Centro Universitario de la Defensa en la Escuela Naval Militar

TRABAJO FIN DE GRADO

Plataforma big data para el análisis de flujos de información marítima

Grado en Ingeniería Mecánica

ALUMNO: Ramón González Guitart

DIRECTOR: Norberto Fernández García

CURSO ACADÉMICO: 2019-2020

Universidade Vigo

Centro Universitario de la Defensa
en la Escuela Naval Militar

TRABAJO FIN DE GRADO

Plataforma big data para el análisis de flujos de información marítima

Grado en Ingeniería Mecánica

Intensificación en Tecnología Naval

Cuerpo General

Universidade Vigo

RESUMEN

Desde hace unos años es necesario que ciertas embarcaciones porten un sistema de identificación automática (AIS, por su acrónimo en inglés). Este sistema emplea radiobalizas para transmitir cada cierto tiempo eventos que dan a conocer datos tales como la identidad, posición o velocidad de la embarcación a otros barcos u organismos de vigilancia y coordinación del tráfico marítimo. Analizar el flujo continuo de los eventos AIS de multitud de embarcaciones tiene gran interés para aplicaciones de seguridad, pero puede suponer un problema de *big data*, dado el volumen de eventos a considerar.

Teniendo esto en cuenta, en este proyecto se realiza un estudio exhaustivo de las más populares herramientas de *big data* para el análisis de flujos de información. Tras este proceso, se comparan y se elige una de ellas, Apache Storm, principalmente en base a las características cualitativas que la singularizan y distinguen de sus homólogas. A continuación, se muestra de manera detallada cómo proceder a la instalación, configuración y prueba en un entorno realista de la herramienta escogida. Para ello, se desarrolla una aplicación Java que permite detectar en tiempo real eventos AIS que se generan dentro de una serie de áreas de interés configurables.

PALABRAS CLAVE

Big data, Análisis, Flujos, AIS, Apache Storm

AGRADECIMIENTOS

Dado que este trabajo pone punto y final a mi etapa en la Escuela Naval Militar, mis agradecimientos van dirigidos principalmente a todos aquellos profesores (civiles y militares) que, con su dedicación y esfuerzo diario, han sido un ejemplo de constancia para mí. Entre ellos, quisiera destacar la labor del profesor Norberto Fernández García, tutor de este TFG, por su ayuda constante en el desarrollo del mismo.

Así mismo, querría mostrar mi gratitud al COVAM (Centro de Operaciones y Vigilancia de Acción Marítima) de la Armada Española por haber proporcionado los datos que se han usado en las pruebas.

De manera más personal, me gustaría dar las gracias a mis compañeros de promoción y a mi familia por su apoyo constante y por los consejos dados a lo largo del desarrollo de este proyecto.

CONTENIDO

Contenido 1
Índice de Figuras 3
Índice de Tablas....6
1 Introducción y objetivos....7
1.1 Big data 7
1.1.1 El problema del big data....7
1.1.2 El big data en el entorno marítimo 8
1.1.3 Las 5Vs del big data marítimo....9
1.2 Objetivos del TFG....11
1.3 Estructura del TFG....11
2 Estado del arte 13
2.1 Empleo del big data en el análisis del tráfico marítimo....13
2.1.1 Primer caso: Extracción de rutas marítimas a partir de datos AIS 14
2.1.2 Segundo caso: Predicción de riesgo en colisiones....17
2.1.3 Tercer caso: Detección de anomalías 18
2.2 Empleo del big data en la gestión medioambiental marítima 20
2.3 Proyectos de interés a nivel europeo y nacional 21
2.3.1 European Maritime Safety Agency (EMSA)....21
2.3.2 Sea Traffic Management (STM)....24
3 Desarrollo del TFG....27
3.1 Análisis de las plataformas 27
3.1.1 Apache Spark....27
3.1.2 Apache Storm 37
3.1.3 Apache Flink....48
3.1.4 Apache Samza 57
3.1.5 Apache Flume....64
3.1.6 Amazon Kinesis Data Analytics (comercial) 67
3.2 Comparación de las herramientas 70
4 Instalación y prueba de Apache Storm....79
4.1 Aplicación de ejemplo 79
4.2 Prueba de la herramienta....81
4.2.1 Entorno de pruebas 82
4.2.2 Configuración de Apache Storm 82

4.2.3 Ejecución de la aplicación en modo local....86
4.2.4 Ejecución de la aplicación en modo remoto....88
5 Conclusiones y líneas futuras....97
6 Bibliografía....99
Anexo I: Fuentes de datos de los Servicios Marítimos Integrados de EMSA....106
Anexo II: Cielo Único Europeo y Proyecto Monalisa 2.0....107
Anexo III: Compañías que usan las herramientas analizadas....108
Anexo IV: Código de la aplicación....109

ÍNDICE DE FIGURAS

Figura 1-1 Las 5Vs del *big data* marítimo [5]....10
Figura 1-2 Mapa producido por MarineTraffic con información en tiempo real sobre la localización de distintos tipos de barcos [8]....11
Figura 2-1 Rutas extraídas a partir de los datos AIS proporcionados por cargueros [9]....16
Figura 2-2 Predicción de riesgo de colisión en un área dinámica [12] 17
Figura 2-3 Procedimiento de detección de anomalías según el proyecto H2020 Big Data Ocean [13] 19
Figura 2-4 Patrón de ruta en el mar Egeo basado en trayectos de ferris durante el año 2016 [13] ..19
Figura 2-5 Mapa de lo que ofrece EMODnet Physics [14]....21
Figura 2-6 ABM detecta comportamientos sospechosos de los barcos [16]....22
Figura 2-7 ABM detecta comportamientos sospechosos de los barcos [16]....22
Figura 2-8 Interfaz Gráfico de Usuario que da acceso a las aplicaciones de EMSA [16] 24
Figura 2-9 Datos del proyecto STM en España según [18] 25
Figura 3-1 Oferta de Apache Spark al usuario final [3] 28
Figura 3-2 Ejemplos de librerías compatibles con Spark [10] 29
Figura 3-3 Diagrama de relaciones establecidas en la ejecución de una aplicación Spark [3] 31
Figura 3-4 Programación con Python o R en Apache Spark [3] 31
Figura 3-5 Transformaciones estrechas [3] 32
Figura 3-6 Transformaciones amplias (*shuffles*) [3] 33
Figura 3-7 Logo de la plataforma AdCreator de Celtra [27]....35
Figura 3-8 Celtra analiza dos mil millones de eventos (1 TB de nuevos datos no comprimidos) por día [30] 36
Figura 3-9 Celtra usa Spark en el análisis de datos [30] 37
Figura 3-10 Logotipo de Apache Storm [32] 38
Figura 3-11 Ejemplo del funcionamiento del paradigma de las colas y los trabajadores [36]....39
Figura 3-12 Ejemplo del funcionamiento del paradigma de colas y trabajadores [36]....40
Figura 3-13 La topología de Apache Storm está formada por *spouts* y *bolts* [31]....41
Figura 3-14 Arquitectura de Apache Storm [39]....42
Figura 3-15 Trident procesa flujos como pequeños lotes de tuplas 44
Figura 3-16 Idea de topología inicial....46
Figura 3-17 Topología anterior implementada con Storm Trident 46
Figura 3-18 Logo de Flink [52]....48
Figura 3-19 Flujo de eventos en desorden donde el orden del tiempo de procesamiento es distinto al del tiempo de evento [52]....50
Figura 3-20 Diagrama con los componentes principales de Flink [52] 51

Figura 3-21 APIs de Flink apiladas en capas [51].....52
Figura 3-22 Aplicación tradicional vs Aplicación basada en eventos [51].....53
Figura 3-23 Análisis por lotes vs Análisis por streaming [51].....54
Figura 3-24 ETL (periódico) vs Canalización de datos (continuo) [51].....55
Figura 3-25 Flink ofrecía a Bouygues la baja latencia y programabilidad deseada [64].....56
Figura 3-26 Gráfico informativo sobre el procesamiento de datos con Flink por parte de Bouygues [64].....56
Figura 3-27 Logo de Apache Samza [65].....57
Figura 3-28 Asignación de tareas en Apache Samza [65].....59
Figura 3-29 Ejecución distribuida en Apache Samza [65].....59
Figura 3-30 Funcionamiento de los *checkpoints* con Apache Samza [65].....60
Figura 3-31 Las tareas tienen su propio estado en Apache Samza [65].....61
Figura 3-32 Samza transforma flujos de entrada y emite resultados a una base de datos.....62
Figura 3-33 Redfin usa Samza para procesamiento de flujo en etapas múltiples [69].....63
Figura 3-34 Logo de Apache Flume [69].....64
Figura 3-35 Entidades principales que componen Apache Flume [69].....65
Figura 3-36 Tecnologías implementadas en CounterTack y su impacto en la detección de anomalías. [74].....67
Figura 3-37 *Streaming* de ETL para el Internet de las cosas (IoT) con aplicaciones Java [81].....69
Figura 3-38 Análisis de registros en tiempo real con SQL [81].....69
Figura 3-39 Tecnología publicitaria y marketing digital con SQL [81].....70
Figura 3-40 Árbol del modelo de decisión [82].....74
Figura 3-41 Impacto de la ventana de tiempo en el número de eventos procesados (100 KB por mensaje).....75
Figura 3-42 Comsumo de CPU [83].....75
Figura 3-43 Consumo de memoria RAM [83].....76
Figura 3-44 Consumo de memoria en disco [83].....76
Figura 3-45 Consumo de ancho de banda [83].....76
Figura 4-1 Esquema de la topología Storm empleada en la aplicación.....80
Figura 4-2 Introducción del comando *ssh* de conexión remota.....82
Figura 4-3 Comandos para la descarga y extracción de Apache Storm.....83
Figura 4-4 Comandos para la descarga y extracción de Apache ZooKeeper.....83
Figura 4-5 Directorio donde se encuentra el archivo “zoo.cfg”.....83
Figura 4-6 Archivo “zoo.cfg” con las modificaciones de los parámetros realizadas.....84
Figura 4-7 Situamos el ejecutable binario de Apache Storm, de Apache ZooKeeper y de OpenJDK en el PATH.....84
Figura 4-8 Formato de los datos AIS que vamos a emplear.....85

Figura 4-9 Número de eventos AIS en el archivo “datos.csv” 85
Figura 4-10 Entrada al directorio “aistorm” y visualización de los subdirectorios y archivos contenidos en él 85
Figura 4-11 Directorio donde se encuentra “storm.yaml” 85
Figura 4-12 Configuración de Apache Storm en el fichero storm.yaml 86
Figura 4-13 Empleo del comando “mvn –e compile” para compilar la aplicación 86
Figura 4-14 Tras ejecutar el comando “mvn –e compile” aparece el directorio “*target*” 87
Figura 4-15 Visualización del directorio “target” con sus respectivos subdirectorios 87
Figura 4-16 Nombres de los “archivos.class” creados en su directorio final de localización 87
Figura 4-17 Utilización del comando “mvn package” para empaquetar los “archivos.class” generados al compilar 87
Figura 4-18 Inicialización del conjunto en modo local 88
Figura 4-19 Alertas contenidas en el fichero “alert.log” 88
Figura 4-20 Equipamiento empleado en la prueba en modo remoto 89
Figura 4-21 Uso del comando “mvn clean” 89
Figura 4-22 Uso del comando “mvn –e compile” 90
Figura 4-23 Uso del comando “mvn package” 90
Figura 4-24 Inicialización del servidor ZooKeeper 90
Figura 4-25 Inicialización del Nimbus de Storm 90
Figura 4-26 Obtención de la dirección IP mediante “ifconfig” 91
Figura 4-27 Uso del comando “scp” para copiar el archivo “area-alert.conf” en el Supervisor 91
Figura 4-28 Contenido del archivo “area-alert.conf” 91
Figura 4-29 Inicialización del Supervisor 91
Figura 4-30 Ejecución de la aplicación en modo remoto 92
Figura 4-31 Activación de la topología de la aplicación 92
Figura 4-32 Los registros en “alert.log” crecen con el tiempo 92
Figura 4-33 Uso del comando “cat alert.log | wc -l” para conocer el número de alertas en cada instante 93
Figura 4-34 Proceso de búsqueda de información por patrones con el comando “fgrep” 93
Figura 4-35 Uso del comando “cut” para recortar información de utilidad 94
Figura 4-36 La información se almacenó en el fichero de texto “salida.txt” para su análisis 94
Figura 4-37 Activación de Storm UI 94
Figura 4-38 Información sobre la configuración del conjunto en Storm 95
Figura 4-39 Métricas de la topología *area-alert* 95
Figura A4-0-1 Código de la aplicación 118

ÍNDICE DE TABLAS

Tabla 1 Áreas de aplicación del *big data* en la industria marítima [7].....9
Tabla 2 Precios de Amazon Kinesis Data Analytics [80] 68
Tabla 3 Comparación resumida de las herramientas de análisis de flujo que son objeto de estudio 71
Tabla 4 Herramientas de procesamiento de flujos analizadas en cada estudio de investigación [82] 72
Tabla 5 Criterios tratados en cada uno de los estudios de investigación [82].....72
Tabla 6 Principales empresas que hacen uso de las herramientas estudiadas 108

1 INTRODUCCIÓN Y OBJETIVOS

1.1 Big data

1.1.1 El problema del big data

La capacidad actual para producir información (datos) se ha incrementado exponencialmente respecto a años anteriores. Es en este contexto donde el Mckinsey Global Institute (MGI) [1], en el año 2011, presentó al mundo por primera vez el término *big data* y lo definió como: "Los conjuntos de datos cuyo tamaño está más allá de la habilidad de las herramientas *software* de base de datos para capturar, almacenar, gestionar y analizar." [2]

Actualmente, el *big data* está jugando un papel fundamental en la llamada Cuarta Revolución Industrial (Industria 4.0), que se fundamenta en el empleo de los grandes volúmenes de datos obtenidos a través de los sistemas ciberfísicos y el Internet de las cosas, para conseguir una industria de factorías inteligentes en las que las máquinas y los distintos recursos se comunican como en una red social. [2]

Para que las factorías y productos sean verdaderamente inteligentes, grandes cantidades de datos deben ser recogidos, analizados e interpretados en tiempo real. Esta necesidad ha llevado al desarrollo de herramientas que permiten el procesado y análisis de los mismos para identificar y extraer la información relevante. [2]

Ahora bien, ¿por qué se necesitan nuevas herramientas para el análisis de datos? De acuerdo a [3], esto se debe principalmente a los cambios en las tendencias económicas que subyacen a las aplicaciones informáticas y de *hardware*.

A lo largo de la historia de la computación, los ordenadores se fueron haciendo más rápidos a medida que aumentaba la velocidad de sus procesadores. Como resultado de esto, se formó un gran ecosistema de aplicaciones (en su mayoría diseñadas para ejecutarse en un solo procesador), las cuales adquirieron, con el paso del tiempo, la propiedad de poder ofrecer resultados de una manera cada vez más veloz. [3]

Desafortunadamente, alrededor del año 2005, los límites en la disipación de calor de los materiales que componían los procesadores provocaron la detención de esta tendencia. Los desarrolladores de *hardware* dejaron de canalizar sus esfuerzos en lograr procesadores individuales más rápidos y se centraron en conseguir altas velocidades de procesamiento mediante la utilización de núcleos que trabajaran en paralelo. Este cambio provocó que un gran número de aplicaciones tuvieran que ser adaptadas para poder aprovechar al máximo el potencial que la computación en paralelo ofrecía. [3]

De la misma manera, el coste de almacenar 1 TB de datos se reducía aproximadamente a la mitad cada catorce meses, por lo que para las organizaciones resultaba muy económico el almacenamiento de grandes cantidades de datos. Además, las tecnologías para recopilar información (sensores, cámaras,

etc.) se abarataron y mejoraron su resolución. Estos factores provocaron que fuera más rentable recolectar amplias y diversas gamas de datos. Ejemplo de ello es como el precio de las cámaras, que son los sensores esenciales en numerosos dispositivos de recolección de datos (telescopios, máquinas de secuenciación de genes, etc.), se vio reducido drásticamente. [3]

Las ideas comentadas previamente han dado lugar a un mundo en el que la recopilación de datos es extremadamente económica: muchas organizaciones podrían incluso considerar negligente el no registrar datos relevantes para el negocio. Sin embargo, el procesamiento de toda esta información requiere de grandes cálculos en paralelo, que a menudo tienen que ser llevados a cabo por un conjunto de máquinas. En este contexto es donde han surgido una serie de potentes herramientas (las cuáles serán explicadas y analizadas en el apartado 3.1) que permiten procesar gran cantidad de información con muy baja latencia. [3]

Respecto a estas herramientas, también resulta de interés destacar que pueden procesar los datos de dos maneras diferenciadas: por lotes (*batch processing*), como MapReduce, o en flujos (*stream processing*), como Apache Storm. El procesamiento por lotes se aplica a grandes volúmenes de datos no continuos, pero no cumple con los requisitos de tiempo real que necesitan hoy en día algunas tareas analíticas. El procesamiento en flujos sí se aplica a los datos continuos y es realmente la solución perfecta para aplicaciones en tiempo real. [4]

Este trabajo centra su interés en el uso de tecnologías de *big data* para el análisis de información marítima, por ello, a lo largo de los siguientes apartados se explorarán los problemas que estas tienen asociados, así como algunos de los requerimientos y utilidades de sus aplicaciones.

1.1.2 El big data en el entorno marítimo

Los datos provenientes del dominio marítimo están creciendo a gran velocidad. De hecho, se dispone de una gran cantidad de datos de fuentes heterogéneas (sensores, boyas, barcos, satélites, etc.) que pueden nutrir de manera potencial un gran número de aplicaciones de protección medioambiental, seguridad, predicción de fallos, optimización de rutas de transporte marítimo y producción de energía. Sin embargo, la existencia de una serie de retos relacionados con el *big data* y la alta heterogeneidad de las fuentes de datos son el principal motivo de que estas aplicaciones estén todavía subdesarrolladas y fragmentadas. [5]

Esta realidad contrasta con el valor de las distintas áreas marítimas, cuya explotación económica es una verdadera fuente de riqueza. Claro ejemplo de este potencial son la pesca, el crecimiento estratégico de las regiones costeras (motivado por el turismo, el transporte o la logística) y las fuentes de energía renovable presentes en estas regiones. [5]

Aparte de los sensores *Global Positioning System* (GPS), cuyos datos se transmiten constantemente a través del *Automatic Identification System* (AIS), la industria de transporte marítimo integra también otro tipo de sensores en sus productos, incluyendo equipamiento que proporciona información del rendimiento del buque o datos de temperatura y humedad. Además, es cada vez más común que los barcos posean medidores de la energía malgastada, o que los pesqueros hagan uso de sónares de búsqueda de peces. Desde un punto de vista medioambiental, otro tipo de sensores están siendo desplegados en el mar para realizar tareas tan diversas como: medir temperaturas, corrientes, viento, luminosidad, nivel del mar y olas, concentración de sal, polución en el agua, movimientos del hielo, riesgo de tsunamis, etc. [5] Así mismo, algunas organizaciones científicas como la *National Oceanic and Atmospheric Administration* (NOAA), la *National Aeronautics and Space Administration* (NASA) o el *National Oceanographic Data Centre* (NODC) proporcionan infraestructuras web integradas para poner a disposición pública datos de sensores marítimos y metadatos (como los que se muestran en [6]).

Por lo tanto, es necesario destacar el enorme valor que se puede obtener del empleo del *big data* en el entorno marítimo. La Tabla 1 muestra solo algunos ejemplos de lo que puede llegar a conseguir en este ámbito: ahorro de energía, optimización de diseño en los buques, seguridad, etc.

PAPEL	FUNCIÓN	EJEMPLO DE APLICACIÓN BIG DATA
OPERADOR DEL BARCO	Operación	Operaciones de ahorro de energía
		Funcionamiento seguro
		Gestión de horarios
	Planificación de flotas	Asignación de flotas
		Planeamiento del servicio
		Alquiler de barcos
PROPIETARIO DEL BARCO	Gestión técnica	Funcionamiento seguro
		Monitorización de condiciones y mantenimiento
		Cumplimiento de la regulación medioambiental
		Limpieza del casco y de la hélice
		Acondicionamiento y modificación
	Nueva construcción	Optimización del diseño

Tabla 1 Áreas de aplicación del *big data* en la industria marítima [7]

En relación con el ámbito tecnológico, el *big data* tiene 5 retos comúnmente reconocidos, en muchos casos llamados las 5Vs del *big data*, que son: volumen, variedad, velocidad, valor y veracidad. Para el análisis de datos heterogéneos procedentes de fuentes diferentes es necesario diseñar los sistemas muy cuidadosamente. Estos sistemas deben estar basados en Internet, para así asegurar acceso continuo y universal a los datos. Además, el paradigma de *High Performace Computing* (HPC) juega un papel muy importante, especialmente cuando es necesario el análisis de datos en tiempo real. [5]

1.1.3 Las 5Vs del big data marítimo

El *big data* se define como los datos cuyo volumen, velocidad de adquisición, variedad, veracidad y valor potencial sobrepasan la capacidad de los sistemas de gestión de datos tradicionales. A continuación se explica el modelo de las 5Vs (Figura 1-1) en su aplicación al escenario marítimo, según [5]:

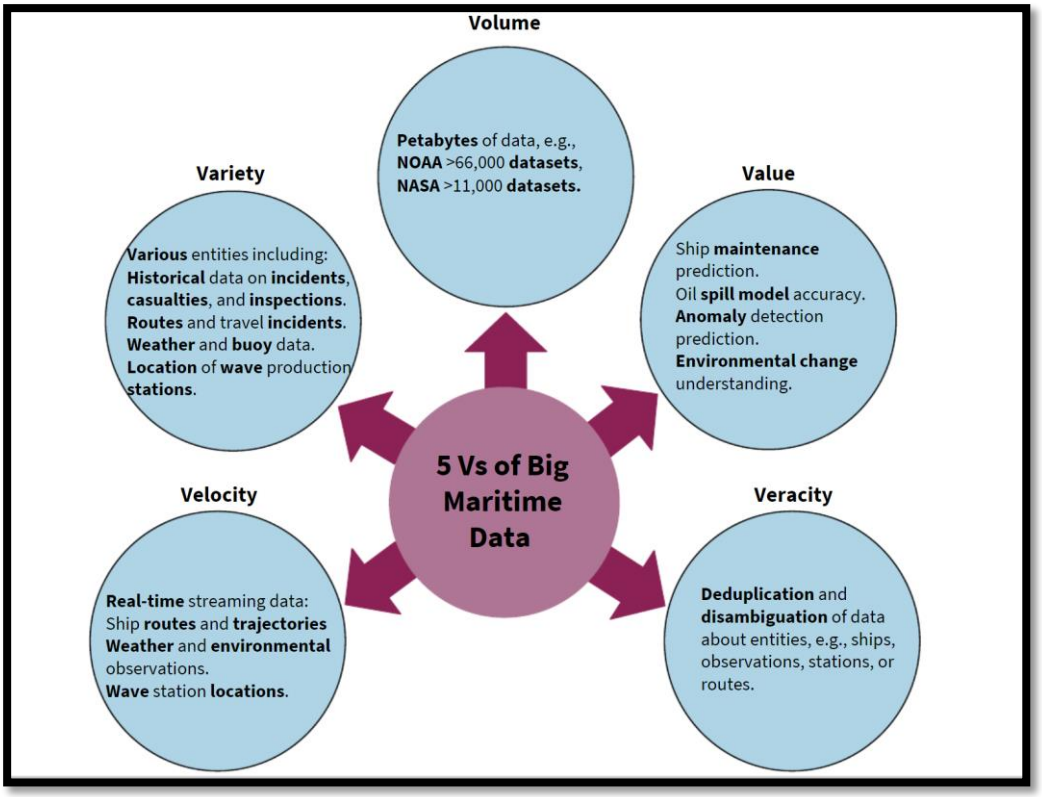

Figura 1-1 Las 5Vs del *big data* marítimo [5]

- Volumen: actualmente existen fuentes de información marítima que proporcionan una enorme cantidad de datos. Existen páginas web públicas de organizaciones científicas, como las ya citadas NOAA, NASA y NODC, que recogen *terabytes* de datos por día, permitiendo el acceso a una gran cantidad de información oceanográfica y de biodiversidad marina.
- Velocidad: las regulaciones marítimas existentes establecen que los barcos deben recoger y almacenar gran cantidad de datos (posición, velocidad, recorrido, estado de las turbinas, etc.). Con estos datos se desarrollan productos que muestran información en tiempo real. De manera similar, la NOAA ofrece en tiempo real las condiciones metereológicas y oceanográficas recogidas cada media hora vía los *Geostationary Operational Enviromental Satellites* (GOES) y otros satélites comerciales. Solo el *NOAA National Centre for Enviromental Information* (NOAA-NCEI) almacena aproximadamente 20 *petabytes* de información atmosférica, costera, oceánica, paleoclimatológica y geofísica cada mes. Además, muchas compañías marítimas publican mapas en tiempo real (Figura 1-2) que muestran los recorridos y la localización de más de medio millón de barcos alrededor del mundo.
- Variedad: los datos marítimos son recogidos de muy distintas maneras y guardados en formatos y estructuras muy variados.
- Veracidad: dado que los datos son recolectados usando métodos muy distintos y mediante instrumentos muy dispares, existen distintos grados de precisión e incertidumbre en ellos.
- Valor: el *big data* marítimo se puede usar para mejorar la calidad de gran número de aplicaciones marítimas. Por ejemplo, los datos meteorológicos y de los barcos representan la base del mantenimiento predictivo en buques y estaciones. A mayores, el historial de datos almacenados puede usarse para llevar a cabo mejoras de eficiencia energética, mientras que los datos de rutas y trayectorias permiten detectar anomalías y proporcionar mayor seguridad en la navegación. Por último, la gestión y monitorización de accidentes y riesgos ambientales puede también conseguirse analizando rutas, datos históricos o en tiempo real, datos meteorológicos, etc.

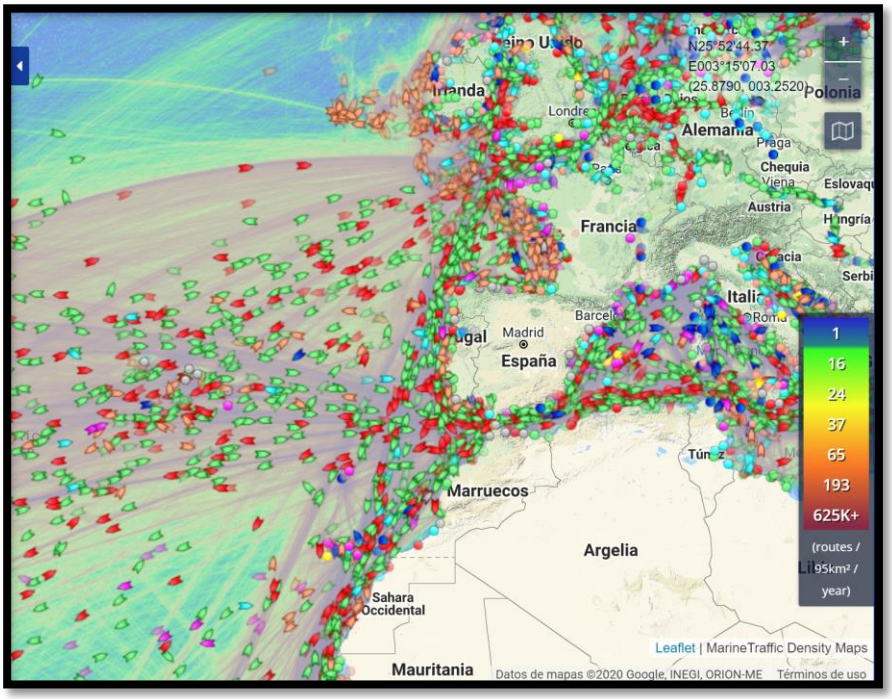

Figura 1-2 Mapa producido por MarineTraffic con información en tiempo real sobre la localización de distintos tipos de barcos [8]

1.2 Objetivos del TFG

Como se ha podido observar a lo largo del apartado 1.1, el *big data* está adquiriendo una importancia suprema en el sector marítimo. Sin embargo, el análisis de esta gran cantidad de datos no es trivial y necesita de potentes herramientas para su procesamiento. Es en este complejo entorno donde surge la necesidad de hacer uso de *frameworks* capaces de llevar a cabo análisis complejos de flujos de datos en tiempo real.

En los últimos diez años han surgido potentes herramientas que, mediante la paralelización de las tareas, consiguen muy bajas latencias en el análisis de datos, así como fiabilidad, tolerancia a fallos y fácil escalabilidad en las aplicaciones desarrolladas.

En la actualidad existen numerosas herramientas de *software* libre dedicadas al análisis de flujos de datos. El objetivo de este trabajo es analizar las más populares para posteriormente compararlas. Tras discutir los pros y contras que tiene cada una de ellas, se procederá a la elección de la que resulte más apropiada para ser instalada y configurada en equipamiento ubicado en la Escuela Naval Militar. Después, sobre el marco de trabajo seleccionado, se ejecutará una aplicación para analizar flujos de datos marítimos en tiempo real. Dicha aplicación extraerá la información de un fichero proporcionado por el COVAM que contiene datos de varios millones de eventos AIS. Por último, se estudiará el rendimiento de la herramienta a partir de los resultados obtenidos.

1.3 Estructura del TFG

El presente proyecto se estructura en los siguientes capítulos:

- Capítulo 1: Introducción y objetivos. Se busca contextualizar el trabajo, así como exponer los objetivos del mismo. Con este propósito se explica qué es el *big data,* incidiendo en la importancia que tiene en la actualidad.
- Capítulo 2: Estado del arte. Se exponen algunos casos prácticos en los que el *big data* procedente del entorno marítimo ha sido de utilidad, así como proyectos relacionados en vías de desarrollo tanto a nivel nacional como europeo.

- Capítulo 3: Desarrollo del TFG. Se analizan distintas herramientas de análisis de *big data*, se realiza una comparación y se escoge la que se considera más adecuada.
- Capítulo 4: Instalación y prueba. Se describe el código de la aplicación que se ejecutará sobre la herramienta escogida. Posteriormente, se explica como instalar, configurar y probar la herramienta tanto en modo local como en modo remoto.
- Capítulo 5: Conclusiones y líneas futuras. Se presentan las principales conclusiones extraídas del desarrollo del trabajo y se proponen nuevas vías de investigación relacionadas con el empleo de las herramientas analizadas.
- Para finalizar, se anexa información que resulta de interés para la documentación del proyecto, como el código de la aplicación utilizada.

2 ESTADO DEL ARTE

2.1 Empleo del big data en el análisis del tráfico marítimo

Para el desarrollo del presente proyecto se han utilizado datos AIS, los cuales están íntimamente relacionados con la monitorización del tráfico marítimo. Por ello resulta interesante conocer lo útil que este tipo de información puede llegar a ser si se analiza convenientemente.

A diferencia de las carreteras, las rutas marítimas varían en forma, límites y contenido a lo largo del espacio y del tiempo. Sobre ellas influyen el comercio y los patrones de los barcos de transporte, así como la inversión en infraestructura, el cambio climático, el desarrollo político y otros eventos de complejidad. [9]

Mientras que en tiempos pasados se carecía de datos para hacer un reconocimiento adecuado del entorno marítimo, actualmente la tecnología de seguimiento ha transformado este problema en una sobreabundancia de información. Esto es debido principalmente a la gran cantidad de datos de seguimiento que lentamente comienzan a estar disponibles, gracias fundamentalmente al AIS. Dada la enorme cuantía de información, se complica el uso de técnicas tales como la extracción de datos tradicional (*data mining)* o el aprendizaje automático (*machine learning*). [9]

A lo largo de la historia, los marinos han definido rutas marítimas para aprovechar los vientos predominantes y las corrientes oceánicas, dando lugar a las denominadas grandes rutas marítimas, las cuales siguen, en su mayoría, actualmente en uso. Estas populares rutas, entre las que se pueden destacar las que cruzan el océano Atlántico, el océano Pacífico y el océano Índico, se ven influenciadas por varios factores que pueden provocar cambios en su tamaño, contenido o conexiones. Uno de estos factores es el calentamiento global, fenómeno por el cual están surgiendo nuevas rutas como la que transcurre por el norte de Rusia en el océano Ártico (que permite acortar el trayecto entre los puertos de China y de Europa en miles de kilómetros) o por el cual las tradicionales rutas marítimas de la costa canadiense están siendo interrumpidas por *icebergs*. Así mismo, la inversión en terminales portuarias y las expansiones de ciertos canales han aumentado el uso de ciertas rutas; como es la reciente ampliación del Canal de Panamá, que ha influido en las tarifas de transporte y en la inversión en determinados puertos. [9]

La importancia de tener un conocimiento bien desarrollado y completo de los patrones de tráfico marítimo y de las rutas de comercio es fundamental para todos los navegantes. Desde el punto de vista de la seguridad, resulta necesario para reconocer áreas de gran congestión. De esta manera los barcos pequeños pueden evitar colisiones con otros de mayor tamaño. Además, un conocimiento de los patrones marítimos puede asistir en la identificación de comportamientos anómalos y puede ayudar a predecir la futura localización de los buques. Adicionalmente, si se combinan rutas marítimas con modelos para

estimar las emisiones de los barcos (que dependen de la distancia navegada, velocidad, calado, condiciones climatológicas y las características propias del barco) se podría llegar a calcular la cantidad de emanaciones por barco y por territorio nacional. En el ámbito económico, la decisión de mandar un barco por una determinada ruta u otra debe ser resultado de alcanzar una óptima relación entre un número de variables como: el camino más corto entre dos puertos, el coste de una ruta, el tráfico esperado, la duración del trayecto, el tamaño del barco, etc. [9]

En los apartados 2.1.1, 2.1.2 y 2.1.3 se exponen tres casos de empleo del *big data* en aspectos relacionados con el tráfico marítimo. Con ellos se pretende mostrar lo útil que puede llegar a ser esta tecnología.

2.1.1 Primer caso: Extracción de rutas marítimas a partir de datos AIS

El AIS es un sistema colaborativo que permite a las embarcaciones transmitir su información a barcos y estaciones costeras cercanas. Los barcos equipados con transceptores AIS envían periódicamente mensajes que incluyen información de identificación o destino junto con otras características provenientes del equipamiento de abordo, como la localización, la velocidad o el rumbo. Los datos AIS proporcionan casi cobertura global, ya que los métodos de recolección de datos no están restringidos a un determinado país o continente, proporcionando la oportunidad de realizar profundos análisis de patrones a escala global, los cuales previamente eran inviables.

El equipo AIS tiene dos fuentes de entrada de información. Una le proporciona los datos aportados por el GPS y la otra, el resto de parámetros introducidos por la tripulación. Todos estos datos son emitidos a través de dos frecuencias de banda marina de VHF: 161,975 MHz (canal 87 B) y 162,025 MHz (Canal 88 B).

El radio de alcance es el propio del VHF (aproximadamente 30 millas náuticas). La información se transmite de forma continua (el intervalo nominal mínimo es de 2 segundos, incrementándose en virtud de la menor velocidad o de las maniobras que ejecuta la nave) y los datos se actualizan automáticamente sin que sea precisa ninguna acción por parte del usuario.

Aparte de la información aportada sobre los barcos de la zona, el AIS es capaz de mostrar si un cruce entre barcos es seguro o no. Para ello, utilizando la cinemática de las embarcaciones involucradas, calcula y muestra el *Closest Point of Aproachment* (CPA) y el *Time to the Closest Point of Aproachment* (TCPA).

Sin embargo, las actuales Tecnologías de la Información y la Comunicación (TIC) y los métodos tradicionales de extracción de datos son puestos a prueba cuando se pide enfrentar los complejos problemas asociados al *big data* para producir inteligencia de utilidad. Los datos geoespaciales del AIS contienen miles de millones de registros. Además, están sesgados y pueden contener unos más datos que otros, haciendo que el procesamiento y el almacenamiento con métodos convencionales sean muy demandantes.

Teniendo lo anterior en consideración, el proyecto descrito en [9] extrae un volumen masivo de datos históricos del AIS para estimar rutas de comercio, sin recurrir a fuente externa de información alguna. Para lograr dicho objetivo, lleva a cabo un método que consta de cuatro fases (las cuales serán descritas con posterioridad) y recurre al paradigma de programación MapReduce. Por último, intenta demostrar la efectividad y la validez de los resultados en condiciones del mundo real.

Este trabajo resulta ser de gran interés, ya no solo por los resultados que ofrece, sino también por las novedades que presenta:

- Computación distribuida: presenta un prototipo arquitectónico capaz de procesar eficientemente miles de millones de mensajes AIS (más de 500 GB) en pocas horas.
- Precisión del algoritmo: hace uso de un algoritmo que genera rutas de comercio apropiadas mediante la superación de la ya conocida inexactitud del AIS. Esto lo consigue de una manera distribuida mediante la adopción de MapReduce.

- Dominio específico: descubre rutas de comercio marítimo global que pueden ser usadas como método de detección de anomalías, investigación, entendimiento y predicción de variaciones en patrones comerciales.

A mayores del problema que impide a los métodos tradicionales analizar *big data* de manera eficiente y rápida, existe otro que consiste en que, a la hora de analizar los datos, los métodos tradicionales suponen que estos poseen una distribución uniforme y espacialmente continua. El AIS, sin embargo, presenta con frecuencia grandes huecos de cobertura geográfica, colisión de mensajes o mensajes erróneos cuando se procesan grandes áreas o conjuntos de datos de gran tamaño.

En el año 2008, Google describió la técnica de programación MapReduce, mientras que poco después, en 2011, Yahoo hizo pública Hadoop (disponible bajo licencia Apache). A pesar de no ser la panacea, Hadoop introdujo a millones de programadores y científicos a la computación paralela y distribuida, comenzando la denominada *big data wave*.

Siguiendo el éxito de Hadoop, numerosas herramientas de fuentes abiertas aparecieron en el ecosistema *big data*. Apache Spark, originalmente diseñada por investigadores en la Universidad de Berkeley, fue desarrollada en respuesta a las limitaciones que presentaba el paradigma MapReduce en Hadoop. Esta herramienta (según la información disponible en [10]) es capaz de procesar datos en memoria cien veces más rápido que Hadoop.

Tras esta breve referencia al uso de *frameworks* que procesan en paralelo y su utilidad sobre lo datos AIS, se describe a continuación cómo se llevó a cabo el proyecto.

De los 64 tipos de mensajes AIS que pueden ser transmitidos por los transceptores (como define la *International Communication Union Recommendation M.1371-5)* [11], el trabajo se centró únicamente en los seis más relevantes, que constituyen aproximadamente el 90% de los escenarios típicos del AIS. Los tipos 1, 2, 3, 18 y 19 son informes de posición, que incluyen latitud, longitud, velocidad sobre fondo, rumbo sobre fondo y otros campos relativos al movimiento de los barcos. Además de los anteriores, los mensajes del tipo 5 contienen información estática y del viaje, que incluye el identificador de la *International Maritime Organization* (IMO), el distintivo de llamada, el nombre, las dimensiones del barco, y el tipo de barco y de carga. En todos los mensajes, cada buque es identificado por el número identificador del servicio móvil marítimo.

Para llevar a cabo los estudios se usaron aproximadamente 5 mil millones de mensajes (525 GB) grabados entre enero y diciembre de 2016. Para las tareas de procesado distribuido, confiaron en un conjunto de ordenadores HDInsight Azure Spark (versión 2.1.0). El grupo estaba formado por 6 nodos de trabajo (D4v2 nodos Azure), cada uno equipado con 8 núcleos de procesado y 28 GB de memoria RAM, sumando un total de 56 núcleos computacionales y 224 GB RAM.

Los pasos seguidos en el procesado fueron los que se exponen a continuación:

1. Limpieza de la trayectoria de los datos y preprocesamiento: los datos AIS no proporcionan información de confianza en relación a la salida y la llegada a puertos. De hecho, los únicos datos relevantes relacionados con este aspecto y recogidos del AIS son los del tipo 5, que incluyen información sobre el puerto de destino. Sin embargo, esta información es dada manualmente por los miembros de la dotación y está sujeta a errores. Por lo tanto, es esencial conocer la información de los datos posicionales del AIS para poder analizar las rutas. Usando un algoritmo calcularon el puerto de destino y el de salida de cada mensaje AIS mediante Spark, así como los tiempos de salida y de llegada respectivos basados en las marcas horarias de los mensajes AIS. El campo de tiempo transcurrido les permitió incluir la dimensión del tiempo en el análisis de los mensajes AIS.
   El trabajo se basó en mensajes AIS procedentes únicamente de cargueros y petroleros, ya que estos siguen patrones repetitivos en sus trayectos, a diferencia de otros como los pesqueros. Así mismo, los mensajes en los que las velocidades eran menores a 0,5 nudos

fueron filtrados. El *dataset* resultante de este proceso fueron mil millones de mensajes AIS de aproximadamente 28.000 barcos diferentes.

2. Enriquecimiento semántico de los datos y clasificación: en el paso de preprocesamiento se usó un conjunto de datos que contenía información de más de 20.000 puertos, lo que suponía la existencia de alrededor de 400 millones de combinaciones entre ellos, número que aumentaba considerablemente si se tenía también en cuenta el tipo de barco. Para poder clasificar convenientemente estos datos se aprovechó la capacidad de paralelización que Spark ofrecía, asignando cada uno de los mensaje a un par de valores clave que identificaban rutas según tipos de barco.
3. Extracción de conocimiento de las trayectorias: de la fase anterior se obtuvo un conjunto de columnas con distintos valores clave. De ellas se pudo deducir que solo existían 368.734 rutas únicas asociadas al valor clave "tipo de barco".
   Como Spark no permite procesos anidados de *MapReduce*, se procesaron simultáneamente múltiples rutas mediante la distribución de sus claves por múltiples nodos. Con el objetivo de captar rutas únicas a partir de todos los puntos incluidos en cada set se llevó a cabo el bien conocido agrupamiento *K-means* usando WEKA (una herramienta de código abierto con algoritmos de aprendizaje automático para tareas de extracción de datos). Las características seleccionadas para realizar el agrupamiento *K-means* fueron la latitud, la longitud y la marca horaria relativa. Esto permitió detectar grupos de posiciones de barcos basados en la localización y en el tiempo, mejorando así la percepción de la ruta.
4. Descubrimiento de rutas alternativas: en los pasos anteriores se ha supuesto que barcos del mismo tipo navegando en la misma ruta seguían trayectorias similares. Sin embargo, esto no es siempre así, ya que las condiciones meteorológicas, el calado, etc. pueden hacer que los buques varíen sus rumbos de manera considerable. En esos casos se observa que la salida de la fase de *clustering* sufre eventos de fluctuación y las trayectorias producidas tienen cambios continuos de rumbo. Esto lleva a pensar que existen al menos dos recorridos diferentes para una misma ruta. Para identificar estos recorridos se aplicó un algoritmo de diferenciación de trayectorias.

El conjunto de datos resultante consistió en un archivo de tan solo 850 MB que incluía información de todos los barcos. El número total de rutas detectadas fueron 440.854, las cuales fueron representadas con 10.847.328 puntos. En la Figura 2-1 se pueden observar las rutas obtenidas.

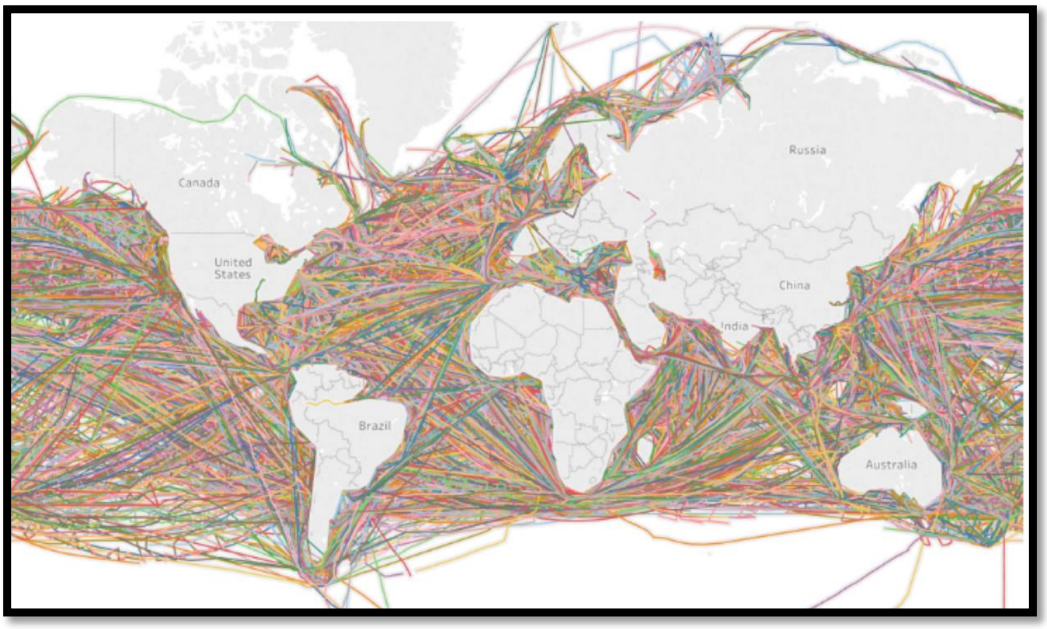

Figura 2-1 Rutas extraídas a partir de los datos AIS proporcionados por cargueros [9]

2.1.2 Segundo caso: Predicción de riesgo en colisiones

A nivel global, especialmente en vías fluviales concurridas, los centros de Control y Gestión del Tráfico Marítimo o *Vessel Traffic Service* (VTS) confían en la advertencia anticolisión de sus sistemas, que utilizan la técnica del punto de aproximación más cercano para alertar a los buques cuando estos se acercan de una manera inusual. A medida que las operaciones e interacciones de los buques se vuelven más complejas, la capacidad de detectar y predecir los movimientos de los barcos por adelantado, especialmente en áreas de tráfico de alta densidad como Singapur, resulta ser clave para administrar y reducir los riesgos de colisión.

Mediante una nota de prensa (cuyo enlace está en [12]) Fujitsu y la Autoridad Marítima y Portuaria de Singapur dejaron constancia de la realización de una prueba basada en inteligencia artificial para predecir el riesgo de colisión de barcos. El día 2 de abril de 2019 anunciaron los resultados obtenidos con el fin de demostrar la utilidad de una nueva tecnología para predecir el riesgo de colisión entre buques.

Esta tecnología de inteligencia artificial fue denominada *Fujitsu Human Centric AI Zinrai*. Entre sus capacidades destacan la detección de riesgo de colisión de barcos y la predicción de las áreas dinámicas críticas donde dichos riesgos se concentran. Adicionalmente, tiene el potencial de poder ser implementada en sistemas VTS, pudiendo así ayudar a los controladores marítimos a gestionar el tráfico de manera proactiva y mejorar la seguridad en la navegación.

La investigación duró 24 meses y contó con la asistencia de 10 oficiales de la Autoridad Marítima y Portuaria de Singapur. Basándose en datos de tráfico del estrecho de Singapur, esta empresa aprovechó su tecnología para extraer información acerca de colisiones o situaciones complejas que involucraban a múltiples embarcaciones, así como de zonas activas de riesgo dinámico (Figura 2-2). Tras esto, se compararon los resultados obtenidos por esta tecnología con los conseguidos por operadores humanos.

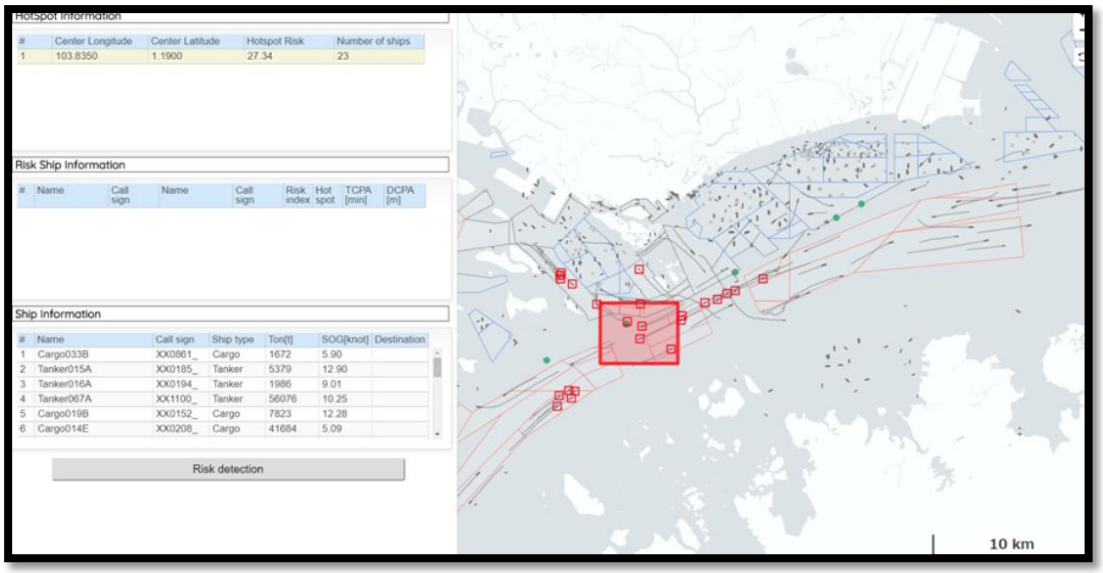

Figura 2-2 Predicción de riesgo de colisión en un área dinámica [12]

Según los estudios de evaluación comparativa, la tecnología de detección pudo identificar riesgos potenciales aproximadamente 10 minutos antes de la situación de choque y, al hacerlo, teóricamente proporcionaba aproximadamente 5 minutos más de tiempo a los operadores humanos para asesorar a los buques. Por otro lado, cabe destacar que el sistema también fue capaz detectar peligro en escenarios en los que se podía pasar por alto el riesgo de colisión.

Sobre la base de estos resultados, Fujitsu continúa trabajando en su mejora. Para el año 2020, la multinacional nipona espera poder ofrecer un servicio de calidad tanto a controladores de tráfico marítimo como a operadores de barcos.

2.1.3 Tercer caso: Detección de anomalías

El centro de Ciencia y Tecnología de la Organización del Tratado del Atlántico Norte (NATO STO *centre)* que trabaja en el ámbito de la investigación y experimentación marítima, programó en mayo del año 2018 un seminario relacionado con el empleo del *big data* bajo el título "*The Maritime Big Data Workshop"*. [13] Este encuentro fue de gran relevancia, pues reunió a grandes investigadores y proveedores de tecnología con el fin de intercambiar su experiencia en campos como la innovación en *big data* para la seguridad marítima, el transporte, la sostenibilidad de la pesca o la explotación de los recursos del océano.

El incremento del tráfico marítimo global y de la explotación de los recursos del océano han contribuido al desarrollo de una serie de innovaciones tecnológicas que garantizan la protección y la seguridad en la navegación marítima, así como un "crecimiento azul". Con este propósito se han desarrollado sistemas de monitorización automática y redes de sensores marítimos. Según un informe publicado por la Agencia Europea de Seguridad Marítima en el año 2017, las aguas europeas son navegadas diariamente por aproximadamente 12.000 barcos, que comparten sus posiciones para evitar colisiones, generando un total de 200 millones de mensajes de posicionamiento cada mes, que a su vez son recibidos por alrededor de 700 estaciones costeras a lo largo de toda Europa. Los guardacostas y oficiales de las armadas de los distintos países monitorizan constantemente el tráfico marítimo en aguas europeas y analizan estos mensajes, cooperando para detectar riesgos potenciales y para adoptar los procedimientos de seguridad necesarios en caso de accidente.

El Conocimiento del Dominio Marítimo o *Maritime Domain Awareness* (MDA) consiste en el efectivo entendimiento de actividades, eventos y riesgos ocurridos en el entorno marítimo que pueden impactar en la protección global, seguridad, actividades económicas o medio ambiente. El principal reto que existe hoy en día en el ámbito de la seguridad marítima es la necesidad de identificar patrones que emergen dentro de una gran cantidad de datos. El entendimiento del complejo entorno marítimo nunca puede estar limitado a simplemente sumar o conectar las posiciones de varios barcos mientras navegan por el mar. Conseguir entender la situación y comprender los elementos y su significado contextual en un ambiente dentro de un tiempo y espacio determinado, constituye un aspecto fundamental en el MDA.

Es en este contexto donde surge el servicio de detección de anomalías de la H2020 Big Data Ocean Platform, que se explicará en los próximos párrafos; sin embargo, este proyecto es de mayores dimensiones y ofrece más recursos tecnológicos como se indica en [13].

La detección anómala consiste en un método que evalúa situaciones que se desvían del comportamiento esperado, conocido o normal y que, por lo tanto, deben tenerse en consideración para futuras investigaciones. El objetivo de este servicio es analizar las características de los barcos y de sus trayectos con el fin de encontrar anomalías y clasificarlas en estáticas o dinámicas, asegurando una posterior reacción proactiva y una minimización de los riesgos en el mar. En la Figura 2-3 se muestra un esquema del procedimiento de detección de anomalías usado por la H2020 Big Data Ocean Platform.

Las anomalías estáticas están relacionadas con cambios inesperados o desajustes en la información de la identidad de los barcos (por ejemplo en el número IMO o en el nombre del barco) mientras que las anomalías dinámicas están mayoritariamente relacionadas con comportamientos anormales durante el viaje (detenciones en el mar, desvíos de la ruta establecida, etc.).

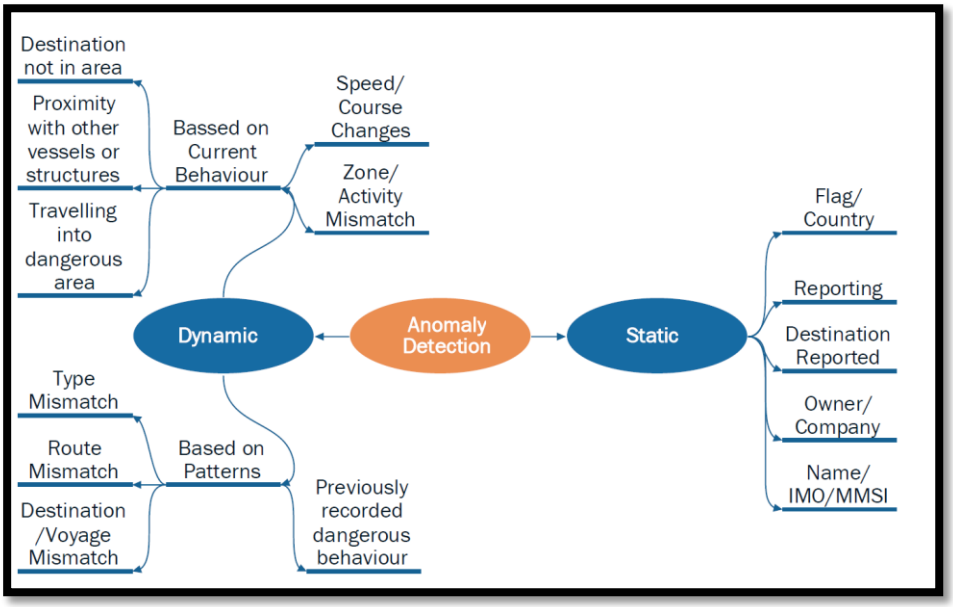

Figura 2-3 Procedimiento de detección de anomalías según el proyecto H2020 Big Data Ocean [13]

El comportamiento de un barco puede definirse como la suma de una serie de características (la posición, el rumbo, la proa, la velocidad, etc.). Por definición, un patrón está compuesto por eventos recurrentes que se repiten de una manera predecible. El comportamiento de los barcos monitorizados durante un periodo largo de tiempo da a conocer los patrones seguidos por cada embarcación en rutas específicas. Por otro lado, el análisis de estos comportamientos y patrones es usado para identificar rutas comunes de barcos realizando el mismo itinerario y gracias a ellos se pueden determinar anomalías dinámicas.

Para alcanzar su meta, el servicio de detección de anomalías de la H2020 Big Data Ocean Platform adopta un enfoque de agrupamiento distribuido, el cual está basado en el paradigma MapReduce, capaz de procesar *terabytes* de datos marítimos espaciotemporales en un conjunto de ordenadores. Este enfoque permite que sea posible extraer patrones de comportamiento similares a escala que son posteriormente usados para identificar desviaciones de ellos y que pueden asistir en la predicción de la futura localización de un barco. Los principales *datasets* usados en esta investigación fueron obtenidos del AIS y del World Port Index. La Figura 2-4 muestra un ejemplo de patrones de rutas basados en trayectos de ferris que operan en el mar Egeo.

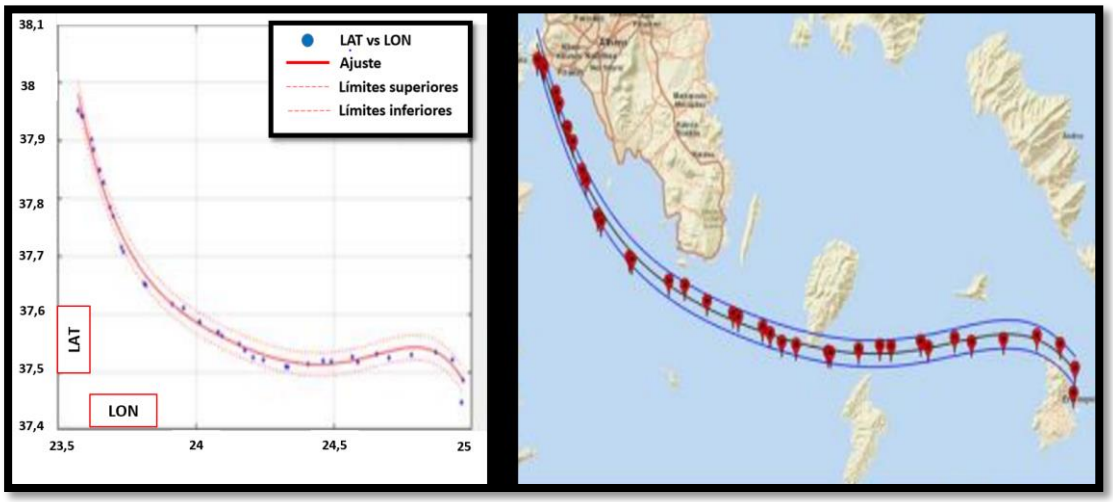

Figura 2-4 Patrón de ruta en el mar Egeo basado en trayectos de ferris durante el año 2016 [13]

2.2 Empleo del big data en la gestión medioambiental marítima

Actualmente el *big data* marítimo no solo es de gran utilidad en el análisis del tráfico marítimo. De hecho, cada vez adquiere una mayor relevancia en otros muchos ámbitos, como el medioambiental. Un ejemplo claro de esta tendencia lo encontramos en la *European Maritime Observation and Data Network* (EMODnet), cuyos productos de datos medioambientales ofertados se obtienen a partir del procesamiento de *big data*.

El objetivo de EMODnet es proveer a los países miembros de la Unión Europea de datos e información para tomar decisiones que preocupen en el ámbito de la gestión medioambiental. Existe una urgente necesidad de adquirir un conocimiento científico detallado de la biosfera del planeta y de los recursos del fondo del mar, así como del uso de las áreas marítimas y costeras. La medición continuada de aspectos físicos, químicos, geológicos y biológicos en el oceáno y en su lecho es necesaria para entender tendencias y cambios cíclicos. La mejora de las capacidades para realizar medidas sostenidas en el océano abrirá nuevas oportunidades de investigación y llevará a la mejora en la detección y en el pronóstico de los cambios medioambientales y de sus respectivos efectos sobre la biodiversidad, ecosistemas y clima. De igual manera, estos avances proveerán de herramientas para mejorar la gestión de los recursos del océano y la toma de decisiones a la hora de usar zonas costeras con fines recreativos o comerciales. [13]

EMODnet es una iniciativa de datos marítimos de larga duración de la Dirección General de Asuntos Marítimos y Pesca de la Comisión Europea (DG MARE) que involucra a más de 150 organizaciones para reunir datos marítimos, productos y metadatos. Ha sido desarrollada con un enfoque gradual y está actualmente en su fase final de desarrollo. Las organizaciones involucradas en ella trabajan juntas para observar el mar, para procesar los datos de acuerdo a estándares internacionales y para hacer que la información esté disponible. Sin embargo, esto no siempre ha sido así. Durante muchos años en Europa, la recolección, el almacenamiento y el acceso a estos datos ha sido llevada a cabo de una manera fragmentada, usándose a menudo para satisfacer las necesidades únicas de organizaciones públicas y privadas, frecuentemente aisladas entre ellas.

EMODnet provee de acceso a datos europeos de 7 tipos: batimetría, geología, hábitats del fondo marino, química, biología, física y actividades humanas. Para cada uno, ha creado una puerta de entrada a una serie de archivos de datos gestionados por organizaciones locales, regionales, nacionales e internacionales. Los usuarios tienen acceso libre a observaciones estandarizadas, a indicadores de la calidad de los datos y a los productos de los datos procesados.

EMODnet Physics [14] es una plataforma horizontal que agrega datos y metadatos desde diversos portales, ofreciendo una matriz combinada de servicios y funcionalidades tanto a clientes internos como externos. Así mismo, proporciona facilidad de visualización y descarga, información mediante *dashboards*, servicios de comunicación entre máquinas, y obtención de datos, metadatos y productos de las condiciones físicas del océano desde *datasets* diferentes y distribuidos.

La adquisición de parámetros físicos es un proceso extendido y automatizado que permite la diseminación de información casi en tiempo real. EMODnet Physics provee actualmente acceso sencillo a metadatos, datos y productos como: altura y periodo de la ola, temperatura y salinidad de la columna de agua, velocidad y dirección del viento, velocidad horizontal de la columna de agua, atenuación de la luz, cobertura de hielo y tendencia del nivel del mar. Además, recientemente empezó a trabajar con datos de vertidos en ríos, con el total de materia suspendida y con el sonido bajo el agua (polución acústica). En la Figura 2-5 se observa una imagen de lo que EMODnet Physics ofrece. [14]

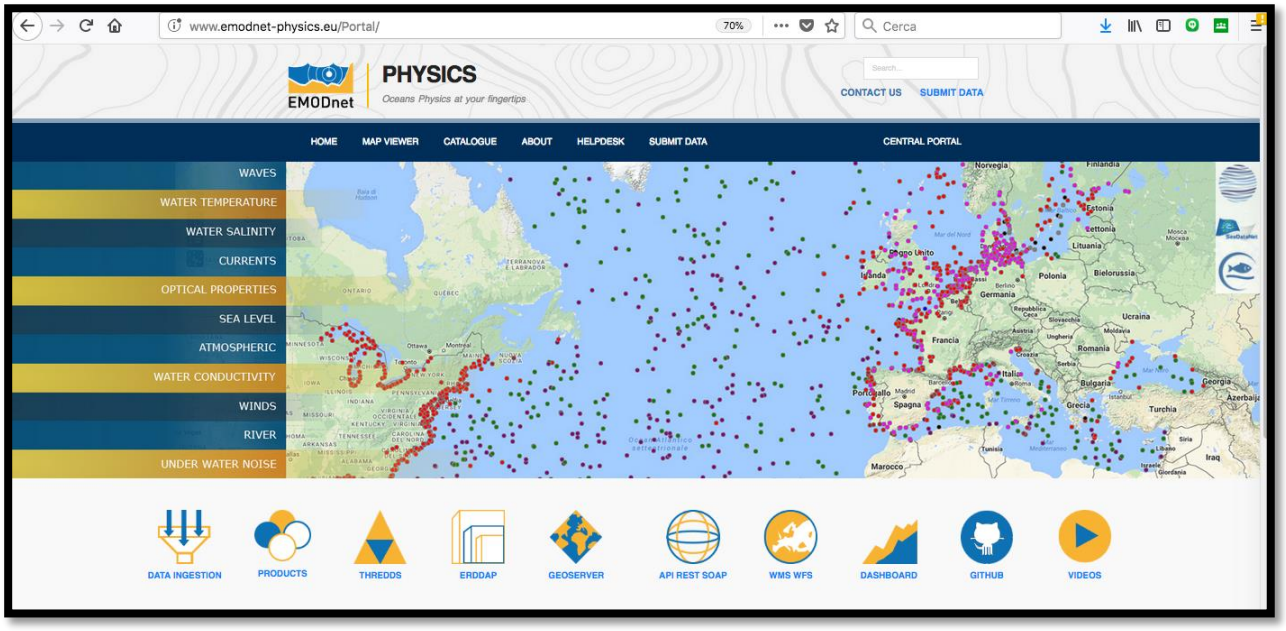

Figura 2-5 Mapa de lo que ofrece EMODnet Physics [14]

EMODnet Physics está continuamente aumentando el número y tipo de plataformas en el sistema y desbloqueando datos de alta calidad; tiene a disposición del usuario cerca de 30.000 plataformas y más de 400.000 sets de datos, así como más de 350 mapas. También proporciona servicios web que permiten la vinculación a servicios externos con flujos de datos casi en tiempo real.

2.3 Proyectos de interés a nivel europeo y nacional

Con el objetivo de mostrar los esfuerzos que se están realizando tanto a nivel europeo como nacional en el entorno del *big data* relacionado con el dominio marítimo, se explican en 2.3.1 y 2.3.2 una serie de proyectos de relevancia.

2.3.1 European Maritime Safety Agency (EMSA)

Desde el año 2007, la empresa Exprivia [15], que desarrolla sotisficada tecnología espacial, se ha volcado con éxito en el sector marítimo para fusionar datos heterogéneos de los sensores de abordo y visualizarlos en tableros de mando 3D. Desde entonces, ha desarrollado diversas soluciones operativas que hoy en día permiten a la marina militar y comercial monitorizar en tiempo real el tráfico marítimo, optimizar las rutas y gestionar los procedimientos de seguridad. [16]

En el Anexo I se encuentran disponibles las fuentes de datos que nutren los *Integrated Maritime Services* (IMS) de EMSA.

A lo largo del siguiente apartado se explica el sistema *Integrated Maritime Data Enviroment* (IMDATE), principal proyecto desarrollado por Exprivia para EMSA. [17]

2.3.1.1 **IMDatE**

El sistema *Integrated Maritime Data Enviroment* (IMDatE) es un marco técnico de trabajo de EMSA para elaborar en tiempo real correlaciones ocultas en enormes archivos de datos heterogéneos recogidos por barcos y satélites. Este proyecto, que aplica la filosofía *big data*, permite nuevas soluciones y nuevas oportunidades en el ámbito marítimo. El sistema integra, en una plataforma web 3D, todos los tipos de datos marítimos disponibles. Adicionalmente, permite la detección automática de comportamientos sospechosos mediante un motor de búsqueda configurable. [16]

IMDatE integra datos del AIS, del *Satellite-AIS* (Sat-AIS), del *Long Range Tracking and Identification System* (LRIT), del *Vessel Monitoring System* (VMS) y de radares costeros y de observación de tierra. [16]

Actualmente IMDatE es empleado por países de Europa y de fuera de Europa, como la EU Navfor (Fuerza Naval Europea), la Frontex (Agencia Europea de la Guardia de Fronteras y Costas), la AECP (Agencia Europea de Control de la Pesca), el Paris MOU (inspecciones portuarias), la Guardia Civil (control de fronteras de España) o la Guardia Costera Italiana. [16]

2.3.1.2 **Monitorización Automatizada del Comportamiento**

Automated Behaviour Monitoring (ABM) es una herramienta de análisis de posición perteneciente a los servicios marítimos integrados de EMSA que proporciona información para alertar sobre comportamientos anormales de barcos. [16]

ABM permite detectar entradas en áreas de interés, encuentros en el mar, aproximaciones a costa y desvíos de las rutas usuales (Figura 2-6 y Figura 2-7). Los operadores son automáticamente alertados en tiempo real vía e-mail o a través de la interfaz gráfica. [16]

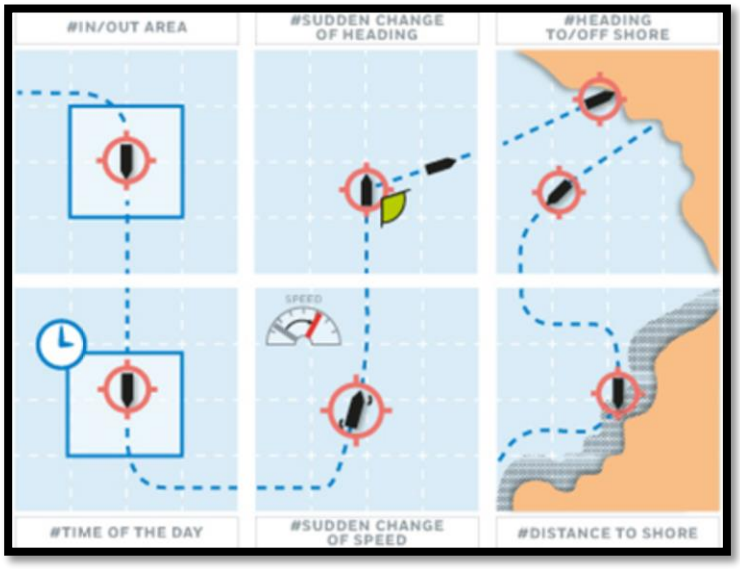

Figura 2-6 ABM detecta comportamientos sospechosos de los barcos [16]

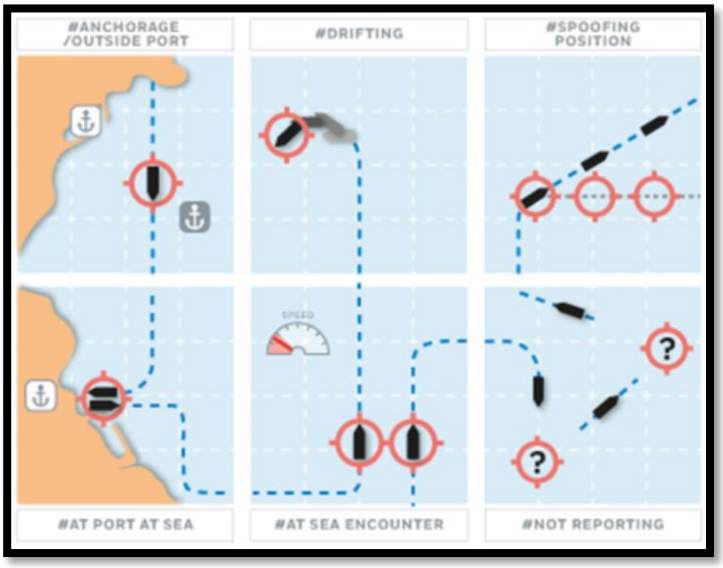

Figura 2-7 ABM detecta comportamientos sospechosos de los barcos [16]

2.3.1.3 **CleanSeaNet**

CleanSeaNet es el servicio europeo para la detección satelital del denominado *oil spill* (contaminación marina por hidrocarburos). Además de monitorizar durante 24 horas todos los mares europeos, contribuye significativamente a la identificación de las fuentes de contaminación combinando imágenes de SAR (Radares de Apertura Sintética que permiten obtener imágenes en alta definición desde el espacio) con datos de posición naval obtenidos de SafeSeaNet. Los productos que proporciona CleanSeaNet según [16] son:

- Imágenes satelitales en alta resolución.
- Detección de manchas de petróleo.
- Inspección visual de las imágenes en cuanto están disponibles.
- Datos del AIS y batimétricos, información sobre el estado del mar y posición de plataformas y naufragios.
- Corrección de la posición.
- Análisis de la imagen: posición de la mancha, forma, área, longitud.
- Determinación del nivel de fiabilidad del avistamiento.

2.3.1.4 **Servicios operacionales de EMSA**

A lo largo de esta sección se exponen las distintas capacidades que EMSA proporciona (Figura 2-8) de acuerdo a lo establecido en [16]:

- Monitorización del tráfico: los servicios están siendo desarrollados para incluir datos meteorológicos y oceanográficos, así como un número de algoritmos de aprendizaje automático que se usarán con propósitos de vigilancia y monitorización. Los algoritmos serán configurables y alertarán según las políticas del usuario.
- Búsqueda y rescate: el servicio se llama *The Enhanced* SAR SURPIC *(Search and Rescue Surface picture)* y puede ser utilizado por las autoridades marítimas durante las operaciones de rescate. El SAR SURPIC da una imagen general de los barcos presentes en cualquier región oceánica del mundo. Los barcos cercanos pueden ser contactados para ir al rescate de los buques en peligro. El SAR SURPIC combina información de posición de los barcos de todas las fuentes disponibles, incluyendo satélites, AIS y LRIT. El sistema también puede incluir información de barcos pesqueros mediante VMS.
- Monitorización de la polución: los servicios marítimos integrados de EMSA combinan información que proviene de barcos con datos de posición proporcionados por fuentes nacionales, además de datos de imágenes satelitales, así como meteorológicos y oceanográficos.
- Control marítimo de las fronteras: este servicio incluye interfaces entre sistemas para el intercambio de información en tiempo real acerca de la posición de los barcos y para la monitorización automatizada del comportamiento de los barcos. La información del buque, que proviene de sistemas satelitales, es correlacionada con la obtenida a partir de otras fuentes, como las imágenes ópticas.
- Antipiratería: las Fuerzas Navales de la Unión Europea (EU NAVFOR) llevan a cabo acciones contra la piratería en el área de la costa somalí y en el océano Índico, apoyando así a la flota mercante de la Unión Europea. Existe un servicio marítimo integrado que es utilizado por EU NAVFOR, el cual incluye la correlación e integración de un gran número de informes de barcos en un entorno marítimo.
- Monitorización pesquera: EMSA proporciona apoyo a la Agencia de Control Pesquera Europea (EFCA) en coordinación con el *Joint Deployment Plan Operations* (JDP) para actividades pesqueras en el Mediterráneo, en el Atlántico Norte y Este y en las aguas del Mar del Norte. Ofrece también una imagen del entorno operacional en tiempo real mediante el

fusionado y la correlación de datos de distintas fuentes y establece un registro común de barcos pesqueros. El servicio permite realizar un análisis de comportamientos, una evaluación de riesgos de objetivos no colaborativos y conduce la monitorización de la actividad pesquera.

- Operaciones antidroga: *The Maritime Analysis and Operations Centre- Narcotics* (MAOC (N)), con base en Lisboa, es una iniciativa de seis estados miembros de la Unión Europea: Francia, Irlanda, Italia, España, Holanda y Portugal, que cuenta con la financiación de la Comisión Europea. MAOC-N proporciona un foro de cooperación multilateral para suprimir el tráfico ilícito de drogas por mar y aire. EMSA apoya las operaciones antidroga de MAOC-N permitiéndoles el acceso a sus servicios marítimos, a través de los cuales pueden hacer uso de información integrada para monitorizar y seguir a barcos sospechosos operando en el Atlántico y el Mediterráneo.

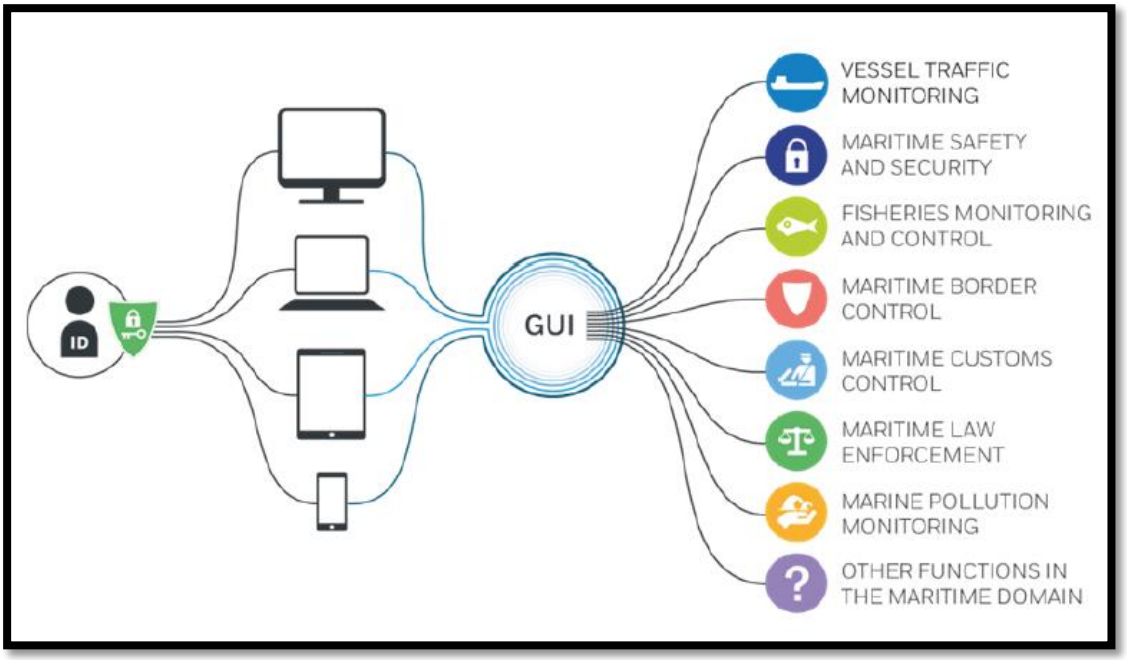

Figura 2-8 Interfaz Gráfico de Usuario que da acceso a las aplicaciones de EMSA [16]

2.3.2 Sea Traffic Management (STM)

En [18] el Ministerio de Fomento de España propone STM, una iniciativa con la que pretende gestionar de manera inteligente el tráfico marítimo.

Sea Traffic Management (STM) es un concepto inspirado en el proyecto *Single European Sky ATM Research* (SESAR) de gestión integral de tráfico aéreo. Además del impulso de las autopistas del mar, STM busca el desarrollo de nuevos paradigmas de gestión de tráfico marítimo cuyo eje común es la integración de los datos y el flujo de información. Para ello pretende conectar y actualizar el mundo marítimo en tiempo real. A partir de la información de barcos, proveedores de servicios y compañías navieras, está creando una infraestructura digital para el intercambio de información marítima.

STM es la continuación del proyecto Monalisa 2.0, que tiene como objetivo la digitalización y estandarización del transporte marítimo, basándose en el concepto de Cielo Único Europeo desarrollado para el transporte aéreo. En el Anexo II, se explican brevemente los proyectos Monalisa 2.0 y Cielo Único Europeo.

El objetivo principal de este programa es lograr un sector marítimo más seguro, más eficiente y más respetuoso con el medio ambiente. Algunos de los datos más relevantes quedan recogidos en la Figura 2-9. Así mismo, las metas que pretende alcanzar para el año 2030 son:

- Seguridad: reducción de los accidentes en un 50%.
- Eficiencia: reducción del 10% en la duración de trayectos y del 30% en los tiempos de espera para el atraque.
- Medio Ambiente: reducción del 7% en el consumo de combustible y en la emisión de gases de efecto invernadero.

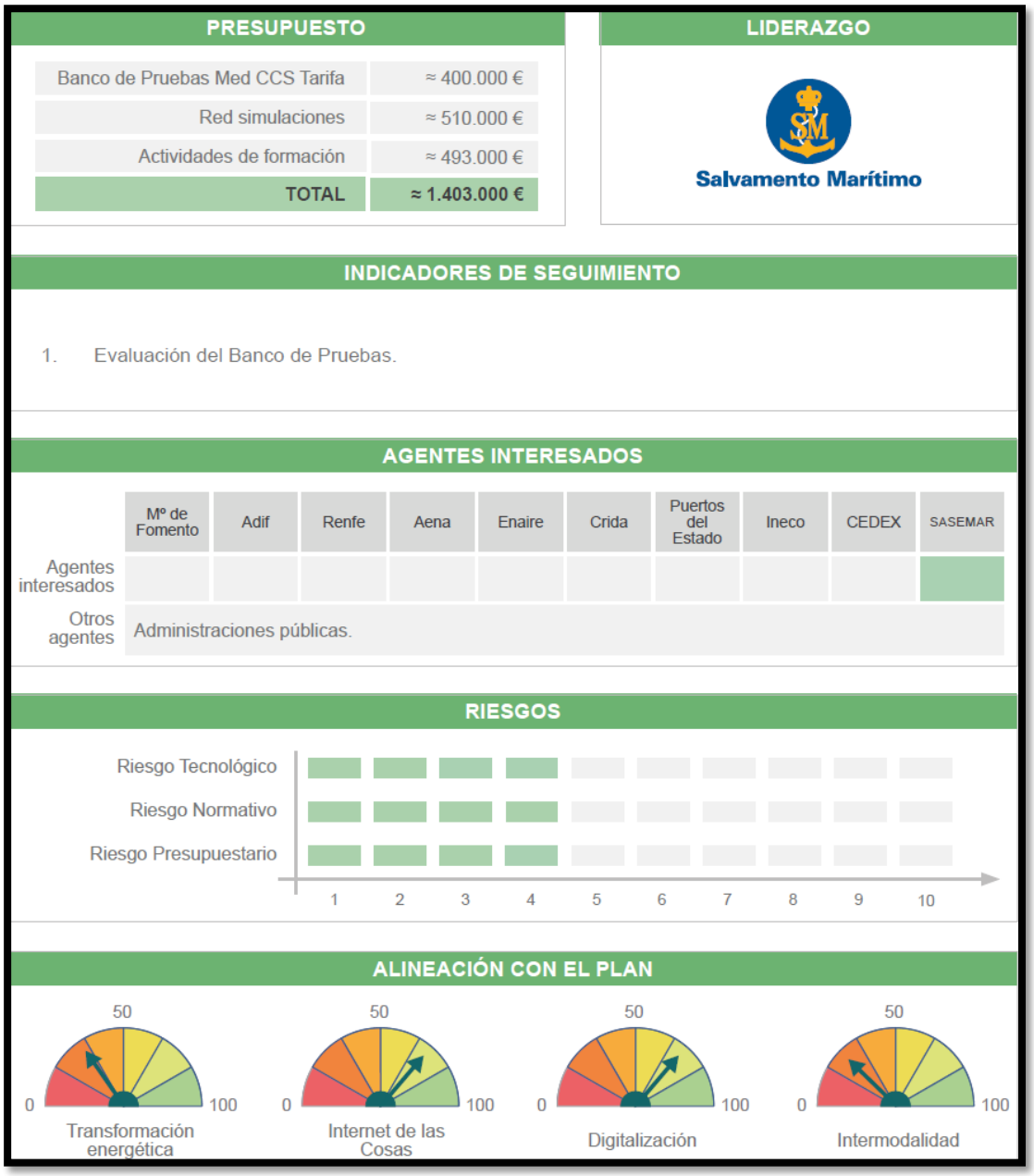

Figura 2-9 Datos del proyecto STM en España según [18]

3 DESARROLLO DEL TFG

A lo largo de este capítulo se aborda el estudio de diferentes herramientas de análisis de *big data*. Así mismo, se realiza una comparación cualitativa final de todas ellas. Tras esto, se elige la que se considera más apropiada para el análisis de datos AIS.

3.1 Análisis de las plataformas

En el año 2004, dada la creciente necesidad de gestionar la explosión de datos en Internet, nació Apache Hadoop, *framework* de código abierto que permite el procesamiento en paralelo distribuido de enormes cantidades de datos en servidores estándares y económicos, que almacenan y procesan datos, y que pueden escalarse horizontalmente. Sin embargo Hadoop, que rápidamente se convirtió en el máximo referente en el procesado por lotes, no era capaz de analizar flujos de datos para proporcionar resultados en tiempo real.

Este hecho llevó al desarrollo de una serie de herramientas que permitían analizar flujos de datos con una latencia muy inferior a la de Hadoop. De entre todas las existentes, este proyecto se centrará en seis de ellas (cinco de código libre y una de pago), las cuales han sido seleccionadas principalmente en base a su popularidad en la red.

3.1.1 Apache Spark

3.1.1.1 ¿Qué es Apache Spark?

Apache Spark [10] es un motor informático unificado y un conjunto de librerías para el procesamiento en paralelo de datos en un conjunto de ordenadores. Spark admite múltiples lenguajes de programación ampliamente utilizados (Python, Java, Scala y R) e incluye librerías para diversas tareas como consultas *Structured Query Language* (SQL), *streaming* y aprendizaje automático. Además, puede ser ejecutado en cualquier lugar, desde un simple portátil hasta una agrupación de miles de servidores. Esto lo convierte en un sistema fácil con el que iniciarse en el procesamiento de datos a gran escala. En la Figura 3-1 se puede visualizar lo que Spark ofrece al usuario final. [3]

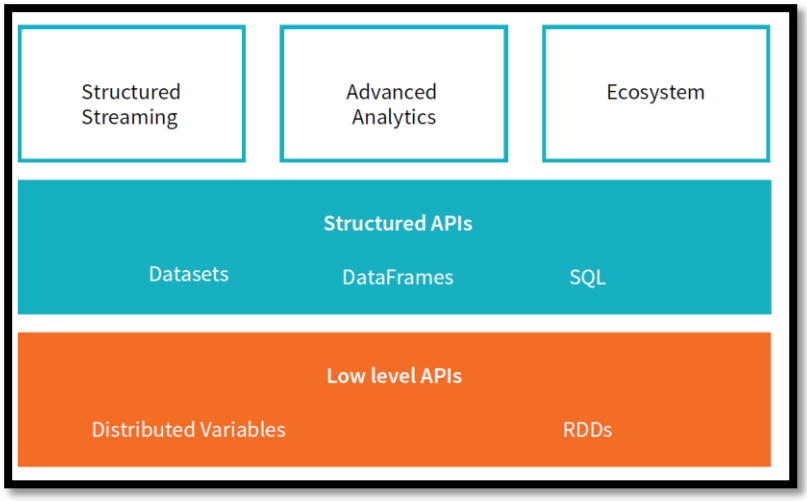

Figura 3-1 Oferta de Apache Spark al usuario final [3]

A continuación se va a proceder a analizar las tres características principales que distinguen a Apache Spark: es un motor informático, está unificado e incluye librerías. [3]

- Es un motor informático: Spark lee datos de un sistema de almacenamiento y realiza cálculos sobre ellos; no tiene como fin en sí mismo almacenar datos de manera permanente. Como los datos son costosos de mover, Spark se enfoca en realizar cálculos sobre ellos, sin importar dónde residan. Spark se puede usar con una amplia variedad de sistemas de almacenamiento persistente, incluidos sistemas de almacenamiento en la nube (Azure Storage y Amazon S3), sistemas de archivos distribuidos (HDFS), almacenes de clave-valor (Apache Cassandra) y plataformas distribuidas de transmisión de datos (Apache Kafka). Mediante interfaces de programación de aplicaciones (APIs) orientadas al usuario, Spark intenta que los sistemas de almacenamiento se vean similares para que la localización de los datos no influya negativamente en las aplicaciones.
- Está unificado: el objetivo principal de Spark es ofrecer una plataforma unificada para escribir aplicaciones de *big data*. Spark es capaz de realizar una amplia gama de tareas de análisis de datos (carga y lectura de datos, consultas SQL, aprendizaje automático, análisis de flujos, etc.) a través de un mismo motor informático y un conjunto de APIs consistentes. Las tareas de análisis de datos del mundo real tienden a combinar muchos tipos de procesamientos y librerías diferentes. La naturaleza unificada de Spark hace que estas tareas sean más fáciles de escribir. Primero, Spark proporciona APIs consistentes y componibles que se pueden usar para crear una aplicación a partir de piezas más pequeñas o de librerías existentes, y sobre ellas facilita escribir librerías propias de análisis. La combinación de APIs y la ejecución de alto rendimiento hacen de Spark una plataforma poderosa para aplicaciones interactivas.
- Incluye librerías: las librerías estándares constituyen la mayor parte del proyecto de código abierto: mientras que el motor central de Spark en sí mismo ha cambiado poco desde sus inicios, las librerías han crecido, proporcionando gran variedad de funcionalidades. Spark incluye librerías de consultas SQL (Spark SQL), de aprendizaje automático (MLlib), de procesamiento de flujo (Spark *Streaming*), de *streaming* estructurado *(Structured Streaming)* y de análisis gráfico (GraphX). Existen también cientos de librerías externas de código abierto compatibles con Spark.

Cabe destacarse que una de las mayores fortalezas de Spark es el ecosistema de paquetes y herramientas que la comunidad ha creado. Algunas de estas herramientas se utilizan ampliamente y han sido adquiridas por el proyecto principal de Spark. El mayor índice de paquetes Spark se puede encontrar en [19]; cualquier usuario puede publicar en este repositorio de paquetes. También hay otros proyectos

y paquetes que se pueden encontrar a través de la web, por ejemplo en [20]. La Figura 3-2 muestra algunas de las librerías más importantes compatibles con Spark.

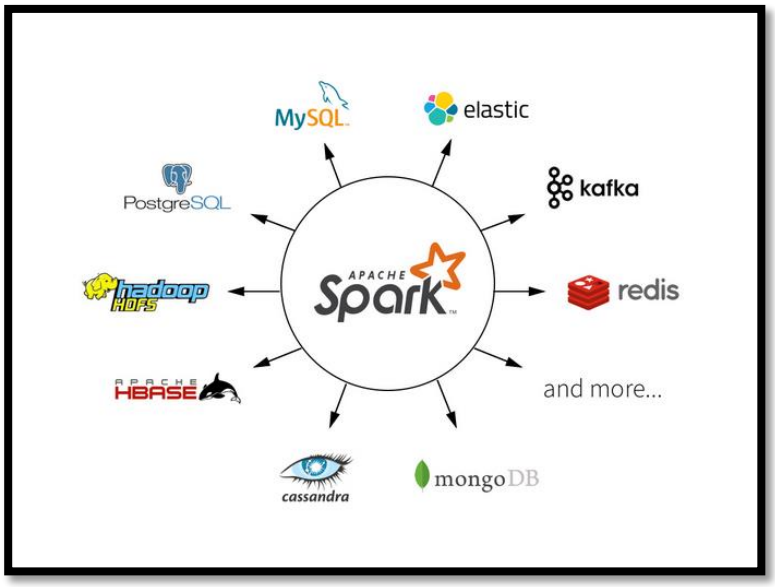

Figura 3-2 Ejemplos de librerías compatibles con Spark [10]

3.1.1.2 **Origen e historia de Apache Spark**

Apache Spark, que comenzó en 2009 como un proyecto de investigación en la Universidad de California, en Berkeley, se dio a conocer mediante el artículo de investigación [21], publicado en 2010 por miembros AMPlab, laboratorio de análisis de *big data* perteneneciente a dicha institución. En ese momento, MapReduce de Hadoop era el motor de programación paralela dominante, pues este fue el primer sistema de código abierto que abordaba el procesamiento en paralelo de datos en agrupaciones formadas por miles de nodos.

A través de la experiencia, los científicos de AMPlab llegaron a la conclusión de que la computación en grupo tenía un tremendo potencial: MapReduce permitía la creación de nuevas aplicaciones a partir de datos existentes y muchas empresas se interesaron; sin embargo, no era eficiente en la ejecución de grandes aplicaciones. Claro ejemplo de esta citada ineficiencia es el hecho de que con MapReduce el algoritmo de aprendizaje automático típico necesitaba recorrer los datos entre 10 y 20 veces. Cada recorrido tenía que escribirse e iniciarse por separado y necesitaba cargar los datos en cada ocasión.

Para abordar este problema, el equipo de Spark diseñó primero una API que podía escribir aplicaciones de múltiples pasos, y luego la implementó sobre un nuevo motor que podía realizar intercambio eficiente de datos en la memoria.

La primera versión de Spark solo admitía aplicaciones por lotes. Posteriormente, en el año 2011, AMPLab desarrolló Shark, un motor que permitía ejecutar consultas SQL sobre Spark (en [22] Shark es descrito por sus desarrolladores de manera más profunda). Después de estos lanzamientos iniciales, el proyecto siguió el enfoque de "librería estándar", pues estas serían las adiciones más poderosas a Spark. Con este objetivo en mente, se crearon APIs altamente interoperables como MLlib, Spark Streaming y GraphX.

En 2013, habiendo el proyecto alcanzado una extensión considerable, AMPlab decidió que Apache Software Foundation se convirtiera en proveedor independiente a largo plazo de Spark. De igual manera, AMPlab también lanzó una empresa denominada Databricks, cuyo fin era fortalecer el proyecto, uniendo empresas y organizaciones para que contribuyeran en el desarrollo de la herramienta.

Resulta relevante reseñar que la idea central de Spark, basada en APIs componibles, también se ha perfeccionado con el tiempo. Las primeras versiones de Spark, previas a la versión 1.0, definieron el paralelismo de las operaciones funcionales de las APIs. A partir de la versión 1.0, el proyecto agregó Spark SQL, una nueva API para trabajar con datos estructurados. Con el tiempo, Spark introdujo nuevas APIs basados en cimientos más fuertemente estructurados, incluidos *DataFrames, Structured Streaming* y APIs de transmisión de alto nivel y de optimización automática. La última versión de Apache Spark, publicada en febrero de 2020, es la versión 2.4.5.

3.1.1.3 **Funcionamiento**

Comprender el funcionamiento de Spark es esencial para poder luego decidir si utilizar esta herramienta o no en el proyecto. Como muestra [3], su funcionamiento es relativamente sencillo y fácil de entender, lo que podría convertirla en una herramienta adecuada para el análisis de flujos.

Los ordenadores individuales no tienen suficiente potencia ni recursos para realizar cálculos sobre grandes cantidades de información (o el usuario puede no tener tiempo para esperar a que terminen las operaciones).

Un conjunto de máquinas agrupa los recursos de muchas máquinas juntas, lo que permite utilizar todos los elementos como si fueran uno. Ahora bien, un grupo de máquinas por sí solo no es suficiente, necesita un marco que coordine el trabajo a través de ellos. Spark es una herramienta para administrar y coordinar la ejecución de tareas sobre datos a través de un grupo de computadoras.

El grupo de máquinas que Spark utiliza son gestionadas por un administrador de conjuntos, como YARN [23] o Mesos [24]. Este elemento es vital, pues se encarga de conceder recursos a las aplicaciones para que puedan completarse las tareas.

Las aplicaciones Spark constan de un proceso controlador y un conjunto de procesos de ejecución. El proceso controlador, que ejecuta la función principal, se encuentra en un nodo del conjunto y es responsable de tres aspectos: gestionar la información de la aplicación, responder a la entrada de un usuario y analizar, distribuir y programar el trabajo entre los ejecutores.

Los ejecutores son los responsables de realizar el trabajo que el controlador les asigna. Esto significa que cada ejecutor es responsable de solo dos tareas: ejecutar el código que le asigne el controlador e informar al controlador del estado del cómputo en ese ejecutor.

El administrador del conjunto controla las máquinas físicas y asigna recursos a las distintas aplicaciones de Spark. Cabe destacar que puede haber múltiples aplicaciones de Spark que se ejecutan en un grupo al mismo tiempo.

En la Figura 3-3 se observa, a la izquierda, el director del proceso y, a la derecha, cuatro ejecutores. En ella, se ha eliminado el concepto de nodos de la agrupación. El usuario puede especificar cuántos ejecutores debe haber en cada nodo a través de configuraciones.

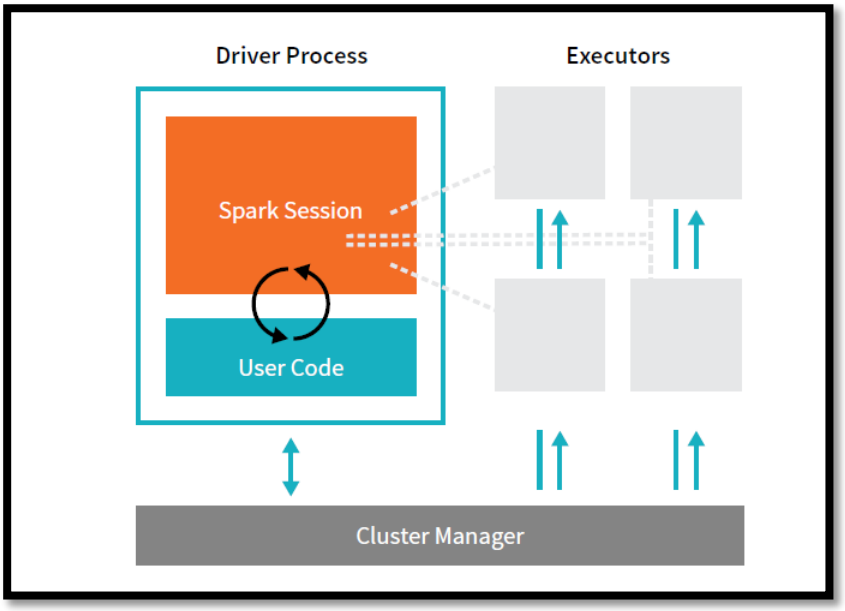

Figura 3-3 Diagrama de relaciones establecidas en la ejecución de una aplicación Spark [3]

Spark está escrito principalmente en Scala, por lo que es su lenguaje predeterminado, aunque se puede escribir el código en Java. Además, Spark es compatible con Python, mediante la API de Python; con el estándar ANSI SQL 2003; y también tiene dos librerías de R de uso común, una como parte del núcleo de Spark (SparkR) y otra como un paquete impulsado por la comunidad R (sparklyr).

Para permitir la ejecución del código a partir de otros lenguajes de programación, la API de lenguaje de Spark presenta algunos conceptos centrales que son comunes a todos ellos. Estos conceptos se traducen al código Spark y son ejecutables en un conjunto de máquinas. En la Figura 3-4 se muestra el proceso de programación con Python y R.

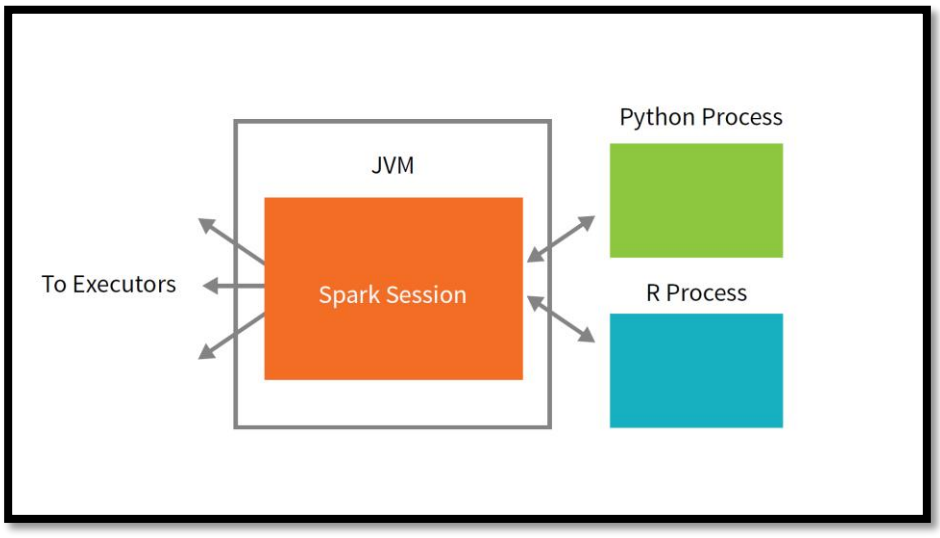

Figura 3-4 Programación con Python o R en Apache Spark [3]

Spark distingue entre dos conjuntos fundamentales de APIs: las no estructuradas o de bajo nivel y las estructuradas o de nivel superior. A continuación se enumeran las APIs de las que dispone Apache Spark:

- *SparkSession*: es el punto de entrada para ejecutar el código. La *SparkSession*, que está a disposición del usuario, le permite a este ejecutar sus peticiones en todo el grupo. Existe una correspondencia de uno a uno entre una *SparkSession* y una aplicación Spark.
- *DataFrame*: tabla de datos con filas y columnas, es la API estructurada más común. Una analogía simple sería una hoja de cálculo con el nombre de las columnas. La diferencia fundamental es que mientras una hoja de cálculo se encuentra en una computadora en una ubicación específica, el *DataFrame* puede abarcar miles de ordenadores.
  Para permitir que cada ejecutor realice trabajos en paralelo, Spark divide los datos en fragmentos, llamados particiones. Las particiones de un *DataFrame* representan la distribución física de los datos en un grupo de máquinas durante la ejecución.
  Una cosa importante a tener en cuenta es que con el *DataFrame* no se manipulan particiones manualmente. Simplemente se especifican transformaciones de datos de alto nivel en las particiones físicas y Spark determina cómo se ejecutará este trabajo en el conjunto.
  En Spark, las estructuras de datos centrales son inmutables, lo que significa que no se pueden cambiar una vez creadas. Para modificar un *DataFrame* habrá que indicarle a Spark cómo transformar el que tiene para convertirlo en el deseado. Estas instrucciones se llaman transformaciones. Si se realiza una transformación abstracta Spark no actuará sobre dicha transformación hasta que llamemos a una acción; es decir, no devolverá ningún resultado.
  Hay dos tipos de transformaciones, aquellas que especifican dependencias estrechas (Figura 3-5) y aquellas que definen dependencias amplias (Figura 3-6).
  Las transformaciones que consisten en dependencias estrechas (transformaciones estrechas) son aquellas donde cada partición de entrada contribuirá a una sola partición de salida.

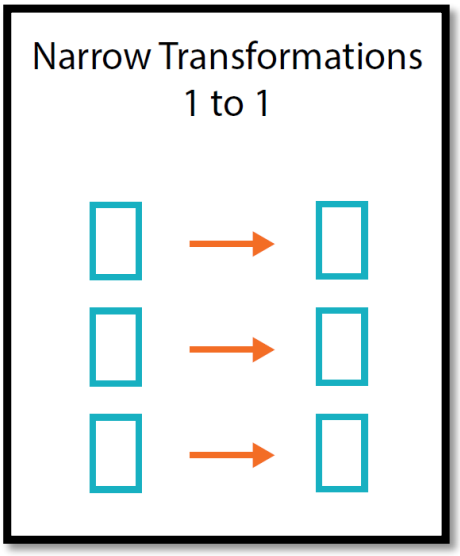

Figura 3-5 Transformaciones estrechas [3]

Las transformaciones de dependencias amplias (transformaciones amplias) tendrán particiones de entrada que contribuirán a muchas particiones de salida. El intercambio de particiones a través del racimo es conocido como *shuffle*. Con transformaciones estrechas, Spark realiza automáticamente una operación llamada canalización en dependencias estrechas, lo que significa que si se aplican filtros en *DataFrames*, todos se guardarán en la memoria. Sin embargo, si se realiza un *shuffle*, Spark escribirá los resultados en el disco.

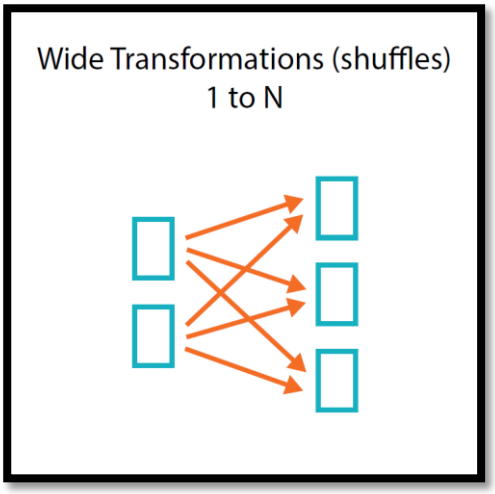

Figura 3-6 Transformaciones amplias (*shuffles*) [3]

- *DataSets:* son una versión de API estructurada en Spark para Java y Scala. Esta API no está disponible en Python ni en R porque son lenguajes de tipado dinámico. Los *DataSets* son una colección distribuida de objetos de tipo *row*, que pueden contener varios tipos de datos tabulares. La API *DataSet* permite a los usuarios asignar una clase Java a los registros dentro de un *DataFrame*, y manipularlos como una colección de objetos escritos. Los *DataSets* son de tipo estático, lo que significa que no pueden acceder accidentalmente a los objetos de otro *DataSet* si son de una clase distinta a la establecida al inicio.
- *StructuredStreaming*: API de alto nivel para el procesamiento de flujos que se implementó en la versión Spark 2.2. El streaming estructurado realiza las mismas operaciones que se harían en el modo por lotes y las ejecuta de forma continua. Esto reduce la latencia y permite un procesamiento incremental.
- API de aprendizaje automático y análisis avanzado: Spark tiene capacidad de realizar aprendizaje automático a gran escala. También posee una librería interna, denominada MLlib, que contiene algoritmos de aprendizaje automático. Estos algoritmos permiten preprocesamiento, *munging*, entrenamiento de modelos y hacer predicciones a escala sobre los datos. Spark proporciona una API sofisticada para realizar diversas tareas de aprendizaje automático, como clasificación, regresión, agrupamiento o aprendizaje profundo. En [25] se pueden conocer más detalles de MLlib.
- APIs de nivel inferior: Spark incluye una serie de primitivas de nivel inferior para permitir la manipulación arbitraria de objetos Java y Python a través de *Resilient Distributed Datasets* (RDDs). Prácticamente todo en Spark está construido sobre RDDs. Las operaciones de *DataFrame* se construyen sobre RDDs y se compilan en estas herramientas de nivel inferior para una ejecución distribuida extremadamente eficiente. Los RDDs son de más bajo nivel que los *DataFrames* porque revelan a los usuarios finales características de ejecución física (como las particiones). Un motivo de uso de RDDs sería, por ejemplo, la paralelización de los datos en bruto almacenados en la memoria de la máquina de control. En [26] se explican más detalladamente.
- SparkR: es una herramienta para ejecutar R en Spark. Para usar SparkR, simplemente se importa al entorno y se ejecuta el código.

El concepto *DataFrame* no es exclusivo de Spark; R y Python tienen conceptos similares. Sin embargo, los *DataFrames* de Python y R existen en una sola máquina en lugar de en varias, lo que limita sus capacidades. Por ello, Spark dispone de interfaces de lenguaje para Python y R que convierten fácilmente Pandas (Python) *DataFrames* y R *DataFrames* en Spark *DataFrames.*

Por otro lado, Spark lleva a cabo lo que sus desarrolladores han acuñado como “evaluación perezosa”. Este comportamiento implica que Spark espera hasta el último momento para ejecutar la computación según las instrucciones dadas; es decir, en lugar de modificar los datos inmediatamente cuando se expresa alguna operación, acumula el plan de transformaciones que se aplicará a los datos de origen. Esto proporciona enormes beneficios para el usuario final porque así optimiza el flujo de datos entre extremos.

Spark posee una herramienta llamada *spark-submit*, incluida en el núcleo de Spark, que permite enviar aplicaciones a un administrador de conjuntos para su ejecución. Las aplicaciones de producción se pueden escribir en cualquier lenguaje admitido por Spark.

3.1.1.4 **Escenarios de uso**

Spark continúa ganando popularidad masiva. Muchos de los nuevos proyectos dentro de su ecosistema continúan ampliando los límites de lo que es posible con el sistema. Por ejemplo, *Structured Streaming* fue diseñado e introducido en Spark en 2017. Esta tecnología es usada por una gran cantidad de compañías que resuelven desafíos de datos a gran escala, desde compañías de tecnología como Uber y Netflix, que aprovechan las herramientas de aprendizaje automático de Spark, hasta instituciones como el *Broad Institute* de la *Massachussets International Technology* (MIT) y Harvard, que emplean Spark para el análisis de datos genéticos. En un futuro próximo, Spark continuará siendo una piedra angular para las empresas que realizan análisis de *big data*, principalmente porque todavía está en su infancia.

Con el objetivo de medir la popularidad actual de la herramienta se han utilizado dos métodos: el primero de ellos ha consistido en la búsqueda entrecomillada de la herramienta en Google; mientras que el segundo se ha basado en el número de seguidores que Spark tiene en su cuenta de Twitter: @ApacheSpark. Se ha obtenido lo siguiente:

- Resultados en Google de “Apache Spark”: 4.790.000
- Seguidores en Twitter de Apache Spark: 34.500
- Fecha de búsqueda: 06 de marzo de 2020

En el Anexo I, se muestran algunas de las más grandes empresas que hacen uso de Spark.

La mejor manera de entender la popularidad de Spark es mediante un ejemplo [27] que muestre el uso real que se le da. Para ello, se explica como Celtra [28], compañía para la cual el análisis de datos a gran escala resulta ser una tarea fundamental, utiliza Spark y Databricks [29].

Databricks es una plataforma en la nube especializada en ingeniería de datos a gran escala y ciencia de datos colaborativa para trabajar con Apache Spark. Fue fundada por el mismo creador de Apache Spark y desarrollada entorno al proyecto AMPLab de la Universidad de Berkeley.

Celtra es una empresa estadounidense que ofrece a agencias, proveedores de medios y líderes de marcas tecnológicas, publicidad de marcas en teléfonos inteligentes, tabletas y ordenadores.

La plataforma, AdCreator 4, cuyo logo puede visualizarse en la Figura 3-7, ofrece a clientes como MEC, Kargo, Pepsi y Macy`s la habilidad de crear, administrar y lanzar anuncios dinámicos basados en datos, realizando un seguimiento de su rendimiento mediante análisis.

Figura 3-7 Logo de la plataforma AdCreator de Celtra [27]

Celtra recopila una amplia variedad de datos, incluidos los relacionados con los procesos internos de la empresa, datos basados en el uso del producto por parte de los clientes y, lo más importante, datos que miden el compromiso de los consumidores con los anuncios de sus clientes. Además de proporcionar análisis a sus clientes, Celtra está constantemente explorando nuevas formas de aprovechar esta recopilación de información para mejorar su oferta, por ejemplo:

- Analiza el uso del producto: la adopción de características, patrones de uso y casos de apoyo para dirigirse hacia un mayor enfoque de desarrollo.
- Analiza el entorno: evalúa la viabilidad de nuevos conceptos de productos y detecta tendencias analizando el contexto en el que se ejecutan los anuncios de Celtra, como el editor y el dispositivo de elección.
- Mejora el rendimiento técnico: monitoriza los tiempos de carga de los anuncios en múltiples dimensiones, es decir, en su complejidad, geografía, conectividad y red de distribución de contenidos (CDN). Más recientemente, Celtra ha estado evaluando los beneficios de rendimiento de SPDY y HTTP / 2 para tiempos de carga de página mejorados.
- Controla la calidad: calcula métricas clave de rendimiento para detectar problemas en implementaciones, lo que le permite la detección automática de regresiones que de otro modo se perderían en los promedios.

A medida que el negocio de Celtra creció, fue desafiado a satisfacer el aumento de datos objeto de análisis debido a tres razones:

- Diversidad de fuentes de datos: los datos de producción e ingeniería de los sistemas de Celtra están dispersos en diferentes lugares. Celtra no tenía una manera fácil de combinar los datos de estas fuentes de datos dispares y realizar el análisis necesario en una única plataforma.
- Gran escala de datos: los sistemas de producción de Celtra generan decenas de *terabytes* de datos mensuales. En la Figura 3-8 se muestra la cantidad de información con la que opera Celtra cada día.
- Pequeño equipo de análisis: el equipo de análisis constaba de solo cuatro personas, y rápidamente se convirtió en un cuello de botella para atender las solicitudes de gestión de productos, ingeniería y control de calidad.

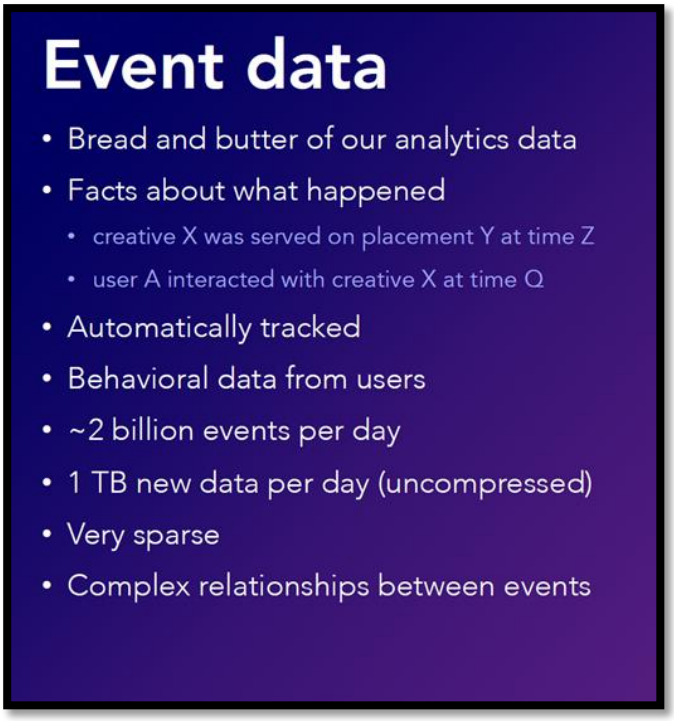

Figura 3-8 Celtra analiza dos mil millones de eventos (1 TB de nuevos datos no comprimidos) por día [30]

Para superar estos desafíos, Celtra necesitaba una poderosa plataforma de datos que fuera capaz de integrar información de fuentes de datos dispares mientras era lo suficientemente rápida como para admitir análisis interactivos a escala de *terabyte*. Esta plataforma también debía ser lo suficientemente fácil de usar como para potenciar que equipos no expertos fueran capaces de realizar análisis por su propia cuenta, eliminando así el cuello de botella creado. La Figura 3-9 Celtra muestra resumidamente las razones por las que decidió trabajar con Spark.

Celtra adoptó Databricks como su plataforma centralizada de análisis porque con ella podía abordar fácilmente todas sus necesidades:

• Gestión cero Apache Spark: Spark es un *framework* de *big data* de código abierto que fue construido para proporcionar velocidad y escalado. Databricks hizo que Spark fuera mucho más fácil de implementar al combinar el poder de Spark con una plataforma de administración cero alojada en *Amazon Web Services* (AWS), que permite a Celtra tomar ventaja de Spark sin las cargas de desarrollo y operación típicamente asociadas a grandes infraestructuras de datos.

• Conexión perfecta a diversas fuentes de datos: Databricks proporciona APIs integradas para acceder a datos de *AWS S3* y a bases de datos relacionales. Como Scala está disponible en Databricks, se puede acceder también a los datos de varias APIs.

• Reutilización del código de producción en análisis *ad-hoc*: ya que Databricks se basa en Apache Spark, que es similar a la línea de análisis de producción de Celtra, gran parte del código de producción podría reutilizarse como la base para análisis *ad-hoc* sin necesidad de reescribir el código en otro marco.

• Espacio de trabajo interactivo y fácil de usar: Databricks incluye un espacio de trabajo interactivo intuitivo y multiusuario para análisis y visualización en tiempo real, lo que le permite a equipos no analíticos trabajar directamente con datos en un ambiente único y fácil de usar.

Con la adopción de Databricks, Celtra ha capacitado a equipos de ingeniería, gestión de productos y control de calidad para realizar análisis de datos complejos por su cuenta; de igual manera, ha aprovechado la producción masiva de datos para mejorar el diseño del producto, abordar las anomalías rápidamente y ajustar el rendimiento de los sistemas de producción.

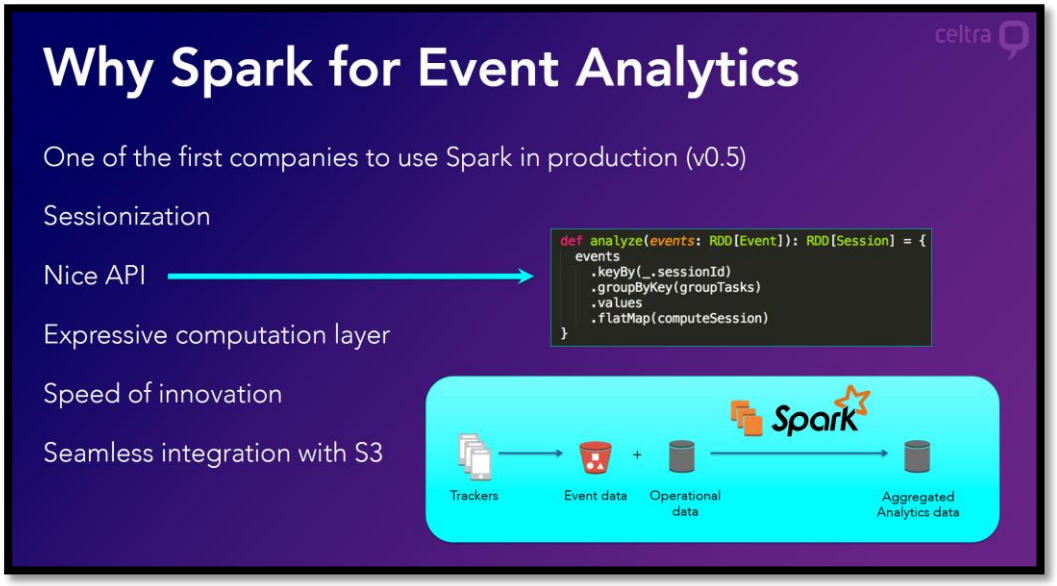

Figura 3-9 Celtra usa Spark en el análisis de datos [30]

Desde su introducción, Databricks ha sido ampliamente utilizado por más de un tercio del personal técnico en ingeniería, control de calidad y gestión de productos. Como resultado de que los empleados trabajen con los datos directamente, han surgido muchas más preguntas y se han probado hipótesis, lo que ha llevado a un diseño de producto con mejor información y más rápido, y a la detección y resolución de problemas. Celtra ha multiplicado por seis la cantidad de análisis realizados y la información obtenida en los primeros cuatro meses después de adoptar Databricks. Así mismo, ha multiplicado por cuatro el número de personas que trabajan con sus datos más valiosos.

Además de incrementar drásticamente la cantidad de análisis realizados, Celtra también ha experimentado dos beneficios adicionales derivados del uso de Databricks:

• Mejor colaboración y reproducibilidad: la naturaleza autodocumentada de los cuadernos en Databricks permite que el código de análisis *ad-hoc* sea almacenado automáticamente en una ubicación centralizada. Esta característica alienta a los equipos a aprovechar la base del código existente en lugar de duplicar esfuerzos escribiendo un nuevo código. Además, al tener todo el trabajo almacenado por defecto, los resultados previos pueden reproducirse fácilmente en casos donde se necesita una visión adicional.

• Coste de infraestructura en la nube reducido: el rápido y fácil aprovisionamiento, cambio de tamaño y desaprovisionamiento de los conjuntos de ordenadores de Spark hizo que los ingenieros de Celtra estuvieran más cómodos cerrando los grupos no utilizados. La agilidad en su gestión también facilitó el uso de instancias puntuales al hacerlo menos arriesgado. Cuando se combinó con la función de "trabajos" de Databricks, Celtra fue capaz de reducir el coste de su infraestructura en la nube al programar tareas de larga duración que aprovisionaban y desaprovisionaban agrupaciones automáticamente según fuera necesario.

3.1.2 Apache Storm

3.1.2.1 **¿Qué es Apache Storm?**

Apache Storm [31], una de las herramientas más potentes en el procesado de flujos de datos, fue creada e introducida en forma de *software* libre por el programador Nathan Marz en el año 2011. Con su desarrollo surgió una nueva categoría de *software* libre que buscaba conseguir un procesamiento escalable y fiable de flujos de datos ilimitados. Su logotipo se puede visualizar en la Figura 3-10.

Figura 3-10 Logotipo de Apache Storm [32]

3.1.2.2 Historia de Apache Storm

Storm se originó cuando su creador, Nathan Marz, trabajaba para una compañía denominada *BackType*, encargada de crear productos analíticos para ayudar a las empresas a entender su impacto histórico y en tiempo real en las redes sociales. Antes de Storm, se usaba el paradigma de colas y trabajadores para el análisis en tiempo real. Sin embargo, este enfoque era frágil y engorroso.

En diciembre de 2010, Marz tuvo su primera gran idea, que consistía en la "transmisión" como una abstracción distribuida. Las corrientes serían producidas y procesadas en paralelo. Eso le llevó a la idea de los *spouts* y de los *bolts*: un *spout* produciría transmisiones nuevas y un *bolt* tomaría *streams* como entrada y produciría flujos como salida. La idea clave era que los *spouts* y los *bolts* fueran inherentemente paralelos, similares a los mapeadores y reductores de Hadoop. Finalmente, la abstracción de nivel superior que se le ocurrió fue la topología, una red de *spouts* y de *bolts*.

Para comprobar el funcionamiento de esas abstracciones, las implementó en los casos que *BackType* le ofrecía y comprobó que todas encajaban muy bien, ya que automatizaban el pesado trabajo de enviar y recibir mensajes, serializar, etc.

Poco después Nathan se embarcó en el diseño de Storm. Su idea inicial pasaba por imitar el paradigma de las colas y trabajadores usado anteriormente y combinarlo con un gestor de colas, como RabbitMQ [33], para pasar mensajes intermedios. Sin embargo, el uso de intermediarios para la transmisión de mensajes no le terminó de convencer por varias razones:

1. Eran una pieza enorme y compleja cuya arquitectura tendría que ser escalada junto a Storm.

2. Creaban situaciones incómodas a la hora de implementar una nueva topología.

3. Hacían más difícil la tolerancia a fallos. Habría que poner solución tanto al fallo de los trabajadores de Storm, como al de los intermediarios.

4. Eran lentos. En lugar de enviar mensajes directamente entre *spouts* y *bolts*, los mensajes debían pasar por un tercero.

El principal problema era encontrar la manera de garantizar el procesamiento de mensajes sin hacer uso de intermediarios. Para ello, Marz desarrolló un algoritmo basado en números aleatorios que solo requería unos 20 *bytes* para rastrear cada lista de valores del *spout*, independientemente de los flujos que se estuvieran procesando en el resto del conjunto. Este algoritmo simplificó masivamente el diseño del sistema, evitando todos los problemas mencionados anteriormente.

Durante los siguientes 5 meses, desarrolló la primera versión de Storm. Con la idea de hacer un proyecto de código libre, Marz tomó algunas decisiones fundamentales al principio. En primer lugar, diseñó todas las APIs de Storm en Java, pero implementó Storm en Clojure. Con Java, Storm se aseguró de tener una gran cantidad de usuarios potenciales; con Clojure, podía ser mucho más productivo y hacer que el proyecto funcionara antes. También planeó desde el principio hacer Storm compatible con lenguajes no JVM. Las topologías se definirían como estructuras de datos *Thrift* [34] (un protocolo de comunicación binario utilizado para definir y crear servicios para numerosos lenguajes) y se enviarían utilizando la API *Thrift*. Además, diseñó un protocolo para que las *spouts* y los *bolts* pudieran

implementarse en cualquier lenguaje, haciendo de esta manera a Storm accesible por un mayor número de personas.

Nathan, que fue usuario de Hadoop durante mucho tiempo, usó su conocimiento para hacer que el modelo de Storm superara al del Hadoop. Por ejemplo, uno de los problemas agravantes con los que trató en Hadoop fueron los denominados "procesos zombies" que ocurrían cuando ciertos trabajadores no se cerraban y los procesos continuaban ejecutándose indefinidamente. El problema principal era que los propios trabajadores eran los encargados de pararse a sí mismos, y había ocasiones en las que fallaban. Para mitigar esto, Marz decidió que esta tarea la realizase el mismo demonio que ponía en marcha a los trabajadores. Esta determinación trajo robustez e hizo que Storm nunca tuviera problemas con los "procesos zombies".

En mayo de 2011, *BackType* entró en conversaciones de adquisición con Twitter y Nathan Marz anunció Storm al mundo en un blog de tecnología. En julio de 2011 *BlackType* se unió oficialmente a Twitter, y Nathan comenzó a plantearse lanzar Storm. El 19 de septiembre de 2011 lo convirtió definitivamente en un proyecto de código abierto. [35]

Para comprender la razón que llevo a Nathan Marz a crear Apache Storm es necesario conocer cuáles eran las herramientas y paradigmas previos a la invención de Storm. Antes de Storm se utilizaba el paradigma de colas y trabajadores. Sin embargo, este modelo no era eficaz y presentaba varias limitaciones entre las que podían destacarse tres: la dificultad de escalado, la poca tolerancia a fallos y la complejidad del codificado.

Para entender mejor lo tedioso que podía llegar a ser el uso del paradigma de colas y trabajadores, Nathan Marz propone en [36] el caso que a continuación se presenta:

Supóngase que se utiliza el modelo de colas y trabajadores para crear una aplicación de análisis de tuits provenientes de Twitter. En primer lugar, los tuits llegarían a través de una "manguera contraincendios" o *firehose* de Twitter que llevaría los datos a una serie de canalizaciones. Después, estos pasarían a un primer grupo de trabajadores que esquematizaría los tuits y los adjuntaría a Hadoop (Figura 3-11).

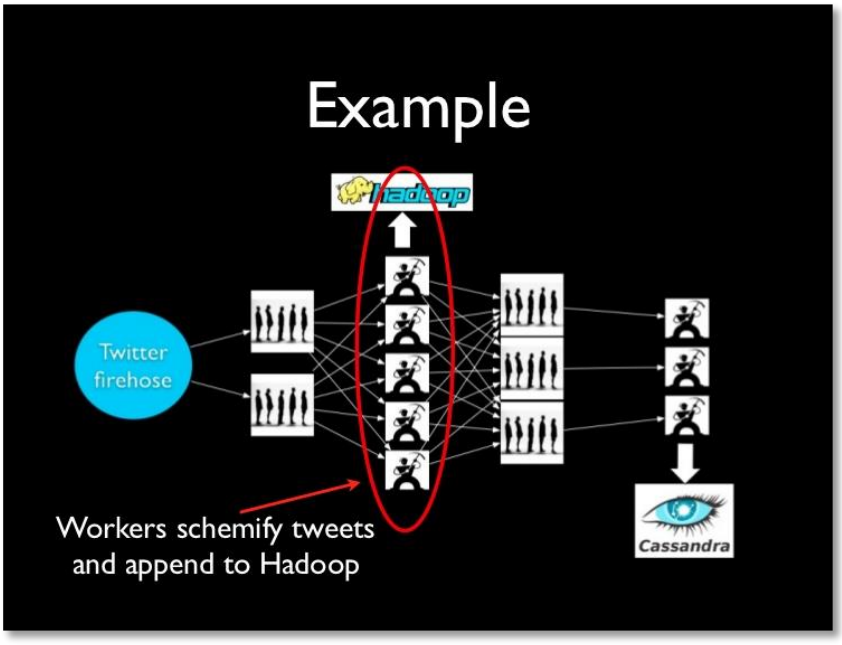

Figura 3-11 Ejemplo del funcionamiento del paradigma de las colas y los trabajadores [36]

Posteriormente, intervendría un segundo grupo de trabajadores que miraría todas las URL tuiteadas y actualizaría las estadísticas incrementando los contadores en Cassandra (Figura 3-12). En este último

proceso existiría la restricción de que una misma URL siempre tendría que ser procesada por el mismo trabajador. La razón que motivaba esta restricción es que Cassandra era no transaccional. Por aquellos tiempos no tenía contadores atómicos, por lo que tener múltiples trabajadores actualizando la misma URL influía en las condiciones y suponía la pérdida de datos por el pisoteo mutuo entre trabajadores.

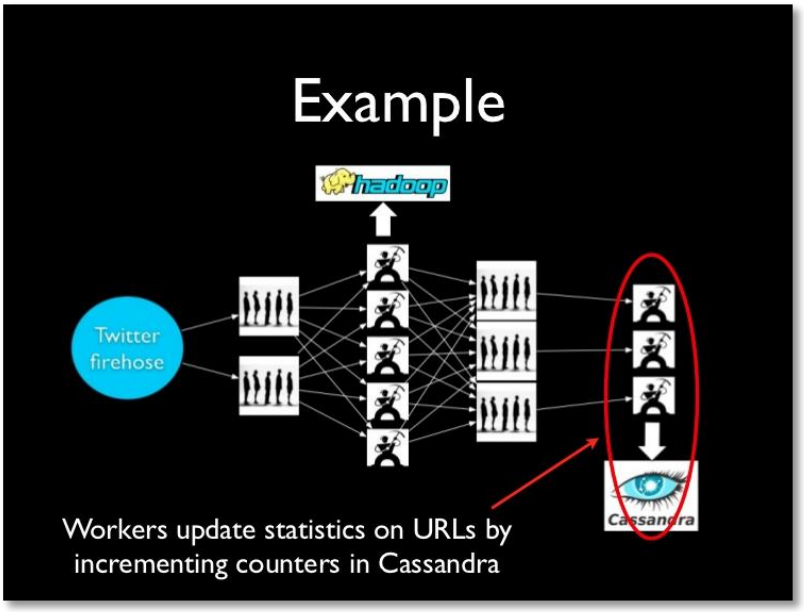

Figura 3-12 Ejemplo del funcionamiento del paradigma de colas y trabajadores [36]

Dentro de este modelo, si se quisiera escalar la aplicación, habría que aumentar el número de trabajadores que actualizan Cassandra. Para añadir un trabajador a ese grupo habría que configurar tanto el trabajador como la nueva canalización de ese trabajador, pero a su vez esto llevaría a reconfigurar el primer set de trabajadores. Por lo tanto, el requerimiento de una nueva pieza en el sistema llevaría a cambiar el resto del sistema. Aparte del ya citado problema de escalado y de la poca tolerancia a fallos, existiría un problema mayor, el codificado, con el que se perdería gran cantidad de tiempo pensando en cómo enviar y serializar los mensajes.

Con objeto de hallar una solución a estos tres problemas y tras analizar otros marcos que trabajaban en tiempo real, Marz llegó a la conclusión de que debían cumplirse seis aspectos para conseguir que un sistema de procesado en tiempo real fuera de calidad:

- Garantía en el procesado de datos.
- Escalabilidad horizontal.
- Tolerancia a fallos.
- Eliminación de intermediarios en los mensajes.
- Mayor nivel de abstracción en la transmisión de mensajes (partición, esterilización, etc.).
- Robustez extrema.

Storm, por tanto, surgió para poner solución a estos problemas.

La última versión de Apache Storm, publicada en octubre de 2019, es la versión 2.1.0, que incluye arreglos en el código que han llevado a mejorar su rendimiento, estabilidad y tolerancia a fallos. [31]

3.1.2.3 **Funcionamiento**

La arquitectura de Storm puede ser comparada con una red de carreteras que conecta una serie de puestos de control. El tráfico comienza en un determinado puesto de control (llamado *spout*) y pasa a través de otros puestos de control (denominados *bolts*). El tráfico o flujo de datos es recogido por el *spout* de una fuente de datos y es enrutado a varios *bolts* donde los datos son filtrados, saneados,

agregados, analizados, etc. La red de *spouts* y *bolts* se denomina topología, y los datos fluyen en forma de tuplas. [37]

A continuación se va a proceder a dar una explicación detallada de los componentes de las topologías de Storm [38]:

- Tupla: lista de valores de diferentes tipos.
- Flujo o *stream*: secuencia ilimitada de tuplas. Los flujos constituyen el núcleo de abstracción de Storm. Cuando se hace uso de Storm se manipula este infinito flujo de datos.
- Grifo o *spout*: fuente de flujos en una topología. En general, los *spouts* leen las tuplas de una fuente externa (por ejemplo de una cola de mensajes o de la API de Twitter) y las emiten por la topología. Los *spouts* pueden ser confiables o no confiables. Un *spout* confiable es capaz de reproducir una tupla si Storm no puede procesarla, mientras que un *spout* no confiable se olvida de la tupla tan pronto como se emite. Cabe destacarse que los *spouts* pueden emitir más de un flujo.
- Rayos o *bolts*: realizan cualquier tipo de procesamiento en las topologías (filtrados, funciones, agregaciones, uniones, diálogos con bases de datos, etc.). Los *bolts* hacen transformaciones de flujo simples. Realizar transformaciones de flujo complejas a menudo requiere múltiples pasos y, por lo tanto, múltiples *bolts*.
- Topología: red de *spouts* y *bolts*. Es el nivel máximo de abstracción que se corresponde con la Figura 3-13. Destacar que su topología es del tipo Direct Acyclic Graph (DAG) y, por lo tanto, no puede tener bucles.

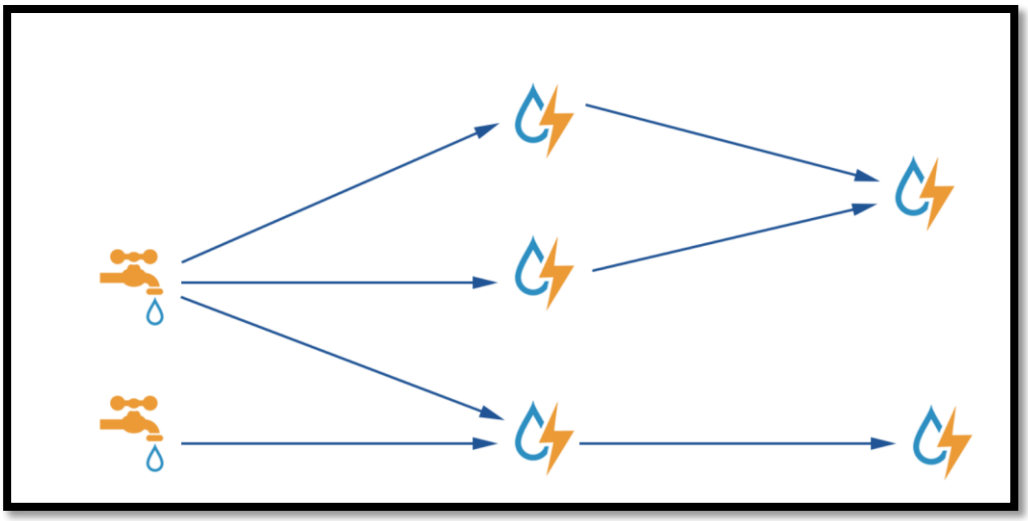

Figura 3-13 La topología de Apache Storm está formada por *spouts* y *bolts* [31]

Para entender la arquitectura de Storm es de vital importancia conocer el significado de los siguientes conceptos, que están estrechamente relacionados con los elementos que se explican a continuación:

- Proceso de trabajo: se encarga de ejecutar tareas relacionadas con una topología específica. Un proceso de trabajo no ejecuta una tarea por sí mismo, sino que crea ejecutores y les pide que realicen una tarea en particular. Por lo tanto, dispone de múltiples ejecutores.
- Ejecutor: es un hilo generado por un proceso de trabajo. Un ejecutor ejecuta una o más tareas, pero únicamente para un *spout* o *bolt* específico.
- Tarea: consiste en el procesamiento de datos.

Un grupo de máquinas de Storm presenta dos tipos de nodos (el Nimbus y el Supervisor) y un servicio denominado ZooKeeper.

- Nimbus: es un proceso que se ejecuta en el nodo que actúa como maestro del conjunto de Storm. Todos los demás nodos del grupo se denominan trabajadores (*workers*). El nodo maestro es responsable de distribuir la aplicación entre todos los nodos de trabajo.También les asigna tareas y monitoriza su estado, reiniciándolos si hubiera algún fallo.
- Supervisor: es un proceso que se ejecuta en todos y cada uno de los nodos que actúan como trabajadores y sigue las instrucciones dadas por el Nimbus. Un Supervisor tiene múltiples procesos de trabajo y los dirige para poder completar las tareas asignadas por el Nimbus.
- Apache ZooKeeper es un servicio utilizado por un conjunto de ordenadores para coordinarse entre sí y mantener datos compartidos con técnicas de sincronización robustas. ZooKeeper ayuda a los Supervisores a interactuar con el Nimbus y es el responsable de mantener el estado del Nimbus y de los Supervisores.

En la Figura 3-14 se muestra un ejemplo de arquitectura básica en Apache Storm.

Resulta interesante saber que Storm carece de estado por naturaleza. Aunque no tener estado tiene sus propias desventajas, ayuda a Storm a procesar datos en tiempo real de la mejor manera posible. Sin embargo, se puede afirmar que Storm almacena su estado en Apache ZooKeeper. Como el estado está disponible en Apache ZooKeeper, un Nimbus que falle puede reiniciarse, comenzando a funcionar exactamente desde donde dejó de funcionar. [38]

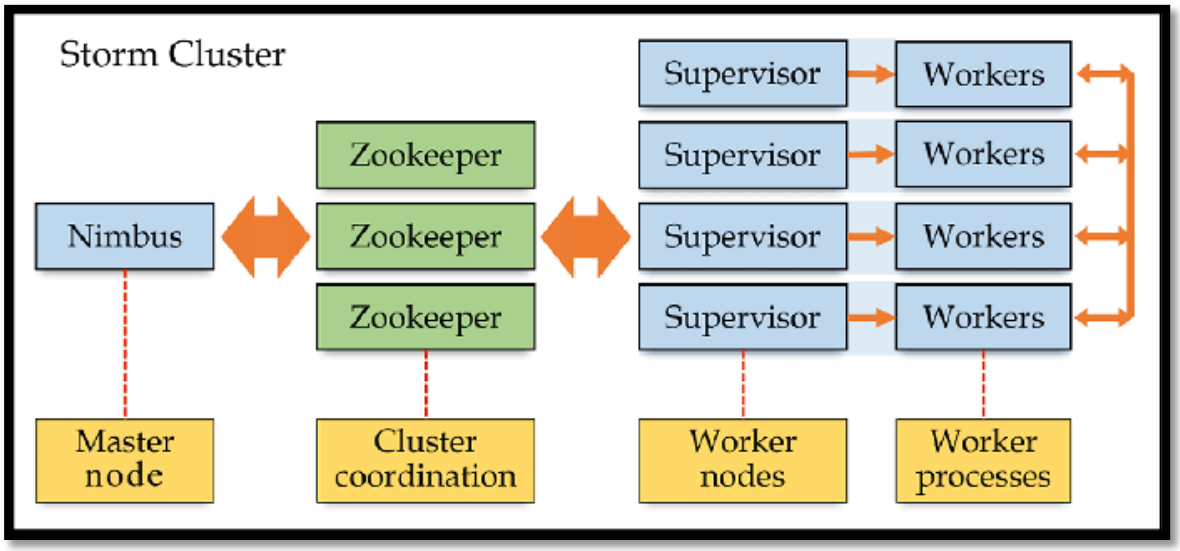

Figura 3-14 Arquitectura de Apache Storm [39]

Las características principales de Storm, según lo dispuesto en [31], son:

- Fácil de instalar y operar: Apache Storm requiere de una configuración mínima para su puesta en marcha y ejecución. Así mismo, es fácil de operar una vez instalado. Su robustez garantiza su ejecución continua durante meses.
- Escalable: las topologías de Apache Storm, formadas por una agrupación de ordenadores, son inherentemente paralelas. Este paralelismo le permite poder procesar una muy alta cantidad de mensajes con una latencia muy baja.
- Gratuito y de *software* libre: Apache Storm es un proyecto gratuito y de *software* libre autorizado por *The Apache License, Versión 2.0*. Cabe destacar que posee un gran y creciente ecosistema de librerías, así como herramientas que le permiten:
    - Integrar sistemas de colas (JMS, Kafka, Redis pub/sub, etc.). Storm realiza esta tarea mediante los *spouts*.
    - Gestionar gran cantidad de estado de memoria. Storm lleva a cabo esta tarea con el denominado *storm-state*.

    - Integrar bases de datos como MongoDB o Cassandra con Storm. Existen *bolts* que se encargan de asegurar esta compatibilidad.
- Garantiza el procesado de datos: Apache Storm asegura que cada tupla sea procesada por completo. Para asegurar esto, resulta ser extremadamente eficiente en el seguimiento del procesado de las tuplas a medida que avanzan por la topología. Las abstracciones básicas de Apache Storm proporcionan garantías de procesamiento de al menos una vez. Los mensajes son solo repetidos cuando hay fallos. Usando Trident, una abstracción de nivel superior sobre las abstracciones básicas de Apache Storm, se pueden lograr semánticas de procesamiento de exactamente una vez.
- Compatibilidad: Apache Storm es compatible con cualquier sistema de colas y con cualquier sistema de base de datos. Los *spouts* facilitan la integración de nuevos sistema de colas. Algunos de los sistemas de colas compatibles con Storm son los que se citan: Kestrel [40], Rabbit MQ/AMQP [33], Kafka [41] y Amazon Kinesis [42]. Así mismo, resulta fácil integrar Apache Storm con sistemas de base de datos. Simplemente hay que abrir una conexión con la base de datos y leer o escribir en ella como normalmente se haría. Apache Storm lleva a cabo el paralelismo, la partición, y se encarga de los fallos que pudieran producirse.
- API simple y fácil de usar: cuando se programa en Apache Storm, se manipulan y transforman flujos de datos entorno a solo tres tipos de abstracciones: grifos (*spouts*), rayos (*bolts*) y topologías.
- Tolerante a fallos: Apache Storm es tolerante a los fallos de los denominados trabajadores o *workers*. Cuando un trabajador deja de funcionar, Storm lo reinicia de manera automática. Si un nodo deja de funcionar, el trabajador será reiniciado en otro nodo. Los demonios de Apache Storm, el Nimbus y los Supervisores están diseñados para no tener estado y para recuperarse rápidamente ante fallos, por lo que si dejan de funcionar se reiniciarán como si nada hubiera ocurrido. Esto significa que pueden dejar de funcionar varios demonios de Apache Storm sin afectar al estado del conjunto o de sus topologías.
- Compatible con cualquier lenguaje de programación: Apache Storm fue diseñado desde cero para poder utilizarse con cualquier lenguaje de programación. En el núcleo de Apache Storm se define *Thrift*, que puede usarse en cualquier lenguaje. Por lo tanto, las topologías pueden ser definidas en cualquier lenguaje, al igual que los *spouts* y los *bolts*. Aunque Storm ofrece soporte a otros lenguajes, la mayoría de topologías están escritas en Java, ya que resulta ser la opción más eficiente.

A continuación se exponen los motivos principales que hacen de Storm una herramienta excepcional en el ámbito de la tolerancia a fallos:

- Si un trabajador de cualquier nodo esclavo deja de funcionar, el demonio Supervisor lo reiniciará. Si el reinicio de un nodo falla repetidamente, el trabajador será reasignado a otra máquina.
- Si un nodo esclavo deja de funcionar en su totalidad, su parte del trabajo le será dada a otro nodo esclavo (Supervisor).
- Si el Nimbus falla, los trabajadores no se verán afectados. Sin embargo, los trabajadores no serán reasignados a otros nodos esclavos si fallan, a no ser que el Nimbus sea restaurado.
- El Nimbus y el Supervisor no tienen estado por sí mismos. Pero el ZooKeeper, almacena cierta información de estado de tal manera que si un nodo falla o si un demonio deja de funcionar inesperadamente, todo puede comenzar desde donde se quedó.
- El Nimbus, el Supervisor y el ZooKeeper se recuperan rápidamente frente a fallos. Esto quiere decir que estos, por su propia cuenta, no son muy tolerantes a errores inesperados y se cerrarán si encuentran alguno. Por esta razón tienen que ser ejecutados bajo supervisión, usando un programa de "perro guardián" (*d supervisor*) que los monitorice constantemente y los reinicie automáticamente si dejan de trabajar correctamente.

Es importante tener en cuenta que en la mayoría de conjuntos de Storm, el Nimbus no es instalado como una única instancia, sino como un grupo de ordenadores. Si esta tolerancia a fallos no es incorporada y el único Nimbus se cae, se perderá la habilidad de enviar nuevas topologías, se perderán fácilmente las topologías ejecutándose, la reasignación de trabajos a otros Supervisores si uno falla, etc. [37]

Storm provee de un nivel de topología más abstracto mediante las denominadas topologías transaccionales, que resuelven este problema. Las topologías transaccionales están construidas en base a los flujos, los *spouts* y los *bolts* primitivos de Storm y sobre, al menos, uno de los procesos de enrutamiento de los mensajes.

Las topologías transaccionales trabajan de manera un poco diferente a las topologías normales. En lugar de procesar tupla a tupla, procesan pequeños grupos de tuplas a la vez, de tal manera que si algo en un grupo falla, se repite de nuevo ese grupo entero. Las topologías transaccionales proporcionan un mayor orden que las normales y pueden proporcionar semánticas de procesamiento exactamente en una ocasión.

Trident es una abstracción de alto nivel para realizar computación en tiempo real sobre Storm. Le permite un alto rendimiento (millones de mensajes por segundo) a la misma vez que un procesamiento del flujo con estado mediante consultas distribuidas de baja latencia.

Trident, que se desarrolló a partir de un esfuerzo previo de Storm para proporcionar garantía de procesamiento en exactamente una ocasión, procesa los flujos como pequeños lotes de tuplas. En general, el tamaño de esos lotes será del orden de miles o millones de tuplas. En la Figura 3-15 se explica el funcionamiento de Trident en Storm.

La API que Trident proporciona es muy similar a las que se puede encontrar en las abstracciones de alto nivel para Hadoop como Pig [43], las cuales permiten agrupaciones, uniones, agregaciones, funciones de ejecución, filtros, etc. Como es evidente, procesar cada lote pequeño de forma aislada no es interesante, por lo que Trident proporciona funciones para realizar agregaciones en lotes y almacenarlas de forma persistente, ya sea en la memoria, en Memcached [44], en Cassandra [45] o en algún otro almacén.

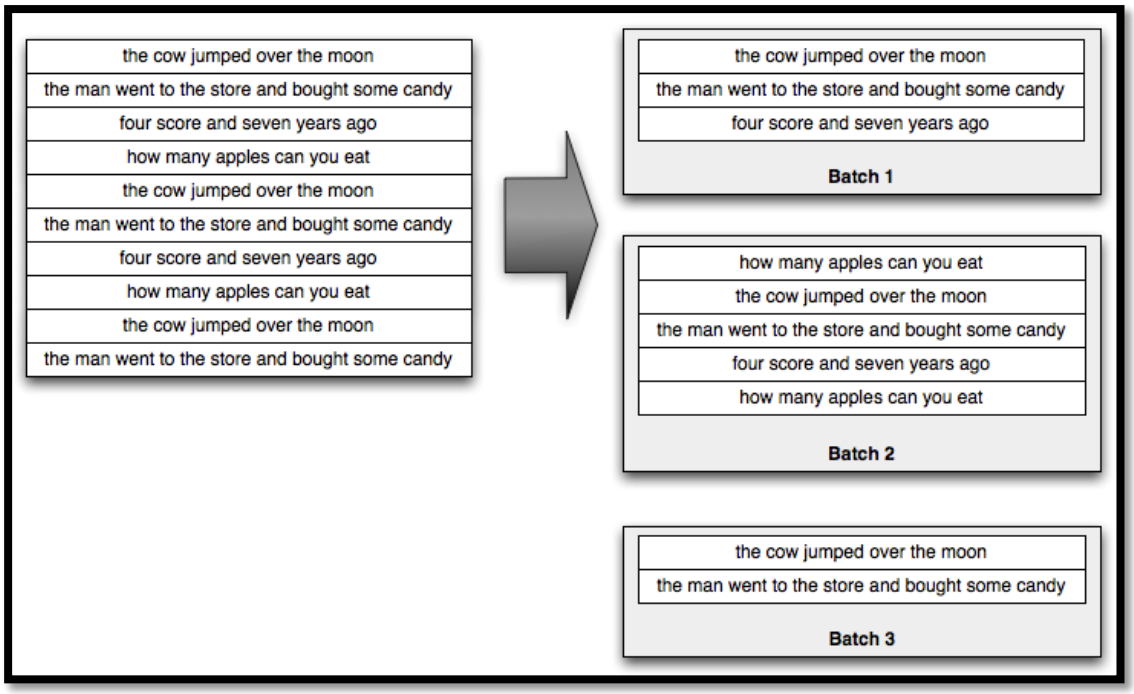

Figura 3-15 Trident procesa flujos como pequeños lotes de tuplas

Trident maximiza el rendimiento de una topología al ser inteligente a la hora de ejecutarla. Hay dos cosas interesantes que suceden automáticamente en sus topologías:

- Las operaciones que leen o escriben en el estado (como *persistentAggregate* y *stateQuery*) realizan automáticamente operaciones por lotes en ese estado. Entonces, si hay 20 actualizaciones que deben realizarse en la base de datos para el lote actual de procesamiento, en lugar de hacer 20 solicitudes de lectura y 20 solicitudes de escritura en la base de datos, Trident agrupará automáticamente las lecturas y escrituras, haciendo solo una solicitud de lectura y una solicitud de escritura (y en muchos casos, puede usar el almacenamiento en caché en la implementación de su estado para eliminar la solicitud de lectura).
- Los agregadores de Trident están muy optimizados. En lugar de transferir todas las tuplas de un grupo a la misma máquina y luego ejecutar el agregador, Trident realizará agregaciones parciales cuando sea posible antes de enviar tuplas a través de la red.

Un problema clave que afecta al cómputo en tiempo real es cómo administrar el estado para que las actualizaciones sean idempotentes ante fallos y reintentos. Es imposible eliminar los fallos, por lo que cuando un nodo muere o algo sale mal, los lotes deben ser reintentados. Para que cada mensaje se procese solo una vez, Trident realiza actualizaciones de estado (ya sean bases de datos externas o estado interno de la topología). Para ello ejecuta dos acciones:

- Cada lote recibe una identificación única llamada "identificación de la transacción". Si se vuelve a intentar un lote, tendrá exactamente el mismo identificador de transacción.
- Las actualizaciones de estado se ordenan entre lotes. Es decir, las actualizaciones de estado para el lote 3 no se aplicarán hasta que las actualizaciones de estado para el lote 2 hayan tenido éxito.

Con estas dos primitivas, puede lograr una semántica “exactamente una vez” con sus actualizaciones de estado.

Como es obvio, no se tiene que hacer esta lógica manualmente en las topologías. Esta lógica está envuelta por la abstracción del estado y se realiza automáticamente. Tampoco se requiere de un objeto de estado para implementar la identificación de la transacción: si no se desea pagar el coste de almacenar la identificación de la transacción en la base de datos, no se tiene que hacer. En ese caso, el estado tendrá semántica de procesamiento de al menos una ocasión en caso de fallo (lo cual puede ser adecuado para la aplicación que se está desarrollando).

Las topologías Trident se compilan en una topología Storm tan eficiente como sea posible. Las tuplas solo se envían a través de la red cuando se requiere un reparto de los datos, por ejemplo, si se hace un *groupBy* o un *shuffle*. Por lo tanto, si se tiene una topología como la de la Figura 3-16 se compilaría en Trident como la de la Figura 3-17.

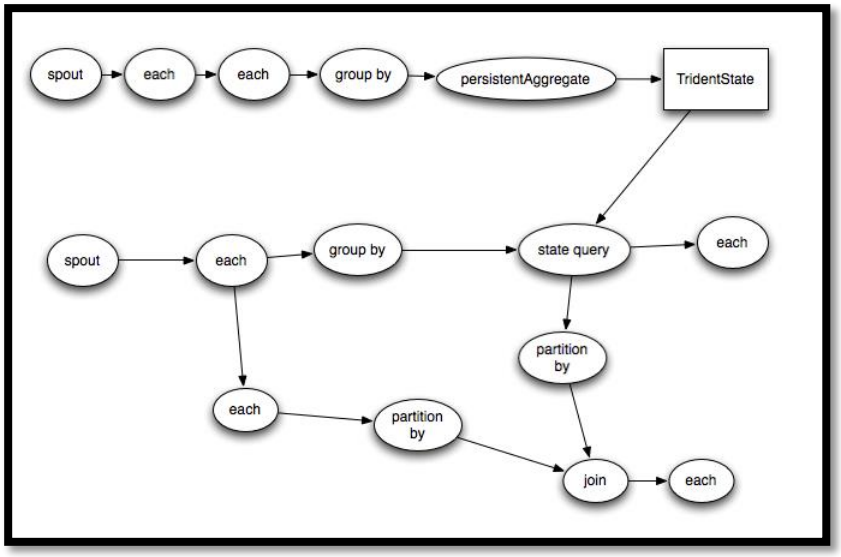

Figura 3-16 Idea de topología inicial

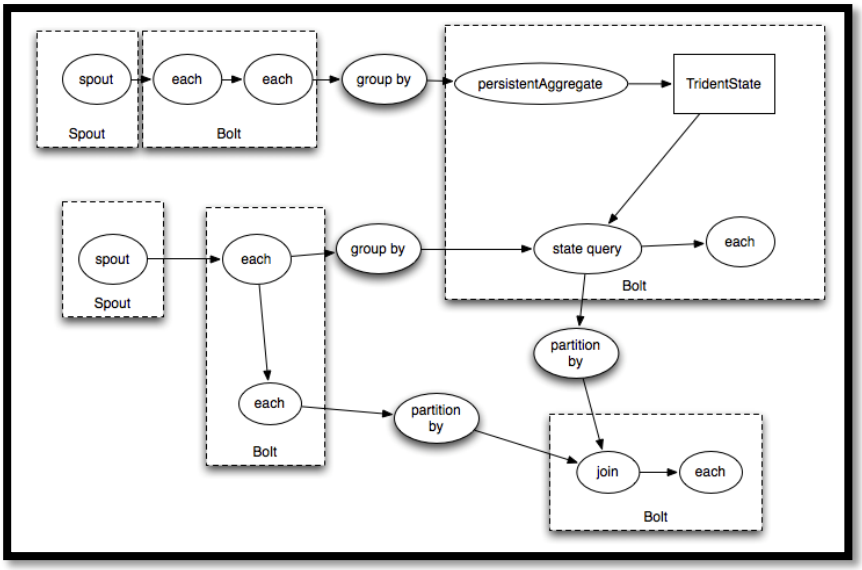

Figura 3-17 Topología anterior implementada con Storm Trident

En resumen, Trident hace que la computación en tiempo real sea elegante. El procesamiento de flujo de alto rendimiento, la manipulación de estado y las consultas de baja latencia se pueden mezclar sin problemas a través de su API. Trident permite expresar cálculos en tiempo real de una manera natural mientras se obtiene el máximo rendimiento.

3.1.2.4 **Escenarios de uso**

Apache Storm consta de una comunidad grande y creciente, así como de un soporte eficaz. Su página web [31] muestra toda la información necesaria para poder entender el proyecto y para poder colaborar en su desarrollo: explica dónde y cómo aprender los detalles de su código base. Así mismo, indica que existen varias maneras de contribuir: para mejorar pequeños problemas en el código base de Storm deben enviarse peticiones a GitHub; para problemas más grandes hay que utilizar el rastreador de problemas JIRA [46].

Por otro lado, [31] dispone de toda la información necesaria para su instalación y configuración, así como ejemplos ilustrativos de cómo proceder. Al ser una herramienta popular, existen además diversos blogs, en los que se encuentra multitud de información complementaria de interés o populares páginas

web (como Udacity [47]) que ofrecen cursos en los que se explica de manera clara y concisa cómo configurarla y ponerla en marcha.

En el caso de necesitar ayuda, Storm deja a disposición de los usuarios varios correos a los que se puede acudir con consultas.

Como se puede observar, Storm es usado por algunas de las compañías más importantes a nivel mundial, lo que evidencia la utilidad de esta herramienta en el análisis de flujos. En el Anexo I, puede comprobarse su gran popularidad.

Basándonos en medidas de popularidad tradicionales, estos son los resultados que se obtienen:

- Resultados en Google de "Apache Storm": 546.000
- Seguidores en Twitter de Apache Storm (@stormprocessor): 8031
- Fecha de búsqueda: 06 de marzo de 2020

A continuación se exponen dos casos reales en los que Storm es de utilidad, ambos descritos en la sección *powered by* de [31].

Caso 1: Laboratorios IDEXX.

Laboratorios IDEXX [48] es el fabricante líder de *software* e instrumentos de diagnóstico para el mercado veterinario. Recopila y analiza datos médicos veterinarios de miles de clínicas. Con el paso del tiempo su infraestructura de procesamiento de datos quedó obsoleta al no poder lidiar con el rápido aumento del volumen, la velocidad y la variedad en los datos que debían procesar. Por ello decidieron embarcarse en un proyecto de actualización que utilizaba Storm.

Storm les permite incorporar datos que se extraen de registros médicos en varios formatos diferentes. Después, con esta herramienta, los transforman en uno estándar y los almacenan en una base de datos relacional. Básicamente es un sistema ETL (de extracción, transformación y carga) distribuido mejorado en el que Storm se encarga de las canalizaciones necesarias en el sistema y les permite escribir el código muy fácilmente. La capacidad de crear pequeñas piezas funcionales y conectarlas les brinda la máxima flexibilidad para paralelizar cada una de las tareas de manera diferente.

El conjunto que emplean consta de 4 Supervisores que ejecutan 110 tareas dentro de 32 procesos de trabajo. Ejecutan dos topologías diferentes que reciben mensajes y se comunican entre sí a través de RabbitMQ.

Caso 2: Navisite.

Navisite es un proveedor internacional para alojamiento en la nube y para servicios y aplicaciones administradas. Navisite ofrece un conjunto completo de servicios administrados confiables y escalables, que incluyen servicios de aplicaciones, escritorio e infraestructura en la nube, así como alojamientos para organizaciones que buscan subcontratar infraestructuras de Tecnología Informática (TI) para ayudar a reducir sus costes operativos y de capital. Los clientes empresariales dependen de Navisite para obtener soluciones personalizadas, las cuales son entregadas a través de centros de datos de última generación con presencia internacional. [49]

Navisite utiliza Storm como parte de su sistema de monitorización y auditoría de registro de eventos del servidor. Envía mensajes de registro de miles de servidores a un conjunto RabbitMQ y luego usa Storm para verificar cada mensaje con un set de expresiones regulares. Si hay una coincidencia (<1% de los mensajes), el mensaje se envía a un *bolt* que almacena datos en una base de datos denominada Mongo [50]. Aproximadamente manejan una carga de entre 5k y 10k mensajes por segundo, sin embargo, llegaron a comprobar que los grupos existentes de RabbitMQ en combinación con Storm permitían lidiar con hasta 50k por segundo. Navisite ha implementado Storm en la plataforma *NaviSite Cloud*; y hace uso de un conjunto ZooKeeper de 3 máquinas virtuales pequeñas, 1 máquina virtual Nimbus y 16 máquinas virtuales de doble núcleo y 4 GB como Supervisores.

3.1.3 Apache Flink

3.1.3.1 **¿Qué es Apache Flink?**

Apache Flink [51] es un *framework* de procesamiento distribuido pensado inicialmente para aplicaciones de *streaming*. Se caracteriza por tratar los procesamientos por lotes como un caso especial de flujo de datos, de forma que se puede considerar un *framework* de procesamiento tanto por lotes como en tiempo real. La herramienta incluye una serie de APIs: una API de *streaming* para Java y Scala, una API de datos estáticos para Java, Scala y Python y una API de consultas *SQL-like* para códigos escritos en Java y Scala. [2]

3.1.3.2 **Origen e historia**

Flink tiene su origen en el proyecto de investigación Stratosphere, con sede en Berlín, y realizado por tres universidades europeas entre los años 2010 y 2014. En abril de 2014 pasó a formar parte de la Apache Software Foundation para continuar su desarrollo. [52]

Durante la incubación en esta Fundación, el nombre del proyecto tuvo que cambiarse de Stratosphere a Flink, pues el nombre Stratosphere hacía que se confundiera con un proyecto no relacionado. El nombre Flink fue seleccionado para honrar el estilo de procesado de secuencias y lotes que lleva a cabo: veloz. En alemán, la palabra *flink* significa rápido o ágil.

El proyecto completó la incubación rápidamente y, en diciembre de 2014, Flink se convirtió en un proyecto de alto nivel de la Apache Software Fundation; de hecho, se encuentra entre los 5 mayores proyectos de *big data* de esta plataforma.

Respecto al lenguaje de programación, Flink es compatible únicamente con Java, aunque actualmente Apache está desarrollando una nueva API que dará soporte a Python en un futuro próximo.

Como curiosidad, resulta interesante apuntar que como logo (Figura 3-18) se eligió una ardilla porque estas son rápidas y ágiles, al igual que Flink.

Figura 3-18 Logo de Flink [52]

La última versión de Apache Flink, publicada en febrero de 2020, es la versión 1.10.0. [51]

3.1.3.3 **Funcionamiento**

Para llegar a comprender cómo funciona Apache Flink resulta imprescindible entender el concepto de "aplicación con estado" y el concepto de "*backend* de estado conectable", los cuales se explican a continuación:

- Aplicaciones con estado: son aquellas que pueden mantener el resultado de una agregación o un resumen de los datos que han sido procesados durante un tiempo determinado. Toda aplicación de *streaming* no trivial tiene estado, es decir, solo las aplicaciones que aplican transformaciones a eventos individuales no requieren estado. Cualquier aplicación que

ejecuta lógica de negocios básica necesita recordar eventos o acceder a resultados intermedios en un momento posterior.

- *Backends* de estado conectable: el estado de la aplicación es administrado y comprobado por un *backend* de estado conectable. Flink presenta distintos *backends* de estado que almacenan el estado en la memoria o en la base de datos RocksDB.

Apache Flink es un motor de procesamiento distribuido para cálculos de estado sobre flujos de datos ilimitados y limitados. Flink se ejecuta en cualquier escala y realiza cálculos a alta velocidad.

En la actualidad, multitud de datos se producen como una secuencia de eventos. Las transacciones con tarjeta de crédito, las mediciones de sensores, los registros de máquinas, las interacciones del usuario en un sitio web o las aplicaciones de dispositivo móvil, son solo unos pocos ejemplos de datos que se generan como una secuencia.

Apache Flink sobresale en el procesamiento de conjuntos de datos ilimitados y limitados. El control preciso del tiempo y del estado permiten a Flink ejecutar cualquier tipo de aplicación de flujos ilimitados. Para el procesamiento de secuencias limitadas Flink recurre a algoritmos y estructuras de datos que están específicamente diseñadas para conjuntos de datos de tamaños fijos, obteniéndose también un excelente rendimiento. [51]

A continuación se sintetizan los principales rasgos que hacen de Flink una herramienta de procesado puntera en su sector:

- Exactitud del estado: Flink presenta algoritmos de recuperación frente a fallos que garantizan la consistencia del estado de la aplicación. Por lo tanto, los errores se manejan de manera transparente y no afectan a la corrección de una aplicación.
- Mantiene aplicaciones con grandes estados: el algoritmo de punto de control asíncrono e incremental permite mantener el estado de una aplicación de varios *terabytes* de tamaño.
- Permite aplicaciones escalables: las aplicaciones se paralelizan en miles de tareas que se distribuyen y ejecutan simultáneamente en un grupo de computadoras. Por lo tanto, una aplicación puede aprovechar cantidades prácticamente ilimitadas de CPU, memoria principal y disco.
- Flink es compatible con todos los administradores de conjunto más comunes como Hadoop YARN, Apache Mesos y Kubernetes [53]. También se puede configurar para ejecutarse como un agrupamiento independiente. Para funcionar bien con cada uno de los administradores de grupos enumerados anteriormente, Flink implementa modos específicos que permiten interactuar con cada administrador de recursos en su forma idiomática.
- Para realizar todos los cálculos asociados a las tareas Flink accede al estado local, disponible con mucha frecuencia en la memoria, produciendo latencias de procesamiento muy bajas.

El tiempo es otro elemento importante en las aplicaciones de *streaming*. Cada evento en *streaming* se produce en un momento específico y muchos de los cálculos más comunes realizados sobre flujos se basan en el tiempo. Debido a esto, un aspecto muy importante en el procesamiento de flujos consiste en la medición del tiempo. Flink puede trabajar en dos modos: con tiempo de evento o con tiempo de procesamiento. La diferencia entre ambos se explica a continuación y puede visualizarse en la Figura 3-19.

- Modo de tiempo de evento: las aplicaciones que procesan flujos con semántica de tiempo de evento calculan los resultados en función de las marcas de tiempo de dichos eventos. De este modo, se obtienen resultados precisos y consistentes sin importar si se procesan eventos almacenados o en tiempo real. Flink emplea marcas de agua para medir el tiempo en las aplicaciones que emplean el tiempo de evento.
- Modo de tiempo de procesamiento: además del modo de tiempo de evento, Flink admite semántica de tiempo de procesamiento. El modo de tiempo de procesamiento es adecuado

para ciertas aplicaciones con estrictos requisitos de baja latencia que pueden tolerar la obtención de resultados aproximados.

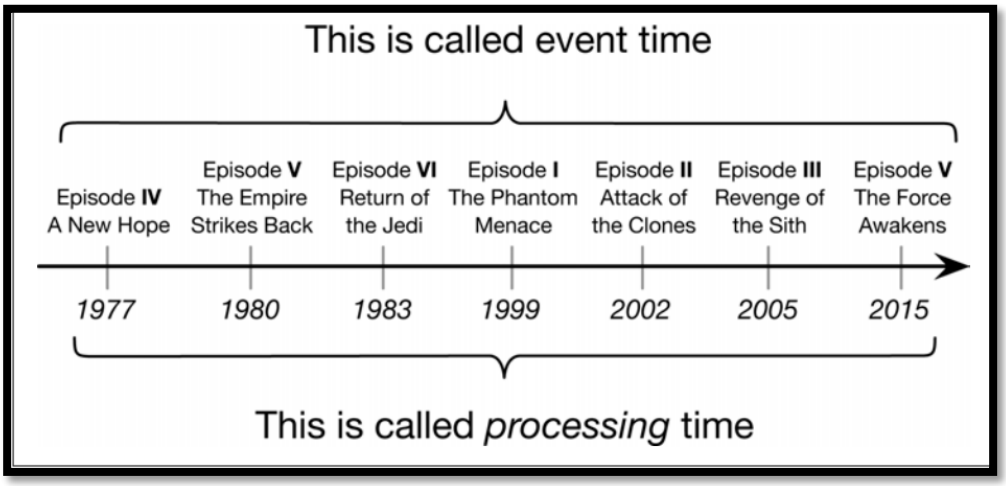

Figura 3-19 Flujo de eventos en desorden donde el orden del tiempo de procesamiento es distinto al del tiempo de evento [52]

Existen muchas aplicaciones de *streaming* diseñadas para ejecutarse continuamente con un tiempo de inactividad mínimo, por ello un procesador de flujo debe proporcionar excelente recuperación frente a fallos, así como herramientas para monitorizar y mantener las aplicaciones mientras se ejecutan. Apache Flink pone un fuerte enfoque en los aspectos operativos del procesado de flujos. Los fallos de máquinas y procesos son ubicuos en los sistemas distribuidos. Según [51], Flink ofrece varias características que garantizan la ejecución y consistencia de las aplicaciones:

- Puntos de control consistentes: el mecanismo de recuperación de Flink se basa en puntos de control consistentes en el estado de una aplicación. En caso de fallo, la aplicación se reinicia y su estado se carga desde el último punto de control. En combinación con fuentes de transmisión reiniciables, esta función puede garantizar la consistencia del estado con semántica de exactamente una ocasión.
- Puntos de control eficientes: Flink utiliza puntos de control asíncronos e incrementales con objeto de mantener una baja latencia.
- De extremo a extremo exactamente una vez: Flink presenta sumideros transaccionales para sistemas de almacenamiento que garantizan que los datos solo se escriben exactamente una vez, incluso en caso de fallo.
- Integración con administradores de conjuntos: Flink está estrechamente integrado con Kubernetes, Hadoop YARN y Mesos, como se puede visualizar en la Figura 3-20.
- Configuración de alta disponibilidad: Flink presenta un modo de alta disponibilidad que elimina todos los puntos únicos de fallo. Este se basa en Apache ZooKeeper.

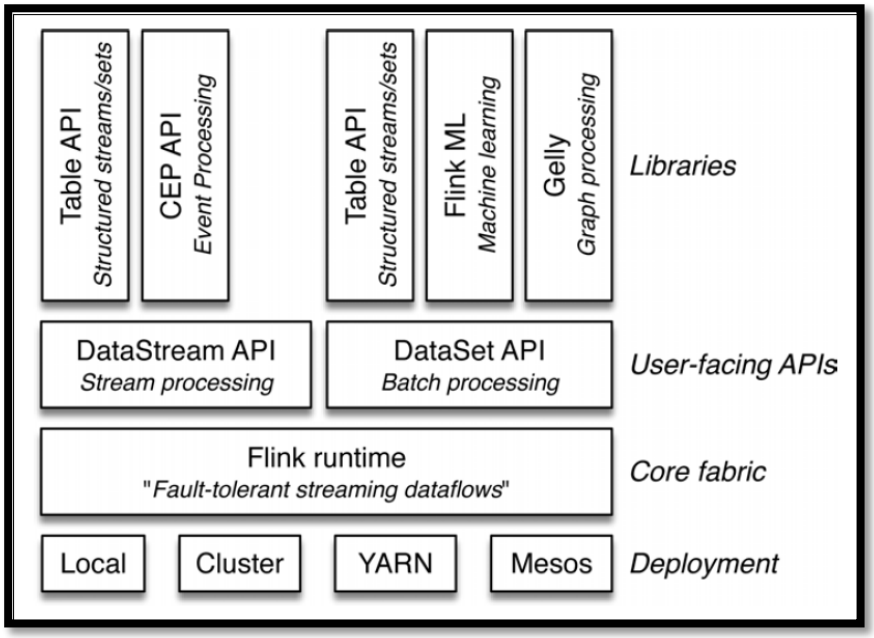

Figura 3-20 Diagrama con los componentes principales de Flink [52]

Los *savepoints*, que consituyen un elemento vital en Flink, son de gran utilidad. Como resulta lógico, las aplicaciones de *streaming* que potencian los servicios críticos del negocio deben ser mantenidas. Los errores deben corregirse y las mejoras o nuevas características deben ser implementadas. Sin embargo, actualizar una aplicación de transmisión con estado no es trivial. A menudo, uno no puede mejorar la versión porque ello podría suponer perder el estado de la aplicación.

Los *savepoints* de Flink son una característica única y poderosa que resuelve el problema de actualizar aplicaciones con estado y otros desafíos relacionados. Un *savepoint* es una instantánea coherente del estado de una aplicación y, por lo tanto, muy similar a un punto de control. Sin embargo, a diferencia de los puntos de control, los *savepoints* son activados manualmente y no son eliminados automáticamente cuando se detiene una aplicación. Un *savepoint* se puede usar para iniciar una aplicación compatible con el estado. Como queda recogido en [51], los *savepoints* habilitan las siguientes funciones:

- Evolución de la aplicación: los *savepoints* se pueden usar para desarrollar aplicaciones. Una versión mejorada de una aplicación se puede reiniciar desde un *savepoint* usado en una versión anterior. Estos permiten también iniciar la aplicación desde un momento anterior si falla la versión.
- Migración de conjunto: mediante el uso de *savepoints*, las aplicaciones pueden clonarse o trasladarse a diferentes agrupaciones.
- Actualizaciones de la versión de Flink: una aplicación puede migrar para ejecutarse en una nueva versión de Flink usando un *savepoint.*
- Escalado de aplicaciones: los *savepoints* disminuyen o aumentan el paralelismo de una aplicación.
- Pruebas A / B y escenarios hipotéticos: el rendimiento de dos (o más) versiones distintas de una aplicación se pueden comparar mediante el inicio de todas las versiones desde el mismo *savepoint.*
- Pausar y reanudar: una aplicación se puede detener en un *savepoint*. En cualquier momento posterior, la aplicación se puede reanudar desde dicho *savepoint*.
- Archivado: los *savepoints* se pueden archivar para restablecer el estado de una aplicación en un momento previo.

El control de las aplicaciones mediante la monitorización y el registro resulta esencial. Por un lado, monitorizar ayuda a anticiparse a los problemas y a reaccionar antes de tiempo frente a ellos. Por otro, el registro permite el análisis de la causa raíz en la investigación de fallos.

De acuerdo a lo reseñado en [51], Flink presenta interfaces de fácil acceso que permiten controlar aplicaciones en ejecución. Además, se integra muy bien con servicios comunes de registro y monitorización.

- Interfaz de usuario web: Flink presenta una interfaz de usuario web para inspeccionar, monitorizar y depurar la ejecución aplicaciones. También se puede utilizar para ordenar ejecuciones o cancelaciones.
- Registro: Flink implementa la popular interfaz de registro *Simple Logging Facade for Java* (slf4j) [54] y se integra con los marcos de registro log4j [55] o logback [56].
- Monitorización: Flink presenta un sofisticado sistema para recopilar métricas del sistema. Estas pueden ser exportadas a sistemas como *Java Management Extensions* (JMX) [57] o Ganglia [58].
- API REST: recoge metadatos y métricas recopiladas de aplicaciones en ejecución o finalizadas. Al implementarse una aplicación, Flink identifica automáticamente las fuentes requeridas basándose en el paralelismo con que ha sido configurada la aplicación y las solicita desde el administrador de recursos. En caso de que uno de los recursos falle, Flink lo reemplaza mediante la solicitud de nuevos recursos. Toda comunicación que se use para enviar o controlar una aplicación se realiza a través de las llamadas REST.

Flink proporciona APIs por capas. Cada API ofrece una compensación diferente entre concisión y expresividad como se indica en la Figura 3-21.

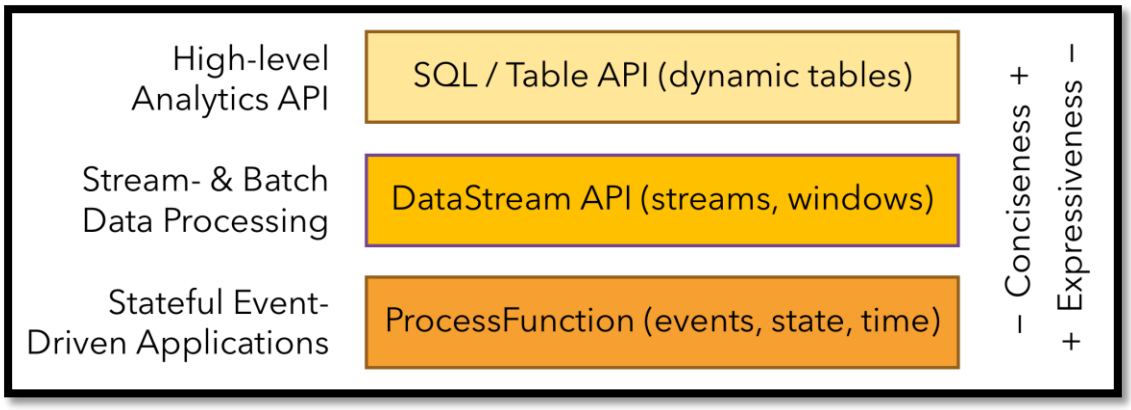

Figura 3-21 APIs de Flink apiladas en capas [51]

Las APIs de Flink se utilizan para procesar eventos individuales de una o dos secuencias de entrada o eventos agrupados en una ventana. Estas APIs proporcionan un control detallado sobre el tiempo y el estado. Una API puede modificar arbitrariamente su estado y programar temporizadores que desencadenen en un futuro una función de devolución de llamada. Por lo tanto, las APIs pueden implementar lógica empresarial compleja en los eventos. Esta lógica se requiere en muchos de los eventos con aplicaciones de estado. A continuación se explican las principales APIs de Flink según [51]:

- *DataStream*: API que proporciona primitivas el procesamiento de flujos. Se encuentra disponible para Java y Scala y contiene funciones como *map*, *reduce* o *aggregate*.
- APIs relacionales: Table API y SQL, diseñadas para facilitar el análisis y la canalización de los datos y las aplicaciones ETL. Ambas se unifican para el procesamiento por lotes y flujos, es decir, las consultas se ejecutan con la misma semántica en secuencias ilimitadas (en tiempo real) o limitadas (almacenadas) obteniéndose los mismos resultados. Table API y SQL aprovechan Apache Calcite [59] para analizar, validar y optimizar consultas. Además,

se pueden integrar perfectamente con las APIs *DataStream* y *DataSet* y admiten funciones escalares, agregadas y con valores de tabla definidos por el usuario.

Flink presenta librerías para casos comunes de empleo del procesamiento de datos:

- Complex Events Processing (CEP): proporciona una API para detectar patrones en eventos. Esta librería se integra con la API *DataStream* de Flink, de modo que los patrones son evaluados sobre *DataStream*. Algunas de las aplicaciones que se pueden encontrar en la librería CEP son: la monitorización de procesos comerciales o la detección de fraude.
- API *DataSet*: es la API principal de Flink para aplicaciones de procesamiento por lotes. Las primitivas de la API DataSet incluyen *map*, *reduce*, *(outer) join*, *co-group* e *iterate*. Todas las operaciones hacen uso de algoritmos y estructuras de datos que operan en datos serializados en la memoria y que, en caso de exceder el tamaño de la memoria, se guardan en el disco.
- Gelly: es una librería para procesamiento y análisis de grafos escalables. Gelly se implementa sobre la API *DataSet* y se integra con ella, beneficiándose de su escalabilidad y robustez. Gelly proporciona la API Graph, que facilita la implementación de algoritmos de grafos personalizados.

Apache Flink es una excelente opción para desarrollar y ejecutar muchos tipos diferentes de aplicaciones. A continuación, se va a proceder a explorar los tipos más comunes de aplicaciones que funcionan con Flink. Como queda recogido en [51], Flink se usa para tres tipos de aplicaciones:

i. Aplicaciones basadas en eventos: aplicación con estado que ingiere uno o más flujos de eventos y reacciona ante estos desencadenando una serie de cálculos, actualizaciones de estado o acciones externas sobre ellos.
Las aplicaciones basadas en eventos son una evolución del diseño tradicional de las aplicaciones. En la arquitectura clásica, las aplicaciones leen datos y los conservan en una base de datos transaccional remota. Por el contrario, las aplicaciones basadas en eventos, acceden a datos locales, lo que mejora el rendimiento, tanto en términos de producción como de latencia. La Figura 3-22 muestra la diferencia entre la arquitectura de una aplicación tradicional y la de otra basada en eventos.

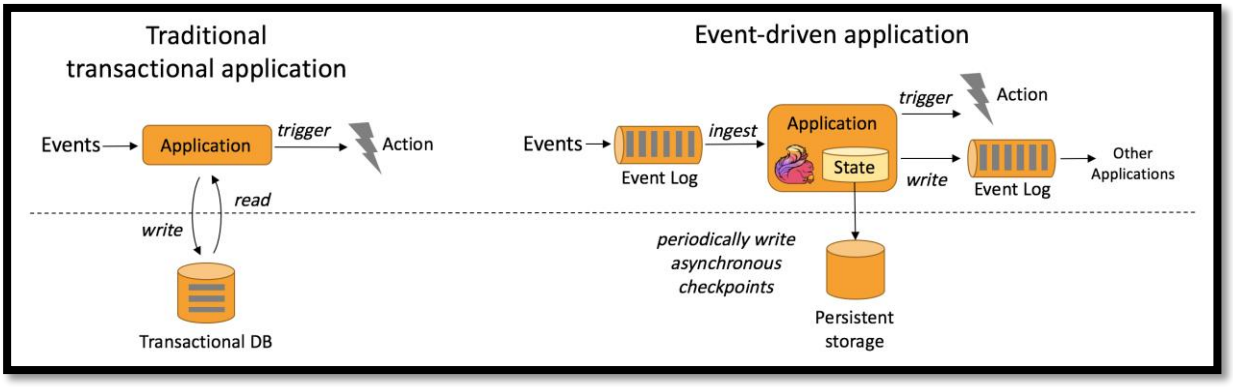

Figura 3-22 Aplicación tradicional vs Aplicación basada en eventos [51]

El diseño de las aplicaciones basadas en eventos proporciona más beneficios que solo el acceso a datos locales. Cualquier cambio en la base de datos, como cambiar el diseño de los datos debido a una actualización de la aplicación o a la ampliación del servicio, debe coordinarse. Dado que cada aplicación basada en eventos es responsable de sus propios datos, los cambios en la representación de los datos o en la escala de la aplicación requiere menos coordinación.

Los límites de las aplicaciones basadas en eventos están definidos por la capacidad del procesador de flujos para manejar el tiempo y el estado. Muchas de las características sobresalientes de Flink están centradas en estos conceptos. Flink proporciona un rico conjunto de primitivas de estado que permiten administrar volúmenes de datos muy grandes (hasta varios *terabytes*) con garantías de consistencia de exactamente una vez.
Sin embargo, la característica sobresaliente de Flink para aplicaciones basadas en eventos es el *savepoint*, que permite actualizar o adaptar la escala de las aplicaciones.
Ejemplos de aplicaciones típicas basadas en eventos son: detección de fraude, detección de anomalías, alertas basadas en reglas, monitoreo de procesos comerciales y las aplicaciones web (red social).

ii. Aplicaciones de análisis de datos: se encargan de extraer información y conocimiento de los datos sin procesar.
Con un sofisticado motor de procesamiento de flujo, se pueden realizar análisis en tiempo real. En lugar de leer conjuntos de datos finitos, las aplicaciones ingieren flujos de eventos en tiempo real y producen y actualizan continuamente los resultados a medida que se consumen los eventos. Los resultados se escriben en una base de datos que puede ser externa o interna. Apache Flink es compatible con el *streaming* y las aplicaciones analíticas por lotes como se muestra en la Figura 3-23.

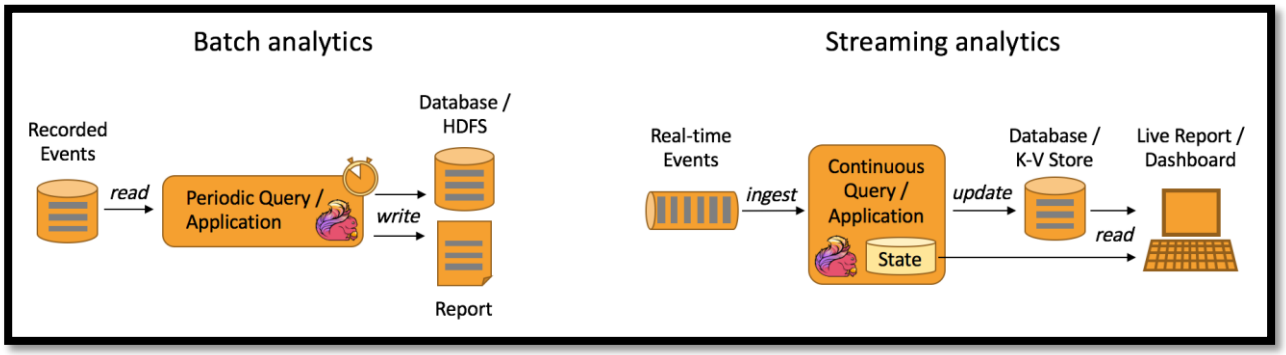

Figura 3-23 Análisis por lotes vs Análisis por streaming [51]

Las ventajas de analizar flujos continuos en vez de por lotes no se limitan simplemente a una latencia mucho más baja de los eventos, sino también a la eliminación de la importación periódica y de la ejecución de las consultas. A diferencia de las consultas por lotes, las consultas de transmisión no tienen que tratar con límites artificiales en los datos de entrada que son causados por importaciones periódicas.
Otro aspecto es que el análisis por *streaming* presenta una arquitectura de aplicación más simple. Una canalización de análisis de lotes consta de varios componentes independientes para programar periódicamente la ingestión de datos y la ejecución de consultas. Operar de manera fiable como canalización no es trivial porque los fallos en un componente afectan a los siguientes pasos de la de la canalización en cuestión. Por el contrario, una solicitud de análisis de flujo que se ejecute en un procesador de flujos sofisticado como Flink incorpora todos los pasos, desde la ingesta de datos hasta el continuo cálculo de resultados.
Flink proporciona muy buen soporte tanto para el análisis de flujos continuos como por lotes mediante el interfaz SQL. Las consultas SQL calculan el mismo resultado independientemente de si se ejecutan en un conjunto de datos estáticos de eventos almacenados o en una secuencia de eventos en tiempo real.
Algunas de las aplicaciones más típicas de análisis de datos son: monitorización de la calidad de redes de telecomunicaciones, evaluación de experimentos en aplicaciones móviles, análisis *ad-hoc* de datos en vivo en tecnología del consumidor y análisis de grafos a gran escala.

iii. Por último, se van a explicar las aplicaciones de canalización de datos, describiendo para ello previamente las tareas ETL.
Extraer-transformar-cargar (ETL) es un enfoque común para convertir y mover datos entre sistemas de almacenamiento. A menudo, los trabajos ETL se activan periódicamente para copiar datos desde sistemas de bases de datos transaccionales a una base de datos analítica.
Las canalizaciones de datos tienen un propósito similar al de los trabajos ETL. Transforman y enriquecen los datos y los mueven de un sistema de almacenamiento a otro. Sin embargo, operan en un modo de transmisión continua en lugar de activarse periódicamente. Por lo tanto, pueden leer los registros de fuentes que continuamente producen datos y los mueven con baja latencia a otro destino. La Figura 3-24 muestra la diferencia entre los trabajos periódicos de ETL y la canalización de datos.

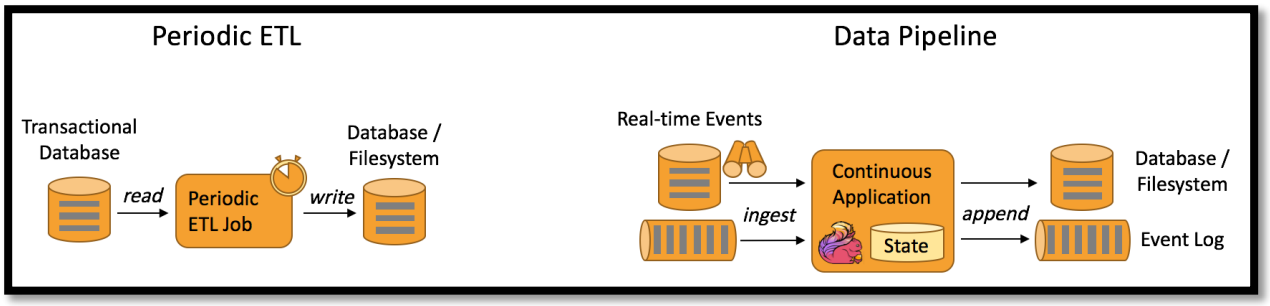

Figura 3-24 ETL (periódico) vs Canalización de datos (continuo) [51]

La ventaja obvia de los canales de datos continuos sobre los trabajos periódicos de ETL es la latencia reducida de mover datos a su destino. Por otra parte, las canalizaciones de datos son más versátiles y pueden emplearse para más casos de uso porque son capaces de consumir y emitir datos continuamente.

Muchas tareas comunes de transformación o enriquecimiento de datos son abordadas por el interfaz SQL de Flink (o Table API). Las canalizaciones de datos con requisitos más avanzados se pueden realizar utilizando la API *DataStream*, que es más genérica. Flink proporciona un rico conjunto de conectores para varios sistemas de almacenamiento como Kafka [60] o Elasticsearch [61] y sistemas de bases de datos como *Java Database Connectivity* (JDBC) [62].

Las aplicaciones típicas de canalización de datos son: creación de índices de búsqueda en tiempo real para comercio electrónico y ETL continuo en comercio electrónico.

3.1.3.4 **Escenarios de uso**

Como se puede observar Flink es usado por algunas de las compañías más importantes a nivel mundial, lo que evidencia la utilidad de esta herramienta en el análisis de flujos. En el Anexo I existe más información al respecto.

Con el objetivo de medir la popularidad de la herramienta, se ha procedido igual que en los sistemas estudiados anteriormente, obteniendo lo siguiente:

- Resultados en Google de “Apache Flink”: 661.000
- Seguidores en Twitter de Apache Flink (@ApacheFlink): 12.100
- Fecha de búsqueda: 06 de marzo de 2020

Un caso de uso real de esta herramienta es el que se expone a continuación. En él se puede observar la utilidad que puede llegar a tener en el desempeño de tareas complejas requeridas por grandes empresas como Bouygues Telecom.

Bouygues Telecom [63] es el tercer proveedor de telefonía móvil en Francia. Bouygues usa Flink para el procesamiento y análisis de eventos en tiempo real de miles de millones de mensajes por día en un sistema que funciona continuamente.

En una publicación de junio de 2015, Mohamed Amine Abdessemed, un representante de Bouygues, describió los objetivos del proyecto de la empresa y las razones por las que eligió Flink para cumplirlos. Bouygues eligió Flink porque el sistema ofrecía verdadero *streaming*, tanto en la API como a nivel de tiempo de ejecución, lo que proporcionaba la programabilidad y baja latencia que estaban buscando. Adicionalmente, pudieron poner su sistema en funcionamiento con Flink en una fracción de tiempo muy pequeña en comparación con otras soluciones. En la Figura 3-25 se puede observar de qué manera Bouygues pretendía implementar Flink.

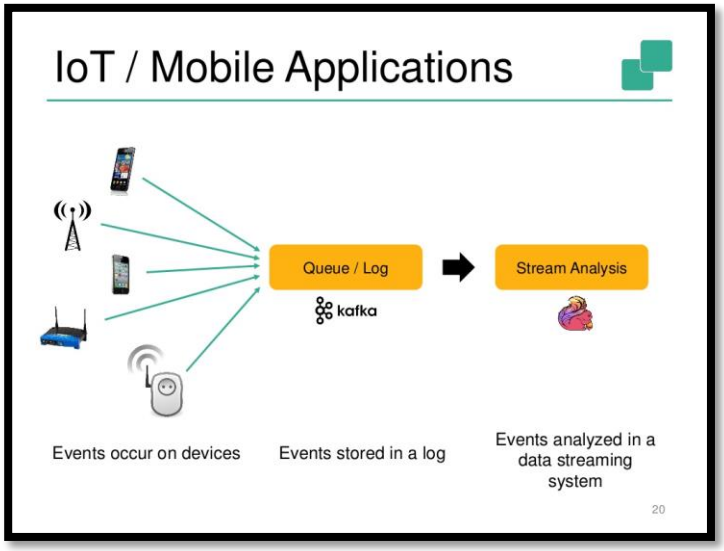

Figura 3-25 Flink ofrecía a Bouygues la baja latencia y programabilidad deseada [64]

Bouygues quería dar a sus ingenieros información en tiempo real sobre la experiencia del cliente, lo que estaba sucediendo a nivel nacional en la red y lo que estaba sucediendo en términos de evolución de la red y operaciones. Para hacer esto, construyeron sistemas capaces de analizar los registros de los equipos de red para identificar los indicadores de la calidad de experiencia del usuario. Cada sistema manejaba dos mil millones de eventos por día (500.000 eventos por segundo) con una latencia de extremo a extremo requerida de menos de 200 milisegundos (incluida la publicación de mensajes por la capa de transporte y el procesamiento de datos en Flink). Esto se logró en un pequeño grupo de solo 10 nodos con 1 *gigabyte* de memoria cada uno. En la Figura 3-26 se observa el número de nodos involucrados en la instalación y otra información relativa al tipo de datos que se manejaban y su volumen, así como la velocidad de procesado de estos.

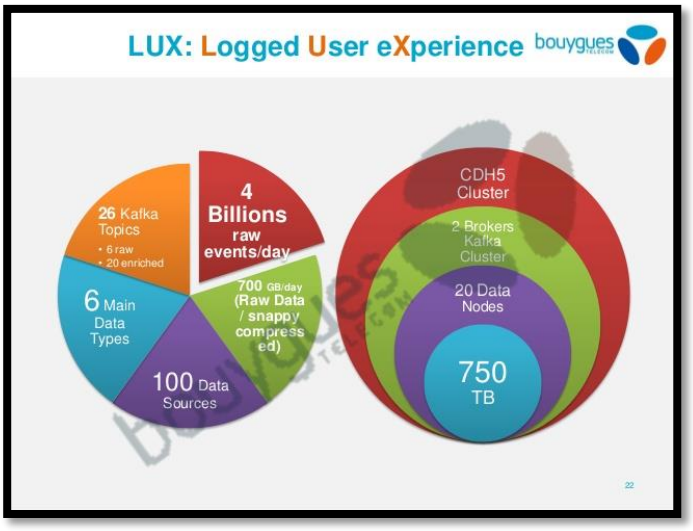

Figura 3-26 Gráfico informativo sobre el procesamiento de datos con Flink por parte de Bouygues [64]

Bouygues también quería que otros grupos pudieran reutilizar datos parcialmente procesados para una variedad de propósitos de *Business Intelligence* (BI), sin interferir los unos con los otros. El plan de la compañía era utilizar el procesamiento continuo de Flink para transformar y enriquecer datos. Los flujos de datos derivados serían entonces transportados de nuevo y se pondrían a disposición para realizar análisis de consumidores.

Este enfoque fue elegido explícitamente en lugar de otras opciones de diseño, como procesar los datos antes de que entren en la cola de mensajes, o delegar el procesamiento a múltiples aplicaciones que consumen de la cola de mensajes. La capacidad de procesamiento de flujo de Flink permitió al equipo de Bouygues completar el procesamiento de datos y la canalización de su movimiento mientras cumplía con los requisitos de baja latencia y alta fiabilidad, alta disponibilidad, y facilidad de uso.

3.1.4 Apache Samza

3.1.4.1 **¿Qué es Apache Samza?**

Apache Samza [65] es un *framework* de procesamiento en *streaming*, compatible con los lenguajes de programación Java y Scala, que está estrechamente relacionado con Apache Kafka. Mientras que Kafka es ampliamente usado en sistemas de procesamiento de *streaming*, Samza está diseñado para aprovechar todas las ventajas arquitectónicas que Kafka ofrece, así como el *buffering*, el almacenamiento de estado y la tolerancia a fallos. Samza usa YARN, por lo que es necesario Hadoop para su ejecución (al menos HDFS y YARN). Como Spark trabaja con registros inmutables de información, Samza maneja *streams* inmutables, esto quiere decir que una transformación en los datos crea un nuevo *stream* sin afectar al original. [66]

Los principales aspectos diferenciadores de Samza son:

- Samza hace uso del estado local mediante tablas ubicadas junto a las tareas de procesamiento. El estado en sí mismo se modela como un flujo. Si el estado local se pierde debido a un fallo de la máquina, podría leer el estado almacenado en el disco y continuar. Si hubiera un fallo de *hardware* (por ejemplo en el disco), entonces acudiría a sistemas de ficheros distribuidos (como HDFS), que almacenan réplicas del contenido en otras máquinas.
- Los flujos son ordenados, particionados, reproducibles y tolerantes a fallos.
- YARN se usa para el aislamiento del procesador, para la seguridad y para la tolerancia a fallos.
- Los trabajos están desacoplados: si un trabajo se ralentiza y acumula mensajes no procesados, el resto del sistema no se ve afectado.

3.1.4.2 **Origen e historia de Apache Samza**

LinkedIn comenzó a procesar *big data* con Apache Hadoop. Con el paso del tiempo, observó que Hadoop no era adecuado para realizar ciertas tareas debido al gran tiempo de respuesta que necesitaba para el procesamiento por lotes. Quería que sus resultados se calcularan de forma incremental y que estuvieran a su disposición de inmediato. Como resultado de esto, en septiembre de 2013, LinkedIn creó una herramienta "ligera" para el procesamiento de flujos de datos llamada Apache Samza, cuyo logo podemos visualizar en la Figura 3-27. Cabe destacar que casi al mismo tiempo, LinkedIn desarrolló Apache Kafka.

Figura 3-27 Logo de Apache Samza [65]

La última versión de Apache Samza, publicada en febrero de 2020, es la versión 1.3.1. [27]

3.1.4.3 **Funcionamiento de Apache Samza**

Con la principal finalidad de entender el funcionamiento de esta herramienta de análisis, se muestra a lo largo de la presente sección información relativa a las características, semántica, arquitectura y APIs que la componen [65]. Aunque antes de entrar en detalle con ello, es necesario entender cuál es el objetivo principal de Samza; esta es la razón que hace imprescindible conocer qué son los sistemas de mensajería, pues estos están íntimamente relacionados con el cometido de Samza.

Un sistema de mensajería es una infraestructura de nivel bastante bajo que almacena mensajes y espera a que vengan consumidores. Cuando se comienza a escribir un código que produce o consume mensajes, rápidamente se aprecia que existen muchos problemas difíciles que deben resolverse en la capa de procesamiento. El principal objetivo de Samza es ayudar a resolver dichos problemas.

Samza presenta una serie de características que lo hacen ideal para la creación de aplicaciones:

- Emplea una API unificada y simple: permite describir la lógica de la aplicación de manera independiente a la fuente de datos. La misma API puede procesar datos por lotes y por transmisión.
- Funciona como librería integrada: puede ser usada como una librería de aplicaciones Java o Scala.
- Administra estados: es capaz de administrar y restaurar el estado de un procesador de flujo. Samza está construido para manejar grandes cantidades de estados (muchos *gigabytes* por partición).
- Tolerancia a fallos: recurre a Apache YARN cada vez que falla una máquina en el conjunto de ordenadores. Esto le permite mover tareas entre máquinas de manera transparente.
- Durabilidad: garantiza que nunca se pierdan mensajes y que estos se procesen en el orden en que se escribieron en una partición.
- Escalabilidad: mediante YARN proporciona un entorno distribuido para que se ejecuten los núcleos de Samza.
- Conectable: aun funcionando de forma inmediata con Kafka y YARN, Samza utiliza APIs de conexión que le permiten relacionar Samza con otros sistemas de mensajería y otros entornos de ejecución. Esta es la razón por la que puede procesar y transformar datos desde cualquier fuente. Samza se integra perfectamente con ElasticSearch, Apache Hadoop o con fuentes propias.
- Aislamiento del procesador: trabaja con Apache YARN, que admite el modelo de seguridad de Hadoop, y el aislamiento de recursos a través de Linux.
- Aplicaciones ejecutables en cualquier lugar: presenta opciones de implementación flexibles, desde nubes públicas a máquinas físicas.

Tras el análisis de los aspectos más importantes de Samza y con el objetivo de definir la forma en que el flujo de datos es manejado, se va a proceder a explicar los principales conceptos de Samza, de acuerdo a lo dispuesto en [66]:

- *Topics* (temas o asuntos): son cada flujo de entrada en Kafka; es decir, son flujos de datos relacionados entre sí y de características parecidas.
- Particiones: división de los mensajes entrantes que se usa para distribuir un conjunto de datos entre los nodos Kafka. Las particiones garantizan que los mensajes que tengan igual clave vayan a la misma partición. Las particiones además garantizan ordenación.
- *Broker*: cada uno de los nodos que compone Kafka.
- Productor: cualquier componente que suministra o escribe datos a Kafka. Los productores asignan las claves que son utilizadas para las particiones.

- Consumidores: componentes que leen datos de los *topics* generados y que son responsables de mantener la información de su propio conjunto de datos, razón por la cual tienen que manejar los registros que procesen en caso de que ocurra un fallo en el sistema.
- Tarea: Samza escala su aplicación al dividirla lógicamente en múltiples tareas. Una tarea es la unidad de paralelismo para su aplicación. Cada tarea procesa datos de una partición de un flujo de entrada. La asignación de particiones a tareas nunca cambia: si una tarea está en una máquina que falla, la tarea se reinicia en otro lugar, todavía procesando las mismas particiones de flujo. (Figura 3-28)

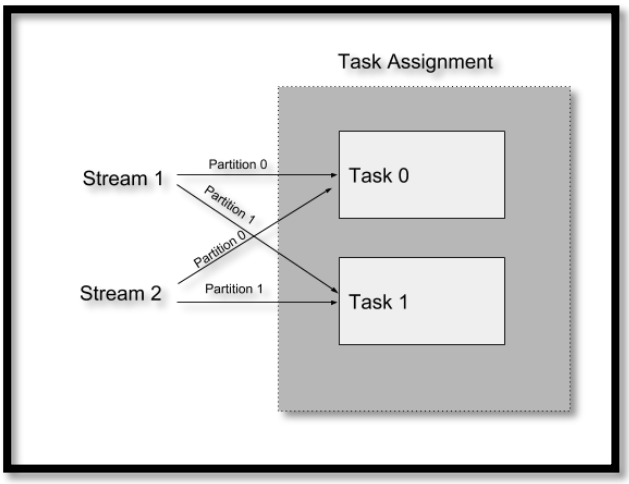

Figura 3-28 Asignación de tareas en Apache Samza [65]

- Contenedor: Al igual que una tarea es la unidad lógica de paralelismo para una aplicación, un contenedor es la unidad física. Puede pensarse que cada contenedor es como un proceso JVM que ejecuta una o más tareas. Una aplicación suele constar de múltiples contenedores distribuidos entre *hosts*.
- Coordinador: cada aplicación también tiene un coordinador que administra la asignación de tareas a través de los contenedores individuales. El coordinador supervisa la vida de los contenedores individuales y redistribuye las tareas entre los restantes cuando se produce un fallo. El coordinador en sí es conectable, lo que permite a Samza admitir múltiples opciones de implementación. Se puede usar Samza como una librería "ligera" que se integra fácilmente con una aplicación más grande. Alternativamente, puede implementarse y ejecutarse como una herramienta de procesamiento de flujos utilizando YARN como administrador del conjunto. (Figura 3-29)

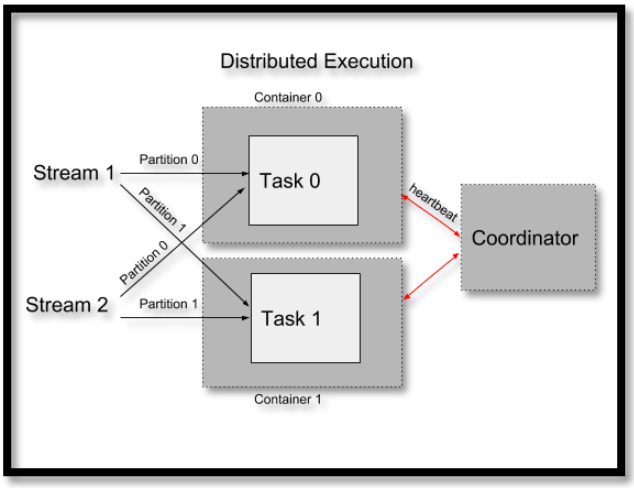

Figura 3-29 Ejecución distribuida en Apache Samza [65]

- Modelo de subprocesos y pedidos: Samza ofrece un modelo de subprocesos flexibles para ejecutar cada tarea. Al ejecutar sus aplicaciones, puede controlar el número de trabajadores necesarios para procesar los datos. También puede configurar el número de hilos que cada trabajador utiliza para ejecutar sus tareas asignadas. Cada hilo puede ejecutar una o más tareas. Las tareas no comparten ningún estado, por lo tanto, no tiene que preocuparse por la coordinación entre estos hilos.
  Otro escenario común en el procesamiento continuo es interactuar con servicios remotos o bases de datos. Por ejemplo, un sistema de notificaciones que procesa cada mensaje entrante, genera un correo electrónico e invoca a una API REST para entregarlo. Samza ofrece una API completamente asíncrona para casos de uso como este, que requieren un alto rendimiento entrada/salida. Por defecto, todos los mensajes entregados a una tarea son procesados por el mismo hilo. Esto garantiza el procesamiento de mensajes dentro de una partición. Sin embargo, algunas aplicaciones no requieren el procesamiento en orden de los mensajes. Para tales casos de uso, Samza también admite el procesamiento mensajes fuera de orden dentro de una sola partición. Esto generalmente ofrece un mayor rendimiento al permitir múltiples mensajes concurrentes en cada partición.
- Punto de control incremental: Samza garantiza que los mensajes no se perderán, incluso si su trabajo falla, si una máquina muere o si hay un fallo de red. Para lograr esta propiedad, cada tarea persiste periódicamente los últimos puntos procesados de sus particiones de flujo de entrada (Figura 3-30). Si una tarea necesita reiniciarse en un trabajador diferente debido a un fallo, se reanuda el procesamiento desde su último punto de control.

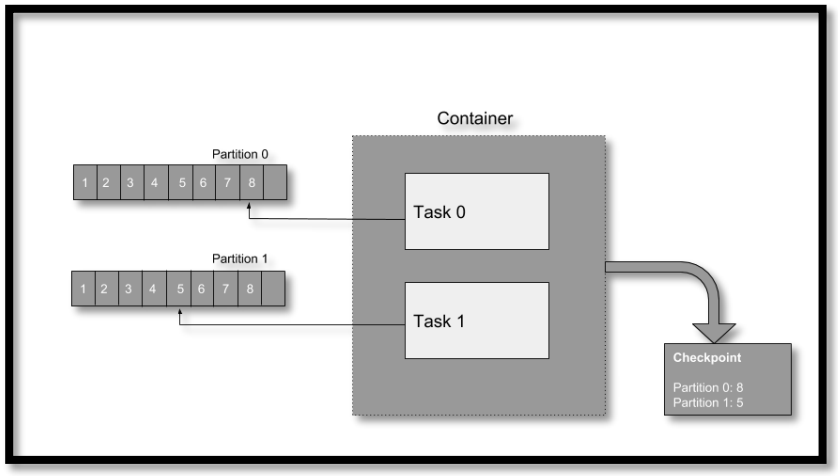

Figura 3-30 Funcionamiento de los *checkpoints* con Apache Samza [65]

- Administración del estado: Samza ofrece almacenamiento escalable y de alto rendimiento que le permite crear aplicaciones de procesamiento de flujo con estado. Esto se implementa asociando cada tarea de Samza con su propia instancia de una base de datos local (también conocido como un almacén de estados). El almacén de estado asociado con una tarea particular solo almacena datos correspondientes a las particiones procesadas por esa tarea (Figura 3-31). Cuando se escala un trabajo dándole más recursos informáticos, Samza migra de manera transparente las tareas de una máquina a otra. Al dar a cada tarea su propio estado, las tareas se pueden reubicar sin que su integridad se vea afectada. Aquí se presentan algunas ventajas clave de esta arquitectura:
  - El estado se almacena en el disco, por lo que el trabajo puede mantener más estado del que tendría en la memoria.
  - Se almacena en la misma máquina que la tarea, para evitar los problemas de rendimiento asociados a la creación de bases de datos de consultas a través de la red.

- Cada tarea tiene su propio almacén, para evitar los problemas de aislamiento en una base de datos remota compartida (si realiza un consulta costosa, afecta solo a la tarea actual, a ninguna más).
- Se pueden conectar diferentes motores de almacenamiento.

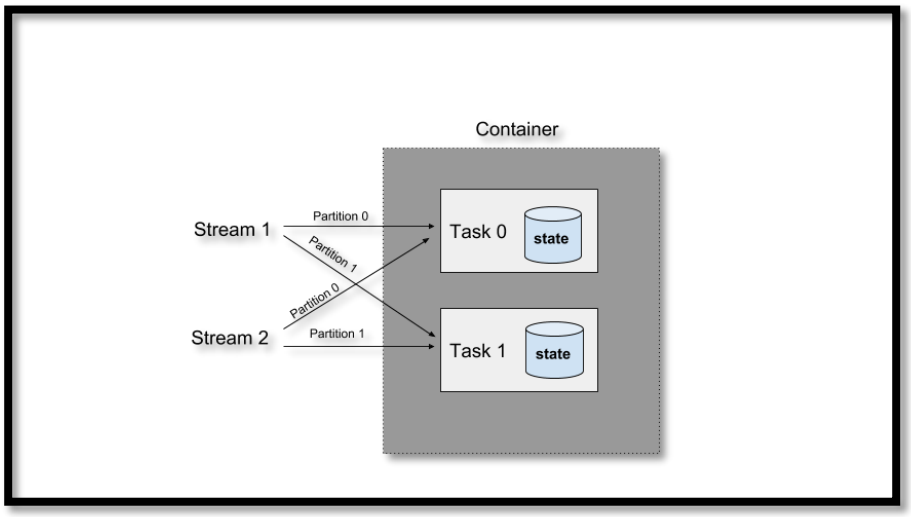

Figura 3-31 Las tareas tienen su propio estado en Apache Samza [65]

- Tolerancia a fallos del estado: los sistemas de procesamiento de flujo distribuido necesitan recuperarse rápidamente de los fallos para reanudar su procesamiento. Si bien tener un almacén local duradero ofrece un gran rendimiento, aún se debe garantizar la tolerancia a fallos. Con este propósito, Samza replica cada cambio en el almacén local (denominado registro de cambios para el almacén). Esto le permite recuperar más tarde los datos del almacén leyendo los contenidos del registro de cambios desde el principio.
- *Host*: si la aplicación tiene varios *terabytes* de estado, ponerla en marcha cada vez mediante la lectura del registro de cambios detendrá el progreso. Por lo tanto, es fundamental poder recuperar el estado rápidamente durante los fallos. Con este propósito, Samza tiene en cuenta la localización de datos cuando se programan las tareas en *hosts*. Esto es implementado por metadatos persistentes sobre el *host* en el que se están ejecutando.
  Durante una nueva implementación de la aplicación, Samza intenta reprogramar las tareas en los mismos *hosts* que se usaron previamente. Esto permite que la tarea reutilice la instantánea del estado local de su ejecución anterior en ese *host*. Llamamos a esta característica *host-affinity*, ya que trata de preservar la asignación de tareas a los *hosts*. Este es un diferenciador clave que permite a las aplicaciones de Samza escalar varios *terabytes* de estado local con cero tiempo de inactividad.

Una aplicación procesa flujos de entrada, los transforma y emite resultados a un flujo de salida o a una base de datos, como muestra la Figura 3-32. Se construye encadenando múltiples operadores, cada uno de los cuales abarca uno o más flujos y los transforman.

Respecto a las APIs, Samza ofrece cuatro de nivel superior para ayudar a construir aplicaciones en *streaming*. Según [65] son:

- API *Stream* de alto nivel: ofrece varios operadores integrados como *map*, *filter*, etc. Esta es la API recomendada para la mayoría de los casos de uso.
- API de tareas de bajo nivel: permite una mayor flexibilidad para definir la lógica de procesamiento y ofrece mayor control.
- Samza SQL: ofrece un interfaz de SQL para crear aplicaciones.

- API Apache Beam: ofrece la API para Java, mientras que la de Python y Go están actualmente en proceso de desarrollo.

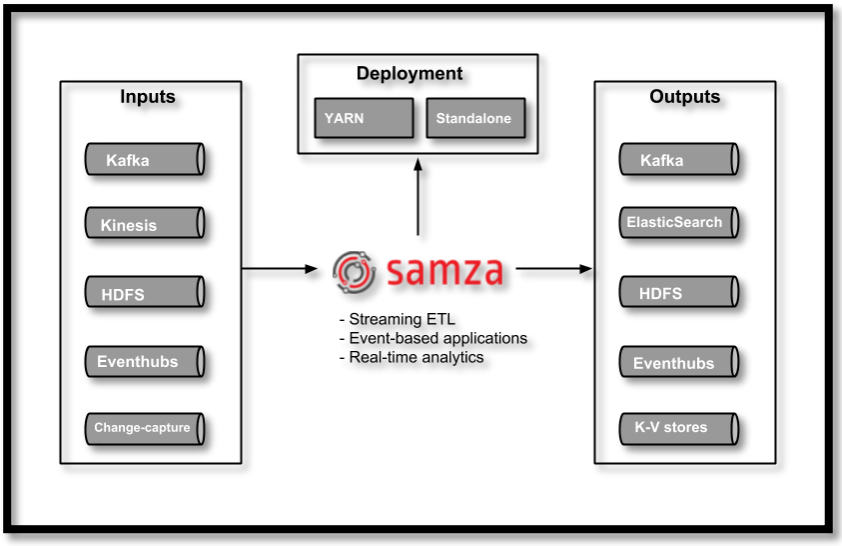

Figura 3-32 Samza transforma flujos de entrada y emite resultados a una base de datos

Como ha podido observarse a lo largo de toda la descripción de Apache Samza, existe un concepto vital (el concepto de estado). Samza admite el procesamiento de flujo sin estado y con estado.

En el procesamiento sin estado, como el nombre indica, el mensaje no tiene ningún estado asociado después de haber sido procesado. Un buen ejemplo de esto es filtrar un flujo entrante de registros de usuario por un campo (por ejemplo: identidad de usuario) y escribir los mensajes filtrados en su propia secuencia.

Por el contrario, el procesamiento con estado requiere el registro de algún estado sobre un mensaje incluso después del procesamiento. Considerese el ejemplo de contar el número de usuarios únicos de un sitio web cada cinco minutos. Esto requiere que se almacene información sobre cada usuario visto hasta ese momento. Samza ofrece tolerancia a fallos; el almacén de estado escalable está para este propósito.

De igual manera, el tiempo es otro concepto fundamental en el procesamiento de flujos, especialmente en cómo es modelado e interpretado por el sistema. Samza admite dos nociones de tiempo: de procesamiento y de evento.

Por defecto, todos los operadores Samza incorporados usan tiempo de procesamiento. En el tiempo de procesamiento, la marca de tiempo de un mensaje está determinada por cuándo es procesado por el sistema. Por ejemplo, un evento generado por un sensor podría ser procesado por Samza varios milisegundos más tarde.

Por otro lado, en el tiempo de evento, la marca de tiempo de un evento está determinada por cuándo ocurrió realmente en la fuente. Por ejemplo, un sensor que genera un evento podría incorporar el tiempo de ocurrencia como parte del evento en sí. Samza proporciona procesamiento basado en tiempo de evento mediante su integración con Apache Beam.

Para concluir con la descripción del funcionamiento, resulta imprescindible hacer referencia a la garantía de procesamiento. Samza ofrece garantías de procesamiento de al menos una ocasión. Esto asegura que cada mensaje del flujo de entrada se procesa en el sistema al menos una vez. Samza es, por tanto, una opción práctica para construir aplicaciones tolerantes a fallos.

3.1.4.4 **Escenarios de uso**

Como se puede observar, Samza es usado por algunas de las compañías más importantes a nivel mundial, lo que evidencia la utilidad de esta herramienta en el análisis de flujos. En el Anexo I pueden visualizarse varios de sus clientes.

Con el objetivo de medir la popularidad de la herramienta se han usado los mismos métodos utilizados en el resto de las herramientas previamente presentadas. De este análisis se han obtenido los siguientes resultados:

- Resultados en Google de "Apache Samza": 56,400
- Seguidores en Twitter de Apache Samza (@samzastream): 1830
- Fecha de búsqueda: 06 de marzo de 2020

Netflix [67] utiliza tareas de Samza de una sola etapa para enrutar más de 700 mil millones de eventos (1 *petabyte*) por día desde conjuntos de máquinas de Kafka a colmenas S3. Una parte de estos eventos se enruta a Kafka y ElasticSearch con soporte para la creación de índices personalizados, filtrado básico y proyección. Netflix ejecuta más de 10.000 tareas de Samza.

Redfin [68] es una inmobiliaria que utiliza tecnología moderna para ayudar a las personas a comprar y vender casas. La notificación es el principal medio para comunicarse con los clientes; esta incluye recomendaciones, correos electrónicos instantáneos, resúmenes programados y notificaciones *push*. Miles de correos electrónicos se envían a los clientes cada minuto.

El sistema de notificación solía ser un sistema monolítico, que servía bien a la empresa. Sin embargo, a medida que el negocio creció y los requisitos evolucionaron, se hizo cada vez más difícil de mantener y escalar.

El equipo de ingeniería de Redfin decidió reemplazar el sistema existente por Samza, principalmente por el rendimiento, la escalabilidad, el soporte que les ofrecía para el procesamiento con estado y la integración de Kafka. Para ello, se desarrolló una canalización de procesamiento de flujo de etapas múltiples (Figura 3-33). En la etapa de identificación, los eventos externos como, por ejemplo, los nuevos listados, se identifican como candidatos para enviar una nueva notificación; luego, los destinatarios potenciales de las notificaciones se determinan analizando los datos en los eventos y los perfiles de los clientes. Los resultados se agrupan por cliente al final de cada ventana de tiempo durante la etapa de coincidencia. Una vez que se identifican las notificaciones y los destinatarios, la etapa de organización los une con fuentes de datos adicionales (por ejemplo: configuración de notificaciones o perfiles de clientes) aprovechando el apoyo de Samza para el estado local. De igual manera, Samza hace un uso intensivo de RocksDB para almacenar y combinar notificaciones individuales antes de enviarlas a los clientes. Finalmente, las notificaciones se formatean en la etapa de formato y se envían al sistema de entrega en la etapa de notificación.

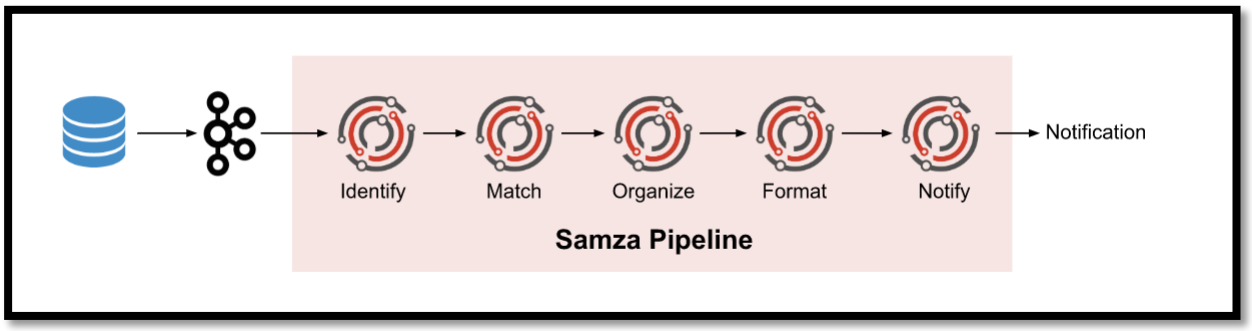

Figura 3-33 Redfin usa Samza para procesamiento de flujo en etapas múltiples [69]

Con el nuevo sistema de notificación basado en Apache Samza, Redfin observó que:

- Resultaba más sencillo agregar soporte para los nuevos casos de uso.
- El nuevo sistema era más eficiente y escalable horizontalmente.
- Se redujo la presión sobre los servicios posteriores debido al uso del almacén de estado local RocksDB.
- Las etapas de procesamiento se podían escalar individualmente ya que estaban aisladas.

Además de para la plataforma de notificaciones, otros equipos de ingeniería en Redfin también usan Samza para calcular métricas comerciales, procesar documentos, programar eventos, etc.

3.1.5 Apache Flume

3.1.5.1 **¿Qué es Apache Flume?**

Apache Flume [69] es un servicio distribuido que mueve de manera fiable y eficiente grandes cantidades de datos, especialmente *logs*. Es ideal para aplicaciones de analíticas en línea en entornos Hadoop. [70] Flume, que está escrito mayoritariamente en Java, permite recoger datos tanto en lotes como en modo *streaming*. El logo de Flume se muestra en la Figura 3-34.

Figura 3-34 Logo de Apache Flume [69]

La tarea principal de Apache Flume es establecer un canal para dirigir los datos desde una fuente de datos a otra. El destino usual de los datos de Apache Hadoop es HDFS, pero podrían ser otros sistemas como HBase [71], Solr [72], etc. Los ejemplos típicos de uso de Apache Flume son recolectar los ficheros de *log* de los servicios web de bancos u otras organizaciones o los tuits de la red social Twitter y almacenarlos en HDFS para su posterior procesamiento. [2]

3.1.5.2 **Origen e historia de Apache Flume**

Apache Flume fue creado por Cloudera [73], compañía que proporciona *software* basado en Apache Hadoop, soporte, servicios y formación para grandes clientes.

Cloudera revolucionó la gestión de datos empresariales al ofrecer la primera plataforma unificada para *big data*. Esta compañía oferta a las empresas un lugar donde almacenar, procesar y analizar todos sus datos, capacitándolas para ampliar el valor de las inversiones existentes. Cloudera ofrece todo lo necesario en el manejo de datos empresarial (*software*, almacenamiento, acceso, gestión, análisis, seguridad y búsqueda). Como principal educador de los profesionales de Hadoop, Cloudera ha capacitado a más de 40.000 personas en todo el mundo. [74]

La última versión de Apache Flume, publicada en enero de 2019, es la versión 1.9.0. [69]

3.1.5.3 **Funcionamiento**

Para llegar a comprender el funcionamiento de Apache Flume se ha llevado a cabo un análisis similar al realizado en las herramientas que preceden: en primer lugar se exponen los distintos elementos con

los que Flume funciona y que constituyen la base de su arquitectura; tras esto, se explican las características que distinguen a Flume y, por último, se comentan particularidades de la herramienta en aspectos relacionados con tolerancia y fiabilidad.

Flume tiene una arquitectura sencilla y flexible basada en flujos de datos en *streaming*, cuyos elementos principales (Figura 3-35) son:

- Evento: representa la unidad de datos entrante transportada por Flume.
- Flujo de datos: describe el movimiento de los eventos desde el origen al destino. Un evento puede pasar por varios agentes antes de llegar a su destino final.
- Cliente: interfaz que opera en el punto de origen de los eventos y los entrega a los agentes. Los clientes suelen operar en el espacio de proceso de la aplicación de la que están consumiendo datos. Log4j Appender es un ejemplo de cliente.
- Agente: máquina virtual de Java en la que Flume se ejecuta. Es un proceso independiente que se encarga de recibir, guardar y enviar eventos. Está a su vez compuesto por:
    - Una fuente: interfaz que ingiere los eventos que le son enviados y los pone en uno o varios canales. Avro [75] es un ejemplo de interfaz.
    - Un canal: conducto entre una fuente y un sumidero. JDBL [76] es un ejemplo de canal.
    - Un sumidero: interfaz que permite consumir eventos de un canal y transmitirlos al siguiente agente o al destino final. Flume Hadoop Distributed File System (HDFS) [77] es un ejemplo de sumidero.

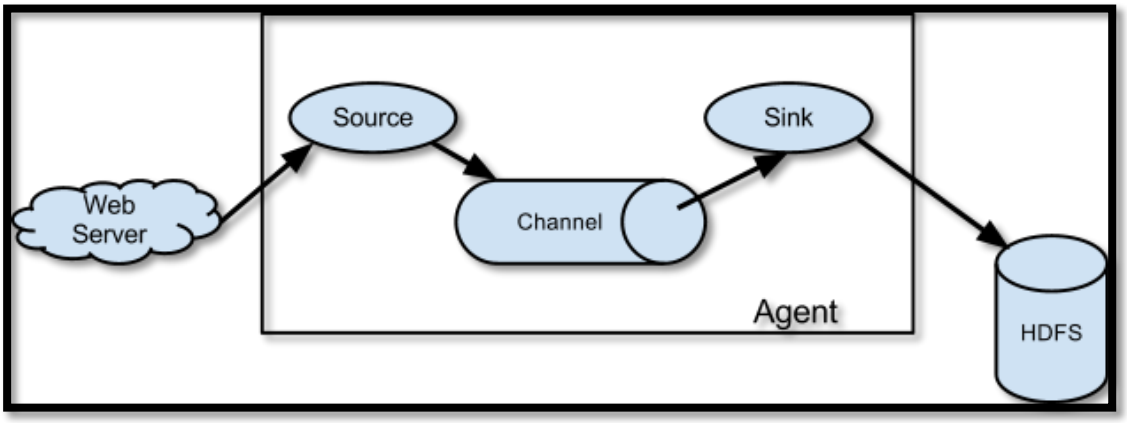

Figura 3-35 Entidades principales que componen Apache Flume [69]

Las características principales de Apache Flume son las que se describen a continuación:

- Está perfectamente integrado en el ecosistema Hadoop, ejemplo de ello es la utilización de sumideros como HDFS o HBase.
- Se especializa en datos de *logs*. Permite la recolección y agregación de *logs* de forma sencilla.
- Está pensado para eventos simples, no para eventos complejos.
- Es robusto y tolerante a fallos, con mecanismos de fiabilidad configurables y de conmutación por error y recuperación.
- Permite la lectura y escritura desde multitud de fuentes de datos.
- Simplifica el filtrado y transformación de datos gracias a los interceptores.
- Dificulta el escalado horizontal. El añadir consumidores implica cambiar la topología de la tubería y añadir un nuevo destino.

- Soporta canales efímeros basados en memoria y canales duraderos basados en ficheros, por lo que no trabaja eficientemente con mensajes de larga duración puesto que los tiene que recuperar del agente.
- Los canales basados en ficheros no replican datos, por lo que están sujetos a errores de disco, para solventarlo se suelen emplear *Storage Area Networks* (SANs).

Flume permite al usuario construir flujos complejos (de múltiples saltos) donde los eventos viajan a través de múltiples agentes antes de llegar al destino final. También permite flujos de agregación y de difusión, enrutamiento contextual y rutas de respaldo (conmutación por error) para saltos fallidos.

Los eventos, organizados en un canal por cada agente, pasan posteriormente a entregarse al repositorio del siguiente agente o a un sumidero de flujo (como HDFS). Por tanto, los eventos se eliminan de un canal solo después de que se almacenen en el canal del siguiente agente o sumidero. De esta manera se proporciona confiabilidad de extremo a extremo en el flujo. Flume utiliza un enfoque transaccional para garantizar la entrega confiable de los eventos.

Los eventos se organizan en el canal, que gestiona la recuperación frente a fallos. Flume admite un canal de archivos duradero respaldado por el sistema de archivos local. También hay un canal de memoria que simplemente almacena los eventos en una cola en memoria, que es más rápida; sin embargo, los eventos que aún quedan en el canal de memoria cuando el proceso de un agente falla no se pueden recuperar.

También existe una integración de Flume con Kafka, denominada Flafka. [78] Esta funcionalidad amplía la capacidad de Flume y le permite explotar todas sus características.

3.1.5.4 **Escenarios de uso**

Los resultados de popularidad de Flume demuestran que no es una de las herramientas más populares en el análisis de flujos de datos, como puede observarse a continuación:

- Resultados en Google de “Apache Flume”: 246.000
- Apache Flume no dispone de cuenta de Twitter
- Fecha de búsqueda: 03 de marzo de 2020

En este caso de uso real [74] no se ha encontrado información relativa al número de máquinas ni a la cantidad ni calidad de los datos procesados. Apache Flume tiene un cometido más parecido a Apache Kafka que al resto de herramientas de análisis de flujos analizadas anteriormente. Es por ello que el ejemplo que se expone se centra más en la utilidad de Cloudera y en las herramientas que nos puede ofrecer (entre estas se encontraría Flume), que en Apache Flume en sí misma. En la Figura 3-36 se muestran el conjunto de herramientas empleadas por Cloudera y su respectivo impacto en CounterTack.

CounterTack [79], con sede en Massachusetts, es una empresa que aprovecha el análisis de *big data* de Cloudera para proteger a las organizaciones de los ataques de ciberdelincuentes que intentan obtener acceso a terminales finales: estaciones de trabajo, computadoras portátiles, teléfonos inteligentes, tabletas, etc. CounterTack analiza la información recopilada por todos los dispositivos de una empresa a nivel de sistema. Esto le permite identificar patrones o anomalías que puedan estar asociadas con comportamientos maliciosos. Entre los clientes de CounterTack se encuentran Alcatel-Lucent, Siemens, Cisco, Zurich y Deloitte.

El número de dispositivos conectados a la infraestructura de red de la empresa crecía a un ritmo inmenso. La revolución móvil, alimentada por la proliferación de teléfonos inteligentes, tabletas y portátiles, suponía un quebradero de cabeza para los equipos encargados de mantener la política y la seguridad de los terminales finales corporativos. Además, con el Internet de las cosas (IoT) y el crecimiento de dispositivos en red en todo en nuestros hogares y oficinas, los piratas informáticos tenían aún más oportunidades de ataque. Ante esta tendencia, el equipo de CounterTack, que utilizaba Apache Hadoop, decidió recurrir a Cloudera.

Desde entonces, la plataforma líder de la industria de CounterTack, denominada Sentinel, utiliza datos empresariales de Cloudera *hub*, construido en Apache Hadoop, para crear un sistema único y masivamente escalable. En él los datos de los terminales finales se recopilan y analizan para proporcionar a los equipos de seguridad la información necesaria para identificar y mitigar las amenazas.

Apache HBase, que es una base de datos distribuida estándar abierta, se utiliza como repositorio para datos de los terminales finales y aquí es donde entra en juego Apache Flume, que se emplea para mover grandes volúmenes de datos desde los terminales finales. Así mismo, ClouderaSearch (que se encuentra totalmente integrado con el estándar abierto para búsqueda empresarial Apache Solr) se implementa para proporcionar búsqueda interactiva e indexación flexible y escalable.

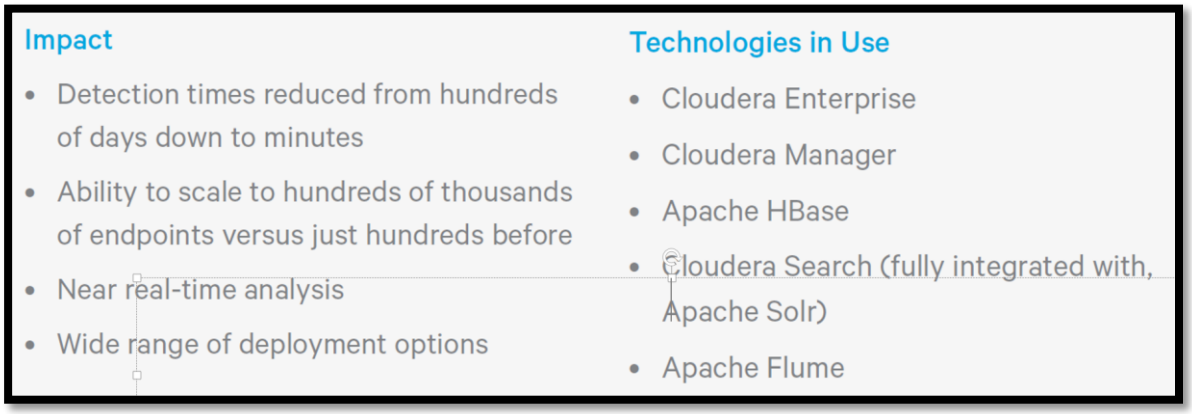

Figura 3-36 Tecnologías implementadas en CounterTack y su impacto en la detección de anomalías. [74]

La integración de las herramientas de Cloudera (entre las que se encuentra Apache Flume) ha conseguido que CounterTack pueda encontrar fragmentos de códigos maliciosos con una rapidez increíblemente alta, de pocos minutos, mucho menor que la de la industria común que tarda una media de 296 días en localizarlos. En la Figura 3-36 se muestran las diferentes tecnologías implementadas en CounterTack y su impacto en la detección de anomalías.

3.1.6 Amazon Kinesis Data Analytics (comercial)

Además de existir herramientas de *software* libre para el procesamiento de flujos de datos, hay otras comerciales. Una de las más conocida es Amazon Kinesis. A continuación se comentarán los aspectos más característicos de esta herramienta, pero de una manera superficial, sin entrar en tanto detalle, ya que no participará en la posterior comparación puesto que, al tratarse de una herramienta comercial, se descarta su uso.

Esta herramienta no conlleva el aprovisionamiento de recursos ni costes iniciales. Amazon cobra una tarifa por hora en función de la cantidad promedio de unidades de procesamiento de Kinesis (o KPU) que se utilicen para ejecutar aplicaciones de procesamiento de transmisiones. Una sola KPU es una unidad de capacidad de procesamiento de transmisiones compuesta de 1 vCPU y 4 GB de memoria. [80]

Para las aplicaciones Java, se paga una KPU adicional única por aplicación para la organización de aplicaciones. En las aplicaciones Java también paga por el almacenamiento de aplicaciones en ejecución y copias de seguridad de aplicaciones duraderas. El almacenamiento de aplicaciones en ejecución se utiliza para capacidades de procesamiento con estado en Amazon Kinesis Data Analytics, y se paga por GB/mes. Las copias de seguridad de aplicaciones duraderas son opcionales, se pagan por GB/mes y proporcionan un punto de recuperación a un momento dado para las aplicaciones. La Tabla 2 muestra los costes asociados a la utilización de Amazon Kinesis Data Analytics. [80]

Unidad de procesamiento de Kinesis, por hora	0,127 USD por hora
Almacenamiento de aplicaciones en ejecución, por GB-mes	0,116 USD por GB-mes
Copias de seguridad de aplicaciones duraderas, por GB-mes	0,024 USD por GB-mes

Tabla 2 Precios de Amazon Kinesis Data Analytics [80]

3.1.6.1 ¿Qué es Amazon Kinesis Data Analytics?

Según lo dispuesto en [81]: "Amazon Kinesis Data Analytics es la manera más sencilla de analizar datos de *streaming*, obtener información procesable y responder a las necesidades de los clientes y su negocio en tiempo real. Amazon Kinesis Data Analytics reduce la complejidad de desarrollar, administrar e integrar las aplicaciones de *streaming* con otros servicios de AWS. Los usuarios de SQL pueden consultar fácilmente los datos de *streaming* o crear aplicaciones de *streaming* completas utilizando plantillas y un editor de SQL interactivo. Los desarrolladores de Java pueden crear rápidamente aplicaciones de *streaming* sofisticadas utilizando bibliotecas de código abierto de Java e integraciones de AWS para transformar y analizar datos en tiempo real."

Amazon Kinesis Data Analytics realiza lo necesario para ejecutar aplicaciones en tiempo real de forma continua y ajusta la escala automáticamente para adaptarse al volumen y rendimiento de los datos de entrada. Con esta herramienta solo se paga por los recursos que se consumen en las aplicaciones de *streaming*. No tiene tarifa mínima ni coste de contratación.

Los beneficios que aporta son: [81]

- Procesamiento eficiente en tiempo real: ofrece funciones integradas para filtrar, agregar y transformar datos de *streaming* para realizar análisis avanzados. Procesa los datos de *streaming* con latencias de menos de un segundo (en tiempo real).
- No es necesario administrar servidores: ejecuta las aplicaciones de *streaming* sin que se tenga que aprovisionar o administrar la infraestructura. Amazon Kinesis Data Analytics presenta una infraestructura "elástica", es decir, se puede incrementar o reducir para ajustarla a la demanda.
- Se paga únicamente por lo que se utiliza: solo se paga por los recursos de procesamiento que se utilizan en la aplicación de *streaming*. No se requieren tarifas mínimas ni compromisos iniciales.

Amazon Kinesis Data Analytics permite crear consultas y aplicaciones sofisticadas de *streaming* de manera fácil y rápida en tres pasos: configurando los orígenes de los datos de *streaming*, escribiendo las consultas o aplicaciones de *streaming* y configurando el destino de los datos procesados. Así mismo, se encarga de ejecutar consultas y aplicaciones continuamente en los datos mientras se encuentran en tránsito, y envía los resultados a los destinos.

También proporciona plantillas y un editor interactivo con el que se permite crear consultas SQL que realizan combinaciones y agregaciones de ventanas, filtros, etc.

Por otro lado, incluye bibliotecas de código abierto basadas en Apache Flink con las que se permite crear una aplicación en horas en lugar de meses. Las bibliotecas ampliables incluyen más de 25 operadores preintegrados para filtrar, agregar y transformar datos de *streaming* e integraciones con los servicios de AWS como Amazon Managed Streaming for Apache Kafka (Amazon MSK), Amazon Kinesis Data Streams, Amazon Kinesis Data Firehose, Amazon Elasticsearch Service, Amazon S3 y Amazon DynamoDB.

3.1.6.2 Escenarios de uso

Amazon Kinesis Data Analytics es ideal para resolver una amplia variedad de casos de uso de datos de *streaming*, que según [81] incluyen:

- *Streaming* de ETL para el Internet de las cosas (IoT) con aplicaciones Java (Figura 3-37): puede escribir aplicaciones Java y usar Amazon Kinesis Data Analytics para transformar, acumular y filtrar datos de dispositivos compatibles con IoT.

Figura 3-37 *Streaming* de ETL para el Internet de las cosas (IoT) con aplicaciones Java [81]

- Análisis de registros en tiempo real con SQL (Figura 3-38): puede hacer *streaming* de miles de millones de pequeños mensajes a Amazon Kinesis Data Analytics y calcular las métricas clave que, posteriormente, pueden utilizarse para actualizar los paneles de rendimiento del contenido en tiempo real y mejorar el rendimiento del contenido.

Figura 3-38 Análisis de registros en tiempo real con SQL [81]

- Tecnología publicitaria y marketing digital con SQL (Figura 3-39): puede recopilar diferentes tipos de registros de datos de sistemas de seguimiento del público, oyentes/oferentes en intercambio de anuncios y servidores publicitarios, y combinarlos en el mismo flujo.

Figura 3-39 Tecnología publicitaria y marketing digital con SQL [81]

3.2 Comparación de las herramientas

La elección de la herramienta se realizará principalmente en base a los estudios de investigación [82] y [83]. El primero resulta de gran utilidad puesto que propone un método de calidad para elegir la arquitectura de procesamiento de flujos adecuada; el segundo, porque compara los diferentes marcos en igualdad de condiciones (mediante un ejemplo).

Como hemos visto a lo largo del capítulo 3, existen varios marcos de procesamiento de flujo con muy diferentes características. Ahora bien, es inevitable hacerse la siguiente pregunta: ¿Qué criterios deben usarse para elegir el marco adecuado? Claramente la respuesta es que todo depende del contexto del análisis en cuestión. Esto a su vez deriva en otra cuestión, ¿qué pautas hay que seguir para conocer lo que resulta ser más conveniente? La respuesta a este interrogante la da [82], que proporciona un método fiable y eficaz para la elección del *framework* más adecuado.

Existen ocho reglas que sirven para ilustrar las características requeridas por cualquier sistema de procesamiento de flujo de tiempo real para aplicaciones de *big data*:

- Mantener los datos en movimiento logrando una baja latencia.
- Consultar usando lenguajes de alto nivel como SQL en *streams* (StreamSQL).
- Manejar las imperfecciones del flujo (retrasos, pérdidas y datos fuera de orden).
- Generar resultados predecibles.
- Integrar datos almacenados y de transmisión.
- Garantizar la seguridad y la disponibilidad de los datos.
- Particionar y escalar aplicaciones automáticamente.
- Procesar y responder instantáneamente.

A continuación se resumen brevemente los aspectos más importantes de las distintas herramientas analizadas (Tabla 3). De las cinco, comentar que desde un principio se ha descartado Apache Flume por dos razones. La primera de ellas es la escasa popularidad de la que goza; es, junto a Apache Samza, la menos conocida y utilizada entre las analizadas. La segunda razón es el aparente estado de inactividad en el que se encuentra sumida esta herramienta (la información a disposición del usuario en la red está desactualizada y es escasa en muchos aspectos, como puede observarse en [84], que es un blog referenciado en [69]).

	Spark	**Storm**	**Flink**	**Samza**
Formato de datos	DataStream	Tupla	*Stream* de datos	Mensaje
Fuentes de datos	HDFS, DBMS, Kafka, etc.	HDFS, DBMS, Kafka, etc.	HDFS, DBMS, Kafka, etc.	Kafka
Modelo de programación	Transformación y acción	*Spouts* y *Bolts*	Funciones de acción (mapear, agrupar, etc.)	MapReduce *Work*
Lenguajes de programación compatibles	Java, Scala, Python, R	Java, Scala, Python, R	Java	Java y Scala
Gestor del conjunto	Hadoop YARN, Apache Mesos	ZooKeeper	Hadoop YARN, Apache Mesos	Hadoop YARN
Latencia	Segundos	Milisegundos	Milisegundos	Milisegundos
Semántica	Exactamente en una ocasión	Exactamente en una ocasión	Exactamente en una ocasión	Exactamente en una ocasión
Aprendizaje automático	Spark MLlib	Compatible con la API SAMOA [85]	Flink ML	Compatible con la API SAMOA
Consultas en *streaming*	Spark SQL	No	No	Sí (API Samza SQL)
Particionamiento de datos	Sí	Sí, con Trident	No	Sí

Tabla 3 Comparación resumida de las herramientas de análisis de flujo que son objeto de estudio

Como se puede visualizar en la Tabla 3, las herramientas analizadas difieren unas de otras en multitud de aspectos. Por ello, es necesario encontrar aquellos criterios que puedan resultar más determinantes en el procesado de flujos de datos. En [82] se proporcionan una serie de pautas que ayudarán a tomar una decisión final. A continuación se procede a su descripción.

- La semántica de mensaje o garantía de mensaje, que puede darse de estas tres formas:
  - Al menos una vez: se asegura la entrega, pero pueden existir duplicados.
  - Como máximo una vez: el mensaje o no se entrega o se entrega en una sola ocasión.
  - Exactamente una vez: garantiza la entrega en una única ocasión.
- La latencia, que es el tiempo que transcurre entre la llegada de nuevos datos y su procesamiento. Está íntimamente relacionada con la recuperación (tolerancia a fallos) porque, cada vez que el sistema tiene errores, debe recuperarse lo suficientemente rápido como para que la latencia no se merme demasiado. Además, los *frameworks* pueden hacer alguna optimización en los datos (procesamiento por lotes de mensajes) para mejorar el rendimiento, pero con el coste de sacrificar la latencia.
- La tolerancia a fallos, la cual garantiza que el sistema esté altamente disponible y opere incluso después de fallos, pudiendo recuperarse de manera transparente.

- El modelo de procesamiento de datos que puede realizarse mediante dos técnicas:
    - Microlote: en lugar de procesar datos almacenados previamente, se procesan directamente datos agrupados en pequeños lotes. Spark, por ejemplo, hace uso de microlotes.
    - Tiempo real: los datos se procesan sobre la marcha como piezas individuales, por lo que no hay espera para agruparlos en un lote. Storm, por ejemplo, procesa datos en tiempo real.

Como se ha podido apreciar, la semántica de mensajes está relacionada con la tolerancia a fallos y con el modelo de procesamiento de datos; así mismo, de acuerdo a cómo se implemente la tolerancia a fallos, la latencia aumentará o disminuirá.

La metodología de comparación de *frameworks* propuesta en [82] está basada en el metaanálisis de varios trabajos sobre el procesamiento continuo. Con el objetivo de hallar qué herramientas son punteras en este tipo de análisis sus autores recurrieron a la redundancia como indicador. Esto les llevó a crear una tabla en la que se presentan documentos consultados frente a *frameworks* populares en el procesamiento de datos, de tal manera que cada vez que un artículo cita a una de las herramientas, lo puntuan positivamente. Tras esto, repitieron exactamente el mismo proceso para seleccionar los criterios válidos para construir un método correcto. Los resultados se presentan en la Tabla 4 y la Tabla 5.

Marco/ Proyecto	An Evaluation of Data Stream Processing for Data Driven Applications	Survey of Big Data Frameweorks for Different Application Characteristics	A Comparative Study on Streaming Frameworks for Big Data	On Big Data Stream Processing	A New Architecture for Real Time Data Stream Processing	A Stream Processing Framework for High Performanc e Computing	Real Time Data Processing Frameworks	Survey of Distributed Stream Processing
Storm	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí
Samza	Sí	Sí	Sí	Sí	No	No	No	Sí
Spark Streaming	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí
Flink	No	Sí	Sí	No	No	Sí	No	Sí

Tabla 4 Herramientas de procesamiento de flujos analizadas en cada estudio de investigación [82]

Marco/ Proyecto	An Evaluation of Data Stream Processing for Data Driven Applications	Survey of Big Data Frameweorks for Different Application Characteristics	A Comparative Study on Streaming Frameworks for Big Data	On Big Data Stream Processing	A New Architecture for Real Time Data Stream Processing	A Stream Processing Framework for High Performance Computing	Real Time Data Processing Frameworks	Survey of Distributed Stream Processing
Latencia	Sí	Sí	No	Sí	Sí	Sí	Sí	Sí
Tolerancia a fallos	Sí	Sí	Sí	Sí	Sí	Sí	Sí	No
Garantías de mensaje	Sí	Sí	No	Sí	No	No	Sí	Sí
Modelo de datos (por lotes o en tiempo real)	Sí	No	Sí	Sí	Sí	Sí	Sí	Sí
Almacenamiento de datos	No	No	Sí	Sí	Sí	Sí	No	No
Colas de *streaming*	Sí	Sí	Sí	No	Sí	Sí	Sí	No
Fuentes de datos	No	Sí	Sí	No	No	No	No	Sí

Tabla 5 Criterios tratados en cada uno de los estudios de investigación [82]

A continuación se recogen los distintos criterios finalmente seleccionados:

1. Modelo de datos.

2. Tolerancia a fallos.

3. Semántica de mensajes.

4. Latencia.

Tras encontrar los criterios, el siguiente esfuerzo se centró en priorizarlos.

La primera decisión que se tiene que tomar consiste en elegir el tipo de modelo de datos. La razón principal es que esta tendrá un impacto considerable en el resto de criterios.

Aun sabiendo que la latencia es de gran importancia, un marco debería poder recuperarse lo suficientemente rápido, por lo que esto no afectaría demasiado al sistema (siempre que procesara dentro de un tiempo mínimo). También cabe destacar que antes de proporcionar semántica de mensajes, la herramienta debe recuperarse de los fallos automáticamente, pues esto influye en los otros criterios que se encuentran por debajo. Esta es la razón por la que se posiciona a la tolerancia a fallos en la segunda posición.

En tercer lugar, se ha situado la semántica de mensaje, pues según esta sea de exactamente una ocasión o de al menos una vez, la latencia cambiará.

En base a lo comentado anteriormente, se presenta el árbol del modelo de decisión (Figura 3-40) para evaluar y elegir un marco de procesamiento de flujo.

Para elegir la herramienta más adecuada, en [83] se proporciona información acerca del desempeño de distintos *frameworks* en lo relativo a tasas de procesamiento y consumo de CPU, memoria RAM y ancho de banda.

El experimento se realizó en un conjunto de ordenadores real llamado Galáctica. La agrupación estaba compuesta por 10 máquinas que funcionaban con Linux Ubuntu 16.04. Cada máquina estaba equipada con 4 CPU, 8 GB de memoria principal y 500 GB de almacenamiento local. Al ser un estudio realizado en 2018 las pruebas fueron realizadas con versiones más antiguas que las que existen actualmente, concretamente fueron Flink 1.3.2, Spark 1.6.0, Samza 0.10.3 y Storm 1.1.1. No obstante, la información sirve como complemento valioso para la toma de la decisión definitiva.

Todos los marcos estudiados tenían implementado YARN como administrador de conjunto. Para el protocolo experimental, utilizaron la API Twitter4J para la transmisión en tiempo real de tuits que contenían los términos *big data.* Cada tuit consistía en un archivo JSON con un conjunto de atributos como la fecha de creación del tuit, el identificador y la información del usuario. El protocolo experimental usado consistía en ejecutar una rutina de extracción, transformación y carga (ETL) que se encargaba de extraer tuits usando Kafka para asegurar la misma velocidad de transmisión al evaluar los marcos estudiados; de transformar los tuits manteniendo solo atributos como el identificador, el contenido, la fecha, las coordenadas y la información del usuario, y de cargar los tuits transformados en ElasticSearch.

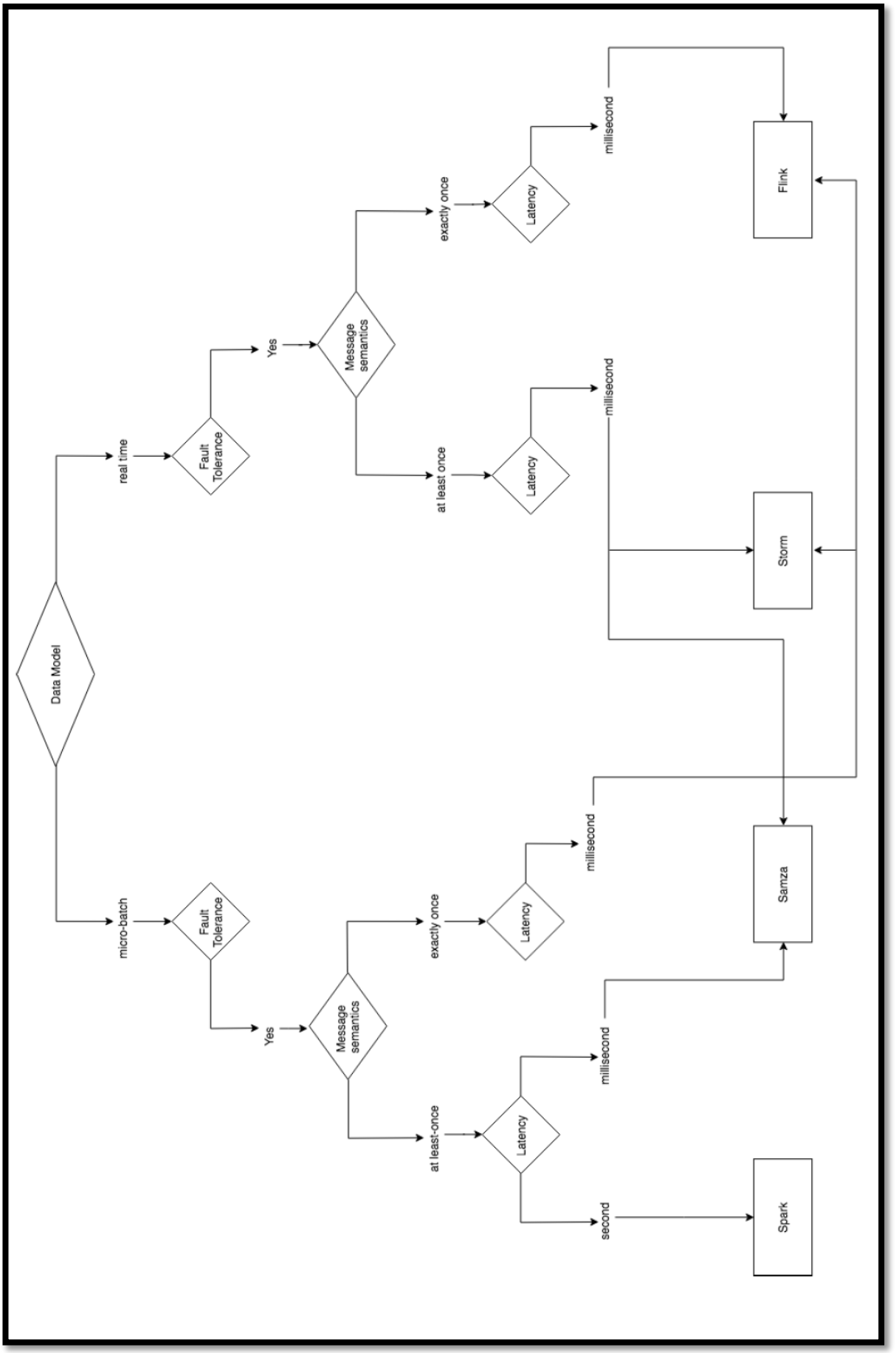

Figura 3-40 Árbol del modelo de decisión [82]

En las siguientes gráficas se presentan los resultados obtenidos:

La Figura 3-41 muestra como Flink, Samza y Storm tienen mejores tasas de procesamiento en comparación con Spark. La latencia de Spark está en el orden de los segundos mientras que la de Flink, Samza y Storm es del orden de los milisegundos.

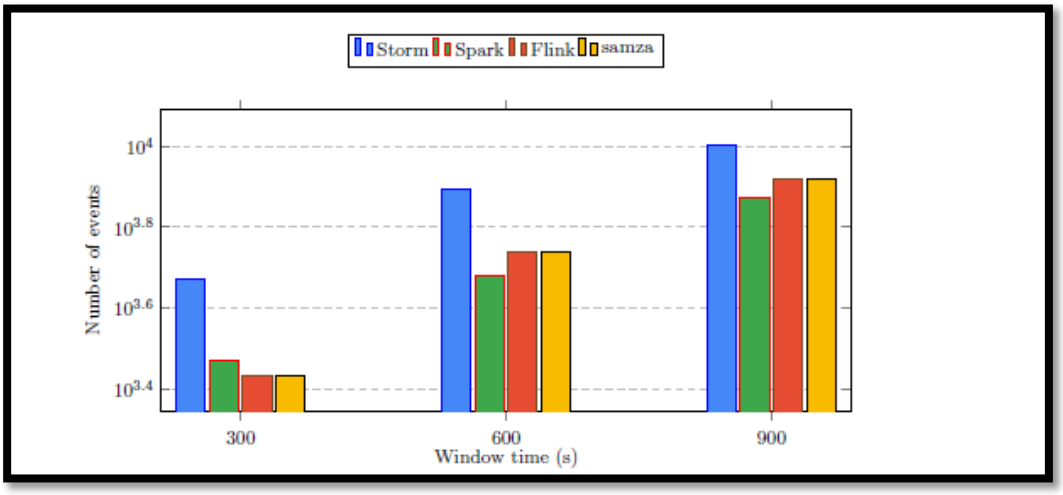

Figura 3-41 Impacto de la ventana de tiempo en el número de eventos procesados (100 KB por mensaje)

Como se muestra en la Figura 3-42, el consumo de CPU de Flink es bajo en comparación con Spark, Samza y Storm. Flink explota aproximadamente el 10% de la CPU disponible, mientras que el uso de CPU de Storm varía entre 15% y 18%. Samza consume alrededor del 55% de la CPU disponible. Flink está diseñado para procesar grandes mensajes, a diferencia de Storm, que funciona mejor manejando mensajes pequeños (por ejemplo, mensajes procedentes de sensores). A diferencia de Flink, Samza y Storm, Spark recopila datos de eventos cada segundo y realiza la tarea de procesamiento después de eso. Por lo tanto, se procesa más de un mensaje, lo que explica el alto uso de CPU de Spark. Debido a la naturaleza de la canalización de Flink, cada mensaje está asociado a un hilo y se consume en cada ventana. Por consiguiente, este bajo volumen de datos procesados no afecta el uso de recursos de la CPU. Samza consume alrededor del 55% de la CPU disponible porque se basa en el concepto de núcleos virtuales y cada trabajo o partición se asigna a varios núcleos virtuales. De hecho, despliega varios subprocesos (uno para cada partición), lo que explica su uso intensivo de la CPU en comparación a los otros marcos.

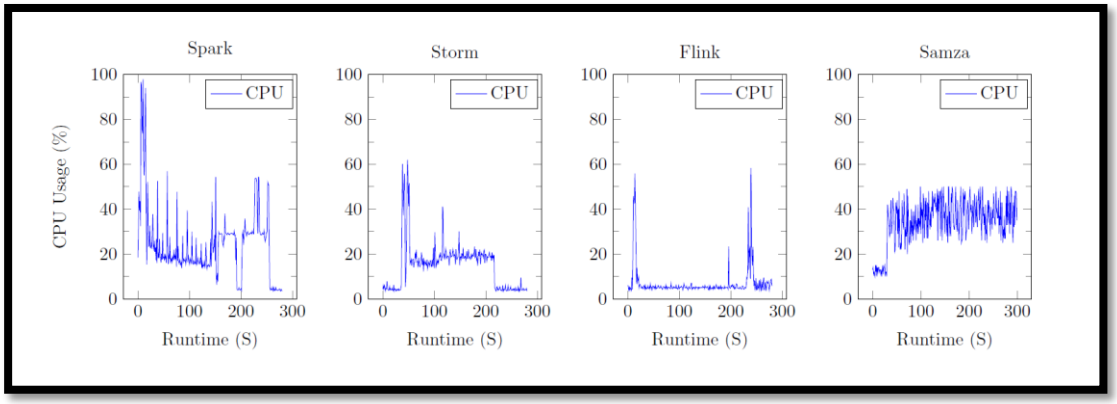

Figura 3-42 Comsumo de CPU [83]

La Figura 3-43 muestra el coste del procesamiento de flujo de eventos en términos de consumo de RAM. Spark alcanzó 6 GB (75% de los recursos disponibles) debido a su comportamiento en memoria

y su capacidad de procesar en microlotes (procesar un grupo de mensajes a la vez). Flink, Samza y Storm no superaron los 5 GB (alrededor del 61% de la RAM disponible) ya que procesan solo mensajes únicos.

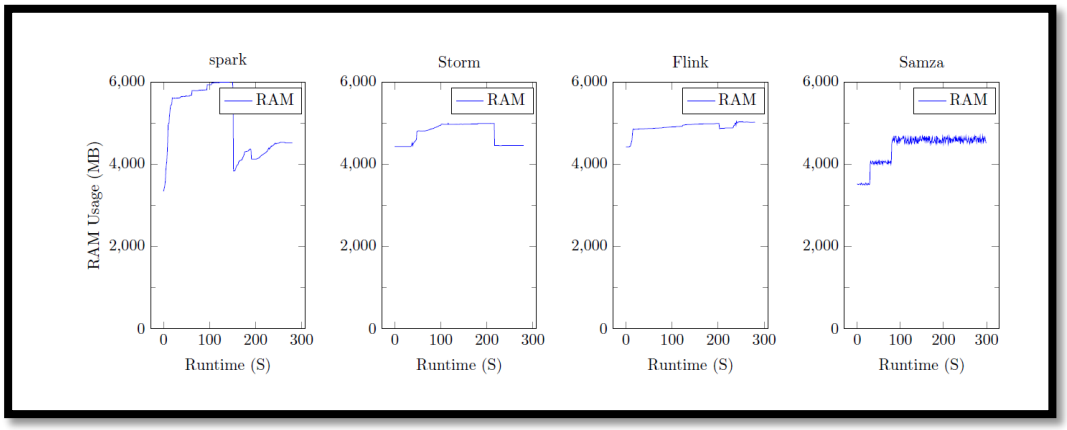

Figura 3-43 Consumo de memoria RAM [83]

La Figura 3-44 muestra el uso de disco por parte de los *frameworks* estudiados. Las curvas denotan la cantidad de operaciones de lectura / escritura. Las cantidades de operaciones de escritura en Flink y Storm son parecidas. Por otro lado, en tiempos grandes Spark tiene unos accesos similares a las otras.

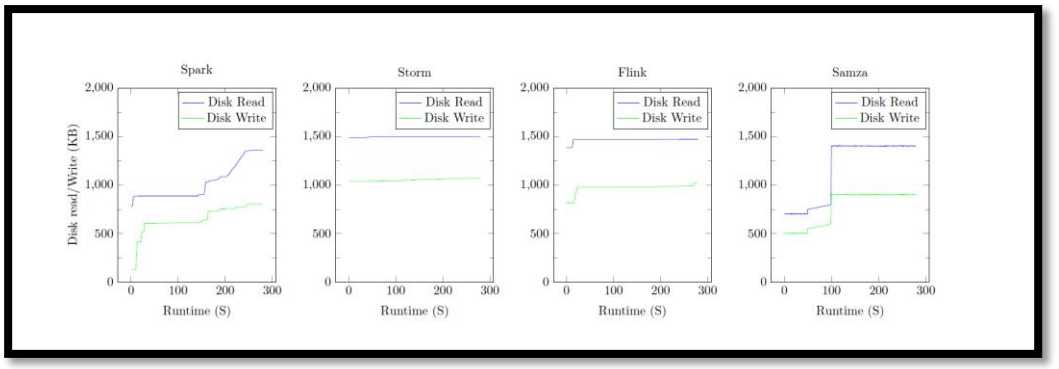

Figura 3-44 Consumo de memoria en disco [83]

Como se muestra en la Figura 3-45, la cantidad de datos intercambiados por segundo varía entre 375 KB/s y 385 KB/s en el caso de Flink, y varía entre 387 KB/s y 390 KB/s en caso de Storm. Es de unos 400 KB/s en el caso de Samza. Esta cantidad es alta comparado con Spark ya que su uso de ancho de banda no supera los 220 KB/s. Esto se debe a la reducción de la frecuencia de las operaciones de serialización y migración entre los nodos del clúster, ya que Spark procesa un grupo de mensajes en cada operación. En consecuencia, la cantidad de datos intercambiados se reduce, mientras que Storm, Samza y Flink están diseñados para el procesamiento por transmisión.

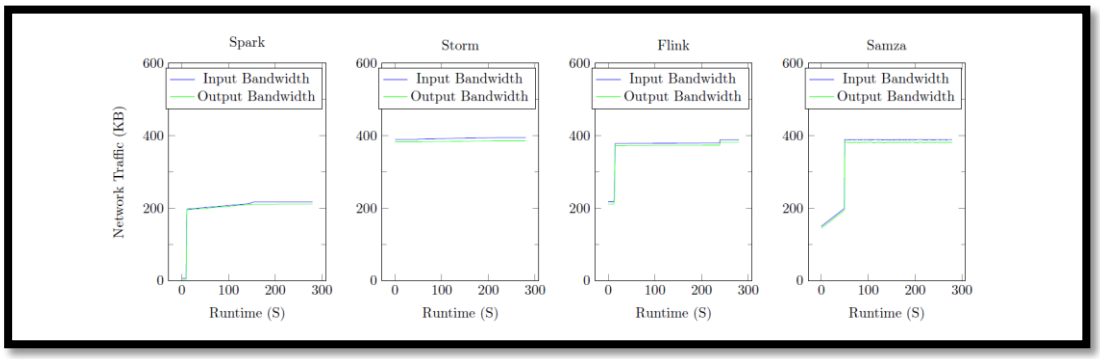

Figura 3-45 Consumo de ancho de banda [83]

Realmente cualquiera de las herramientas analizadas ofrecería un servicio de calidad en el procesamiento de datos que se va a realizar con posterioridad y, por tanto, podrían emplearse sin ningún problema. Sin embargo, el objetivo del trabajo es únicamente utilizar una de ellas. Acto seguido se dan detalles de la herramienta que será empleada en nuestro grupo de ordenadores y el porqué de la elección.

De acuerdo al árbol del modelo de decisión (Figura 3-40) propuesto, se ha decidido escoger Apache Storm por varias razones. En primer lugar, Storm permite un procesado de flujos de datos rápido y eficiente con dos modelos diferentes: en microlotes y en tiempo real. Según se escoja uno u otro tipo se obtendrá una semántica de mensaje y una latencia determinada. La implementación de Storm Trident proporciona, como se ha visto anteriormente, semánticas de exactamente una ocasión. Así mismo, con Trident la latencia aumenta pero sigue siendo del orden de milisegundos, que es menor que la de Spark (del orden de segundos).

Por otro lado, en el ejemplo propuesto por [83] se puede observar como Storm presenta una tasa de procesamiento muy alta (sobre todo en mensajes de pequeño tamaño) y a la misma vez un consumo de CPU, de memoria RAM, de memoria en disco y de ancho de banda aceptables si lo comparamos con el resto de herramientas analizadas que procesan datos en tiempo real.

A mayores, hay otra serie de características de Storm que han llevado a tomar esta decisión final.

- Soporte de lenguajes: aunque Storm está escrito en Java y Clojure, tiene un buen soporte para lenguajes no JVM.
- Flujo de trabajo: Storm permite el modelado de topologías (un grafo de procesamiento de múltiples etapas) en código. Trident proporciona una API de alto nivel que va más allá de esto, incluyendo operadores familiares de tipo relacional como agrupaciones, agregaciones y uniones. [86]
- Mediante Distributed RPC (DRPC) Storm permite que se puedan escribir topologías que no solo procesen una secuencia de eventos fijados, sino que también permitan a los clientes ejecutar cálculos bajo demanda. [86]
- Particionamiento y paralelismo: el modelo multiproceso de Storm tiene la ventaja de aprovechar mejor el exceso de capacidad en una máquina inactiva, a costa de un modelo de recursos menos predecible. Storm admite el reequilibrio dinámico, lo que significa poder agregar más subprocesos o procesos a una topología sin reiniciar la topología o el conjunto. Esta es una característica conveniente, especialmente durante el desarrollo. [86]
- Madurez: Storm tiene un conjunto de características sólidas y está en desarrollo activo. Se integra bien con muchos sistemas de mensajería comunes como RabbitMQ, Kestrel, Kafka, etc.
- Popularidad: junto a Spark es de las herramientas de procesado de flujos más populares y de las que más información se puede encontrar al respecto. De igual manera, existe un gran número de empresas de renombre internacional que confían en Storm.

4 INSTALACIÓN Y PRUEBA DE APACHE STORM

4.1 Aplicación de ejemplo

En este apartado se procede a explicar detalladamente el código de la aplicación que se ejecutará en Storm, cuyo principal objetivo es el de detectar la presencia de barcos en áreas de interés (*geofencing*).

El algoritmo de la aplicación implementada, denominada *Area Alert Topology*, consta de cuatro etapas principales:

- Primera etapa (*File Read Spout*): lee el fichero de datos en la carpeta del proyecto, coge cada una de las líneas, que representan eventos AIS, y las transmite hacia los 2 *bolts* de la siguiente etapa mediante *shuffle grouping*, que distribuye las tuplas aleatoriamente entre las tareas de los *bolts* garantizando que cada *bolt* obtenga el mismo número de tuplas.
- Segunda etapa (*AIS Event Builder*): coge cada uno de los eventos y trocea sus campos (que se encuentran separados por comas). Tras esto, comprueba que cada una de las líneas tenga todos sus fragmentos completos; si es así analiza cada uno de ellos por separado para comprobar que no estén mal formados, puesto que si lo están se descarta el evento. A esta etapa también le corresponde la tranformación de las coordenadas de su formato textual a una variable tipo *float* y la conversión de las horas de los distintos eventos al formato *timestamp*. Para finalizar genera tuplas de salida que, identificadas por el nombre del barco, se distribuyen mediante *fields grouping* por los 8 *bolts* de la tercera fase. *Fields grouping* particiona el flujo por campos especificados en la agrupación, de tal manera que las tuplas con el mismo campo irán a la misma tarea siempre.
- Tercera etapa (*AIS Event Analyzer*): se encarga de analizar los eventos que han superado el filtrado de la etapa anterior. Su tarea principal consiste en detectar aquellos eventos cuyas coordenadas GPS se encuentran dentro de ciertas áreas circulares, determinadas por un centro (definido por unas coordenadas GPS) y un radio (en millas náuticas). Así mismo, como en los datos AIS son frecuentes los errores de duplicidad, en caso de aparecer secuencias de eventos iguales, solo se dejaría pasar el primero de ellos. Por tanto, si un evento se encuentra en el interior de una de las áreas y no está duplicado, se emitirá una nueva tupla hacia el único *bolt* de la última etapa mediante *global grouping*, que dirige todo el flujo a una única tarea de un *bolt*.
- Cuarta etapa (Alert Logger): recibe las tuplas, las vuelca en un fichero y las imprime por pantalla.

En la Figura 4-1 se muestra un esquema de la topología de Storm empleada en la aplicación.

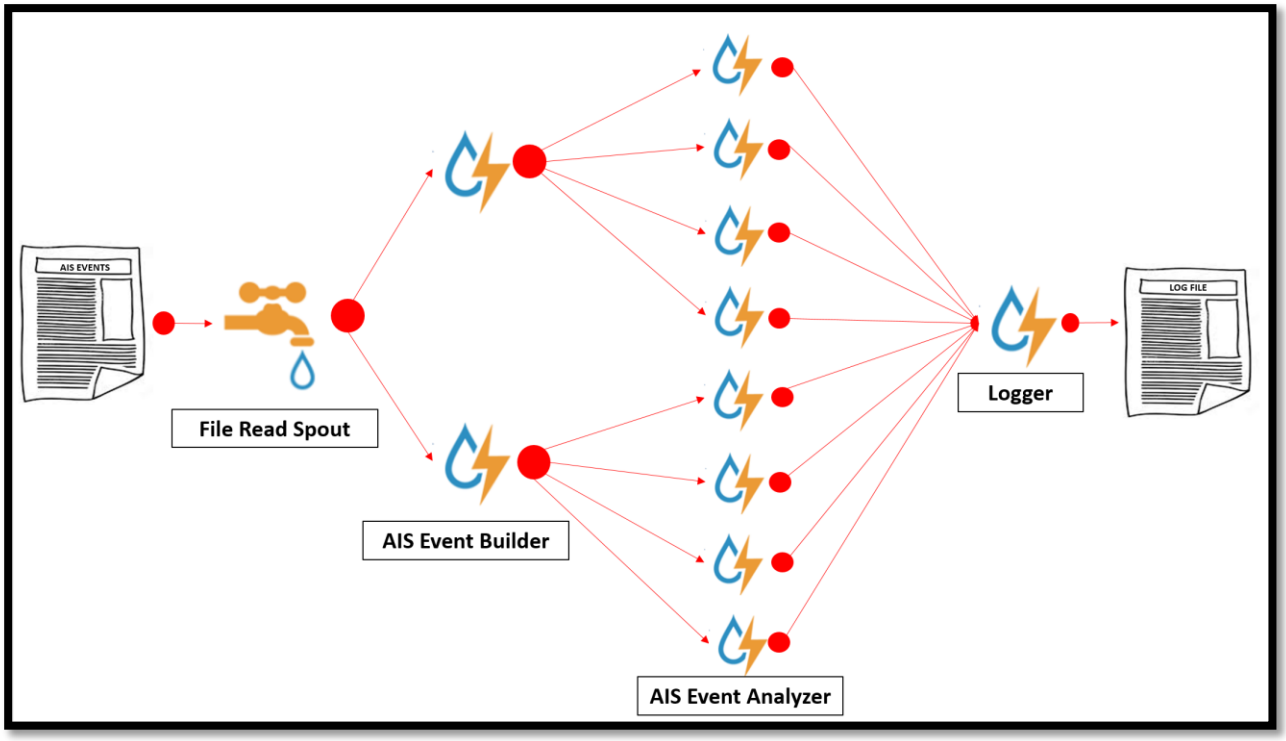

Figura 4-1 Esquema de la topología Storm empleada en la aplicación

En el Anexo IV se puede encontrar la Figura A4-0-1, la cual recoge el código completo de la aplicación. A lo largo de los siguientes párrafos se ofrece una visión más detallada de los conceptos más importantes de la aplicación.

- Líneas 1-30: se emplean los comandos *package* e *import:*
  - *package:* indica el paquete en el que se agrupan las clases.
  - *import*: importa las librerías necesarias para poder escribir la aplicación.
- Líneas 31-40: se define la clase *AreaAlertTopology* como subclase de *ConfigurableTopology*. Tras esto, se define la función principal *main*.
- Líneas 41-88: se define el método *run* por el que comenzará la ejecución. En él se lee un fichero de configuración con los datos de las áreas de interés almacenando el resultado en una lista (*LinkedList*)
  Tras esto, se emplea *TopologyBuilder* para crear la topología de la aplicación y se llevan a cabo una serie de configuraciones:
  - *setDebug*: se usa para decidir si se quiere información acerca de los errores.
  - *setNumWorkers:* indica el número de *workers* que se desea que sean utilizados para ejecutar la aplicación.
  - *setMaxSpoutPending*: se utiliza para controlar el flujo del *spout.* Su objetivo es evitar el desbordamiento de los *bolts* debido a una lectura acelerada de los eventos por parte del *FileReadSpout.* Básicamente limita el número máximo de tuplas emitidas por el *spout* cuyo procesamiento no ha sido confirmado por Storm.
  - *registerSerialization:* se emplea para registrar las nuevas clases creadas.
  - *setMessageTimeoutSecs:* se corresponde con el tiempo de espera máximo para recibir un *acknowledge* cuando se procesa una tupla en un *bolt*.
  - *registerMetricsConsumer*: necesario para registrar clases que van a encargarse de recibir información (métricas) de rendimiento para su procesado (por ejemplo, volcarlas en un fichero de *log*).

- Líneas 89-146: desarrollan la clase *AISEventBuilder*, que se emplea durante la primera etapa de *bolts*. Como se comentó con anterioridad, se crean una serie de filtros cuyos cometidos son:
    - Analizar si el evento está completo mediante la medida de su número de campos (que en todo caso debe ser 11).
    - Comprobar si la longitud, la latitud y la marca de tiempo de los distintos eventos AIS son correctas. Si pasan este filtro, los campos citados son sometidos a una serie de transformaciones de formato.
    - Asegurar que el campo con el nombre de los barcos es el correcto.
- Líneas 147-182: describen la clase *AISEventAnalyzer*. Como se apuntó previamente, dentro de su método *execute* se establece un filtro que detecta si existen eventos consecutivos duplicados. A continuación, se usa un bucle *for* para comprobar si un determinado evento se encuentra dentro alguna de las áreas de interés. En cuanto encuentra una coincidencia, emite una tupla con la alerta.
- Líneas 183-229: desarrollan la clase *AlertLogger*. En ella se define método constructor *AlertLogger* que crea un nuevo archivo para registrar los eventos de salida si este no existiera. También establece cómo se va a escribir esa información dentro del método *execute*.
- Líneas 229-385: en esta última parte se definen el resto de clases que se utilizan a lo largo de la aplicación. A continuación se procede a la descripción de cada una de ellas:
    - AISEvent (229-282): primero se lleva a cabo una definición de variables (*serialversionUID, ship, timestamp, coordinates*). Tras esto, se llama al método constructor *AISEvent* que define 3 variables (*this.ship, this.timestamp y this.coordinates*) a partir de sus 4 argumentos de entrada. Por otro, se definen los siguientes métodos *getShip,* que devuelve el *string this.ship; getTimeStamp,* que devuelve la variable de clase *long this.timestamp; getCoordinates,* que entrega la variable *this.coordinates* que es de la clase GPS; *equals,* que recibiendo como argumento de entrada la variable de clase *Object o,* devuelve verdadero si *o* es igual a *this* o falso si es distinto y, por último, se escribe la función *toString,* que devuelve un *string* con información de las variables *this.ship, this.timestamp* y *this.coordinates*.
    - GPS (255-351): como en el caso anterior se definen las variables necesarias, en este caso *serialversionUID, longitude y latitude.* Tras esto, se definen los métodos: GPS, que dado los argumentos de entrada *longitude* y *latitude*, asigna sus valores a *this.longitude* y *this.latitude*; *getLongitude*, devuelve la variable *this.longitude; getLatitude, devuelve la variable this.latitude; distanceTo,* que devuelve la distancia entre la posición GPS de un barco y el centro de un área; equals; y *toString*, que devuelve un *string* con información de las variables *this.longitude* y *this.latitude*.
    - Area (352-385): se definen las variables *name, coordinates, range y serialversionUID*. Así mismo, se declaran los siguientes métodos: *Area,* que recibe los argumentos *name, coordinates y range* en la entrada y asocia sus valores a las variables *this.name, this.coordiantes y this.range*; *getName,* que devuelve *this.name*; *get Coordinates*, que devuelve *this.coordinates*; *getRange,* que devuelve *this.range*; y *toString* para convertir el área a una cadena de texto.

4.2 Prueba de la herramienta

A lo largo de este apartado se explica, paso por paso, cómo se ha procedido a usar Apache Storm, describiendo detalladamente todos los programas y comandos empleados para ello. En primer lugar se describe brevemente el entorno de pruebas y, tras esto, se explica el proceso de prueba de Storm que presenta tres fases claramente diferenciadas: la configuración de Apache Storm, la ejecución de la

aplicación en el modo local y la ejecución de la aplicación en el modo remoto dentro de un conjunto de máquinas virtuales.

4.2.1 Entorno de pruebas

A lo largo de este apartado se explica la infraestructura usada en las distintas pruebas realizadas con Apache Storm.

Primero, cabe destacar que para el proyecto se configuraron un total de 5 VMs. Entre ellas había una que poseía mayores recursos (30 GB de RAM, 50 GB de disco duro y 16 núcleos), mientras que las otras 4 tenían 15 GB de RAM, 15 GB de disco duro y 8 núcleos cada una.

Para la prueba en modo local, se empleó una sola VM (asociada al nodo 2030), concretamente la más importante en recursos.

Para la prueba en modo remoto se utilizaron las 5 VMs (asociadas a los nodos 2030, 2031, 2032, 2033 y 2034).

4.2.2 Configuración de Apache Storm

La primera de las etapas consistió en la configuración de Apache Storm en una máquina virtual Ubuntu ubicada en el centro de cálculo del Centro Universitario de la Defensa (CUD). Para ello fue necesario hacer uso del comando *ssh* de Linux (Figura 4-2), que permitió acceder remotamente a un terminal de comandos en dicha máquina virtual.

Con el fin de poder disponer del comando *ssh* y de otros de potencial interés se llevó a cabo la descarga en [87] de la *"desktop image Ubuntu 18.04.4 LTS"*, que daba la posibilidad de emplear Ubuntu en un ordenador con sistema operativo Windows.

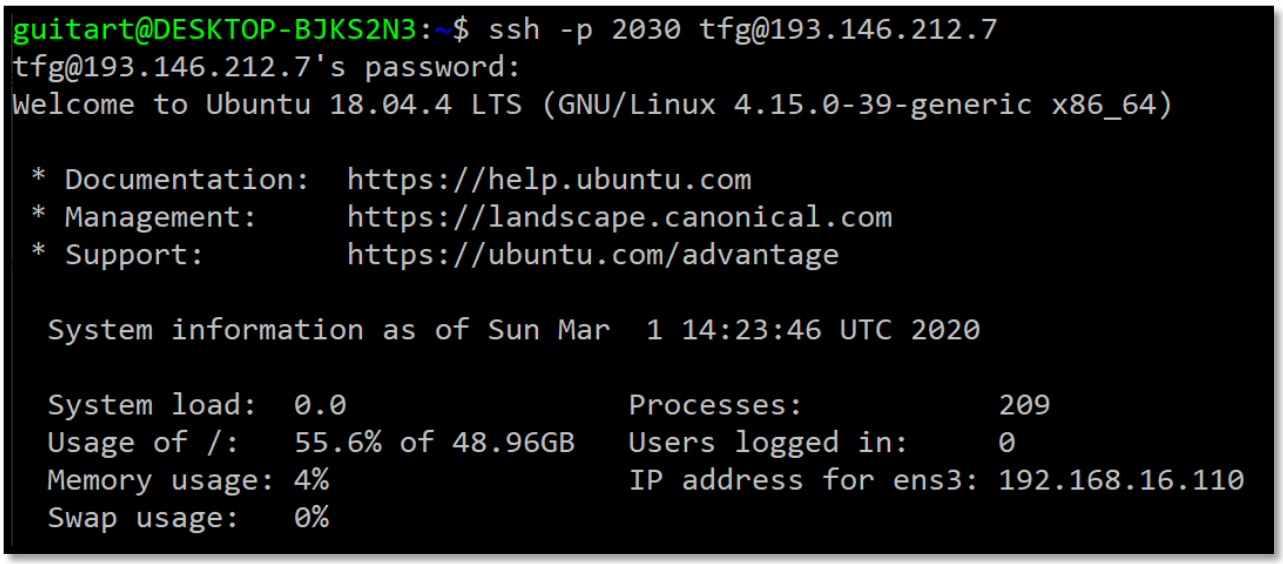

Figura 4-2 Introducción del comando *ssh* de conexión remota

Una vez establecida la conexión con la *Virtual Machine* (VM) del CUD, se instalaron Java 11.0.6 y Python 2.7.17 con los comandos "sudo apt-get install default-jre" y "sudo apt-get install python". Tras esto, se descargó la última versión de Storm (2.1.0) y se extrajo mediante el comando "tgz" (Figura 4-3). A partir de esta descarga se creó un directorio denominado "apache-storm-2.1.0" dentro de "/home/tfg/" que contenía las librerías necesarias para poder ejecutar cualquier aplicación con Apache Storm.

```
tfg@tfgp02:~$  wget http://apache.uvigo.es/storm/apache-storm-2.1.0/
apache-storm-2.1.0.tar.gz
tfg@tfgp02:~$ tar -xzvf apache-storm-2.1.0.tar.gz
```

Figura 4-3 Comandos para la descarga y extracción de Apache Storm

También se instaló Apache ZooKeeper para la coordinación del conjunto de ordenadores que se emplearían en el modo remoto. En el caso particular de nuestra aplicación, la carga que Storm delega en ZooKeeper es bastante baja, por lo que este servicio solo se descargó en un simple nodo (el nodo 2030). Tras la descarga, al igual que con Storm, se utilizó el comando "tgz" para extraerlo (Figura 4-4).

```
wget http://apache.rediris.es/zookeeper/stable/apache-zookeeper-3.5.7-bin.tar.gz

tar xvzf apache-zookeeper-3.5.7-bin.tar.gz
```

Figura 4-4 Comandos para la descarga y extracción de Apache ZooKeeper

Para completar la instalación de Apache ZooKeeper, se tuvo que acceder en el directorio "~/apache-zookeeper-3.5.7-bin/conf" (Figura 4-5). Allí se encontraba el archivo "zoo.sample.cfg", a partir del cual se creó un nuevo archivo denominado"zoo.cfg" que introducía cambios en una serie de parámetros (el *tickTime*, el *dataDir* y el *clientPort*) con respecto al primero (Figura 4-6). A continuación se describen los tres parámetros modificados:

- *tickTime*: es la unidad de tiempo básica (en milisegundos) utilizada por ZooKeeper para provocar "latidos". El tiempo de espera mínimo de la sesión es el doble del *tickTime*. Su valor se estableció como 2000.
- *dataDir*: determina la ubicación para almacenar las instantáneas de las bases de datos en memoria y, a menos que se especifique lo contrario, el registro de transacciones de actualizaciones a la base de datos. El directorio elegido fue, en este caso concreto, "datos/zoo".
- *clientPort*: se corresponde con el puerto al que los clientes se conectarán. El elegido fue el 2181.

```
tfg@ubuntu:~/apache-zookeeper-3.5.7-bin/conf$ ls
configuration.xsl  log4j.properties  zoo.cfg
```

Figura 4-5 Directorio donde se encuentra el archivo "zoo.cfg"

```
tickTime=2000
# The number of ticks that the initial
# synchronization phase can take
initLimit=10
# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5
# the directory where the snapshot is stored.
# do not use /tmp for storage, /tmp here is just
# example sakes.
dataDir=datos/zoo
# the port at which the clients will connect
clientPort=2181
```

Figura 4-6 Archivo "zoo.cfg" con las modificaciones de los parámetros realizadas

Después se situaron los paquetes descargados de Storm y de ZooKeeper en el PATH *(*conjunto de carpetas donde el sistema operativo espera encontrar los ejecutables). Para ello se utilizó el comando "nano" para editar el fichero: "$ nano /home/tfg/.bashrc" añadiendo el texto mostrado en la Figura 4-7. Para que los cambios en el PATH fueran asimilados por el sistema se introdujo el comando "$ source ~/.bashrc".

```
export PATH=$PATH:/home/tfg/apache-storm-2.1.0/bin
export PATH=$PATH:/home/tfg/apache-zookeeper-3.5.7-bin/bin

export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64

export PDSH_RCMD_TYPE=ssh
```

Figura 4-7 Situamos el ejecutable binario de Apache Storm, de Apache ZooKeeper y de OpenJDK en el PATH

La aplicación que se ejecutó sobre Storm tenía como fuente de datos un archivo en formato *comma-separated values* (CSV), que se corresponde con un archivo de texto en el que se almacenan los datos en forma de columnas (separadas por comas) y en el que las filas se distinguen por saltos de líneas (Figura 4-8). [88]

De los 11 campos que presentaba cada uno de los eventos de este archivo, a nuestra aplicación le interesaban especialmente la longitud, la latitud, la marca horaria del evento y el nombre del barco.

```
8119540582,367606370,0,0,-95.1016235351562,29.7730503082275,313,2019-12-11 00:00:00.321,MSSIS,WDH3305,PAT BOONE
8119540583,367743640,0,2,-88.0387420654297,30.1528301239014,22.2000007629395,2019-12-11 00:00:00.33,MSSIS,WDI8935,MASTER PAUL
8119540584,994010813,NA,1,-152.397583007812,-1.13196837902069,98.5999984741211,2019-12-11 00:00:00.342,MSSIS,NA,NA
8119540585,374668000,9757785,9,-144.023529052734,32.7933578491211,276.5,2019-12-11 00:00:00.342,MSSIS,H3IW,NADESHIKO
8119540585,374668000,9757785,9,-144.023529052734,32.7933578491211,276.5,2019-12-11 00:00:00.342,MSSIS,H3IW,NADESHIKO
8119540585,374668000,9757785,9,-144.023529052734,32.7933578491211,276.5,2019-12-11 00:00:00.342,MSSIS,H3IW,NADESHIKO
8119540585,374668000,9757785,9,-144.023529052734,32.7933578491211,276.5,2019-12-11 00:00:00.342,MSSIS,H3IW,NADESHIKO
8119540586,990118002,NA,0,-153.648834228516,31.8064498901367,23.7000007629395,2019-12-11 00:00:00.343,MSSIS,NA,RED-002    7V9
8119540587,257289000,9777395,3,-5.39118337631226,36.0653915405273,218.199996948242,2019-12-11 00:00:00.343,MSSIS,LADH8,BOW TITANIUM
8119540588,305240000,9436953,13,-5.96612310409546,35.9565467834473,272.200012207031,2019-12-11 00:00:00.343,MSSIS,V2DT5,BBC KWIATKOWSKI
8119540588,305240000,9436953,13,-5.96612310409546,35.9565467834473,272.200012207031,2019-12-11 00:00:00.343,MSSIS,V2DT5,BBC KWIATKOWSKI
8119540588,305240000,9436953,13,-5.96612310409546,35.9565467834473,272.200012207031,2019-12-11 00:00:00.343,MSSIS,V2DT5,BBC KWIATKOWSKI
8119540588,305240000,9436953,13,-5.96612310409546,35.9565467834473,272.200012207031,2019-12-11 00:00:00.343,MSSIS,V2DT5,BBC KWIATKOWSKI
```

Figura 4-8 Formato de los datos AIS que vamos a emplear

Este archivo, que se encontraba en /home/tfg/apache-storm-2.1.0./examples/aistorm, contenía un total de 36.167.467 eventos AIS (Figura 4-9) y se guardó en /home/tfg/apache-storm-2.1.0./examples/aistorm, directorio entorno al cual se va a proceder a ejecutar nuestra aplicación (Figura 4-10).

```
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm$ cat datos.csv | wc -l
36167467
```

Figura 4-9 Número de eventos AIS en el archivo "datos.csv"

```
tfg@ubuntu:~$ cd apache-storm-2.1.0/examples/aistorm
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm$ ls
alert.log        datos       dependency-reduced-pom.xml  pasos.txt  src
area-alert.conf  datos.csv   Makefile                    pom.xml
```

Figura 4-10 Entrada al directorio "aistorm" y visualización de los subdirectorios y archivos contenidos en él

El siguiente paso consistió en configurar Storm en el fichero "storm.yaml" que se encuentra en el directorio "~/apache-storm-2.1.0/conf" (Figura 4-11). En él se introdujo la información que se presenta en la Figura 4-12, que está relacionada con la configuración de los puertos y direcciones IP del conjunto de ordenadores empleados. Así mismo, se creó un directorio de datos local en todas y cada una de las máquinas para que Storm pudiese almacenar en él los datos necesarios.

```
tfg@ubuntu:~/apache-storm-2.1.0/conf$ ls
datos                    storm_env.ini   storm-env.sh  storm.yaml.old
path-to-data-directory   storm-env.ps1   storm.yaml
```

Figura 4-11 Directorio donde se encuentra "storm.yaml"

Explicando con más detalle la Figura 4-12, en ella se indica la dirección del equipo que actúa como coordinador (ZooKeeper), del maestro Storm del conjunto (Nimbus) y de los puertos que usarán los Supervisores dentro de los nodos esclavos (*workers*). Se indica también el puerto de funcionamiento de Storm UI, una herramienta que a través de la red y de una API REST permite monitorizar el grupo de computadoras y controlarlo.

```
#Storm cluster configuration
storm.zookeeper.servers:
  - "192.168.16.110"

#Nimbus node configuration
nimbus.seeds: ["192.168.16.110"]

#Supervisor ports
supervisor.slots.ports:
  - 6700
  - 6701
  - 6702
  - 6703

# UI port
ui.port: 8822

#Storm local Directory
storm.local.dir: "/home/tfg/apache-storm-2.1.0/datos"
```

Figura 4-12 Configuración de Apache Storm en el fichero storm.yaml

4.2.3 Ejecución de la aplicación en modo local

Como se apuntó con anterioridad, los siguientes esfuerzos estuvieron encaminados a conseguir ejecutar la aplicación en modo local. Este suele ser un primer paso de utilidad a la hora de comprobar su funcionamiento y detectar posibles errores antes de hacer un despliegue en un conjunto de máquinas.

Para ello, primero se descargó Maven [89], de manera parecida a como se procedió con Java. Apache Maven es una herramienta de gestión de proyectos de *software* basada en el concepto de *Project Object Model* (POM). Su principal cometido es administrar la construcción, los informes y la documentación de un proyecto a partir de una información central.

Una vez descargado Maven, se procedió a la compilación de la aplicación mediante el comando "mvn –e compile", que generó un "archivo.class" a partir del código fuente Java. En la Figura 4-13 puede visualizarse este paso.

```
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm$ mvn -e compile
[INFO] Error stacktraces are turned on.
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------------< org.apache.storm:aistorm >----------------------
[INFO] Building aistorm 2.1.0
[INFO] --------------------------------[ jar ]---------------------------------
```

Figura 4-13 Empleo del comando "mvn –e compile" para compilar la aplicación

Como se puede observar en la Figura 4-14, tras compilar la aplicación aparece un nuevo directorio denominado "target" en el cual se almacenan los "archivos.class" generados durante este proceso.

```
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm$ ls
alert.log        datos      dependency-reduced-pom.xml  pasos.txt  src
area-alert.conf  datos.csv  Makefile                    pom.xml    target
```

Figura 4-14 Tras ejecutar el comando "mvn –e compile" aparece el directorio "*target*"

Si se entra en el directorio "target" (Figura 4-15) se encuentran los distintos subdirectorios creados en el proceso de compilación. Dentro del subdirectorio "classes" quedan almacenados los ya citados "archivos.class".

```
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm$ cd target
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm/target$ ls
classes  generated-sources  maven-shared-archive-resources  maven-status
```

Figura 4-15 Visualización del directorio "target" con sus respectivos subdirectorios

En la siguiente captura de pantalla (Figura 4-16) se muestran los nombres de los"archivos.class" generados, así como su directorio final de localización que, como cabría esperar, coincide con la estructura definida en el código previamente.

```
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm$ cd target
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm/target$ ls
classes  generated-sources  maven-shared-archive-resources  maven-status
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm/target$ cd classes
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm/target/classes$ ls
es  META-INF
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm/target/classes$ cd es
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm/target/classes/es$ ls
uvigo
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm/target/classes/es$ cd uvigo
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm/target/classes/es/uvigo$ ls
cud
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm/target/classes/es/uvigo$ cd cud
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm/target/classes/es/uvigo/cud$ ls
'AreaAlertTopology$AISEventAnalyzer.class'  'AreaAlertTopology$Area.class'
'AreaAlertTopology$AISEventBuilder.class'    AreaAlertTopology.class
```

Figura 4-16 Nombres de los "archivos.class" creados en su directorio final de localización

Una vez finalizado el proceso de compilado, se procedió al empaquetado de los "archivos.class". La Figura 4-17 muestra el empleo del comando "mvn package" para crear un "fichero.jar" que contiene todas las clases y librerías que necesita la aplicación para funcionar.

```
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm$ mvn package
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------------< org.apache.storm:aistorm >----------------------
[INFO] Building aistorm 2.1.0
[INFO] --------------------------------[ jar ]---------------------------------
```

Figura 4-17 Utilización del comando "mvn package" para empaquetar los "archivos.class" generados al compilar

Tras esto, lo único que quedaba ya por hacer era ejecutar la aplicación en modo local. Para controlar el tiempo de ejecución se estableció un *time to live* de 60 segundos, mayor que el que tiene por defecto Storm (que es de 30 segundos). Con este fin se usó el comando "storm local target/aistorm-2.1.0.jar es.uvigo.cud.AreaAlertTopology –local-ttl 60". En la Figura 4-18 se muestra el lanzamiento de Storm en modo local.

```
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm$  storm local target/aistor
m-2.1.0.jar es.uvigo.cud.AreaAlertTopology --local-ttl 60
Running: /usr/lib/jvm/java-8-openjdk-amd64/bin/java -client -Ddaemon.name= -
Dstorm.options= -Dstorm.home=/home/tfg/apache-storm-2.1.0 -Dstorm.log.dir=/h
ome/tfg/apache-storm-2.1.0/logs -Djava.library.path=/usr/local/lib:/opt/loca
l/lib:/usr/lib:/usr/lib64 -Dstorm.conf.file= -cp /home/tfg/apache-storm-2.1.
0/*:/home/tfg/apache-storm-2.1.0/lib/*:/home/tfg/apache-storm-2.1.0/extlib/*
:target/aistorm-2.1.0.jar:/home/tfg/apache-storm-2.1.0/conf:/home/tfg/apache
-storm-2.1.0/bin: -Dstorm.local.sleeptime=60 -Dstorm.jar=target/aistorm-2.1.
0.jar -Dstorm.dependency.jars= -Dstorm.dependency.artifacts={} org.apache.st
orm.LocalCluster es.uvigo.cud.AreaAlertTopology
16:15:26.689 [main] INFO  o.a.s.LocalCluster -

                STARTING LOCAL MODE CLUSTER
```

Figura 4-18 Inicialización del conjunto en modo local

Tras la ejecución se observó la aparición del nuevo fichero "alert.log", en el que se vuelcan las alertas detectadas por la aplicación, comprobándose así el correcto funcionamiento del sistema (Figura 4-19).

```
[ROBIN M. LEE Wed Dec 11 01:01:57 UTC 2019 (-8.695032,42.39752)] IN
[Vigo,(-8.712447,42.231358) 10.0] AT 9.999158 nm
[VB MARHABA Wed Dec 11 01:02:00 UTC 2019 (-5.4888635,35.898693)] IN
[Gibraltar,(-5.485833,35.97167) 5.0] AT 4.38082 nm
[VB MARHABA Wed Dec 11 01:02:00 UTC 2019 (-5.4888635,35.898693)] IN
[Gibraltar,(-5.485833,35.97167) 5.0] AT 4.38082 nm
```

Figura 4-19 Alertas contenidas en el fichero "alert.log"

4.2.4 Ejecución de la aplicación en modo remoto

Configurar y ejecutar una aplicación en el modo remoto de Storm requiere de un proceso más elaborado que el necesario para el modo local. El primer paso consiste en decidir los recursos a emplear. Dependiendo de las características de la aplicación empleada en Storm, el número de equipos usados variará.

En el caso concreto de nuestra aplicación se optó por emplear cinco máquinas virtuales (Figura 4-20). Entre ellas existía una con mayor memoria y más núcleos que el resto: 30 GB de RAM, 50 GB de disco duro y 16 núcleos. Esta máquina se emplearía como ZooKeeper, Nimbus y *User Interface* (UI). Las cuatro restantes, que presentaban características similares (15 GB de RAM, 15 GB de disco duro y 8 núcleos), serían los *workers* en los que se ejecutaría un Supervisor. En la aplicación desarrollada, como se vio en 4.1, se registró una clase para la lectura de métricas de rendimiento que, por defecto, proporciona Apache Storm y, por lo tanto, en uno de los Supervisores habrá un *worker* más que se encargará de ellas.

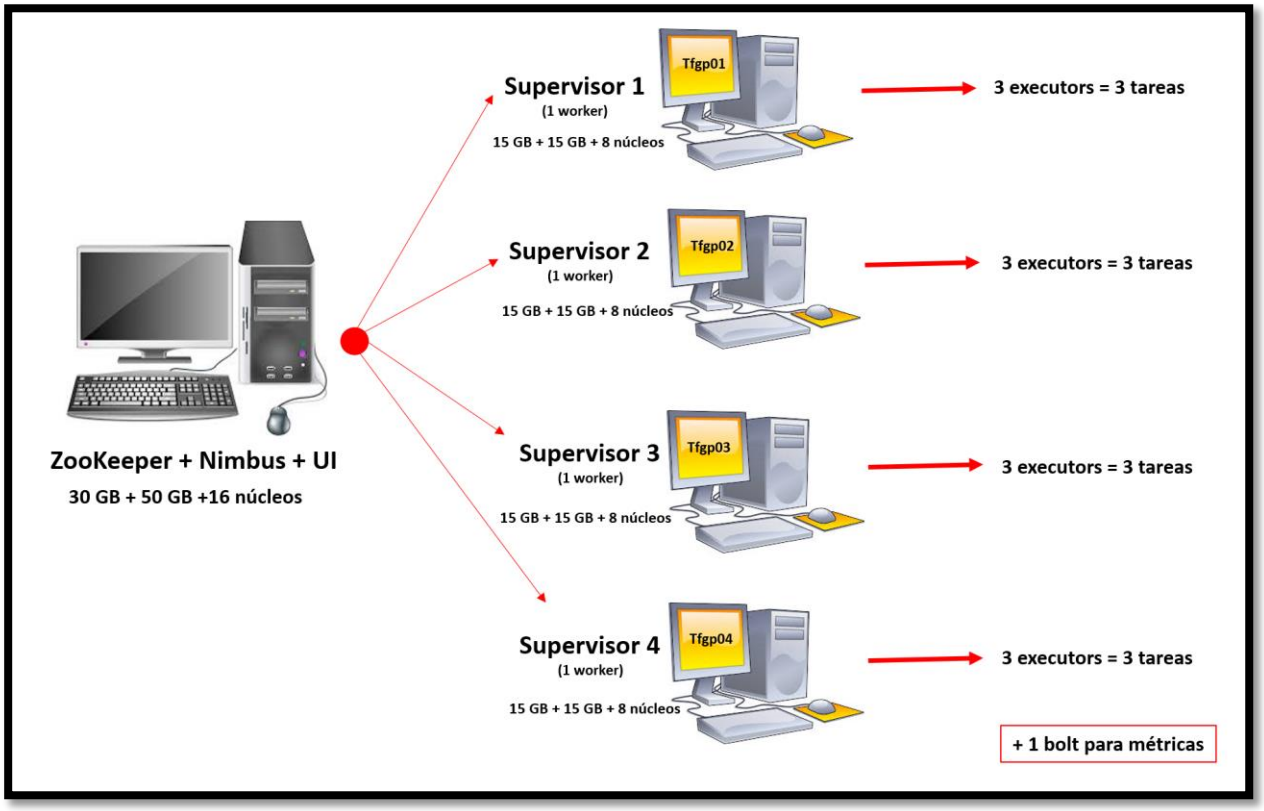

Figura 4-20 Equipamiento empleado en la prueba en modo remoto

Para conseguir ejecutar Storm en modo remoto, hay que llevar a cabo una serie de pasos, los cuales se describen a continuación.

Estos están relacionados con la configuración del Nimbus de Storm, por lo tanto solo se llevarán a cabo en el nodo maestro.

Como paso previo a la puesta a punto del Nimbus, es importante borrar todos los archivos derivados de usos anteriores de Maven. Para ello se empleará el comando "mvn clean", apreciable en la Figura 4-21.

```
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm$ mvn clean
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------------< org.apache.storm:aistorm >----------------------
[INFO] Building aistorm 2.1.0
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ aistorm ---
[INFO] Deleting /home/tfg/apache-storm-2.1.0/examples/aistorm/target
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (cleanup) @ aistorm ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
```

Figura 4-21 Uso del comando "mvn clean"

Después se procede de manera similar a como se procedió en modo local, con el empleo de los comandos "mvn –e compile" y "mvn package" (Figura 4-22 y Figura 4-23).

```
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm$ mvn -e compile
[INFO] Error stacktraces are turned on.
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------------< org.apache.storm:aistorm >----------------------
[INFO] Building aistorm 2.1.0
[INFO] --------------------------------[ jar ]---------------------------------
```

Figura 4-22 Uso del comando "mvn –e compile"

```
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm$ mvn package
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------------< org.apache.storm:aistorm >----------------------
[INFO] Building aistorm 2.1.0
[INFO] --------------------------------[ jar ]---------------------------------
```

Figura 4-23 Uso del comando "mvn package"

Una vez realizado esto, se inicializa el servidor ZooKeeper mediante el comando "zkServer.sh start &" y el Nimbus de Storm mediante "storm nimbus &" (Figura 4-24 y Figura 4-25).

```
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm$  zkServer.sh start &
[1] 28962
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm$ ZooKeeper JMX enabled by default
Using config: /home/tfg/apache-zookeeper-3.5.7-bin/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED

[1]+  Done                    zkServer.sh start
```

Figura 4-24 Inicialización del servidor ZooKeeper

```
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm$ storm nimbus &
[1] 29036
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm$ Running: /usr/lib/jvm/java-8-openjdk-
amd64/bin/java -server -Ddaemon.name=nimbus -Dstorm.options= -Dstorm.home=/home/tfg/apa
che-storm-2.1.0 -Dstorm.log.dir=/home/tfg/apache-storm-2.1.0/logs -Djava.library.path=/
usr/local/lib:/opt/local/lib:/usr/lib:/usr/lib64 -Dstorm.conf.file= -cp /home/tfg/apach
e-storm-2.1.0/*:/home/tfg/apache-storm-2.1.0/lib/*:/home/tfg/apache-storm-2.1.0/extlib/
*:/home/tfg/apache-storm-2.1.0/extlib-daemon/*:/home/tfg/apache-storm-2.1.0/conf -Xmx10
24m -Djava.deserialization.disabled=true -Dlogfile.name=nimbus.log -Dlog4j.configuratio
nFile=/home/tfg/apache-storm-2.1.0/log4j2/cluster.xml org.apache.storm.daemon.nimbus.Ni
mbus
```

Figura 4-25 Inicialización del Nimbus de Storm

Los pasos que se decriben ahora están relacionados con la configuración de los Supervisores de Storm, por lo tanto se llevarán a cabo en las 4 máquinas que tienen este cometido.

En primer lugar se usa el comando "ifconfig" para obtener la dirección IP de la máquina supervisora (Figura 4-26). Este dato será vital para el empleo de futuros comandos (como el "scp").

```
tfg@tfgp02:~/apache-storm-2.1.0/examples/aistorm$ ifconfig
ens3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.16.112  netmask 255.255.255.0  broadcast 192.168.16.255
        inet6 fe80::5054:ff:fe50:a62b  prefixlen 64  scopeid 0x20<link>
        ether 52:54:00:50:a6:2b  txqueuelen 1000  (Ethernet)
        RX packets 9564341  bytes 9379157671 (9.3 GB)
        RX errors 0  dropped 501783  overruns 0  frame 0
        TX packets 5755711  bytes 2653356174 (2.6 GB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Figura 4-26 Obtención de la dirección IP mediante "ifconfig"

Tras esto, se emplea el comando "scp" para copiar el archivo "area-alert.conf" en cada uno de los Supervisores (Figura 4-27). Aquí es donde se describen las áreas de interés para la monitorización. Como se observa en la Figura 4-28, la descripción incluye información relativa al nombre del área, a las coordenadas GPS de su centro y al radio.

```
tfg@tfgp02:~/apache-storm-2.1.0/examples/aistorm$ scp area-alert.conf tfg@192.168.16.112:/home/tfg
/apache-storm-2.1.0/examples/aistorm/
tfg@192.168.16.112's password:
area-alert.conf                                100%  147   117.2KB/s   00:00
```

Figura 4-27 Uso del comando "scp" para copiar el archivo "area-alert.conf" en el Supervisor

```
Madrid,-3.70379019,40.4167754,10.0
Japon,137.005813598633,33.9819831848145,20.0
Gibraltar,-5.48583333,35.9716666,5.0
Vigo,-8.712447,42.231356,10.0
area-alert.conf (END)
```

Figura 4-28 Contenido del archivo "area-alert.conf"

El siguiente paso se corresponde con la creación de un fichero de registro de alertas a través del comando "touch". Tras esto, como muestra la Figura 4-29, se inicializa el Supervisor de Storm.

```
tfg@tfgp02:~/apache-storm-2.1.0/examples/aistorm$ touch alert.log
tfg@tfgp02:~/apache-storm-2.1.0/examples/aistorm$  storm supervisor &
[1] 13162
tfg@tfgp02:~/apache-storm-2.1.0/examples/aistorm$ Running: java -server -Ddaemon.name=supervisor -
Dstorm.options= -Dstorm.home=/home/tfg/apache-storm-2.1.0 -Dstorm.log.dir=/home/tfg/apache-storm-2
.1.0/logs -Djava.library.path=/usr/local/lib:/opt/local/lib:/usr/lib:/usr/lib64 -Dstorm.conf.file=
 -cp /home/tfg/apache-storm-2.1.0/*:/home/tfg/apache-storm-2.1.0/lib/*:/home/tfg/apache-storm-2.1.
0/extlib/*:/home/tfg/apache-storm-2.1.0/extlib-daemon/*:/home/tfg/apache-storm-2.1.0/conf -Xmx256m
 -Djava.deserialization.disabled=true -Dlogfile.name=supervisor.log -Dlog4j.configurationFile=/hom
e/tfg/apache-storm-2.1.0/log4j2/cluster.xml org.apache.storm.daemon.supervisor.Supervisor
ls
```

Figura 4-29 Inicialización del Supervisor

Para finalizar, es necesario volver a acceder al Nimbus, ya que desde este nodo se ejecuta la aplicación y se activa la topología creada, lo que permite a los *spouts* comenzar a emitir tuplas para su procesado por la topología. La Figura 4-30 y la Figura 4-31 muestran esto último.

```
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm$ storm jar target/aistorm-2.1.0.jar es
.uvigo.cud.AreaAlertTopology
Running: /usr/lib/jvm/java-8-openjdk-amd64/bin/java -client -Ddaemon.name= -Dstorm.opti
ons= -Dstorm.home=/home/tfg/apache-storm-2.1.0 -Dstorm.log.dir=/home/tfg/apache-storm-2
.1.0/logs -Djava.library.path=/usr/local/lib:/opt/local/lib:/usr/lib:/usr/lib64 -Dstorm
.conf.file= -cp /home/tfg/apache-storm-2.1.0/*:/home/tfg/apache-storm-2.1.0/lib/*:/home
/tfg/apache-storm-2.1.0/extlib/*:target/aistorm-2.1.0.jar:/home/tfg/apache-storm-2.1.0/
conf:/home/tfg/apache-storm-2.1.0/bin: -Dstorm.jar=target/aistorm-2.1.0.jar -Dstorm.dep
endency.jars= -Dstorm.dependency.artifacts={} es.uvigo.cud.AreaAlertTopology
17:54:08.969 [main] INFO  o.a.s.StormSubmitter - Generated ZooKeeper secret payload for
 MD5-digest: -7824209410801372900:-5937101279696996011
```

Figura 4-30 Ejecución de la aplicación en modo remoto

```
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm$ storm activate area-alert
Running: /usr/lib/jvm/java-8-openjdk-amd64/bin/java -client -Ddaemon.name= -Dstorm.opti
ons= -Dstorm.home=/home/tfg/apache-storm-2.1.0 -Dstorm.log.dir=/home/tfg/apache-storm-2
.1.0/logs -Djava.library.path=/usr/local/lib:/opt/local/lib:/usr/lib:/usr/lib64 -Dstorm
.conf.file= -cp /home/tfg/apache-storm-2.1.0/*:/home/tfg/apache-storm-2.1.0/lib/*:/home
/tfg/apache-storm-2.1.0/extlib/*:/home/tfg/apache-storm-2.1.0/extlib-daemon/*:/home/tfg
/apache-storm-2.1.0/conf:/home/tfg/apache-storm-2.1.0/bin org.apache.storm.command.Acti
vate area-alert
17:55:49.319 [main] WARN  o.a.s.v.ConfigValidation - task.heartbeat.frequency.secs is a
 deprecated config please see class org.apache.storm.Config.TASK_HEARTBEAT_FREQUENCY_SE
CS for more information.
17:55:49.577 [main] INFO  o.a.s.u.NimbusClient - Found leader nimbus : ubuntu:6627
17:55:49.602 [main] INFO  o.a.s.c.Activate - Activated topology: area-alert
```

Figura 4-31 Activación de la topología de la aplicación

Una vez realizado todo el proceso descrito, se procedió a comprobar si verdaderamente la ejecución de la aplicación se estaba realizando. Una manera sencilla de asegurarse, consiste en emplear el comando "ls –l" en dos ocasiones, ya que así se puede observar si el número de registros en "alert.log" crece con el tiempo a medida que se detectan nuevas alertas y se vuelcan en el fichero. La Figura 4-32 muestra este proceso.

```
tfg@tfgp01:~/apache-storm-2.1.0/examples/aistorm$ ls -l
total 4503860
-rw-rw-r-- 1 tfg tfg     344815 feb 29 18:13 alert.log
-rw-rw-r-- 1 tfg tfg        147 feb 29 17:47 area-alert.conf
tfg@tfgp01:~/apache-storm-2.1.0/examples/aistorm$ ls -l
total 4503872
-rw-rw-r-- 1 tfg tfg     359725 feb 29 18:14 alert.log
-rw-rw-r-- 1 tfg tfg        147 feb 29 17:47 area-alert.conf
```

Figura 4-32 Los registros en "alert.log" crecen con el tiempo

Así mismo, también se puede emplear el comando "cat alert.log | wc -l" para conocer exactamente el número de alertas contenidas en "alert.log"que, como se observa en la Figura 4-33, van aumentando con el paso del tiempo.

```
tfg@tfgp02:~/apache-storm-2.1.0/examples/aistorm$ cat alert.log | wc -l
1119
tfg@tfgp02:~/apache-storm-2.1.0/examples/aistorm$ cat alert.log | wc -l
1163
tfg@tfgp02:~/apache-storm-2.1.0/examples/aistorm$ cat alert.log | wc -l
1186
```

Figura 4-33 Uso del comando "cat alert.log | wc -l" para conocer el número de alertas en cada instante

Una vez lanzado Storm en remoto, conocer las métricas del rendimiento puede resultar de gran interés. Por ello, se va a explicar como sacar información de las métricas que por defecto proporciona Storm.

En primer lugar hay que señalar que los registros de las métricas de Storm solo se encuentran disponibles en uno de los cuatro Supervisores, concretamente en aquel en el que el Nimbus de Storm haya decidido ejecutar el *bolt* responsable de volcar a fichero de *log* las métricas de rendimiento. En ese nodo se creará un fichero "worker.log.metrics" conteniendo los datos.

Tras esto, se utilizó el comando "fgrep", que permite buscar patrones en un archivo, de tal manera que solo se mostraban aquellas métricas que nos fueran de utilidad. Se muestra en la Figura 4-34 un ejemplo de búsqueda de los patrones "analyzer:default" y "__process-latency" que nos permite obtener estadísticas de la latencia de procesamiento del último *bolt* de la topología (el *AlertLogger* , que recibe su entrada de la etapa de análisis).

```
tfg@tfgp01:~/apache-storm-2.1.0/examples/aistorm$ fgrep logger ../../logs/workers-artifacts/area-
alert-1-1582998853/6700/worker.log.metrics | grep "analyzer:default" | grep "__process-latency"
2020-02-29 17:55:41,530 66788    1582998941              tfgp01:6700     17:logger       __process
-latency        {analyzer:default=1.0}
2020-02-29 17:56:41,512 126770   1582999001              tfgp01:6700     17:logger       __process
-latency        {analyzer:default=1.0}
2020-02-29 17:57:41,513 186771   1582999061              tfgp01:6700     17:logger       __process
-latency        {analyzer:default=0.8}
2020-02-29 17:58:41,517 246775   1582999121              tfgp01:6700     17:logger       __process
-latency        {analyzer:default=0.7142857142857143}
2020-02-29 17:59:41,515 306773   1582999181              tfgp01:6700     17:logger       __process
-latency        {analyzer:default=0.8571428571428571}
2020-02-29 18:00:41,517 366775   1582999241              tfgp01:6700     17:logger       __process
-latency        {analyzer:default=0.8571428571428571}
2020-02-29 18:01:41,518 426776   1582999301              tfgp01:6700     17:logger       __process
```

Figura 4-34 Proceso de búsqueda de información por patrones con el comando "fgrep"

Una vez obtenidos los resultados anteriores, se utilizó el comando "cut" para cortar exactamente la información de interés (Figura 4-35), volcándola posteriormente en un fichero de texto (Figura 4-36).

```
tfg@tfgp01:~/apache-storm-2.1.0/examples/aistorm$ fgrep logger ../../logs/workers-artifacts/area-
alert-1-1582998853/6700/worker.log.metrics | grep "analyzer:default" | grep "__process-latency" |
 cut -d\{ -f2 | cut -d= -f2 | cut -d\} -f1
1.0
1.0
0.8
0.7142857142857143
0.8571428571428571
0.8571428571428571
0.3333333333333333
0.625
0.375
2.25
0.5
0.5714285714285714
1.4285714285714286
0.5
0.8333333333333334
```

Figura 4-35 Uso del comando "cut" para recortar información de utilidad

```
tfg@tfgp01:~/apache-storm-2.1.0/examples/aistorm$ fgrep logger ../../logs/workers-artifacts/area-
alert-1-1582998853/6700/worker.log.metrics | grep "analyzer:default" | grep "__process-latency" |
 cut -d\{ -f2 | cut -d= -f2 | cut -d\} -f1 > salida.txt
```

Figura 4-36 La información se almacenó en el fichero de texto "salida.txt" para su análisis

Posteriormente, se emplearon los valores de *process latency* (término que, referido a los *bolts*, indica el tiempo promedio entre la llamada a "execute" para procesar una tupla y el momento el que el *bolt* finaliza su procesamiento). El resultado promedio de *process latency* obtenido a partir de dichos valores fue de 0,83 milisegundos.

Otra manera de realizar un análisis del funcionamiento de la aplicación es mediante la API REST de Storm, que proporciona datos de métricas e información de configuración, así como operaciones de administración. Para que dicha API REST esté disponible es necesario activar en un nodo (en nuestro caso el mismo en el que se ejecuta el maestro) la herramienta UI de Storm, ejecutando para ello el comando "*storm ui &*" (Figura 4-37).

```
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm$ storm ui &
[3] 10568
tfg@ubuntu:~/apache-storm-2.1.0/examples/aistorm$ Running: /usr/lib
/jvm/java-8-openjdk-amd64/bin/java -server -Ddaemon.name=ui -Dstorm
.options= -Dstorm.home=/home/tfg/apache-storm-2.1.0 -Dstorm.log.dir
=/home/tfg/apache-storm-2.1.0/logs -Djava.library.path=/usr/local/l
ib:/opt/local/lib:/usr/lib:/usr/lib64 -Dstorm.conf.file= -cp /home/
tfg/apache-storm-2.1.0/*:/home/tfg/apache-storm-2.1.0/lib/*:/home/t
fg/apache-storm-2.1.0/extlib/*:/home/tfg/apache-storm-2.1.0/extlib-
daemon/*:/home/tfg/apache-storm-2.1.0/lib-webapp/*:/home/tfg/apache
-storm-2.1.0/conf -Xmx768m -Djava.deserialization.disabled=true -Dl
ogfile.name=ui.log -Dlog4j.configurationFile=/home/tfg/apache-storm
-2.1.0/log4j2/cluster.xml org.apache.storm.daemon.ui.UIServer
```

Figura 4-37 Activación de Storm UI

La Figura 4-38 proporciona información acerca de la configuración del conjunto.

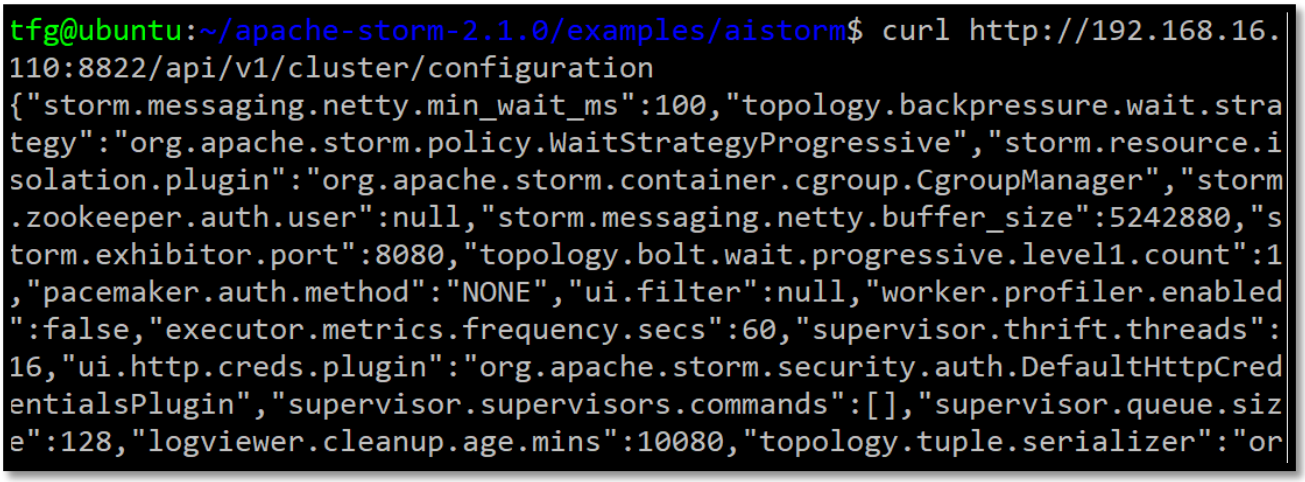

Figura 4-38 Información sobre la configuración del conjunto en Storm

La Figura 4-39 muestra las métricas de la topología *area-alert* según la API REST.

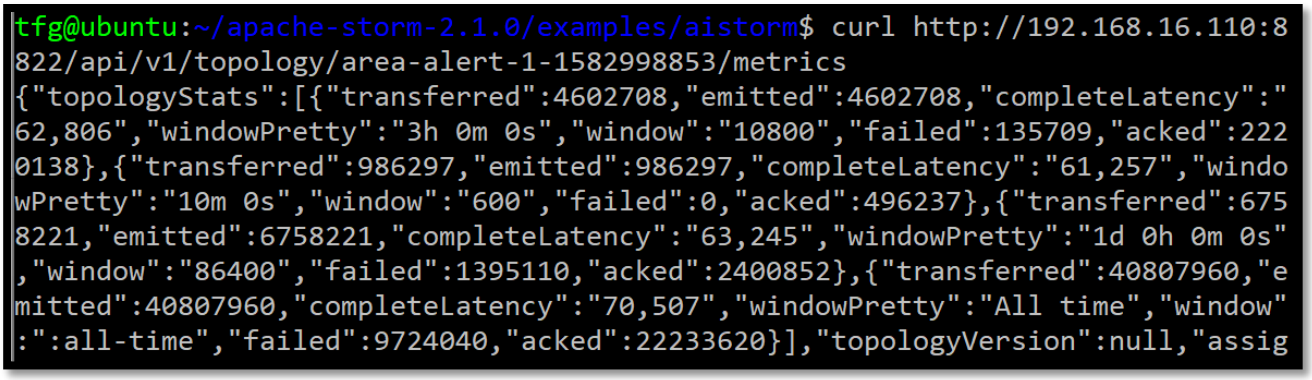

Figura 4-39 Métricas de la topología *area-alert*

5 CONCLUSIONES Y LÍNEAS FUTURAS

A lo largo del desarrollo de este TFG se ha podido comprobar como las herramientas de análisis de *big data* son de gran utilidad en muy diversos campos. La posibilidad de paralelizar y distribuir las tareas entre varias máquinas permite realizar un análisis de información en tiempo real. Dada la relativa corta edad de esta tecnología, se puede afirmar que estas herramientas todavía se encuentran en su infancia. Muchas de ellas siguen introduciendo importantes mejoras con el lanzamiento de cada nueva versión y luchan por ser más competitivas, potenciando sus virtudes y corrigiendo aquellos defectos de semántica, latencia o compatibilidad que presentan.

Por lo tanto, la conclusión principal extraída de este proyecto es que el conocimiento y el dominio de herramientas de análisis de datos, como Apache Storm, puede proporcionar grandes ventajas a aquellas empresas o instituciones que las empleen. Más específicamente, en nuestro escenario de aplicación, se ha demostrado su utilidad para el procesado de flujos de información marítima, como podrían ser los eventos AIS. Por lo tanto, la explotación del sinfín de aplicaciones compatibles con el uso de esta reciente tecnología llevará a solventar las vicisitudes que puedan surgir de una manera más rápida y eficiente. No cabe la menor duda que estas herramientas vanguardistas son el presente y el futuro inmediato de los análisis.

Una de las posibles líneas futuras de trabajo sería aplicar estas herramientas para realizar análisis predictivos de fallos en equipos de la Armada. El análisis de flujos de información proveniente de sensores ubicados en equipos fundamentales de los barcos (como motores, hélices o diésel generadores) puede llegar a ser decisivo para evitar situaciones de riesgo innecesarias y para alargar la vida útil de los buques. Actualmente existe en la Armada Española un programa de IA, denominado Soprene [90], que emplea Apache Spark, una de las herramientas analizadas en el TFG, con este fin.

Otra línea de investigación futura podría centrarse en la representación de los datos obtenidos en *dashboards,* incluyendo estadísticas sobre los buques que se encuentran en una determinada zona u ofreciendo de manera visualmente atractiva toda la información obtenida.

También podría centrarse en el estudio de plataformas para el almacenamiento de datos. Storm permite la interacción con herramientas de almacenamiento de *big data*, como HBase, con lo que además de permitir procesar los eventos en tiempo real, posibilitaría guardar dichos eventos de cara a su posible uso en tareas de análisis complejas que puedan requerir, por ejemplo, de un histórico a largo plazo de datos. Un ejemplo de las mismas podrían ser las técnicas de análisis basadas en *machine learning*, que suelen requerir importantes volúmenes de datos como entrenamiento.

Por último, destacar que por limitaciones temporales no fue posible desarrollar en este trabajo aplicaciones de tiempo real de mayor complejidad e interés, limitándonos a desarrollar un ejemplo sencillo de aplicación de *geofencing*. Sin embargo, cabe estudiar como línea futura la viabilidad de

emplear Storm para implementar analíticas más complejas, como aquellas que permiten detectar anomalías, cambios de rutas, apagado de AIS, etc.

6 BIBLIOGRAFÍA

[1] Mckinsey, «Mckinsey Global Institute,» [En línea]. Available: https://www.mckinsey.com/mgi/overview. [Último acceso: 28 febrero 2020].

[2] Xunta de Galicia, «Oportunidades Industria 4.0 en Galicia,» [En línea]. Available: http://www.atiga.es/web/wp-content/uploads/2017/03/Resumen-ejecutivo.pdf. [Último acceso: 31 enero 2020].

[3] Databricks, «A Gentle Introduction to Apache Spark,» [En línea]. Available: http://www.dcs.bbk.ac.uk/~dell/teaching/cc/book/databricks/spark-intro.pdf. [Último acceso: 31 enero 2020].

[4] N. Piette, «talend: Procesamiento por lotes o en flujo: ¿Cuál escoger y en qué momento?,» [En línea]. Available: https://es.talend.com/blog/2018/02/01/batch-vs-stream-processing/. [Último acceso: 28 febrero 2020].

[5] I. Lytra, M.-E. Vidal, F. O. y J. Attard, «A Big Data Architecture for Managing Oceans of Data and Maritime Applications,» [En línea]. Available: https://www.researchgate.net/publication/322997369_A_big_data_architecture_for_managing_oceans_of_data_and_maritime_applications. [Último acceso: 03 febrero 2020].

[6] NOAA, «NODC: World Ocean Database,» [En línea]. Available: https://www.nodc.noaa.gov/OC5/WOD/pr_wod.html. [Último acceso: 08 febrero 2020].

[7] Trelleborg, «Use of Big Data in the Maritime Industry,» [En línea]. Available: https://www.patersonsimons.com/wp-content/uploads/2018/06/TMS_SmartPort_InsightBee_Report-to-GUIDE_01.02.18.pdf. [Último acceso: 04 02 2020].

[8] Marine Traffic developers, «Marine Traffic,» [En línea]. Available: https://www.marinetraffic.com/en/ais/home/centerx:-2.7/centery:46.0/zoom:4. [Último acceso: 23 febrero 2020].

[9] G. Spiliopoulos, D. Zissis y K. Chatzikokolakis, «A big data driven approach to extracting global trade patterns,» [En línea]. Available: http://ai-group.ds.unipi.gr/mates17/spiliopoulos.pdf. [Último acceso: 04 febrero 2020].

[10] Apache Spark, «Apache Spark en la plataforma databricks,» databricks, [En línea]. Available: https://databricks.com/spark/about. [Último acceso: 22 enero 2020].

[11] International Telecommunication Union, «Recommendation ITU-R M.1371-5,» [En línea]. Available: https://www.itu.int/dms_pubrec/itu-r/rec/m/R-REC-M.1371-5-201402-I!!PDF-E.pdf. [Último acceso: 23 febrero 2020].

[12] Fujitsu, «Fujitsu and Maritime and Port Authority of Singapore Determine Effectiveness of AI Ship Collision Risk Prediction Technology,» [En línea]. Available: https://www.fujitsu.com/global/about/resources/news/press-releases/2019/0402-01.html. [Último acceso: 04 febrero 2020].

[13] E. Camossi y A.-L. Jousselme, «Proceedings of the Maritime Big Data Workshop,» [En línea]. Available: https://www.cmre.nato.int/research/publications/other-publications/1303-proceedings-of-the-maritime-big-data-workshop/file. [Último acceso: 04 febrero 2020].

[14] EMODnet, «EMODnet Physics,» [En línea]. Available: https://www.emodnet-physics.eu/Portal/. [Último acceso: 05 febrero 2020].

[15] Exprivia, «Web de Exprivia,» [En línea]. Available: https://www.exprivia.it/en/we-design-systems-that-make-the-most-complex-scenarios-simple/7247/we-continue-to-raise-our-competences-to-promote-projects-of-excellence.php. [Último acceso: 23 febrero 2020].

[16] European Maritime Safety Agency, «EMSA: Integrated Maritime Services,» [En línea]. Available: http://www.emsa.europa.eu/operations/maritime-monitoring.html. [Último acceso: 05 febrero 2020].

[17] Exprivia, «IMDatE: Big Data for maritime traffic control,» [En línea]. Available: https://www.exprivia.it/en/we-design-systems-that-make-the-most-complex-scenarios-simple/we-continue-to-raise-our-competences-to-promote-projects-of-excellence/7251/imdate-big-data-for-maritime-traffic-control.php. [Último acceso: 23 febrero 2020].

[18] Ministerio de Fomento, «Plan de Innovación para el Transporte y las Infraestructuras,» [En línea]. Available: https://www.mitma.gob.es/recursos_mfom/paginabasica/recursos/plan_de_innovacion_20182020_1.pdf. [Último acceso: 05 febrero 2020].

[19] Apache Spark, «Spark-packages,» [En línea]. Available: https://spark-packages.org/. [Último acceso: 25 enero 2020].

[20] GitHub developers, «Web de GitHub,» [En línea]. Available: https://github.com/. [Último acceso: 31 enero 2020].

[21] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker y I. Stoica, «Spark: Cluster Computing with Working Sets,» [En línea]. Available: https://www.usenix.org/legacy/event/hotcloud10/tech/full_papers/Zaharia.pdf. [Último acceso: 08 marzo 2020].

[22] I. Stoica, «Shark: Fast Data Analysis Using Coarse-grained,» [En línea]. Available: https://amplab.cs.berkeley.edu/wp-content/uploads/2012/03/mod482-xin1.pdf. [Último acceso: 08 marzo 2020].

[23] YARN developers, «Web de YARN,» [En línea]. Available: https://yarnpkg.com/. [Último acceso: 23 febrero 2020].

[24] Mesos developers, «Web de Mesos,» [En línea]. Available: http://mesos.apache.org/. [Último acceso: 23 febrero 2020].

[25] X. Meng, «MLlib: Machine Learning in Apache Spark,» 07 marzo 2020. [En línea]. Available: http://www.jmlr.org/papers/volume17/15-237/15-237.pdf.

[26] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker y I. Stoica., «Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing,» [En línea]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-82.pdf. [Último acceso: 07 marzo 2020].

[27] Databricks, « Customer Case Study Celtra,» [En línea]. Available: https://pages.databricks.com/CaseStudy-Celtra.html. [Último acceso: 31 enero 2020].

[28] C. Inc, «Web de Celtra,» [En línea]. Available: https://www.celtra.com/. [Último acceso: 06 marzo 2020].

[29] Databricks developers, «Página web de Databricks,» [En línea]. Available: https://databricks.com/product/unified-data-analytics-platform. [Último acceso: 23 febrero 2020].

[30] G. Kespret y D. Lee, «SlideShare: How Delta Optimizes Deltra with Databricks,» 9 Diciembre 2015. [En línea]. Available: https://www.slideshare.net/gregak/how-celtra-optimizes-its-advertising-platformwith-databricks. [Último acceso: 27 enero 2020].

[31] Apache Storm, «Web de Apache Storm,» [En línea]. Available: https://storm.apache.org/. [Último acceso: 10 enero 2020].

[32] D. Calvo, «Apache Storm,» [En línea]. Available: https://www.freecodecamp.org/news/apache-storm-is-awesome-this-is-why-you-should-be-using-it-d7c37519a427/. [Último acceso: 20 enero 2020].

[33] Pivotal developers, «Web de RabbitMQ,» [En línea]. Available: https://www.rabbitmq.com/. [Último acceso: 23 febrero 2020].

[34] Apache developers, «Web de Apache Thrift,» [En línea]. Available: https://thrift.apache.org/. [Último acceso: 29 febrero 2020].

[35] N. Marz, «Thoughts from the Red Planet: History of Apache Storm and lessons learned,» 6 Octubre 2014. [En línea]. Available: http://nathanmarz.com/blog/history-of-apache-storm-and-lessons-learned.html. [Último acceso: 26 enero 2020].

[36] N. Marz, «YouTube: Conferencia de Nathan Marz sobre Apache Storm,» [En línea]. Available: https://www.youtube.com/watch?v=bdps8tE0gYo. [Último acceso: 25 enero 2020].

[37] U. Ashraf, «Apache Storm is awesome. This is why (and how) you should be using it.,» [En línea]. Available: https://www.freecodecamp.org/news/apache-storm-is-awesome-this-is-why-you-should-be-using-it-d7c37519a427/. [Último acceso: 19 enero 2020].

[38] ChariotSolutons, «ETE 2012-Nathan Marz on Storm-You Tube,» [En línea]. Available: https://www.youtube.com/watch?v=bdps8tE0gYo. [Último acceso: 25 enero 2020].

[39] «Web Research Gate,» [En línea]. Available: https://www.researchgate.net/figure/Physical-architecture-of-Apache-Storm_fig2_334368601. [Último acceso: 24 enero 2020].

[40] Github developers, «Información sobre Kestrel,» [En línea]. Available: https://twitter-archive.github.io/kestrel/. [Último acceso: 28 febrero 2020].

[41] Apache developers, «Web de Apache Kafka,» [En línea]. Available: https://kafka.apache.org/. [Último acceso: 29 febrero 2020].

[42] Amazon, «Web de Amazon Kinesis Data Strams,» [En línea]. Available: https://aws.amazon.com/es/kinesis/data-streams/?nc=sn&loc=2&dn=2. [Último acceso: 29 febrero 2020].

[43] developers, Apache, «Apache Pig,» [En línea]. Available: https://pig.apache.org/. [Último acceso: 23 febrero 2020].

[44] developers, Memcached, «Web de Memcached,» [En línea]. Available: https://memcached.org/. [Último acceso: 23 febrero 2020].

[45] Apache developers, «Web de Apache Cassandra,» [En línea]. Available: http://cassandra.apache.org/. [Último acceso: 23 febrero 2020].

[46] Atlassian, «Web de JIRA,» [En línea]. Available: https://www.atlassian.com/software/jira. [Último acceso: 23 febrero 2020].

[47] Udacity, «Curso gratuito de Apache Storm en Udacity,» [En línea]. Available: https://www.udacity.com/course/real-time-analytics-with-apache-storm--ud381. [Último acceso: 23 febrero 2020].

[48] IDEXX, «Web de Laboratorios IDEXX en España,» [En línea]. Available: https://www.idexx.es/es/. [Último acceso: 23 febrero 2020].

[49] Navsite developers, «Web de Navsite,» [En línea]. Available: https://www.navisite.com/. [Último acceso: 27 enero 2020].

[50] Mongo developers, «Página web de Mongo DB,» [En línea]. Available: https://www.mongodb.com/. [Último acceso: 23 febrero 2020].

[51] Apache developers, «Web de Apache Flink,» [En línea]. Available: https://flink.apache.org/. [Último acceso: 26 enero 2020].

[52] Tzoumas, «Introduction to Apache Flink,» [En línea]. Available: https://cdn2.hubspot.net/hubfs/4757017/Ververica/Docs/Introduction_to_Apache_Flink_book_9781491998809.pdf. [Último acceso: 03 marzo 2020].

[53] Kubernetes developers, «Página web de Kubernetes,» [En línea]. Available: https://kubernetes.io/. [Último acceso: 23 febrero 2020].

[54] Java developers, «web de Simple Logging Facade for Java (SLF4J),» [En línea]. Available: http://www.slf4j.org/. [Último acceso: 25 febrero 2020].

[55] Log4j developers, «web de log4j,» [En línea]. Available: https://logging.apache.org/log4j/2.x/. [Último acceso: 25 febrero 2020].

[56] Logback developers, «logback,» [En línea]. Available: http://logback.qos.ch/. [Último acceso: 25 febrero 2020].

[57] Java, «Java Management Extensions,» [En línea]. Available: https://openjdk.java.net/groups/jmx/. [Último acceso: 25 febrero 2020].

[58] Ganglia developers, «Web de Ganglia,» [En línea]. Available: http://ganglia.sourceforge.net/. [Último acceso: 25 febrero 2020].

[59] Apache developers, «web de Apache Calcite,» [En línea]. Available: https://calcite.apache.org/. [Último acceso: 25 febrero 2020].

[60] Apache developers, «Página web de Kafka,» [En línea]. Available: https://kafka.apache.org/. [Último acceso: 25 febrero 2020].

[61] Elastic, «web de Elasticsearch,» [En línea]. Available: https://www.elastic.co/es/elasticsearch. [Último acceso: 25 febrero 2020].

[62] Javapoint developers, «JDBC Explanation,» [En línea]. Available: https://www.javatpoint.com/java-jdbc. [Último acceso: 25 febrero 2020].

[63] Bouygues, «Web de Bouygues Telecom,» [En línea]. Available: https://www.bouyguestelecom.fr/. [Último acceso: 25 febrero 2020].

[64] M. Abdessemed, «Real-time Data Integration with Apache Flink & Kafka (Bouygues Telecom),» [En línea]. Available: https://2015.flink-forward.org/index.html%3Fp=405.html. [Último acceso: 08 marzo 2020].

[65] Apache Samza developers, «Web de Apache Samza,» [En línea]. Available: http://samza.apache.org/. [Último acceso: 28 enero 2020].

[66] V. d. l. Heras, «TFM: Procesamiento y visualización de datos a gran escala aplicado al comercio electrónico,» [En línea]. Available: http://oa.upm.es/51574/1/TFM_SERGIO_VICENTE_DE_LAS_HERAS.pdf. [Último acceso: 05 03 2020].

[67] Netflix, «Web de Netflix,» [En línea]. Available: https://www.netflix.com/es/. [Último acceso: 25 febrero 2020].

[68] Redfin, «Web de Redfin,» [En línea]. Available: https://www.redfin.com/. [Último acceso: 25 febrero 2020].

[69] Flume developers, «Web de Apache Flume,» [En línea]. Available: https://flume.apache.org. [Último acceso: 28 enero 2020].

[70] D. Calvo, «Apache Flume,» [En línea]. Available: http://www.diegocalvo.es/en/apache-flume/. [Último acceso: 28 enero 2020].

[71] Apache developers, «Web de HBase,» [En línea]. Available: https://hbase.apache.org/. [Último acceso: 25 febrero 2020].

[72] Apache developers, «web de Solr,» [En línea]. Available: https://lucene.apache.org/solr/. [Último acceso: 25 febrero 2020].

[73] Cloudera, «Web de Cloudera,» [En línea]. Available: https://www.cloudera.com/. [Último acceso: 25 febrero 2020].

[74] Cloudera, «Cloudera CounterTack Case Study,» [En línea]. Available: http://pages.cloudera.com/delete/cloudera-counter-tac. [Último acceso: 31 enero 2020].

[75] Apache developers, «Web de AVRO,» [En línea]. Available: https://avro.apache.org/. [Último acceso: 25 febrero 2020].

[76] Sourceforge, «Web de JDBL,» [En línea]. Available: http://jdbl.sourceforge.net/. [Último acceso: 29 febrero 2020].

[77] Hadoop, «Web de HDFS,» [En línea]. Available: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html. [Último acceso: 29 febrero 2020].

[78] G. Shapira y J. Holoman, «Flafka: Apache Flume Meets Apache Kafka for Event Processing,» 6 noviembre 2014. [En línea]. Available: https://blog.cloudera.com/flafka-apache-flume-meets-apache-kafka-for-event-processing/. [Último acceso: 29 enero 2020].

[79] Gosecure , «Countertack,» [En línea]. Available: https://www.gosecure.net/countertack-platform. [Último acceso: 27 febrero 2020].

[80] Amazon, «Precios de Java Kinesis Data Analytics,» [En línea]. Available: https://aws.amazon.com/es/kinesis/data-analytics/pricing/?nc=sn&loc=3. [Último acceso: 29 febrero 2020].

[81] Amazon developers, «Web de Amazon,» [En línea]. Available: https://aws.amazon.com/es/kinesis/data-analytics/?nc=sn&loc=1. [Último acceso: 30 enero 2020].

[82] Y. Dendane, F. Petrillo, H. Mcheick y S. B. Ali, «Quality model for evaluating and choosing a stream,» 25 enero 2019. [En línea]. Available: https://www.researchgate.net/publication/330700749_A_quality_model_for_evaluating_and_choosing_a_stream_processing_framework_architecture. [Último acceso: 01 febrero 2020].

[83] W. Inoubli, S. Aridhi, H. Mezni y M. Maddouri, «A Comparative Study on Streaming Frameworks for Big Data,» [En línea]. Available: https://hal.inria.fr/hal-01835437/document. [Último acceso: 01 febrero 2020].

[84] A. Lorenz, «Blog de Apache Flume,» [En línea]. Available: https://cwiki.apache.org/confluence/display/FLUME/Home. [Último acceso: 29 febrero 2020].

[85] Apache developers, «Web de Apache Samoa,» [En línea]. Available: https://samoa.incubator.apache.org/. [Último acceso: 29 febrero 2020].

[86] Samza developers, «Samza vs Storm Comparison,» [En línea]. Available: http://samza.apache.org/learn/documentation/0.7.0/comparisons/storm.html. [Último acceso: 05 febrero 2020].

[87] Ubuntu developers, «Ubuntu releases: Ubuntu 18.04.4 LTS,» [En línea]. Available: http://releases.ubuntu.com/18.04/. [Último acceso: 18 febrero 2020].

[88] L. Parra, «Qué es un CSV y para qué sirve,» [En línea]. Available: https://lolap.wordpress.com/2015/01/14/que-es-un-csv-como-se-hace-y-para-que-sirve/. [Último acceso: 18 febrero 2020].

[89] Apache Maven developers, «Página web de Apache Maven,» [En línea]. Available: https://maven.apache.org/. [Último acceso: 18 febrero 2020].

[90] Grupo Edefa S.A, «Soprene: Inteligencia artificial para potenciar la operatividad de la Armada Española,» [En línea]. Available: https://www.defensa.com/espana/soprene-inteligencia-artificial-para-potenciar-operatividad. [Último acceso: 01 marzo 2020].

[91] Udacity developers, «Apache Storm Course in Udacity,» [En línea]. Available: https://www.udacity.com/course/real-time-analytics-with-apache-storm--ud381. [Último acceso: 18 febrero 2020].

ANEXO I: FUENTES DE DATOS DE LOS SERVICIOS MARÍTIMOS INTEGRADOS DE EMSA

Uno de los principales atributos de los servicios marítimos integrados es la habilidad para combinar información de muy diversas fuentes de datos, y de esta manera enriquecer el conocimiento del dominio marítimo.

- *Automatic Identification System* (AIS): es un servicio marítimo de retransmisión VHF.
- *Long Range Identification and Tracking* (LRIT): consiste en un sistema de identificación y seguimiento basado en comunicaciones satelitales.
- *Additional Ship and Voyage Information*: intercambio de datos entre los estados miembros a través del Safe SEANet system.
- *Satellite AIS*: son nuevos sistemas que se están desarrollando para permitir a los satélites recibir las posiciones AIS.
- *Synthetic aperture radar satellite images*: Los sensores de los satélites radar miden la rugosidad de la superficie del mar independientemente de las condiciones del tiempo o de la luz solar. Esto es usado en los sistemas de detección de barcos así como en la monitorización de la polución.
- *Optical satellite images*: Imágenes de observación de la tierra desde satélites operando en el espectro óptico, proporcionando imágenes de alta resolución de barcos o áreas costeras.
- *Metereological- oceanographic area*: está siendo desarrollado en la actualidad e incluirá un rango de campos: velocidad del viento y dirección, altura de la ola y dirección, periodo de la ola.

También pueden ser integradas otras fuentes de datos provenientes de sistemas nacionales:

- *Vessel Monitoring System*: VMS usa comunicaciones satelitales para el seguimiento comercial de barcos pesqueros.
- *Coastal radar*: los servicios de control del tráfico marítimo de los estados miembros monitorizan constantemente los movimientos de los buques a lo largo de la costa con la ayuda de los rádares locales.
- Uso de datos específicos: EMSA puede también procesar otra variedad de datos distinta proporcionada por usuarios nacionales. Estos datos incluyen informes de posición encriptados de barcos de patrulla, informes de posición provenientes de barcos de recreo y datos adicionales meteorológicos y oceanográficos provenientes de boyas.

ANEXO II: CIELO ÚNICO EUROPEO Y PROYECTO MONALISA 2.0

El Cielo Único Europeo (CUE) es una iniciativa creada en 1999 por la Comisión Europea, que busca integrar y reestructurar el sistema de control de tráfico aéreo europeo con el objetivo de mejorar el funcionamiento de la gestión del tránsito aéreo y los servicios de navegación.

Esta reestructuración persigue aumentar la capacidad para atender la demanda futura prevista, así como aumentar el rendimiento global de la gestión del tráfico aéreo europeo.

El Cielo Único Europeo se ha marcado como objetivos para el año 2020 conseguir mejorar el sistema de Control de Tráfico Aéreo (CTA) en términos de seguridad, medioambiente, capacidad y rentabilidad.

Algunos de los avances que pretende introducir el CUE son:

- Mejorar la calidad en el servicio a la vez que se atiende a la creciente demanda aérea esperada.
- Ahorro en tiempo y combustible, mejorando el impacto medioambiental gracias a rutas más cortas.
- Disminución de retrasos y cancelaciones de vuelos.
- Mejora de la eficiencia en el transporte aéreo.

Antecedentes: Monalisa 2.0, que fue aprobado por Decisión de la Comisión C (2013) 7588 de 05 de noviembre, concede una ayuda financiera de la Unión Europea a proyectos de interés común en el ámbito de la Red Transeuropea de Transporte (RTE-T). El objetivo global es contribuir al desarrollo de las autopistas del mar (MOS) en la UE en línea con las políticas de transporte marítimo, así como reforzar la eficiencia, la seguridad y la protección del medio ambiente en el transporte marítimo. La Sociedad de Salvamento y Seguridad Marítima ha participado en el Proyecto Monalisa 2.0 en calidad de beneficiario, junto con otros organismos del sector público y privado de los siguientes estados miembros: Suecia, Italia, Alemania, España, Grecia, Reino Unido, Dinamarca, Malta y Finlandia, y siendo el coordinador del Proyecto la Swedish Maritime Administration.

ANEXO III: COMPAÑÍAS QUE USAN LAS HERRAMIENTAS ANALIZADAS

En la Tabla 6 se exponen algunas de las más importantes empresas que utilizan las herramientas de análisis de flujo de datos de *software* libre analizadas. En la sección "powered by" de la página web de cada herramienta se especifica con más detalle la manera en que estas son empleadas. Dada la aparente inactividad de la página "powered by" asociada a Apache Flume, la información sobre las empresas que usan Apache Flume no será proporcionada.

Apache Spark [10]	**Apache Storm** [31]	**Apache Flink** [51]	**Apache Samza** [65]
Alibaba	Groupon	Alibaba	Linkedin
Amazon	Twitter	AWS	Intuit
Autodesk	Yahoo!	bouygues	Tripadvisor
eBay	Spotify	Capital One	Tivo
NASA JPL	Flipboard	eBay	Slack
Shopify	Ooyala	Ericsson	Redfin
Tripadvisor	Taobao	Huawei	Netflix
Yahoo!	Alibaba	Uber	Optimizely
RocketFuel	Yelp	Yelp	Banno
Premise	Klout	Xiaomi	Movio
PanTera	Wego	Zalando	Ntent
Ooyala	RocketFuel	Vip.com	Fortscale
Groupon	Navisite	Pinterest	Cavulus
Flyxt	TwiSprout	OVH	Metamarkets
Conviva	Trovit	Mux	Vmware
Concur	Akazoo	Comcast	Vintank

Tabla 6 Principales empresas que hacen uso de las herramientas estudiadas

ANEXO IV: CÓDIGO DE LA APLICACIÓN

```
1  package es.uvigo.cud;
2
3  import java.util.HashMap;
4  import java.util.Map;
5  import java.util.List;
6  import java.util.LinkedList;
7  import java.util.Date;
8  import java.text.SimpleDateFormat;
9  import java.io.Serializable;
10 import java.io.File;
11 import java.io.FileReader;
12 import java.io.BufferedReader;
13 import java.io.IOException;
14 import java.nio.file.Paths;
15 import java.nio.file.Files;
16 import java.nio.file.StandardOpenOption;
17 import java.nio.charset.StandardCharsets;
18 import org.apache.storm.perf.spout.FileReadSpout;
19 import org.apache.storm.topology.BasicOutputCollector;
20 import org.apache.storm.topology.ConfigurableTopology;
21 import org.apache.storm.topology.OutputFieldsDeclarer;
22 import org.apache.storm.topology.TopologyBuilder;
23 import org.apache.storm.topology.base.BaseBasicBolt;
24 import org.apache.storm.metric.LoggingMetricsConsumer;
25 import org.apache.storm.tuple.Fields;
26 import org.apache.storm.tuple.Tuple;
27 import org.apache.storm.tuple.Values;
28 import org.slf4j.Logger;
29 import org.slf4j.LoggerFactory;
30
31
32 public class AreaAlertTopology extends ConfigurableTopology {
33
34     // private static final Logger LOG = LoggerFactory.getLogger(ShipCountTopology.class);
35
36     public static void main(String[] args) throws Exception {
37
38         ConfigurableTopology.start(new AreaAlertTopology(), args);
39     }
40
```

```
41
42      @Override
43      protected int run(String[] args) throws Exception {
44
45          String line = "";
46          LinkedList<Area> areas = new LinkedList<Area>();
47          try (BufferedReader br = new BufferedReader(new FileReader("/home/tfg/apache-
    storm-2.1.0/examples/aistorm/area-alert.conf"))) {
48
49              while ((line = br.readLine()) != null) {
50                  String[] datosArea = line.split(",");
51                  String nombre = datosArea[0];
52                  float longitud = Float.parseFloat(datosArea[1]);
53                  float latitud = Float.parseFloat(datosArea[2]);
54                  float radio = Float.parseFloat(datosArea[3]);
55                  areas.add(new Area(nombre, new GPS(longitud, latitud), radio));
56              }
57
58          } catch (IOException e) {
59              e.printStackTrace();
60          }
61
62          TopologyBuilder builder = new TopologyBuilder();
63          builder.setSpout("input", new FileReadSpout("/home/tfg/apache-storm-
    2.1.0/examples/aistorm/datos.csv"), 1);
64          builder.setBolt("builder", new AISEventBuilder(), 2).shuffleGrouping("input");
65          builder.setBolt("analyzer", new AISEventAnalyzer(areas), 8).fieldsGrouping("builder",
    new Fields("ship"));
66          builder.setBolt("logger", new AlertLogger("/home/tfg/apache-storm-
    2.1.0/examples/aistorm/alert.log"), 1).globalGrouping("analyzer");
67
68          String topologyName = "area-alert";
69
70          // Disable debug logging
71          conf.setDebug(true);
72
73          // Set the number of worker nodes
74          conf.setNumWorkers(4);
75
76          // Listen and log performance metrics
77
    conf.registerMetricsConsumer(org.apache.storm.metric.LoggingMetricsConsumer.class, 1);
78
```

```
79          // Register classes to be sent into tuples
80          conf.registerSerialization(AISEvent.class);
81          conf.registerSerialization(Area.class);
82
83          // Spout flow control
84          conf.setMaxSpoutPending(1000);
85
86          return submit(topologyName, conf, builder);
87      }
88
89
90      public static class AISEventBuilder extends BaseBasicBolt {
91
92          SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd
    HH:mm:ss.SSS");
93
94          @Override
95          public void execute(Tuple tuple, BasicOutputCollector collector) {
96              String line = tuple.getString(0);
97              String[] tokens = line.split(",");
98
99              if (tokens.length != 11) {
100                 return;
101             }
102
103             Float longitude;
104             String strLongitude = tokens[4].trim();
105             if (strLongitude.equals("") || strLongitude.equals("NA")) {
106                 return;
107             } else {
108                 longitude = Float.parseFloat(strLongitude);
109             }
110
111             Float latitude;
112             String strLatitude = tokens[5].trim();
113             if (strLatitude.equals("") || strLatitude.equals("NA")) {
114                 return;
115             } else {
116                 latitude = Float.parseFloat(strLatitude);
117             }
118
119             Long timestamp;
```

```
120             String strTimestamp = tokens[7].trim();
121             if (strTimestamp.equals("") || strTimestamp.equals("NA")) {
122                 return;
123             } else {
124                 try {
125                     Date date = formatter.parse(strTimestamp);
126                     timestamp = date.getTime();
127                 } catch(Exception e) {
128                     return;
129                 }
130             }
131 
132             String ship = tokens[10].trim();
133             if (ship.equals("") || ship.equals("NA")) {
134                 return;
135             }
136 
137             AISEvent event = new AISEvent(ship, timestamp, longitude, latitude);
138             collector.emit(new Values(ship, event));
139         }
140 
141         @Override
142         public void declareOutputFields(OutputFieldsDeclarer declarer) {
143             declarer.declare(new Fields("ship", "event"));
144         }
145     }
146 
147 
148     public static class AISEventAnalyzer extends BaseBasicBolt {
149 
150         AISEvent lastEvent = null;
151 
152         List<Area> areas = new LinkedList<Area>();
153 
154         public AISEventAnalyzer(List<Area> areas) {
155             this.areas = areas;
156         }
157 
158         @Override
159         public void execute(Tuple tuple, BasicOutputCollector collector) {
160             String ship = tuple.getString(0);
161             AISEvent event = (AISEvent)tuple.getValue(1);
```

```
162
163             // Filter some duplicated AIS events
164             if (!event.equals(this.lastEvent)) {
165                 GPS gps = event.getCoordinates();
166                 for (Area area : this.areas) {
167                     GPS areaCoordinates = area.getCoordinates();
168                     float areaRange = area.getRange();
169                     float distance = gps.distanceTo(areaCoordinates);
170                     if (distance <= areaRange) {
171                         collector.emit(new Values(ship, event, area, distance));
172                     }
173                 }
174                 lastEvent = event;
175             }
176         }
177
178         @Override
179         public void declareOutputFields(OutputFieldsDeclarer declarer) {
180             declarer.declare(new Fields("ship", "event", "area", "distance"));
181         }
182     }
183
184
185     public static class AlertLogger extends BaseBasicBolt {
186
187         boolean logToFile = true;
188         String filePath = null;
189
190         public AlertLogger(String filePath) {
191             try {
192                 File file = new File(filePath);
193                 if (!file.exists()) {
194                     file.createNewFile();
195                 }
196                 this.filePath = filePath;
197                 logToFile = true;
198             } catch (Exception e) {
199                 logToFile = false;
200                 e.printStackTrace();
201             }
202         }
```

```
203
204         @Override
205         public void execute(Tuple tuple, BasicOutputCollector collector) {
206
207             String ship = tuple.getString(0);
208             AISEvent event = (AISEvent)tuple.getValue(1);
209             Area area = (Area)tuple.getValue(2);
210             Float distance = (Float)tuple.getValue(3);
211
212             String messageToWrite = event + " IN " + area + " AT " + distance + " nm\n";
213             System.out.print(messageToWrite);
214             if (logToFile) {
215                 try {
216                     Files.write(Paths.get(filePath),
    messageToWrite.getBytes(StandardCharsets.UTF_8), StandardOpenOption.WRITE,
    StandardOpenOption.APPEND);
217                     // Files.writeString(Paths.get(filePath), messageToWrite,
    StandardCharsets.UTF_8, StandardOpenOption.WRITE, StandardOpenOption.APPEND);
218                 } catch(Exception e) {
219                     logToFile = false;
220                     e.printStackTrace();
221                 }
222             }
223         }
224
225         @Override
226         public void declareOutputFields(OutputFieldsDeclarer declarer) {}
227     }
228
229
230     public static class AISEvent implements Serializable {
231
232         private static final long serialversionUID = 202002131205L;
233
234         String ship;
235         long timestamp;
236         GPS coordinates;
237
238         public AISEvent() {}
239
240         public AISEvent(String ship, long timestamp, float longitude, float latitude) {
241             this.ship = ship;
242             this.timestamp = timestamp;
```

```
243             this.coordinates = new GPS(longitude, latitude);
244         }
245
246         public String getShip() {
247             return this.ship;
248         }
249
250         public long getTimestamp() {
251             return this.timestamp;
252         }
253
254         public GPS getCoordinates() {
255             return this.coordinates;
256         }
257
258         @Override
259         public boolean equals(Object o) {
260
261             if (o == this) {
262                 return true;
263             }
264
265             if (!(o instanceof AISEvent)) {
266                 return false;
267             }
268
269             AISEvent event = (AISEvent) o;
270             return (
271                 this.ship.equals(event.getShip()) &&
272                 this.coordinates.equals(event.getCoordinates()) &&
273                 this.timestamp == event.getTimestamp()
274             );
275         }
276
277         @Override
278         public String toString() {
279             return new String("[" + this.ship + " " + new Date(this.timestamp) + " " +
    this.coordinates + "]");
280         }
281     }
282
283
```

```
284     public static class GPS implements Serializable {
285
286         private static final long serialversionUID = 202002131207L;
287
288         float longitude;
289         float latitude;
290
291         public GPS() {}
292
293         public GPS(float longitude, float latitude) {
294             this.longitude = longitude;
295             this.latitude = latitude;
296         }
297
298         public float getLongitude() {
299             return this.longitude;
300         }
301
302         public float getLatitude() {
303             return this.latitude;
304         }
305
306         // Code from: https://www.geodatasource.com/developers/java
307         // Output in Nautical Miles, check reference to change to KM
308         public float distanceTo(GPS other) {
309
310             float lat1 = this.latitude;
311             float lon1 = this.longitude;
312             float lat2 = other.getLatitude();
313             float lon2 = other.getLongitude();
314
315             if ( (Math.abs(lat1 - lat2) < 1e-6) && (Math.abs(lon1 - lon2) < 1e-6) ) {
316                 return 0;
317             }
318             else {
319                 double theta = lon1 - lon2;
320                 double dist = Math.sin(Math.toRadians(lat1)) * Math.sin(Math.toRadians(lat2)) +
    Math.cos(Math.toRadians(lat1)) * Math.cos(Math.toRadians(lat2)) *
    Math.cos(Math.toRadians(theta));
321                 dist = Math.acos(dist);
322                 dist = Math.toDegrees(dist);
323                 dist = dist * 60 * 1.1515 * 0.8684;
324                 return ((float)dist);
```

```
325             }
326         }
327
328         @Override
329         public boolean equals(Object o) {
330
331             if (o == this) {
332                 return true;
333             }
334
335             if (!(o instanceof GPS)) {
336                 return false;
337             }
338
339             GPS gps = (GPS) o;
340             return (
341                     (Math.abs(this.longitude - gps.getLongitude()) < 1e-6) &&
342                     (Math.abs(this.latitude - gps.getLatitude()) < 1e-6)
343                 );
344         }
345
346         @Override
347         public String toString() {
348             return new String("(" + this.longitude + "," + this.latitude + ")");
349         }
350     }
351
352
353     public static class Area implements Serializable {
354
355         private static final long serialversionUID = 202002131308L;
356
357         String name;
358         GPS coordinates;
359         float range;
360         public Area() {}
361
362         public Area(String name, GPS coordinates, float range) {
363             this.name = name;
364             this.coordinates = coordinates;
365             this.range = range;
```

```
366         }
367
368         public String getName() {
369             return this.name;
370         }
371
372         public GPS getCoordinates() {
373             return this.coordinates;
374         }
375
376         public float getRange() {
377             return this.range;
378         }
379
380         @Override
381         public String toString() {
382             return new String("[" + this.name + "," + this.coordinates + " " + this.range + "]");
383         }
384     }
385 }
```

Figura A4-0-1 Código de la aplicación