IMPLEMENTACIÓN DE UNA ARQUITECTURA EN TIEMPO REAL PARA EL PROCESAMIENTO Y CONTROL DE EVENTOS PROCEDENTES DE SENSORES Y ALARMAS

Contenido

1. Introducción

Han pasado ya 14 años desde que en octubre del 2003 Google publicara su artículo denominado *"The Google File System"* en el que detallaban cómo manejaban grandes volúmenes de datos a través de la implementación de un sistema de ficheros completamente distribuido y escalable. Un año más tarde, Google publicaría otro artículo denominado *"Map Reduce: Simplified Data Processing on Large Clusters"*, en el que argumentaban la forma en que ellos entendían e implementaban este framework de procesamiento como es MapReduce. Estas dos publicaciones sentarían las bases para que posteriormente la comunidad Open Source sacase a la luz la primera versión de Hadoop.

Poco a poco, el procesamiento y almacenamiento de grandes volúmenes de datos comenzó a convertirse en algo habitual y requerido por las empresas. Este tipo de procesamiento de datos era principalmente de tipo Batch, es decir, el Job encargado de procesar el dato tenía un inicio y un fin marcado.

Una de las principales ideas que llegó con Hadoop y que todavía se sigue empleando en el mundo Batch es el concepto de llevar el procesamiento al dato, es decir, el objetivo es tener una gran cantidad de datos almacenados en algún sistema de ficheros como por ejemplo HDFS y que sean los algoritmos los que una vez implementados lleguen hasta donde se encuentran los datos para procesarlos.

Durante los últimos años no solo se le da importancia al hecho de procesar los datos y obtener una salida, si no que lo que se pretende es tener la salida en un tiempo de respuesta cada vez más cercano al tiempo real. Esto obliga a cambiar la mentalidad del tipo de procesamiento, si antes teníamos la idea de llevar el procesamiento al dato, la meta de este nuevo concepto es lo contrario, los algoritmos permanecen siempre en funcionamiento, es decir, no tienen un final marcado, y son los datos en un flujo de streaming los que llegan hasta ellos. El gran avance en los últimos años de motores de procesamiento en tiempo real hace posible este concepto.

2. Alcance del proyecto

Este proyecto es el resultado de la finalización del III Máster de Arquitectura Big Data, impartido por KSchool y dirigido por Rubén Casado. En él, se implementará un sistema basado en una arquitectura en tiempo real cuyo objetivo es el control de un amplio conjunto de alarmas de detección de movimiento, humo y temperatura. Dicho sistema se encargará de obtener todos los mensajes generados por los sensores de las alarmas, los cuales se están emitiendo continua e ininterrumpidamente, procesar toda esta información y activar protocolos de emergencia ante la activación de cualquier sensor.

La implementación será un sistema End To End, que por sí solo y sin ninguna ayuda manual, realizará todo el proceso de obtención y procesamiento de los datos, almacenamiento de la información generada, recuperación de datos de interés de los sensores, activación de protocolos de actuación dependiendo de la alarma generada y visualización en tiempo real del estado de todos los sensores del sistema.

Se implementará también un sistema de monitorización de todos los servicios del entorno en el que se reflejará en todo momento los recursos consumidos por cada servicio y la cantidad de mensajes que están entrando en el sistema.

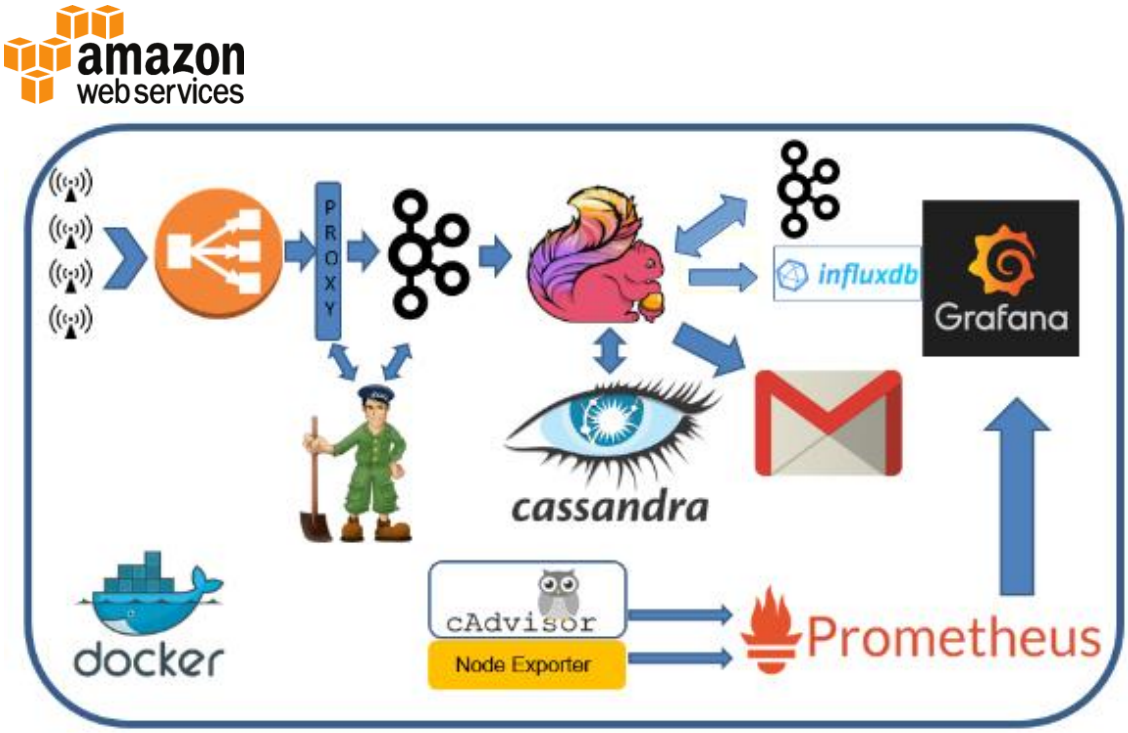
3. Requisitos

Los requisitos que se han marcado para este proyecto son los siguientes:

- Necesidad de la existencia de un servicio en alta disponibilidad al que los sensores envíen los mensajes con sus estados. Este servicio sería la entrada a todo el sistema.
- Se requiere de un sistema de almacenamiento donde tener registrados a todos los clientes que tienen contratadas las alarmas con todos sus datos.
- Es necesario la existencia de un motor de procesamiento que permita leer y procesar todos los mensajes recibidos y que aplique la lógica necesaria para determinar si una alarma se ha activado y determinar la acción a seguir.
- Será necesario un sistema de integración y paso de mensajes que una el punto de entrada al sistema con el motor de procesamiento.
- Se requiere de un sistema de visualización para poder mostrar el estado de todos los sensores en cualquier momento.
- Será necesario la existencia de agentes en cada uno de los nodos que sean capaces de reportar todas las métricas necesarias para tener todo el entorno monitorizado desde un único punto común.

4. Solución propuesta

La arquitectura propuesta para la realización del proyecto es la siguiente:



A continuación, se detalla la secuencia de principio a fin desde que un sensor emite un mensaje hasta que ese mensaje es visualizado.

1. Los sensores están continuamente emitiendo mensajes con su estado. Los mensajes serán de tipo Json y estarán formados por el identificador del sensor, un campo de movimiento, otro de humo y un último campo con la temperatura del sensor y su temperatura máxima. Estos mensajes serán enviados en su totalidad a un único punto de entrada al sistema, que será un ELB proporcionado por AWS, el cuál asegura alta disponibilidad. En la configuración del load balancer, se especificará el destino de donde tiene que enviar los mensajes que recibe.
2. Los mensajes que han entrado al ELB vía HTTP serán enviados a un proxy, que tendrá la misión de recoger los mensajes HTTP y enviarlos a una cola de Kafka. El proxy estará replicado en varias instancias para asegurar HA, uno por cada broker de Kafka.
3. Todos los mensajes se almacenarán en un topic de Kafka, el cuál proporciona tolerancia a fallos y persistencia temporal de los datos.

4. El motor de procesamiento será Flink, el cual se subscribirá al topic de Kafka y recibirá los mensajes mediante un flujo de streaming. Una vez que obtiene los mensajes, los analizará y determinará la acción a tomar.
5. El sistema de almacenamiento escogido que contendrá todos los datos de los clientes y su relación con los sensores contratados es Cassandra.

Por otro lado, la arquitectura propuesta incluye un sistema de monitorización de todos los servicios y nodos del sistema, para ello, se instalará cAdvisor y Node Exporter en todos los servidores. Éstos servicios se encargarán de recoger todas las métricas de los servicios y exponerlas en un puerto HTTP para que Grafana a través de Prometheus recoja todos estos datos y puedan ser visualizados en tiempo real.

Por último, cabe indicar que todos los servicios se instalarán y se configurarán mediante Docker. A su vez, todo el sistema estará alojado en AWS utilizando instancias EC2.

5. Implementación

En este capítulo se detallarán los pasos que se han llevado a cabo para la implementación del proyecto en todos los módulos involucrados.

5.1 Cassandra

Como se ha comentado anteriormente, el sistema de almacenamiento escogido ha sido Cassandra. Los principales motivos para la elección de esta base de datos han sido los siguientes:

- Gran escalabilidad horizontal que presenta este servicio.
- Alto rendimiento en consultas, sobre todo si los datos son estáticos como en este caso de uso. No está previsto ejecutar inserciones en el alcance de este proyecto.
- Todas las consultas ejecutadas en cassandra irán en base a un id, (identificador del sensor), por lo que éste será la clave del esquema de cassandra. Este modelo de datos encaja perfectamente en la arquitectura de esta bbdd.

Se ha implementado un clúster de 3 nodos, distribuyendo los datos con factor de replicación igual a 2. El estado, repartición de la carga y tokens de cada uno de los nodos del clúster se puede ver en la siguiente imagen:

```
root@eb2deb1fa587:/# nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address       Load       Tokens       Owns (effective)  Host ID                               Rack
UN  172.31.8.143  6.93 MiB   256          65.7%             cf369487-a980-410b-801c-bfcf18f00cdf  rack1
UN  172.31.7.201  10.52 MiB  256          67.3%             1c7dfbd3-d371-42c0-879b-872af0341b20  rack1
UN  172.31.1.114  10.52 MiB  256          67.0%             37a29c48-9f6e-4ce9-936a-0958982aa5ab  rack1
```

Se ha creado el keyspace ‘world’ y dentro se ha creado la tabla ‘datosclientes’ con la siguiente estructura:

```
CREATE TABLE world.datosclientes (
 id INT PRIMARY KEY,
 name text,
 surname text,
 gender text,
```

```
nationalid text,
birthday text,
streetaddress text,
city text,
zipcode text,
country text,
telephonecountrycode text,
telephonenumber text,
latitude DOUBLE,
longitude DOUBLE,
emailaddress text,
title text,
occupation text,
company text,
username text);
```

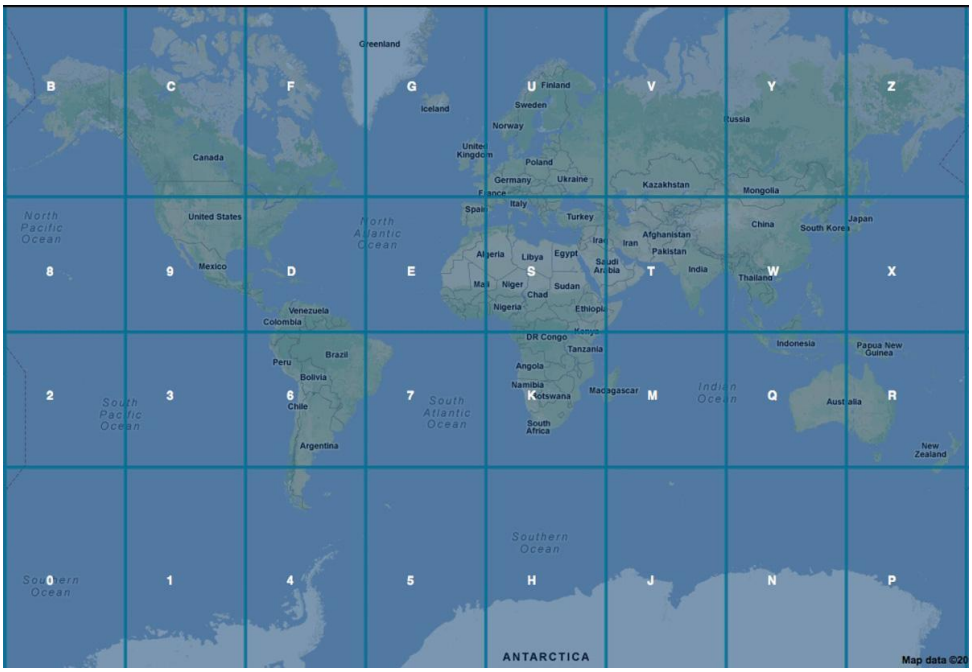
Una vez que la tabla estaba creada, se ha procedido a la carga de los datos de los 50.000 clientes que engloba este proyecto, los cuales estaban contenidos en 10 ficheros con formato csv.

La carga se ha realizado con la siguiente instrucción:

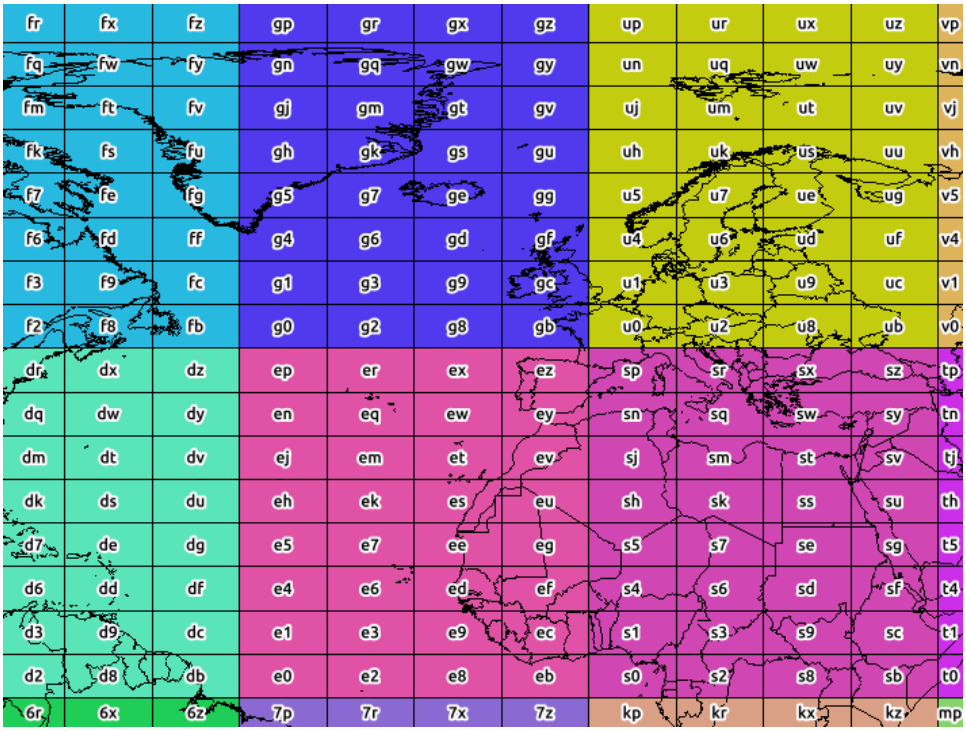
```
COPY world.datosclientes (id,
  name,
  surname,
  gender,
  nationalid,
  birthday,
  streetaddress,
  city,
  zipcode,
  country,
  telephonecountrycode,
  telephonenumber,
  latitude,
  longitude,
  emailaddress,
  title,
  occupation,
  company,
  username) FROM '/home/ubuntu/Data/xaa' WITH HEADER = TRUE;
```

Posteriormente, se detectó la necesidad de añadir información técnica de los clientes en cassandra, esta información fué el código geohash de cada sensor y el nombre del dashboard donde iba a estar ubicado ese sensor en grafana.

➔ Geohash: Es un código de geolocalización que divide el planeta en áreas rectangulares.



A su vez, tiene una estructura jerárquica, es decir, a mayor número de dígitos mayor precisión se obtiene sobre un punto determinado, igualmente a menor número de dígitos menor será la



precisión y mayor el área seleccionado.

En este proyecto se ha implementado un algoritmo para calcular el código geohash de cada alarma dados su latitud y longitud, datos que ya se tenían almacenados en cassandra, por lo que se ejecutó un script que obtenía de cassandra estos datos, calculaba su geohash y lo insertaba en una nueva columna. Este dato es necesario para

almacenar en la bbdd InfluxDB series temporales con el id del sensor, su estado y su geohash para que posteriormente desde grafana se realicen agregaciones y agrupaciones por este código para dibujar los sensores en el mapa.

➔ Dashboards: Con el objetivo de conseguir una buena escalabilidad y usabilidad a la hora de monitorizar y gestionar los paneles de visualización, se ha decidido dividir el planeta en varias zonas utilizando códigos geohash. Con esto se han creado las siguientes zonas: NorthAmerica, SouthAmerica, Oceania, Mediterranean, North, UK, NorthEurope, CentroEurope y Africa. Cada uno de estas zonas tiene definidos uno o varios códigos geohash de tamaño 1, 2 o 3 dígitos, dependiendo de la precisión necesaria.

5.2 ELB

Se ha creado un balanceador en AWS (ELB) con el nombre de DNS *TFM-982204392.us-west-2.elb.amazonaws.com.* Este balanceador será el punto de entrada a todo el sistema, es decir, será el destino donde todos los sensores envíen los mensajes. Amazon asegura HA en su servicio de load balancer distribuyendolo en tres zonas de disponibilidad.

Basic Configuration

Name:	TFM
*** DNS name:**	TFM-982204392.us-west-2.elb.amazonaws.com (A Record)
Scheme:	internet-facing
Availability Zones:	subnet-9cb136d5 - us-west-2a, subnet-a3a4bbfb - us-west-2c, subnet-c989e4ae - us-west-2b

Port Configuration

Port Configuration:	**80 (HTTP) forwarding to 19092 (HTTP)** Stickiness: Disabled

La configuración del balanceador se ha realizado enlazando su puerto 80 con el puerto 19092 de los dos nodos donde está instalado el proxy. De esta manera, todas las peticiones que entran por HTTP y por este puerto se redirigen al puerto de escucha del proxy.

Load Balancer Protocol	Load Balancer Port	Instance Protocol	Instance Port	Cipher	SSL Certificate
HTTP	80	HTTP	19092	N/A	N/A

Por su parte, el balanceador está continuamente monitorizando estos puertos a través de health checks, en caso de que se detecte que alguno de estos puertos no está escuchando se da de baja ese endpoint y se continúa con el resto de nodos disponibles. Si más tarde el servicio del proxy

comienza a estar activo de nuevo, el balanceador vuelve a registrar este endpoint en su lista de listeners. En este caso, el número de nodos que hay conectados al balanceador son 2.

Instance ID	Name	Availability Zone	Status
i-0b3134c353a8693cb	kafka-proxy1	us-west-2c	InService
i-0046e7ba84d9ee8ae	kafka-proxy2	us-west-2a	InService

5.3 Proxy

La utilización de un proxy en este proyecto está pensada para que realice la tarea de obtener todos los mensajes que llegan procedentes del balanceador a través de HTTP y los inserta en Kafka en formato json. En nuestro sistema, estará instalado un proxy en cada nodo que contenga un broker de kafka, de esta manera el proxy recibe los mensajes en el puerto 19092 y los redirige en formato json al 9090 del mismo nodo, de esta manera evitamos más tráfico de red innecesario.

Se ha utilizado la imagen de docker *trovit/docker-hub:proxy-kafka.* Para configurar este servicio, es necesario indicarle los endpoints de conexión con zookeeper y con kafka. El docker-compose de este servicio es el siguiente, únicamente cambiaría de un nodo a otro la IP de Kafka a la que está conectado:

```
kafka-pixy:
  image: trovit/docker-hub:proxy-kafka
  ports:
   - "19092:19092"
  environment:
   KAFKA_PROXY_HOSTS: ip-172-31-6-74:9092
   ZOOKEEPER_PROXY_HOSTS: ip-172-31-32-37:2181,ip-172-31-33-75:2181,ip-172-31-42-
129:2181
  depends_on:
   - kafka
  command: ["./wait-for-kafka.sh"]
  volumes:
   - /var/lib/kafka-pixy/data
```

5.4 Kafka

Kafka es un sistema distribuido de paso de mensajes que se ha convertido en indispensable en todo sistema de procesamiento de datos en tiempo real. Permite construir tuberías de datos a través de topics en modalidad productor/suscriptor.

En nuestro sistema se han instalado dos broker de kafka y se han creado los topics *ENTRY, OK y COUNT,* la configuración de estos topics está pensada para tener alta disponibilidad por lo que se ha establecido replicación igual a 2. El número de particiones se ha establecido en 2 para que los topics estén balanceados de igual forma en los brokers.

A continuación se muestra el docker-compose de configuración del servicio. Se han establecido parámetros como la conexión con zookeeper, los recursos destinados al uso de de la jvm, la configuración por defecto de los topics, el id de cada broker…

```
kafka:
  image: wurstmeister/kafka:0.10.2.1
  ports:
   - "9092:9092"
  environment:
   KAFKA_AUTO_CREATE_TOPICS_ENABLE: 'true'
   KAFKA_ZOOKEEPER_CONNECT: ip-172-31-32-37:2181,ip-172-31-33-75:2181,ip-172-31-42-
129:2181
   KAFKA_HEAP_OPTS: "-Xmx1G -Xms512m"
   KAFKA_ADVERTISED_HOST_NAME: ip-172-31-6-74
   KAFKA_DEFAULT_REPLICATION_FACTOR: 2
   KAFKA_NUM_PARTITIONS: 2
   KAFKA_BROKER_ID: 100
   KAFKA_DELETE_TOPIC_ENABLE: 'true'
   JMX_PORT: 9999
  volumes:
   - /var/run/docker.sock:/var/run/docker.sock
```

Por último se ha habilitado el puerto 9999 para exportar métricas de jmx, de esta manera posteriormente en grafana tendemos un dashboard específico de kafka en el que podremos ver métricas de los mensajes recibidos por topic, mensajes de salida, mensajes almacenados...

Para esto se utiliza el servicio de prometheus-jmx-exporter https://github.com/rama-nallamilli/kafka-prometheus-monitoring

```
kafka-jmx-exporter:
  build: ./prometheus-jmx-exporter
  ports:
   - "8080:8080"
  links:
   - kafka
  environment:
   - JMX_PORT=9999
   - JMX_HOST=kafka
   - HTTP_PORT=8080
   - JMX_EXPORTER_CONFIG_FILE=kafka.yml
```

En el fichero de kafka.yml se le indican todas las métricas que se exportarán por jxm y que serán expuestas en el puerto 8080 para que el servicio de prometheus ubicado en el servidor de grafana los recoja. Posteriormente en el apartado de grafana se mostrará la configuración de prometheus.

5.5 Zookeeper

Tanto el servicio del proxy como kafka necesitan de un servicio externo de coordinación, esta dependencia la solucionamos introduciendo Zookeeper para esta tarea.

Zookeeper necesita tener un número impar de nodos para garantizar mayoría en las votaciones que realiza internamente, por lo que se ha desplegado un clúster de 3 servidores para asegurar así también la alta disponibilidad. Debido a la dependencia que tienen otros servicios en zookeeper, es requisito que este servicio se ejecute en primer lugar, para asegurar este punto, se ha configurado zookeeper como servicio del sistema para que se arranque siempre al levantar el servidor. El resto de servicios, se desplegarán manualmente a través de un playbook de ansible una vez que todos los nodos del entorno estén arrancados.

El comando ejecutado en los nodos de zookeeper para arrancar este servicio es el siguiente.

```
docker run --name zookeeper1 -d --restart=always -p 2181:2181 -p 2888:2888 -p 3888:3888 -v /var/lib/zookeeper:/var/lib/zookeeper -v /var/log/zookeeper:/var/log/zookeeper jeygeethan/zookeeper-cluster ip-172-31-32-37,ip-172-31-33-75,ip-172-31-42-129 1
```

El último parámetro que se incluye en el comando es la lista con los nodos que componen el clúster de zookeeper y un identificador que representa el número del nodo dentro del clúster.

5.6 Flink

Flink es el motor de procesamiento de todo el sistema. Se encarga de procesar todos los mensajes que los sensores han emitido y que previamente han pasado por el ELB - Proxy - Kafka.

Flink proporciona gran escalabilidad y procesamiento continuo en tiempo real, es tolerante a fallos y proporciona un gran rendimiento. También posee conectores con muchas bbdd, entre ellas cassandra, por lo que se ajusta en gran medida a la necesidad de este proyecto.

El mensaje que recibe y procesa Flink tiene el siguiente aspecto:

```
{"42959",
"true",
"false",
"25.5",
"30.0"}
```

Los campos se corresponden respectivamente con: Id del sensor, movimiento, humo, temperatura y temperatura máxima.

La lógica necesaria para este sistema y que Flink deberá de implementar es la siguiente:

a. Si el mensaje llega con sus campos (movimiento, humo y temperatura) correctos, se descartará el mensaje.

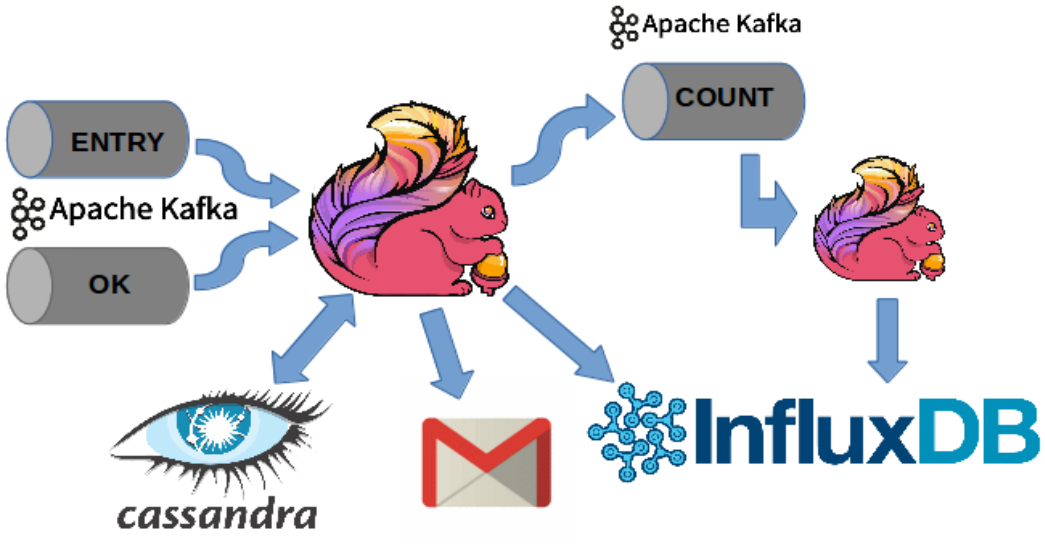
b. Si se recibe un mensaje con el campo movimiento activo, se activará el protocolo de actuación y Flink deberá enviar un correo electrónico a un buzón de monitorización y control de alarmas indicando que esa alarma se ha activado, dicho mensaje irá compuesto por todos los datos de ese cliente, por último se deberá actualizar un panel de visualización mostrando en el mapa la alarma que se acaba de activar quedando diferenciada del resto de sensores.

c. Si un mensaje llega con el campo de temperatura o de humo activo, Flink almacenará el estado de ese sensor y esperará tres mensajes con este campo activo antes de activar el protocolo de actuación y se repetirán los pasos del punto II.

d. Se deberá de habilitar un punto de entrada para permitir a los sensores enviar un mensaje cuando recuperen su estado correcto después de que su alarma haya sido activada.

e. Se desea visualizar en tiempo real el histórico de alarmas que han sido activadas en una determinada franja de tiempo según su tipo.

Para poder cumplir con estos puntos, se ha propuesto implementar la siguiente arquitectura:



Los puntos de entrada a Flink son dos topics de Kafka:

- **ENTRY**: Los sensores envían todos los mensajes con su estado.
- **OK**: Se reciben únicamente los mensajes de los sensores que han recuperado su estado normal después de haberse activado su alarma.

Los mensajes del topic ENTRY que han llegado con el campo movimiento activo o se reciben tres mensajes con el campo humo o temperatura activo, se escribirán en el topic COUNT. A su vez, se activará el protocolo de actuación y Flink lanzará una petición a cassandra para obtener todos los datos del cliente que tiene contratada esa alarma, posteriormente enviará un correo electrónico a un buzón de monitorización y se escribirá el estado de la alarma en InfluxDB para actualizar el dashboard de Grafana.

Por otro lado, Flink recoge los mensajes del topic COUNT, (que ya son todos con su alarma activa), le aplica un filtro para conocer el tipo del mensaje, paraleliza por su tipo (así se asegura que todos los mensajes del mismo tipo van a ir al mismo task de flink para poder obtener la suma de todos ellos), le aplica una ventana de 1 minuto, hace la suma y escribe en InfluxDB. De esta forma se crea un time series en esta bbdd con los datos históricos de la activación de alarmas según su tipo para poder visualizarlo fácilmente en Grafana a través de una gráfica.

Se ha desplegado un clúster de Flink con un JobManager y dos TaskManager para conseguir alta disponibilidad.

El docker compose del Job Manager es el siguiente:

```
jobmanager:
  image: flink:latest
  expose:
    - "6123"
    - "6124"
  ports:
    - "8081:8081"
    - "6123:6123"
    - "6124:6124"
  command: jobmanager
  environment:
    - JOB_MANAGER_RPC_ADDRESS=172.31.18.139
```

El docker compose de los task manager es:

```
flink-taskmanager:
  image: flink:latest
  expose:
    - "6121"
    - "6122"
  command: taskmanager
  environment:
    - JOB_MANAGER_RPC_ADDRESS=172.31.18.139
```

5.7 Grafana

Grafana es el sistema de visualización escogido para este proyecto. Proporciona una interfaz de administración sencilla y permite la utilización de varias bases de datos como Prometheus e InfluxDB especializadas en agregación de series temporales que son ideales para integrarlas con sistemas de monitorización, además grafana facilita la importación y creación de multitud de dashboards, como por ejemplo el worldmap-panel, creado por CartoDB y que integrado con InfluxDB es el principal panel de visualización de este proyecto, ya que será la interfaz donde se mostrará en tiempo real el estado de todas las alarmas del sistema y se podrá consultar algunos datos de los sensores. Además de este panel de control de las alarmas, se han implementado varios dashboards de monitorización completa del sistema, se dispone de un panel con todas las métricas de los dockers y de los servidores con gráficas como el consumo de memoria, cpu y de red, entre otros. También se ha realizado un dashboard para todas las métricas de Flink y otro para kafka, en el que se muestran datos como el número de mensajes recibidos, procesados, enviados…

En la siguiente tabla se muestran las bbdd utilizadas para cada tipo de dashboard.

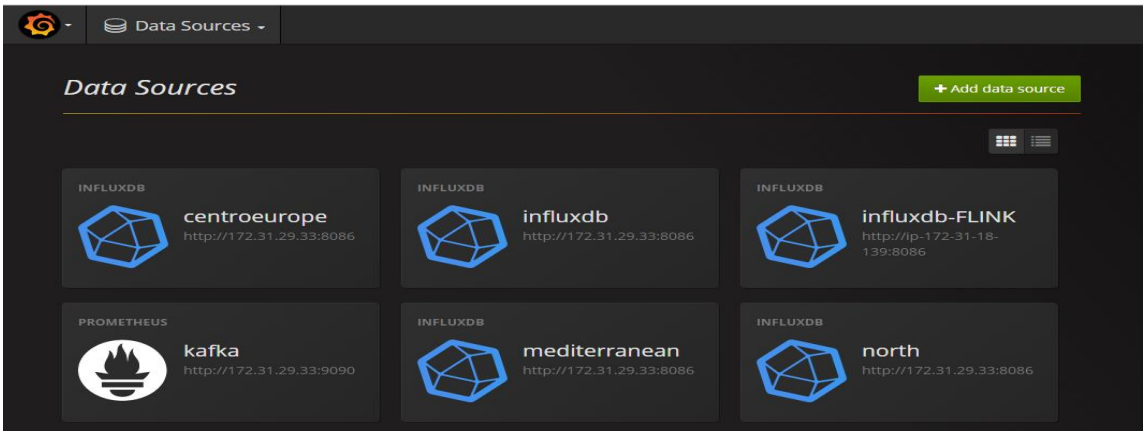
BBDD	Dashboard
InfluxDB	Control de los sensores
InfluxDB	Métricas de Flink
Prometheus	Métricas de los dockers
Prometheus	Métricas de Kafka

El docker-compose utilizado para desplegar Grafana, InfluxDB y Prometheus es el siguiente:

```
prometheus:
  ports:
    - "9090:9090"
  image: prom/prometheus:0.18.0
  volumes:
    - ./mount/prometheus:/etc/prometheus
grafana:
  image: grafana/grafana
  ports:
    - "3000:3000"
  environment:
    GF_SECURITY_ADMIN_PASSWORD: "alvaro"
  volumes_from:
    - graf-db
influxdb:
  ports:
    - "8086:8086"
    - "8083:8083"
```

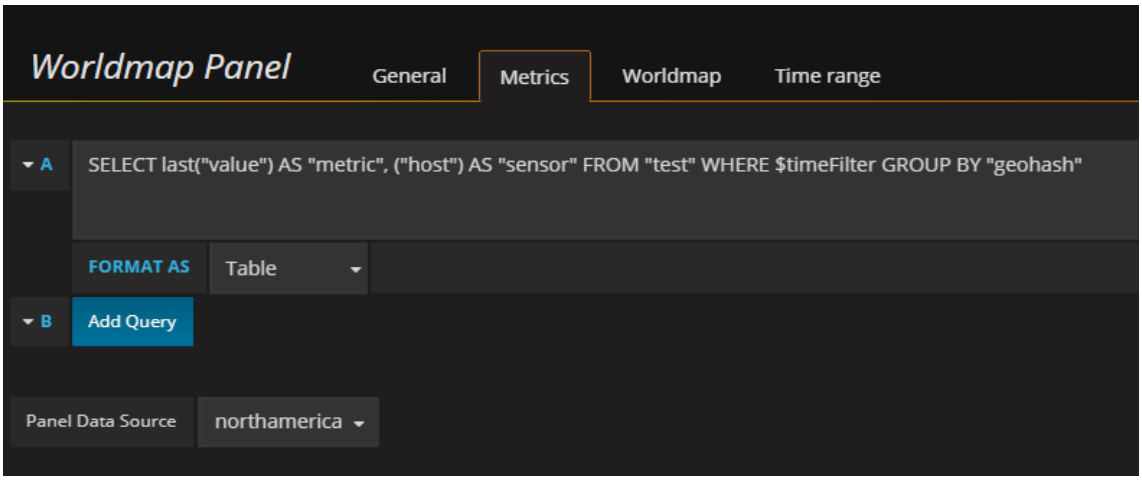
```
  image: influxdb
  volumes:
    - /var/lib/influxdb
  environment:
    INFLUXDB_ADMIN_ENABLED: "true"
```

Grafana permite añadir orígenes de datos y enlazarlos con una bbdd desplegada previamente simplemente con indicar el tipo de base de datos, el endpoint de conexión, el nombre de la instancia y de la tabla, el tipo de acceso y los credenciales para acceder. Es posible crear varios orígenes de datos en grafana y enlazarlos con la misma instancia pero con diferentes tablas.



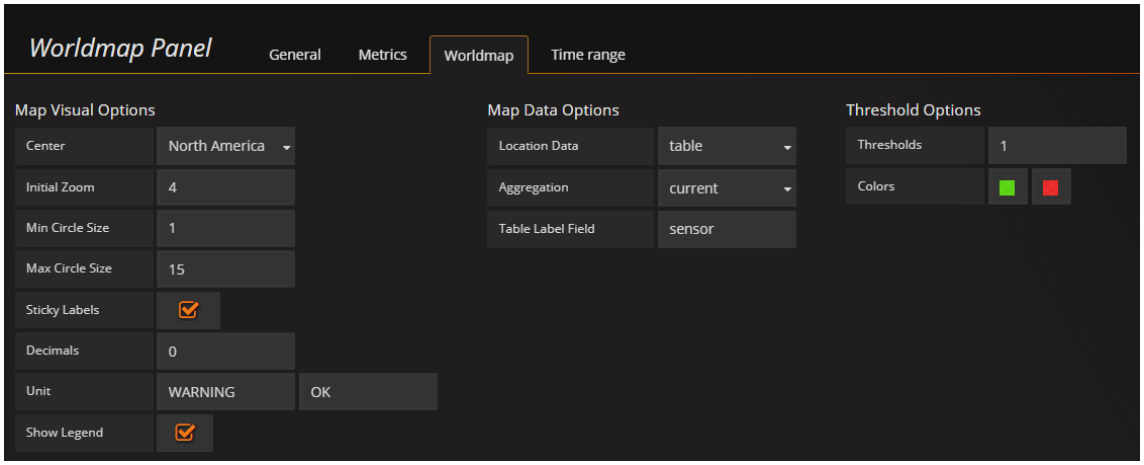
Una vez que se han creado los orígenes de datos es la hora de crear los dashboards. Grafana permite importar multitud de paneles de una manera sencilla, indicando el identificador del dashboard que se puede encontrar en www.grafana.com , importando directamente un fichero json con la estructura del dashboard o comenzar a crearlo desde cero.

Para la creación de los mapas, se ha utilizado el plugin *WorldMap Panel v0.0.17,* creado por CartoDB. En la configuración del dashboard es necesario indicarle la bbdd que va a utilizar y las queries que tiene que realizar sobre ella para mostrar la información. En la siguiente imagen se muestra la consulta sobre la bbdd InfluxDB que realiza grafana cada vez que se refresque el panel.

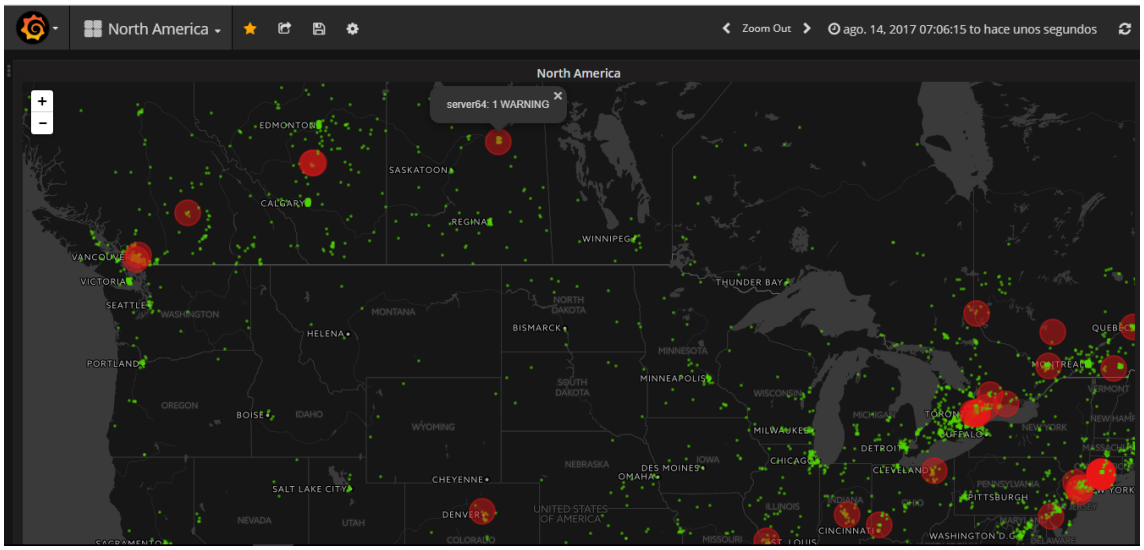


En esta bbdd Flink almacena en series temporales el valor de cada sensor, (activo o inactivo), el identificador de la alarma y su código geohash para poder pintarlo en el mapa posteriormente. Grafana lanza la query y agrega los resultados por series temporales, quedándose con la última serie de cada sensor.

En la siguiente imagen se muestra tanto la configuración con respecto a la visualización de los datos como el tamaño de los puntos que van a representar los sensores, se habilitan labels que van a mostrar información de cada sensor y se utiliza un código de colores para representar el estado de los mismos.



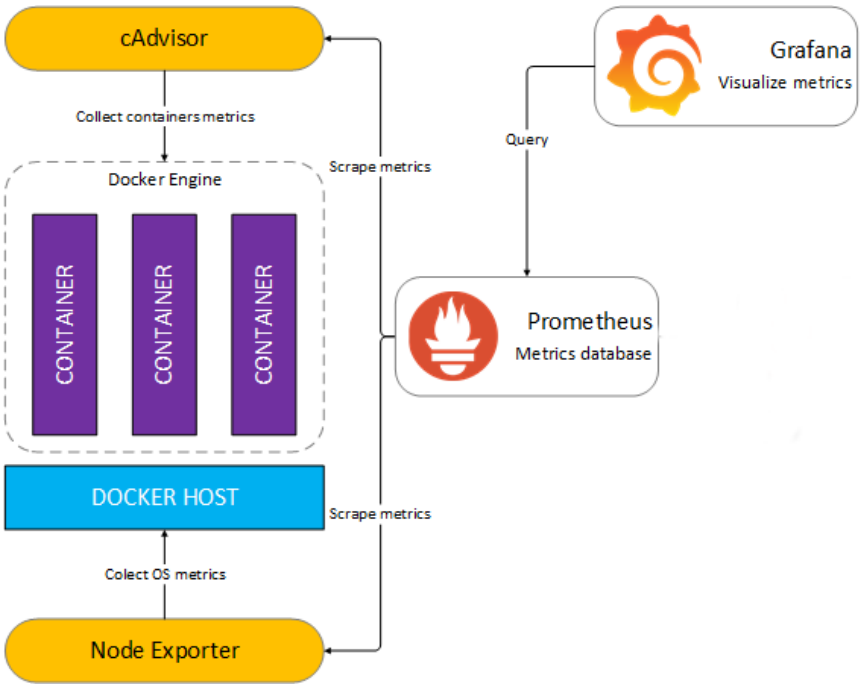
Por último, se muestra el resultado del panel de control de los sensores de América del norte.



5.8 Node Exporter, cAdvisor y Prometheus-docker

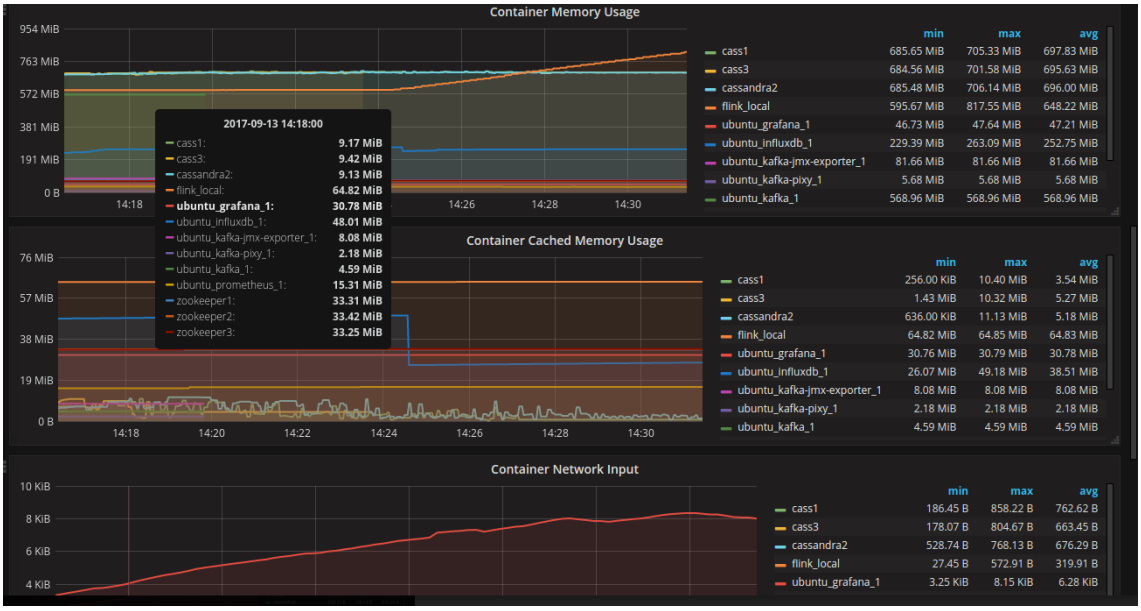
Un punto importante en todo proyecto o sistema informático es la monitorización de sus componentes y servicios.

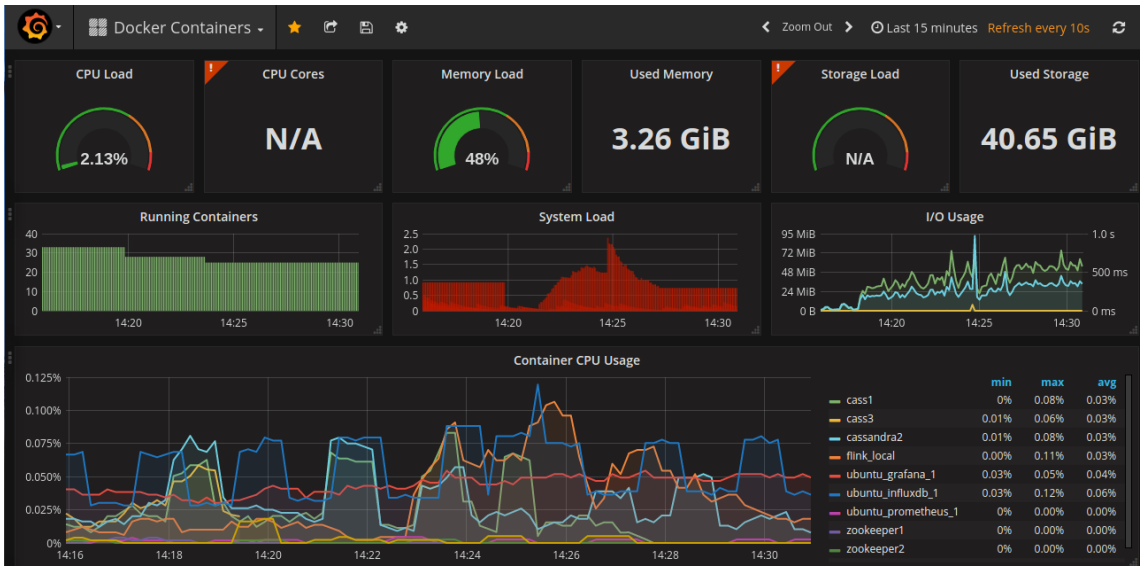
La solución propuesta en cuanto a monitorización es la siguiente:



Se han desplegado contenedores de cAdvisor y Node Exporter en todos los nodos que conforman el sistema. Estos contenedores serán los encargados de obtener las métricas de todos los servicios y recursos de cada servidor y exportarlos en un puerto, de este modo Prometheus se encargará de recoger todos estos valores vía HTTP y almacenarlos en series temporales para que sean accesibles desde Grafana.

Por último, en Grafana se han configurado varios dashboard para que ataquen contra este Prometheus ejecutando queries y pintando varias gráficas donde se pueden visualizar en tiempo real el estado de todos los nodos, contenedores y servicios que componen el sistema. El resultado de estos paneles es el siguiente:





Se ha utilizado el módulo de Stefan Prodan dockprom adaptándolo a nuestro caso de uso *https://github.com/stefanprodan/dockprom.git*

El docker-compose del Prometheus es el siguiente:

```
prometheus-docker:
  image: prom/prometheus
  container_name: prometheus-docker
  volumes:
    - ./dockprom/prometheus/:/etc/prometheus/
    - prometheus_data:/prometheus
  command:
    - '-config.file=/etc/prometheus/prometheus.yml'
    - '-storage.local.path=/prometheus'
```

```
    - '-alertmanager.url=http://alertmanager:9093'
    - '-storage.local.memory-chunks=100000'
  restart: unless-stopped
  expose:
    - 9091
  ports:
    - 9091:9090
  labels:
    org.label-schema.group: "monitoring"
```

En la configuración de este Prometheus, se le indica todos los nodos y los puertos donde tiene que recoger las métricas, es decir, todos los nodos del sistema:

```
global:
 scrape_interval:     15s
 evaluation_interval: 15s

 # Attach these labels to any time series or alerts when communicating with
 # external systems (federation, remote storage, Alertmanager).
 external_labels:
    monitor: 'docker-host-alpha'

# Load and evaluate rules in this file every 'evaluation_interval' seconds.
rule_files:
 - "targets.rules"
 - "host.rules"
 - "containers.rules"

# A scrape configuration containing exactly one endpoint to scrape.
scrape_configs:
 - job_name: 'nodeexporter'
   scrape_interval: 5s
   static_configs:
    - targets: ['172.31.33.75:9100']
    - targets: ['172.31.32.37:9100']
    - targets: ['172.31.42.129:9100']
    - targets: ['172.31.8.143:9100']
    - targets: ['172.31.7.201:9100']
    - targets: ['172.31.1.114:9100']
    - targets: ['172.31.29.33:9100']
    - targets: ['172.31.18.139:9100']
    - targets: ['172.31.6.74:9100']
    - targets: ['172.31.40.250:9100']
    - targets: ['172.31.30.19:9100']
    - targets: ['172.31.29.60:9100']
```

```
- job_name: 'cadvisor'
  scrape_interval: 5s
  static_configs:
    - targets: ['172.31.33.75:8080']
    - targets: ['172.31.32.37:8080']
    - targets: ['172.31.42.129:8080']
    - targets: ['172.31.8.143:8080']
    - targets: ['172.31.7.201:8080']
    - targets: ['172.31.1.114:8080']
    - targets: ['172.31.29.33:8080']
    - targets: ['172.31.18.139:8080']
    - targets: ['172.31.6.74:8081']
    - targets: ['172.31.40.250:8081']
    - targets: ['172.31.30.19:8080']
    - targets: ['172.31.29.60:8080']

- job_name: 'prometheus'
  scrape_interval: 10s
  static_configs:
    - targets: ['ip-172-31-29-33:9091']
```

Por último, el docker-compose que contienen todos los agentes del sistema con los servicios de cAdvisor y Node Exporter para generar las métricas es el siguiente:

```
nodeexporter:
  image: prom/node-exporter
  container_name: nodeexporter
  volumes:
    - /proc:/host/proc:ro
    - /sys:/host/sys:ro
    - /:/rootfs:ro
  ports:
    - "9100:9100"
  command:
    - '-collector.procfs=/host/proc'
    - '-collector.sysfs=/host/sys'
    - '-collector.filesystem.ignored-mount-points=^/(sys|proc|dev|host|etc)($$|/)'
  restart: unless-stopped
  expose:
    - 9100
  labels:
    org.label-schema.group: "monitoring"

cadvisor:
  image: google/cadvisor:latest
```

```
  container_name: cadvisor
  volumes:
    - /:/rootfs:ro
    - /var/run:/var/run:rw
    - /sys:/sys:ro
    - /var/lib/docker/:/var/lib/docker:ro
  ports:
    - "8080:8080"
  restart: unless-stopped
  expose:
    - 8080
  labels:
    org.label-schema.group: "monitoring"
```

6. Resultados

Como resultado final, se han realizado varias pruebas end to end para comprobar que los resultados son los esperados y medir el rendimiento general de todo el sistema. Las características de los nodos que componen el entorno han sido los siguientes.

Name	Instance type
cassandra1	t2.small
cassandra2	t2.small
cassandra3	t2.small
edge	t2.micro
flink_taskmanager	t2.small
flink_jobmanager	t2.small
flink_jobmanager	t2.small
grafana	t2.small
kafka_proxy	t2.small
kafka_proxy	t2.small
grafana	t2.small
zookeeper1	t2.small
zookeeper2	t2.small
zookeeper3	t2.small

El nodo edge es el nodo de acceso a la plataforma y desde él se puede levantar y parar el sistema por completo utilizando el siguiente playbook de ansible:

```
---
- hosts: cassandra
  remote_user: ubuntu
  tasks:
    - name: starting cassandra
      shell: docker-compose start
- hosts: cassandra
  remote_user: ubuntu
  tasks:
    - name: starting cassandra
      shell: docker start cassandra
```

```
- hosts: zookeeper
  remote_user: ubuntu
  tasks:
    - name: starting zookeeper
      shell: docker-compose start
- hosts: kafka
  remote_user: ubuntu
  tasks:
    - name: starting kafka
      shell: docker-compose start
- hosts: grafana
  remote_user: ubuntu
  tasks:
    - name: starting grafana
      shell: docker-compose start
- hosts: flink
  remote_user: ubuntu
  tasks:
    - name: starting flink cluster
      shell: docker-compose start
- hosts: kafka
  remote_user: ubuntu
  tasks:
    - name: starting pixy
      shell: docker start ubuntu_kafka-pixy_1
- hosts: flink_jobmanager
  remote_user: ubuntu
  tasks:
    - name: starting flink influx
      shell: docker start influxdb
```

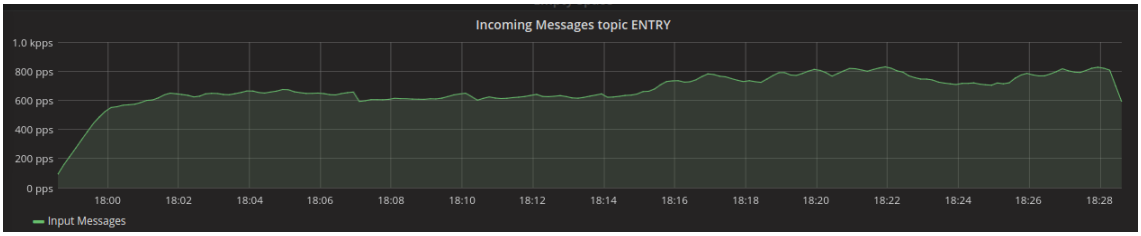
Una vez que todo el sistema está levantado, desde este mismo nodo se ejecutan varias aplicaciones java concurrentes que son las que simulan constantemente el envío de los mensajes por parte de los sensores, su lógica incluye un componente aleatorio que va activando alarmas. Cada uno de estos ficheros jar envían al ELB alrededor de 400 mensajes por segundo.

A continuación, se detalla el rendimiento del sistema dependiendo del número de aplicaciones ejecutadas concurrentemente.

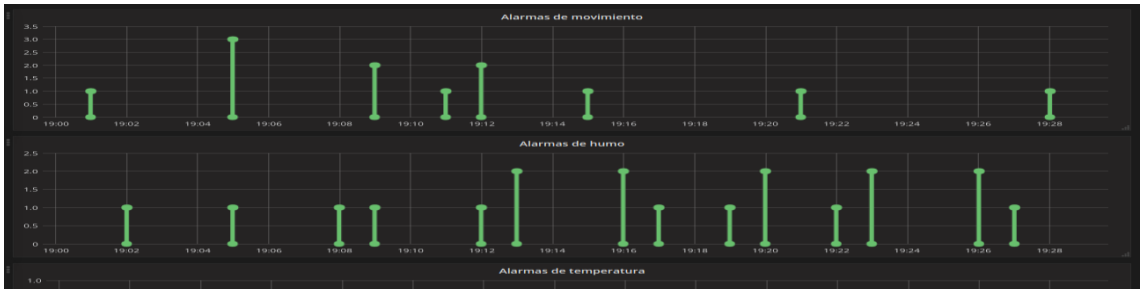
6.1 Caso 1

Apps concurrentes	2
Mensajes totales	1.245.480
Mensajes / segundo	800 m/s
Duración	30 minutos
Resultado	OK
Alarmas activadas	655
Alarmas recuperadas	410

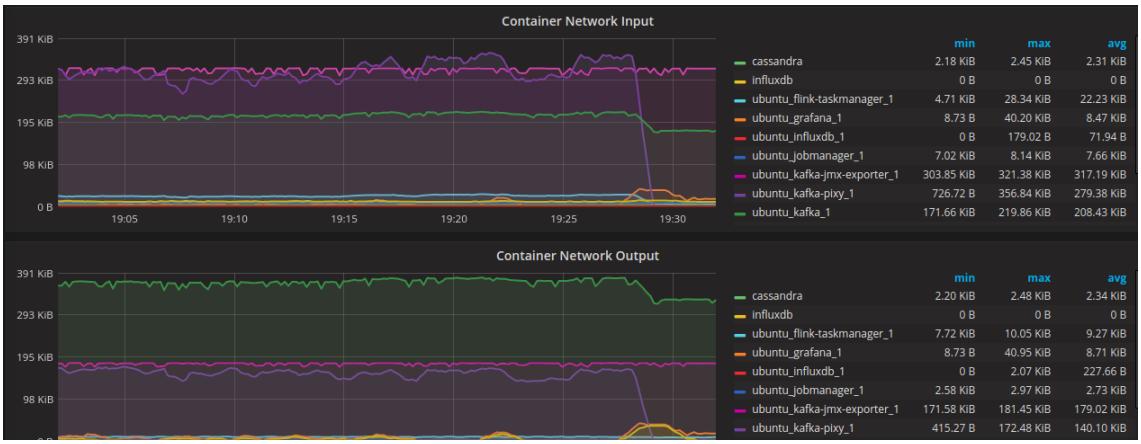
A continuación se muestra el ratio de mensajes por segundo enviados:



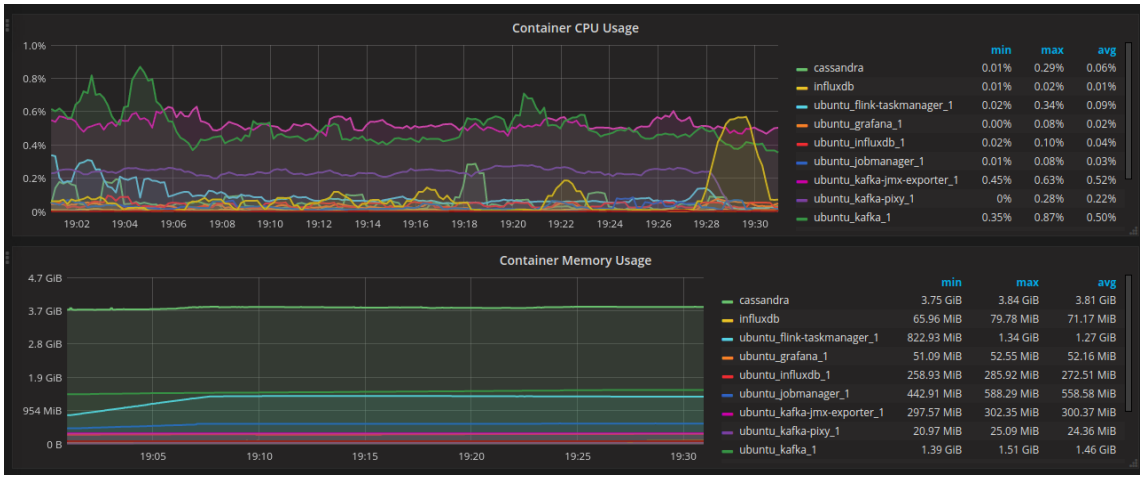
El histórico de alarmas activadas por minuto según su tipo:



Se muestran las métricas de red generadas. Los servicios de Kafka, el proxy y el node exporter son los que más consumo de red han generado:



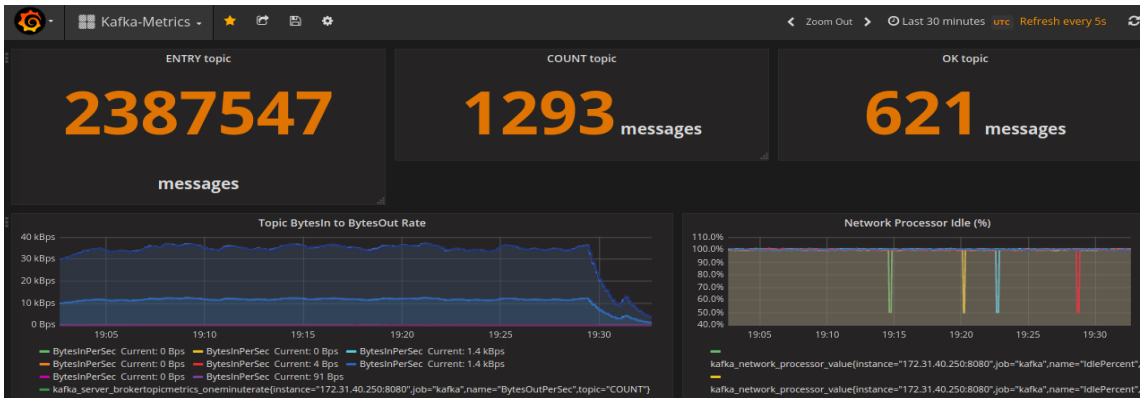
Mientras que Cassandra y Flink son los que más recursos en cuanto a CPU y memoria consumen:



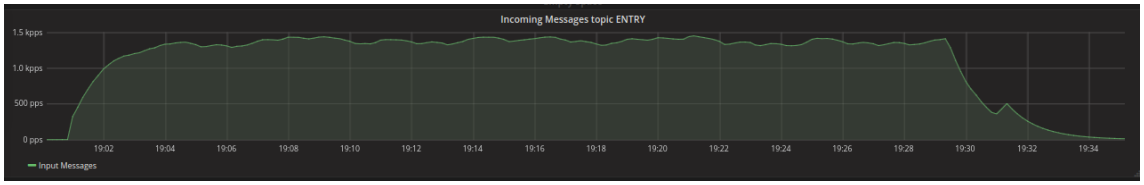
6.2 Caso 2

Apps concurrentes	4
Mensajes totales	2.387.547
Mensajes / segundo	1400 m/s
Duración	30 minutos
Resultado	OK
Alarmas activadas	1293
Alarmas recuperadas	621

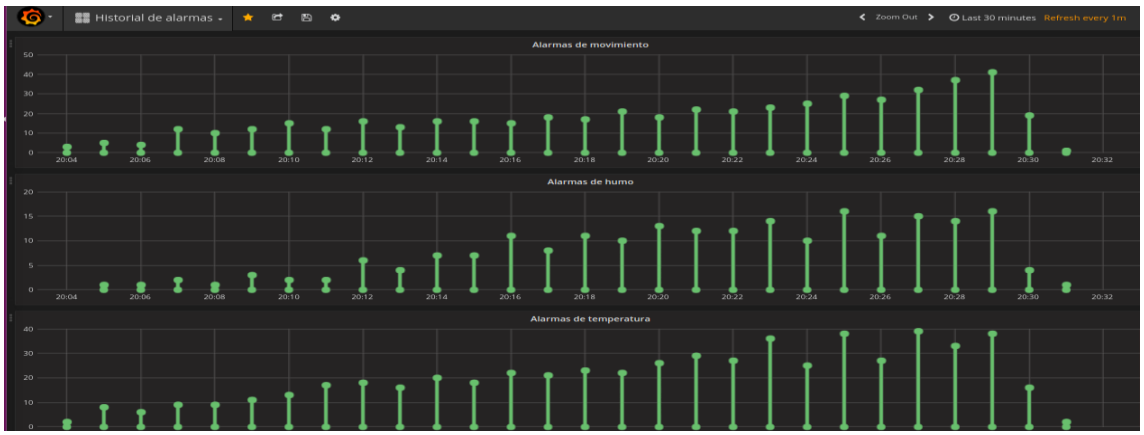
El número total de mensajes enviados, alarmas activadas y recuperadas:



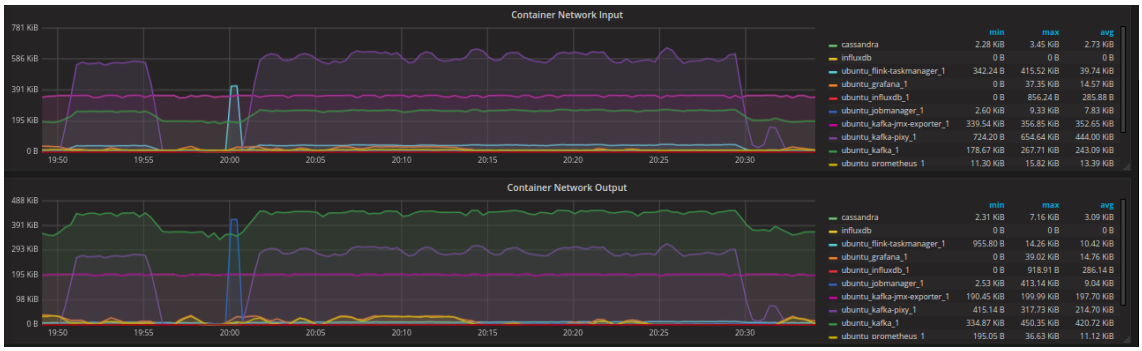
El ratio de mensajes por segundo enviados:



El histórico de alarmas activadas por minuto según su tipo:



Al igual que antes, los servicios de Kafka, el proxy y el node exporter son los que más consumo de red han generado:



En cuanto al uso de CPU y memoria, Cassandra, Flink y Kafka son los que más consumen.