiento es el deseado.
- Se deben codificar tests tanto para los caminos satisfactorios de ejecución, como para los caminos excepcionales.
- Cada batería de test debe estar centrada en una unidad o *unit*, aislada del resto de sus dependencias. Las dependencias externas han de ser sustituidas por versiones controladas o *mocks*.

La Figura 32 muestra un ejemplo de un test que sigue estos principios. Se trata del test implementado para la clase `CachedMetricDefinitionRetriever`. Esta clase depende de otra `MetricDefinitionRetriever`, ya que implementa el patrón decorador. La dependencia se sustituye con una instancia de `MetricDefinitionRetrieverStub`, que se trata de un *mock* controlado de la interfaz `MetricDefinitionRetriever`. En este caso, se testean los dos comportamientos posibles de la caché, cuando debe ser renovada, y cuando debe ser consultado el valor almacenado en la misma.



```

class CachedMetricDefinitionRetrieverSpec extends FunSpec with Matchers with Eventually {

  implicit val defaultPatience =
    PatienceConfig(timeout = Span(5, Seconds), interval = Span(1, Seconds))

  def fixture(results: List[String], duration: Duration) = new {
    val unit = new CachedMetricDefinitionRetriever(new MetricDefinitionRetrieverStub(results), duration)
  }

  describe("A CachedMetricDefinitionRetriever") {
    it("should return the cached element if cache didn't expire") {
      val f = fixture(List("first", "second"), 20.minutes)

      f.unit.get.head.name shouldBe "first"
      f.unit.get.head.name shouldBe "first"
    }

    it("should return the second element if cache expired") {
      val f = fixture(List("first", "second"), 2.seconds)

      f.unit.get.head.name shouldBe "first"
      eventually {
        f.unit.get.head.name shouldBe "second"
      }
    }
  }
}

```

Figura 32 Test implementado para la clase CachedMetricDefinitionRetriever



# Capítulo 5. *Funcionamiento del software*

En este capítulo se explicará cómo funciona el sistema implementado. Se darán detalles de cómo ejecutar el sistema partiendo desde el código fuente, así como un tutorial de como visualizar las gráficas configuradas por defecto, y como extender la interfaz gráfica con una nueva métrica definida por el usuario. [15] [16] [17] [18] [19]

## 5.1. Despliegue del sistema

El sistema se compone de diferentes elementos, pero su despliegue se ha automatizado atendiendo a el primer requisito no funcional presentes en la Sección 4.1. Para la automatización, se han empotrado todos los ejecutables en imágenes Docker, y se ha descrito un fichero de configuración de despliegue que puede ser ejecutado con Docker-compose.

El despliegue se divide en dos fases, una primera de compilación y generación de imágenes docker, y una segunda de despliegue de la aplicación en sí. Para la primera fase, los requisitos del sistema son:

1. Sistema operativo UNIX. Necesitamos generar imágenes Docker que van a ejecutarse en entornos UNIX, por lo que es requisito indispensable disponer de un entorno UNIX donde compilar la solución. Tanto Linux como Mac OS en su nueva versión Sierra nos permiten la ejecución nativa de Docker. Para versiones anteriores de Mac OS, o el sistema operativo Windows, necesitaremos de una máquina virtual con un *kernel* con compatibilidad nativa con Docker.
2. Máquina virtual de Java. Necesitamos tener la máquina virtual de java instalada, además del JDK. La versión necesaria es la 1.8, ya que utilizamos librerías compiladas para dicha versión.
3. Scala. El proyecto está escrito en Scala, por lo que necesitaremos del mismo modo tener el compilador de Scala instalado en el sistema.

4. SBT<sup>23</sup>. SBT es una herramienta de desarrollo Java y Scala similar a Maven<sup>24</sup> o Ant<sup>25</sup>. Se requiere de la versión 0.13 de la misma.
5. Docker. Necesario para la generación de imágenes sobre las cuales se ejecutan los componentes de la solución.

Para el despliegue, no obstante, los requisitos del sistema se reducen. A continuación, encontramos la lista de los mismos:

1. Un sistema operativo basado en Unix. Necesitamos un sistema operativo que nos permita ejecutar de forma nativa las imágenes Docker generadas.
2. Docker. Para la ejecución de las imágenes generadas. En su desarrollo se ha hecho uso de la versión 17.09 para Mac OS, aunque no debería presentar problemas en cualquier versión de 2017 en adelante, ya que no se usan características avanzadas.
3. Docker-compose. Es la librería que nos permitirá ejecutar el despliegue con un único comando. Debemos tener el comando presente en el sistema.

Una vez configurado el entorno de compilación y de ejecución, disponemos de un script *bash* para compilar y generar todos los artefactos necesarios. El script está en el directorio raíz del proyecto y se denomina *build.sh*. Situándonos en el directorio raíz y ejecutando el script, se encargará de realizar todas las compilaciones y de generar todas las imágenes Docker. El contenido del script es bastante sencillo y se muestra en la Figura 33. Dicho código se encarga de limpiar posibles artefactos antiguos, compilar, generar un ejecutable en forma de *.jar*, copiarlo al directorio donde tenemos el *Dockerfile* de cada uno de los proyectos, y ejecutar *docker build* para crear la imagen Docker.

---

<sup>23</sup> [www.scala-sbt.org](http://www.scala-sbt.org)

<sup>24</sup> [maven.apache.org](http://maven.apache.org)

<sup>25</sup> [ant.apache.org](http://ant.apache.org)

```

sbt clean && \
sbt twitter-ingestion/assembly && \
cp twitter-ingestion/target/scala-2.11/twitter-ingestion.jar twitter-ingestion/docker/twitter-ingestion.jar && \
cd twitter-ingestion/docker && \
docker build --no-cache -t twitter-ingestion . && \
cd ../../ && \
sbt aggregator/assembly && \
cp aggregator/target/scala-2.11/aggregator.jar aggregator/docker/aggregator.jar && \
cd aggregator/docker && \
docker build --no-cache -t aggregator . && \
cd ../../ && \
sbt kafka-to-influxdb/assembly && \
cp kafka-to-influxdb/target/scala-2.11/kafka-to-influxdb.jar kafka-to-influxdb/docker/kafka-to-influxdb.jar && \
cd kafka-to-influxdb/docker && \
docker build --no-cache -t kafka-to-influxdb .

```

Figura 33 Script de compilación y generación de imágenes Docker

Una vez que hemos compilado, ya debemos disponer de las imágenes Docker generadas por lo que podemos pasar al despliegue. La ejecución requiere de varias variables de entorno para su ejecución:

- INFLUXDB\_USER. Esta variable de entorno contiene el usuario para acceder a la base de datos InfluxDB.
- INFLUXDB\_USER\_PASSWORD. Esta variable de entorno contiene la contraseña para poder acceder a la base de datos InfluxDB.

Una vez elijamos el usuario y la contraseña de nuestra base de datos, y hagamos *export* de ambas, podemos realizar el despliegue.

```

export INFLUXDB_USER=user
export INFLUXDB_USER_PASSWORD=password

```

Figura 34 Comandos para especificar las credenciales de acceso a la base de datos

Para el despliegue únicamente necesitamos ejecutar el comando ‘docker-compose up’ en el directorio raíz del proyecto, y en unos segundos dispondremos del sistema completo desplegado en nuestro entorno. En el Anexo 2 podemos encontrar el fichero de Docker-compose utilizado para definir el despliegue. Tras la ejecución de *docker-compose up*, el comando *docker ps* muestra información general de cada uno de los componentes. En la Figura 35 podemos observar el resultado de ejecutar dicho comando. Los campos más importantes que debemos fijarnos son la columna IMAGE que describe qué imagen se está ejecutando, y la columna STATUS que nos indica su estado y cuánto tiempo lleva en el mismo.

Para poder observar los logs específicos de cada imagen siempre podemos hacer uso de *docker logs*. Para más información de cómo gestionar los containers docker, encontramos guías de usuario con todo detalle en la web oficial de Docker<sup>26</sup>.

```
➔ ~ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
32a470aa700b	grafana/grafana	"/run.sh"	21 minutes ago	Up 21 minutes
03a78a554c65	kafka-to-influxdb	"java -cp /opt/kaf..."	21 minutes ago	Up 21 minutes
7d5fef1a0cd2	influxdb	"/entrypoint.sh -c..."	21 minutes ago	Up 21 minutes
4f7bdf0910b8	twitter-ingestion	"java -Dconfig.fil..."	21 minutes ago	Up 21 minutes
eddb3e03ded1	aggregator	"java -cp /opt/agg..."	21 minutes ago	Up 21 minutes
e6de6d00421a	spotify/kafka	"supervisord -n"	21 minutes ago	Up 21 minutes

Figura 35 Containers de Docker en ejecución tras el despliegue

## 5.2. Acceso a la interfaz gráfica

Por defecto, el Sistema cuenta con una serie de métricas que podemos encontrar en el directorio *metrics/* en la raíz del proyecto. Así pues, tras ejecutar el sistema ya podemos hacer uso de la interfaz gráfica para poder visualizar los resultados.

Si estamos utilizando la versión nativa de Docker, todas los *containers* estarán ejecutándose en la dirección local de la máquina (*localhost*). En caso de estar ejecutando Docker en una máquina virtual, o una máquina remota, la IP cambiará dependiendo de la configuración de red que dispongamos. En este proyecto se supone que contamos con la instalación Docker nativa, por lo que supondremos que todos los *containers* se ejecutan en local.

Grafana es la interfaz gráfica, y como observamos en la configuración de despliegue, su puerto 3000 está mapeado al puerto 3000 de nuestra máquina local. Conocido esto, podemos ir a su interfaz gráfica accediendo a la dirección <http://localhost:3000>.

<sup>26</sup> docs.docker.com

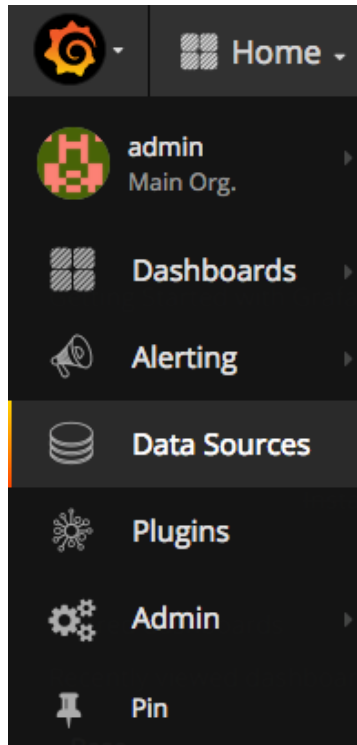



Figura 36 Acceso a la configuración de la base de datos en Grafana

Dentro de Grafana, lo primero que hay que hacer es conectar la interfaz gráfica a una base de datos. Para ello accedemos al panel *Datasources* que encontramos en la esquina superior izquierda de la pantalla, como muestra la Figura 36.

Una vez en el panel de configuración de la base de datos, añadimos una nueva fuente de datos en el botón Add data source  y nos aparecerá el menú de configuración, que para conectarlo con nuestra base de datos InfluxDB, debemos configurarlo como muestra la Figura 36.

Podemos observar que la URL de la base de datos está configurada a `http://influxdb:8086`. Esto funciona puesto que hemos creado un *link* entre el *container* Docker Grafana y el *container* Docker InfluxDB. Grafana tiene configurado su archivo host para apuntar a la IP de InfluxDB mediante el nombre *influxdb*. La base de datos especificada en el campo Database corresponde con la base de datos donde estemos escribiendo los resultados de las agregaciones. Podemos consultar su valor en la propiedad *influx.database* del escritor de InfluxDB. Por otra parte, la configuración de credenciales corresponde con el nombre y usuario que hayamos configurado en el despliegue con los comandos de la Figura 34.

Tras configurar la fuente de datos de InfluxDB, estamos en disposición de crear un *dashboard* para visualizar los datos que el sistema está calculando. Abrimos de nuevo el menú principal y pinchamos en Dashboards -> New como muestra la Figura 38.

Una vez entramos en el nuevo dashboard, seleccionamos el botón de Graph, pinchamos en *Panel Title*, y después en editar, y ya podemos configurar la consulta para nuestra gráfica. En la Figura 39 encontramos un ejemplo de consulta que podemos realizar.

En este caso, se ha realizado una consulta de la serie *autogen*, la métrica count. La métrica count define la agregación más simple que se puede generar en nuestro sistema, que es una cuenta de todos los mensajes consumidos.

En la parte WHERE podemos definir filtros de etiquetas. En nuestro caso, la métrica count no define ningún valor en su campo groupBy que define las etiquetas en InfluxDB, por lo que no podemos filtrar ningún valor.

La parte SELECT indica qué campos queremos consultar en InfluxDB. El único campo que expone nuestro sistema es el campo *count*, puesto que la única función de agregación soportada. A la selección del campo le sigue una función de agregación local. El funcionamiento de ésta se explicará a continuación.

El selector de GROUP BY define las agrupaciones que queremos realizar en la gráfica. Podemos definir tanto agrupaciones temporales como agrupaciones por etiquetas. Las agrupaciones definen qué representa un punto. En este caso, no disponemos de etiquetas por lo que únicamente podemos definir una agrupación temporal. Para la agregación temporal podemos elegir cualquier granularidad que sea múltiplo de la granularidad de origen. Como estamos realizando agregaciones temporales de 30 segundos, podemos elegir 30 segundos, 1 minuto, 1 minuto y 30 segundos, etc.

La diferencia entre este selector y la agregación temporal que se realiza en el componente Agregador del sistema, es que la agregación definida en Grafana se realiza en el momento de consultar los datos, por lo que no es óptimo en caso de consultas con un gran volumen de datos.



Name	Test InfluxDB		Default	<input checked="" type="checkbox"/>
Type	InfluxDB			

### HTTP settings

URL	http://influxdb:8086	
Access	proxy	

### HTTP Auth

Basic Auth	<input checked="" type="checkbox"/>	With Credentials		<input type="checkbox"/>
TLS Client Auth	<input type="checkbox"/>	With CA Cert		<input type="checkbox"/>

Skip TLS Verification (Insecure) ☒

### Basic Auth Details

User	user
Password	*****

### InfluxDB Details

Database	test		
User		Password	

Min time interval 10s

Figura 37 Configuración de la base de datos InfluxDB en Grafana

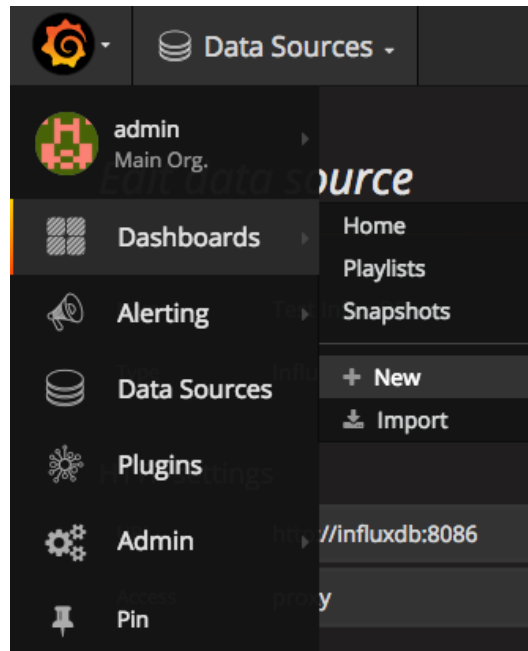


Figura 38 Creación de un nuevo dashboard en Grafana

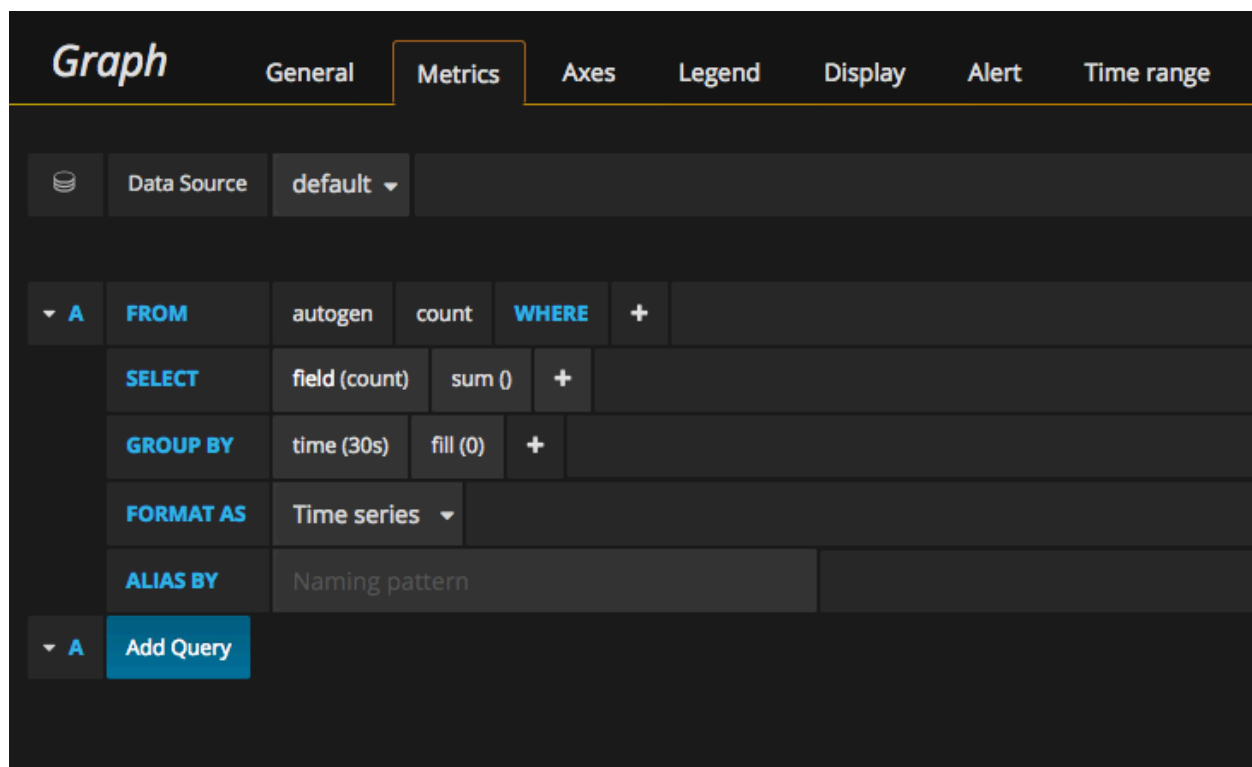


Figura 39 Configuración de una consulta en un gráfico de Grafana

El campo de función de agregación que habíamos dejado pendiente de la parte SELECT aplica a qué función de agregación se realizará en la agregación que especificamos en el GROUP BY. En caso de resultar más de un punto por parte de la consulta a la base de datos, en este caso, queremos sumar sus valores puesto que estamos realizando una cuenta. Si el resultado almacenado en la base de datos fuese una media, por ejemplo, deberíamos especificar una función de media igualmente para no inducir a error en la gráfica.

Por último, podemos elegir cómo queremos actuar ante valores temporales que no tienen ningún dato. En este caso, puesto que de no haber datos de origen que generen una métrica nuestro sistema no manda valores a la cola de agregaciones, queremos hacer un relleno con 0 en caso de no encontrar valores para una fecha en concreto.

El resto de campos disponibles en la configuración atienden a preferencias de visualización como título del gráfico, formato de representación de los puntos, formato de la leyenda, etc. En caso de tener más interés en éstos, es de gran utilidad acceder a la documentación oficial de Grafana.

La Figura 40 muestra el resultado que obtenemos tras configurar la gráfica de ejemplo.

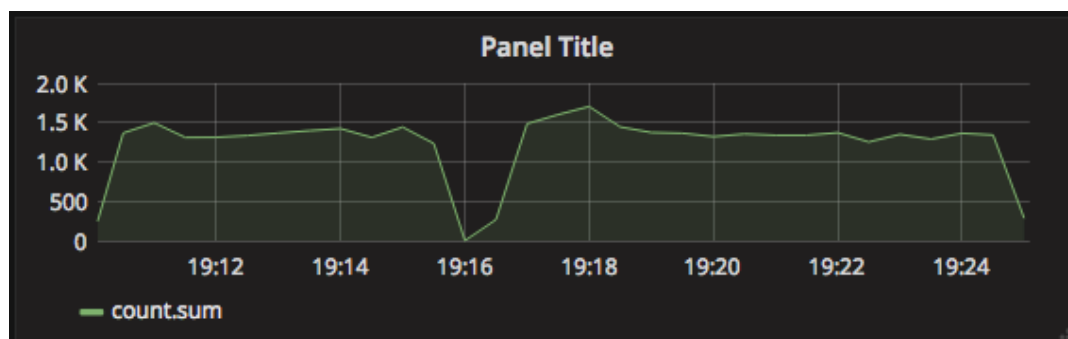


Figura 40 Gráfica del número de mensajes recibidos cada 30 segundos

### 5.3. Configuración de una nueva métrica

En la sección anterior hemos utilizado una métrica ya existente en el sistema. No obstante, es posible definir nuevas métricas de una forma bastante sencilla.

La definición de una métrica sigue el formato JSON detallado en la sección 4.3.e y se debe almacenar en el directorio *metrics/* dentro del directorio raíz del proyecto. Esta métrica será

incluida automáticamente en el conjunto de métricas del Agregador, y podremos ver el resultado en nuestro dashboard.

Así pues, definamos una nueva métrica que cuente el número de veces que aparece la palabra ‘feliz’ agregada por idioma. Ya que la palabra feliz es española, deberíamos recibir un mayor número de mensajes en idioma español. La métrica que queremos definir es la que se muestra en la Figura 41.

```
{
  "textFilters": {
    "text": ["feliz"]
  },
  "name": "containsFelizByLang",
  "groupBy": ["user.lang"],
  "ifHasNot": "delete"
}
```

Figura 41 Definición de métrica 'contiene feliz por idioma'

Desgranando la definición tenemos que:

- El campo **textFilters**, que indica los filtros de texto, tiene un único par clave valor: filtraremos aquellos mensajes de Twitter que contengan en el campo *text* el texto “feliz”.
- El campo **name** contiene el nombre de la nueva métrica, en nuestro caso *containsFelizByLang*
- El campo **groupBy** contiene la lista de campos por los que queremos agregar. En este caso, queremos agregar por idioma del usuario, así que elegimos el campo *user.lang*, que en el mensaje de entrada contiene el idioma del usuario que ha creado el tweet.
- Por último, añadimos un filtro **ifHasNot** que nos filtra los mensajes que no tienen el campo especificado. En este caso, dado que los mensajes que indican el borrado de un Tweet no nos interesan, filtraremos aquellos mensajes que no contengan el campo *delete*.

Una vez guardado el fichero en el directorio *metrics/* el Agregador lo aplicará cuando refresque su caché de 10 segundos. Por lo que ya podemos pasar a configurar la nueva gráfica en Grafana.

En grafana, añadimos una nueva gráfica al panel que teníamos en la Sección 5.2, y configuramos la consulta como muestra la Figura 42. Tras unos ajustes de visualización como el título y la leyenda mostrados en las figuras 43 y 44, podemos ver los resultados que obtenemos en el gráfico de la Figura 45. Tras dejarlo ejecutar durante unos minutos, si ordenamos por el total de mensajes recibidos, vemos como el idioma español junto con el portugués aparecen a la cabeza, tal y como era de esperar.

The image shows the Grafana 'Graph' panel configuration interface. The 'Metrics' tab is selected. The configuration is as follows:

Tab	Field	Value
General	Data Source	default
Query A	FROM	autogen
	containsFelizByLang	WHERE +
	SELECT	field (count) sum () +
	GROUP BY	time (30s) tag (user.lang) fill (0) +
	FORMAT AS	Time series
ALIAS BY	[[tag_user.lang]]	
Query B	Add Query	

Figura 42 Configuración de la consulta para la gráfica 'contiene feliz por idioma'

**Graph** General Metrics Axes Legend Display Alert Time range

**Info**

Title	Contiene Feliz por Idioma
Description	Panel description, supports markdown & links

**Dimensions**

Span	3
Height	100px
Transparent	<input type="checkbox"/>

**Drilldown / detail link** ⓘ

+ Add link

Figura 43 Configuración de título para la gráfica 'contiene feliz por idioma'

pt **Graph** General Metrics Axes Legend Display Alert Time range

**Options**

Show	<input checked="" type="checkbox"/>
As Table	<input checked="" type="checkbox"/>
To the right	<input type="checkbox"/>

**Values**

Min	<input checked="" type="checkbox"/>	Max	<input checked="" type="checkbox"/>
Avg	<input checked="" type="checkbox"/>	Current	<input checked="" type="checkbox"/>
Total	<input checked="" type="checkbox"/>	Decimals	auto

**Hide series**

With only nulls	<input type="checkbox"/>
With only zeros	<input type="checkbox"/>

Figura 44 Configuración de leyenda para la gráfica 'contiene feliz por idioma'



Figura 45 Gráfica 'contiene feliz por idioma'





# *Capítulo 6. Conclusiones y extensiones futuras*

## **6.1. Conclusiones**

En este proyecto se ha creado un sistema de generación y visualización de métricas agregadas en tiempo real sobre un *stream* de Twitter, implementando la arquitectura Kappa. El sistema es capaz de generar nuevas métricas en tiempo real a partir de la acción del usuario, así como configurar *dashboards* de visualización en una interfaz gráfica.

Para la implementación de la arquitectura se han analizado en detalle la arquitectura Lambda y la arquitectura Kappa, dos de las principales soluciones al procesamiento de eventos a gran escala, explicando sus ventajas e inconvenientes. Se ha hecho de igual manera un análisis bastante amplio de las tecnologías elegidas para implementar cada pieza de la arquitectura, y los motivos de su elección.

Como resultado de la implementación encontramos un sistema capaz de ingerir sin problemas el flujo de datos de la API gratuita de Twitter. El sistema cumple con los requisitos especificados para el proyecto, tal y como se ha detallado en el Capítulo 4. Se han implementado una serie de principios y patrones que incrementan la calidad del código, facilitando el mantenimiento y la extensibilidad del sistema, así como proveyéndolo de una mayor resiliencia.

El balance personal de la realización del proyecto es en su mayoría positivo, puesto que se han logrado implementar todos los requisitos funcionales, junto con los no funcionales definidos al inicio del mismo. Se han seguido metodologías de desarrollo actuales, así como tecnologías y arquitecturas vanguardistas, que han requerido la consulta de numerosas fuentes de información, algunas de ellas incluso han aparecido durante el transcurso del proyecto, o estaban en fase de revisión.

El proyecto comenzó con la idea de implementar el cálculo de métricas en tiempo real Tweets. Los requisitos cambiaron durante la implementación del proyecto para adaptar el sistema a cualquier formato JSON de entrada y no hacerlo únicamente dependiente del formato que Twitter

proporciona. Puesto que se siguió la metodología Kanban, cambiar los requisitos en mitad del proyecto no fue ningún problema, y no afectó de manera grave a la planificación del mismo.

La implementación final es sorprendentemente sencilla de utilizar, y bastante robusta. Si los componentes fallaran en algún momento, dichos fallos no afectarían al correcto funcionamiento del sistema, sino que únicamente introduciría ciertos retrasos en la entrega de los datos.

Por otro lado, implementar la metodología de Test Driven Development no es sencillo, pero sus resultados son sorprendentes en la calidad del código generado. Esto junto con las pautas y principios de Clean Code hacen que el código acabe siendo muy legible y fácilmente extensible. Si recorremos el total de las clases de la implementación observaremos que el tamaño de cada clase es bastante pequeño, ya que cada una de ellas tiene un único propósito en la implementación. El código carece de comentarios y, pese a ello, es en su mayoría totalmente legible. Los tests ayudan al usuario a entender la funcionalidad esperada de las piezas del sistema que puedan parecer más oscuras en un primer vistazo.

## 6.2. Ampliaciones futuras

El proyecto ofrece una gran cantidad de posibilidades en cuanto a su ampliación. El procesamiento de eventos en tiempo real en gran volumen es una materia en auge en los últimos años y existen numerosas aplicaciones a las que se puede aplicar. A continuación, se resumen algunas de las principales áreas de extensión del proyecto:

### 6.2.a. *Diferentes fuentes de datos de entrada*

Con el cambio de requisito para aceptar mensajes de cualquier fuente JSON, la primera posible ampliación es la de crear nuevos conectores con diferentes fuentes de datos. Cualquier dato que provenga en JSON, o se pueda convertir a un formato JSON es apto para ser inyectado en el sistema.

La inclusión de nuevas métricas puede venir acompañado de un componente intermedio entre el Agregador y los diferentes inyectores que enriquezca los mensajes con nuevos campos, de manera que sea aún más sencillo diferenciarlos en la fase de agregación. Una posible solución quedaría con los inyectores escribiendo a diferentes *topics* Kafka, un tipificador que se encargue

de leer de los diferentes *topics* y les asigne un campo nuevo denominado tipo al JSON de entrada, y de esta manera el Agregador podría filtrar fácilmente el origen de datos sin necesidad de recurrir a los campos ya existentes *IfHas* o *IfHasNot*.

### ***6.2.b. Nuevos tipos de métricas***

Otro de los campos por los que se puede extender el proyecto es dándole más expresividad a las definiciones de métricas. Actualmente la única función permitida es la cuenta de eventos. Se podría realizar otro tipo de agregaciones como medias, percentiles, o incluso cambiando el formato de salida, incluir datos geográficos que puedan ser representados en un mapa.

### ***6.2.c. Lenguaje específico para la definición de métricas e interfaz gráfica***

Durante el proyecto se ha implementado un lenguaje de generación de métricas bastante simple. En caso de ser extendido, el manejo directo de definiciones en JSON podría llegar a ser tedioso.

Un posible punto de extensión sería la generación de un Lenguaje de Dominio Específico con una sintaxis más sencilla que el formato JSON, en conjunto con una interfaz gráfica de usuario para la definición de métricas. Podría incluir una sección donde se definen mensajes de entrada de ejemplo, y se muestran los resultados que se obtendrían tras aplicar las métricas, con lo que el usuario podría testear su definición antes de ser insertada en el sistema.

### ***6.2.d. Despliegue en la nube y escalado***

Como se ha expuesto a lo largo del proyecto, el sistema está especialmente diseñado para ser fácilmente desplegable en la nube. La encapsulación de los componentes en imágenes Docker convierte el despliegue en la nube en una tarea más que factible. Sin embargo, dado que las limitaciones de los datos de entrada causan que el tráfico del sistema pueda ser manejado por un único ordenador personal, el despliegue en la nube se ha quedado fuera del ámbito del proyecto.

Son numerosas las plataformas Cloud que permiten el despliegue de containers Docker. Encontramos entre ellas Kubernetes, Cloud Foundry u OpenShift<sup>27</sup>. En este entorno en la nube,

---

<sup>27</sup> [www.openshift.com](http://www.openshift.com)

probar el escalado y sus efectos en el rendimiento del sistema al conectarlo a un sistema de entrada que disponga de una mayor carga podría aumentar el valor del sistema.

# Referencias

- [1] D. Reinsel, J. Gantz and J. Rydning, "Data Age 2025: The evolution of Data to Life-Critical," IDC, 2017.
- [2] T. Akidau, C. Bradshaw, C. Chambers, C. Slava, R. J. Fernández-Moctezuma, L. Reuven, S. McVeety, D. Mills, F. Perry, E. Schmidt and S. Whittle, "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing," *Proceedings of the VLDB Endowment*, vol. 8, pp. 1792-1803, 2015.
- [3] P. Sareen and P. Kumar, "NoSQL Database and its Comparison with SQL Database," *International Journal of Computer Science & Communication Networks*, vol. 5, no. 5, pp. 293-298.
- [4] N. Marz and J. Warren, *Big Data. Principles and best practices of scalable realtime data systems*, Manning, 2015.
- [5] J. Kreps, "Questioning the Lambda Architecture," 2014.
- [6] J. Kreps, *I Heart Logs*, O'Reilly Nodua, 2014.
- [7] B. Wilder, *Cloud Architecture Patterns*, O'Reilly Media, 2012.
- [8] "Internet Live Stats," Real Time Statistics Project, 2017. [Online]. Available: <http://www.internetlivestats.com/>. [Accessed 10 2017].
- [9] D. J. Andersson, *Kanban: Successful Evolutionary Change for Your Technology Business*, Blue Hole Press, 2010.
- [10] J. Sutherland, *Scrum: The art of Doing Twice the Work in Half the Time*, Crown Business, 2014.
- [11] S. Newman, *Building Microservices*, O'Reilly Media, 2015.
- [12] R. C. Martin, *Agile Software Development, Principles, Patterns, and Principles*, Pearson, 2002.
- [13] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftmanship*, Prentice Hall, 2008.
- [14] M. Fowler, "Fail Fast," *IEEE Computer Society*, pp. 21-25, 2004.
- [15] H. C. M. Andrade, B. Gedik and D. S. Turaga, *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*, Cambridge University Press, 2014.
- [16] M. Barlow, *Real-Time Big Data Analytics: Emerging Architecture*, O'Reilly Media, 2013.
- [17] B. Ellis, *Real-Time Analytics: Techniques to Analyze and Visualize Streaming Data*, John Wiley & Sons, 2015.
- [18] M. Kleppmann, *Making Sense of Stream Processing*, O'Reilly Media, 2016.
- [19] G. Shapira, N. Narkhede and T. Palino, *Kafka: The Definitive Guide*, 1st Edition (Early Access) ed., O'Reilly Media, 2017.

- [20] "Kafka Documentation," [Online]. Available: [kafka.apache.org/documentation](http://kafka.apache.org/documentation). [Accessed 10 2017].
- [21] "InfluxDB Documentation," [Online]. Available: [docs.influxdata.com/influxdb/v1.3/](http://docs.influxdata.com/influxdb/v1.3/). [Accessed 10 2017].
- [22] "Grafana Documentation," [Online]. Available: [docs.grafana.org](http://docs.grafana.org). [Accessed 10 2017].
- [23] "Scala Documentation," [Online]. Available: [docs.scala-lang.org](http://docs.scala-lang.org). [Accessed 10 2017].
- [24] "Docker Documentation," [Online]. Available: [docs.docker.com](http://docs.docker.com). [Accessed 10 2017].

# *Anexo 1. Mensaje ejemplo de Twitter. Creación de un Tweet.*

```
{
  "text": "RT @PostGradProblem: In preparation for the NFL lockout, I will be
spending twice as much time analyzing my fantasy baseball team during ...",
  "truncated": true,
  "in_reply_to_user_id": null,
  "in_reply_to_status_id": null,
  "favorited": false,
  "source": "<a href=\"http://twitter.com/\" rel=\"nofollow\">Twitter for
iPhone</a>",
  "in_reply_to_screen_name": null,
  "in_reply_to_status_id_str": null,
  "id_str": "54691802283900928",
  "entities": {
    "user_mentions": [
      {
        "indices": [
          3,
          19
        ],
        "screen_name": "PostGradProblem",
        "id_str": "271572434",
        "name": "PostGradProblems",
        "id": 271572434
      }
    ],
    "urls": [],
    "hashtags": []
  },
  "contributors": null,
  "retweeted": false,
  "in_reply_to_user_id_str": null,
  "place": null,
  "retweet_count": 4,
  "created_at": "Sun Apr 03 23:48:36 +0000 2011",
  "retweeted_status": {
    "text": "In preparation for the NFL lockout, I will be spending twice as much
time analyzing my fantasy baseball team during company time. #PGP",
    "truncated": false,
    "in_reply_to_user_id": null,
    "in_reply_to_status_id": null,
    "favorited": false,
    "source": "<a href=\"http://www.hootsuite.com\"
rel=\"nofollow\">HootSuite</a>",
    "in_reply_to_screen_name": null,
    "in_reply_to_status_id_str": null,
    "id_str": "54640519019642881",
    "entities": {
```

```

    "user_mentions": [],
    "urls": [],
    "hashtags": [
      {
        "text": "PGP",
        "indices": [
          130,
          134
        ]
      }
    ]
  },
  "contributors": null,
  "retweeted": false,
  "in_reply_to_user_id_str": null,
  "place": null,
  "retweet_count": 4,
  "created_at": "Sun Apr 03 20:24:49 +0000 2011",
  "user": {
    "notifications": null,
    "profile_use_background_image": true,
    "statuses_count": 31,
    "profile_background_color": "C0DEED",
    "followers_count": 3066,
    "profile_image_url":
"http://a2.twimg.com/profile_images/1285770264/PGP_normal.jpg",
    "listed_count": 6,
    "profile_background_image_url":
"http://a3.twimg.com/a/1301071706/images/themes/theme1/bg.png",
    "description": "",
    "screen_name": "PostGradProblem",
    "default_profile": true,
    "verified": false,
    "time_zone": null,
    "profile_text_color": "333333",
    "is_translator": false,
    "profile_sidebar_fill_color": "DDEEF6",
    "location": "",
    "id_str": "271572434",
    "default_profile_image": false,
    "profile_background_tile": false,
    "lang": "en",
    "friends_count": 21,
    "protected": false,
    "favourites_count": 0,
    "created_at": "Thu Mar 24 19:45:44 +0000 2011",
    "profile_link_color": "0084B4",
    "name": "PostGradProblems",
    "show_all_inline_media": false,
    "follow_request_sent": null,
    "geo_enabled": false,
    "profile_sidebar_border_color": "C0DEED",
    "url": null,
    "id": 271572434,
    "contributors_enabled": false,

```



```

        "following": null,
        "utc_offset": null
    },
    "id": 54640519019642880,
    "coordinates": null,
    "geo": null
},
"user": {
    "notifications": null,
    "profile_use_background_image": true,
    "statuses_count": 351,
    "profile_background_color": "C0DEED",
    "followers_count": 48,
    "profile_image_url":
"http://a1.twimg.com/profile_images/455128973/gCsVUnofNqqyd6tdOGevR0vko1_500_normal
.jpg",
    "listed_count": 0,
    "profile_background_image_url":
"http://a3.twimg.com/a/1300479984/images/themes/theme1/bg.png",
    "description": "watcha doin in my waters?",
    "screen_name": "OldGREG85",
    "default_profile": true,
    "verified": false,
    "time_zone": "Hawaii",
    "profile_text_color": "333333",
    "is_translator": false,
    "profile_sidebar_fill_color": "DDEEF6",
    "location": "Texas",
    "id_str": "80177619",
    "default_profile_image": false,
    "profile_background_tile": false,
    "lang": "en",
    "friends_count": 81,
    "protected": false,
    "favourites_count": 0,
    "created_at": "Tue Oct 06 01:13:17 +0000 2009",
    "profile_link_color": "0084B4",
    "name": "GG",
    "show_all_inline_media": false,
    "follow_request_sent": null,
    "geo_enabled": false,
    "profile_sidebar_border_color": "C0DEED",
    "url": null,
    "id": 80177619,
    "contributors_enabled": false,
    "following": null,
    "utc_offset": -36000
},
    "id": 54691802283900930,
    "coordinates": null,
    "geo": null
}

```



## ***Anexo 2. Mensaje ejemplo de Twitter. Borrado de un tweet***

```
{  
  "delete": {  
    "status": {  
      "id": 1234,  
      "id_str": "1234",  
      "user_id": 3,  
      "user_id_str": "3"  
    }  
  }  
}
```