Universidad Politécnica
de Madrid

**Escuela Técnica Superior de
Ingenieros Informáticos**

Grado en Ingeniería Informática

Trabajo Fin de Grado

Implementación de una Herramienta Visual para el Despliegue y Configuración de un Sistema de Data Streaming

Autor: Eduardo Checa Gismero
Tutor(a): Marta Patiño Martínez

Madrid, junio de 2020

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática
Título: Implementación de una Herramienta Visual para el Despliegue y Configuración de un Sistema de Data Streaming
Mes Año

Autor: Eduardo Checa Gismero

Tutor:
Marta Patiño Martínez
Departamento de Sistemas Distribuidos
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

El procesamiento de eventos complejos es una disciplina de la analítica de datos en el mundo del BigData. Trata de analizar grandes volúmenes de datos de una forma rápida y eficaz obteniendo así información valiosa para una organización.

El CEP es una herramienta diseñada para ejecutar queries sobre streamings de datos en un entorno distribuido. Estas queries se construyen a base de operadores de entrada/salida y de transformación de datos. El objetivo de este proyecto, ha sido crear un módulo del CEP llamado CEP-RequestHandler para poder recibir peticiones HTTP con queries en formato json y transformarlas para que el CEP sea capaz de interpretarlas y ejecutarlas.

Abstract

Complex event processing is an analytical data discipline within BigData. It attempts to analyse large volumes of data in a fast and efficient way, therefore obtaining valuable information for an organization.

CEP is a tool designed to execute queries about data streams in a distributed environment. These queries are built based on input/output and data transformation operators. The aim of this project was to create a module of the CEP called CEP-RequestHandler able to receive HTTP requests with json format queries and transform them so that the CEP can interpret and execute them.

Tabla de contenidos

1 Introducción 1
1.1 Complex Event Processing 1
1.2 CEP 2
1.2.1 Operadores con estado 3
1.2.2 Operadores sin estado 3
1.2.3 Arquitectura del CEP 4
1.3 Aplicación Web 5
1.4 OBJETIVO 6
1.5 ESTADO DEL ARTE 6
1.5.1 API 6
1.5.2 REST 7
1.5.3 Servicios Web 8
2 Desarrollo 10
2.1 PROBLEMA PLANTEADO 10
2.2 REQUISITOS 10
2.2.1 Requisitos funcionales 10
2.2.2 Requisitos técnicos 11
2.3 SOLUCIÓN 12
2.3.1 CEP-HTTPEndPoint 12
2.3.2 Algoritmos 15
3 Resultados y conclusiones 18
3.1 Pruebas 18
3.1.1 Pruebas técnicas 18
3.1.2 Pruebas funcionales 18
3.2 Conclusión 22
4 Bibliografía 23

1 Introducción

En este documento se pretende explicar el desarrollo del proyecto 'Implementación de una herramienta visual para el despliegue y configuración de un sistema de data *streaming'*. Para ello se hablará del panorama actual de los sistemas de información relacionados con el flujo de datos en BigData, la necesidad de implementar un proyecto como este, el desarrollo del mismo y las pruebas realizadas.

BigData es un término utilizado para referirse al procesamiento de grandes volúmenes de datos a una velocidad y fiabilidad tales que sería inviable realizarlo con herramientas convencionales. El mundo del BigData está en auge. Cada vez se hace más evidente la necesidad de sacar información útil de cualquier evento que ocurre en la sociedad y esto hace que aparezcan nuevas soluciones a los problemas que van surgiendo en el camino del desarrollo de estas nuevas tecnologías, para poder ser capaces de exprimir al máximo las enormes cantidades de datos que generamos día a día.

Esta necesidad de procesar datos a tan alto rendimiento surge cuando a principios de los 2000 [5] las empresas empiezan a darse cuenta de la cantidad de datos que generaban los usuarios de compañías como YouTube [13] o Facebook [14] y que, procesando y analizando dichos datos de la forma correcta podían convertir esa información en mejoras de sus modelos de negocio. Para explotar toda esta información se empezaron a desarrollar tecnologías y herramientas específicas para el procesamiento, análisis y visualización de los datos. Además, las organizaciones se dan cuenta de que pueden sacar información, no solo de usuarios de ciertas aplicaciones, sino también de balizas meteorológicas, sensores de predicción de errores en trenes o aviones, el valor de las empresas en el mercado, etc. Por lo que las posibilidades de utilizar estas nuevas tecnologías se incrementan enormemente.

Una forma muy común de obtener dichos datos es mediante procesos de *streaming*. El *streaming* es la transmisión en directo o a tiempo real de datos en pequeños bloques de información de forma secuencial y que puede provenir de una o varias fuentes. Es común encontrarlo en plataformas de video o audio, pero también se utiliza para analizar y procesar información en tiempo real. Una forma eficiente de procesar estos datos es mediante el procesamiento de eventos complejos.

1.1 Complex Event Processing

Un evento es un suceso, algo que ocurre. En este panorama, un evento puede ser un correo que llega, un click de un usuario en una página web, una alarma que se activa, un pago con tarjeta o cualquier situación que produzca algún tipo de información.

El Complex Event Processing o 'procesamiento de eventos complejos' es una disciplina de la analítica de seguimiento y análisis de los flujos de información (procesamiento de datos) en forma de eventos, para poder extraer valor de dicho análisis. Son procesos que manejan grandes volúmenes de datos y por ello requieren de una infraestructura escalable y distribuida, entrando así en el

conjunto de metodologías y procesos que forman parte del BigData. Al procesar los datos en tiempo real, otorga a las organizaciones la posibilidad de tomar decisiones de negocio muy rápido. La Figura 1 es una representación gráfica del flujo de eventos que utiliza procesamiento de eventos complejos.

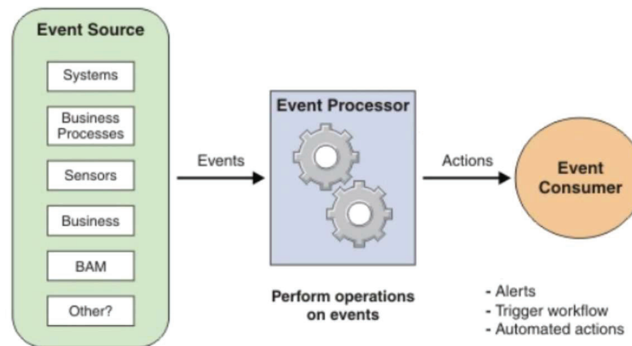


Figura 1. Complex Event Processing [6]

Una herramienta muy utilizada para este tipo de procesos es Apache *Flink:* [13] un motor de procesamiento de datos distribuidos en batch (Procesamiento de datos históricos y estáticos) y en stream (flujo continuo de datos), tolerante a fallos y que puede ejecutarse en las distribuciones más comunes de BigData, como Hadoop [16] o Cloudera [17].

1.2 CEP

El CEP es un proyecto del Laboratorio de Sistemas Distribuidos de la Universidad Politécnica de Madrid cuyo objetivo es analizar y procesar los flujos de datos en entornos distribuidos y ser capaz de repartir la carga de trabajo entre los diferentes nodos del sistema de la manera más eficiente posible. Es una herramienta distribuida y escalable, lo que significa que puede ejecutarse en varias máquinas (o nodos) a la vez y en paralelo. A medida que se añaden más maquinas el procesamiento de los datos se distribuye en todos los nodos del sistema lo que se traduce en una mayor velocidad de procesamiento.

Las queries son grafos acíclicos cuyos nodos son los operadores y las aristas los streams. Las queries se dividen en subqueries, que son desplegadas en cada *Instance Manager* del sistema. Los eventos llegan al sistema en forma de tuplas: conjuntos ordenado e inmutable de elementos o tipos de datos. Existen operadores de entrada (DataSources) que pueden ser de tipo SocketDataSource para recibir datos por una conexión a un *socket*, o FileDataSource para realizar pruebas desde un fichero de datos. Además, existen los operadores que procesan los eventos y se dividen en tres tipos: orientados sin estado, con estado y custom.

1.2.1 Operadores con estado

Son operadores que almacenan eventos en ventanas de tamaño o de tiempo. Cuando se acumula un número de eventos o pasa un tiempo determinado la ventana se desliza y se realizan operaciones sobre los datos almacenados dependiendo del tipo de operador:

- aggregate: devuelve agregaciones de tuplas que se definen con funciones de agregación en la configuración del componente.
- join: recibe tuplas de dos flujos de entrada. Usando dos ventanas deslizantes, empareja cada tupla de una secuencia con las tuplas mantenidas en la ventana deslizante asociada con la otra secuencia.
- selfjoin: funciona igual que el join, pero lee los datos del mismo stream.

Cada operador con estado genera una subquery lo que va dividiendo la query en varias subqueries. Cuando se despliega la query, se crea una instancia por cada subquery por defecto, a menos que el usuario decida cambiar la forma en la que se distribuye la query, en cuyo caso podrá ordenarle al CEP que ejecute la distribución deseada.

1.2.2 Operadores sin estado

Estos operadores filtran y manipulan las tuplas de datos que van llegando, una a una, y devuelven una o más tuplas como resultado de esa transformación.

- map: aplica transformaciones sobre los campos de la tupla.
- filter: aplica una condición o predicado a la tupla. Si esta no cumple dicha condición, no se retorna como resultado.
- demux: asigna la tupla a una de las múltiples salidas configuradas, cada una con una condición o predicado. La tupla es retornada por la salida cuya condición cumpla.
- union: tiene varias entradas de tuplas con mismo esquema de datos. Las une y las devuelve en un único stream de tuplas.
- Splitter: divide los datos de un flujo de entrada, en dos streams de salida.

Capaces de ejecutar sentencias SQL sobre datos almacenados en fuentes externas.

- select: selecciona los campos deseados, de la tabla externa de datos.
- insert: inserta las tuplas resultantes en la tabla destino configurada.
- delete: ejecuta un borrado en la tabla destino usando los campos definidos en el operador.
- update: actualiza la tabla destino usando los campos definidos en el operador.

Además de los operadores ya mencionados, existe otro tipo de operador personalizable o 'custom'. Es un operador no definido. El usuario puede diseñarlo en función de las necesidades del proyecto.

1.2.3 Arquitectura del CEP

La arquitectura del CEP permite escalabilidad a nivel de streaming, es decir, puede recibir eventos de varios flujos distintos y a nivel de subqueries, lo que implica que puede procesar queries sencillas o muy elaboradas y distribuirlas de la forma más óptima entre los nodos aprovechando los recursos del sistema.

La función de la herramienta es abstraer al usuario de la distribución de la carga de procesado de los datos, mientras que este solo tiene que encargarse de elaborar sus queries. Funciona como una caja negra, el usuario se centra en desarrollar las queries y el CEP se encarga de procesar la información y devolver el resultado.

Los principales componentes del CEP son el *Orchestador*, los *Instance Managers*, las *CEP Instances* y el *Reliable Registry*, tal y como se muestra en la Figura 2.

El orquestador (Orchestator) monitoriza el sistema para detectar posibles fallos y controlar el flujo de datos. También se encarga de distribuir las queries por los Instance Manager. Un Instance Manager es un proceso que se encarga de configurar las instancias del CEP (CEP Instances) y les asigna las queries. Estas instancias son los procesos que se encargan del procesado de los eventos. Por último, el Reliable Registry es un cluster con Zookeeper que permite guardar de manera permanente, tolerante a fallo y altamente disponible la información relativa al estado del CEP, tanto de cada uno de los componentes como las consultas registradas y desplegadas.

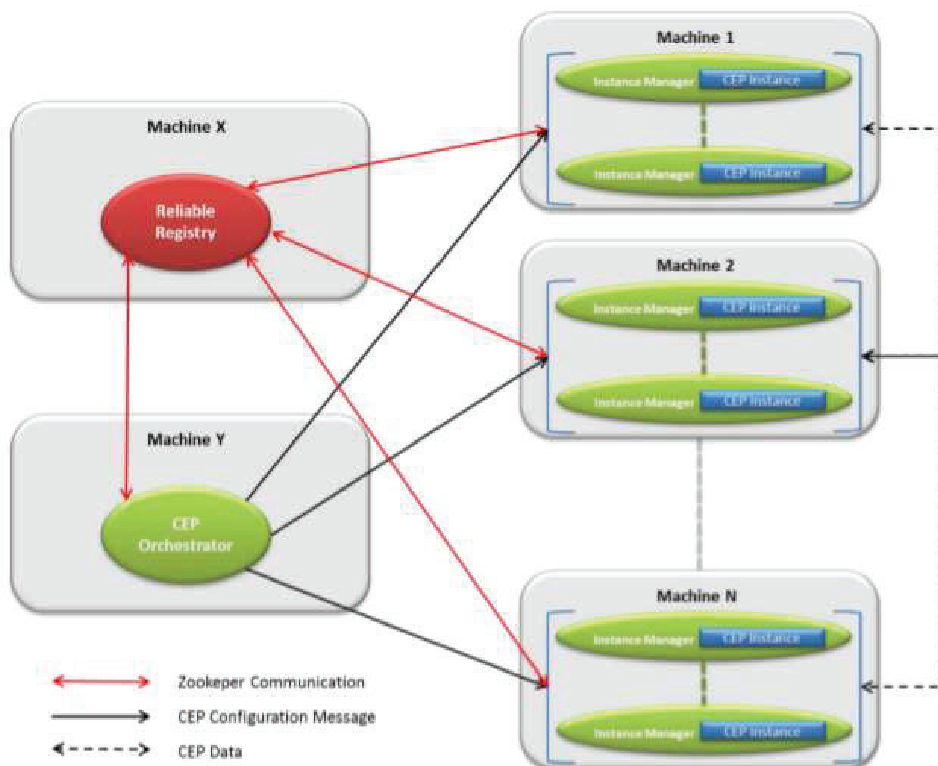


Figura 2. Componentes del CEP [3]

1.3 Aplicación Web

La aplicación se ha diseñado para dar facilidad al usuario a la hora de manejar sus queries. Para ello, el usuario va creando la query uniendo varios operadores entre sí en una interfaz gráfica (GUI) en la que dichos operadores aparecen como elementos visuales y los cuales se pueden configurar desde la propia aplicación. Una vez creada la query, el usuario puede, entre otras cosas, desplegar la query en el CEP para que este empiece a ejecutarla en el sistema y así poder visualizarla.

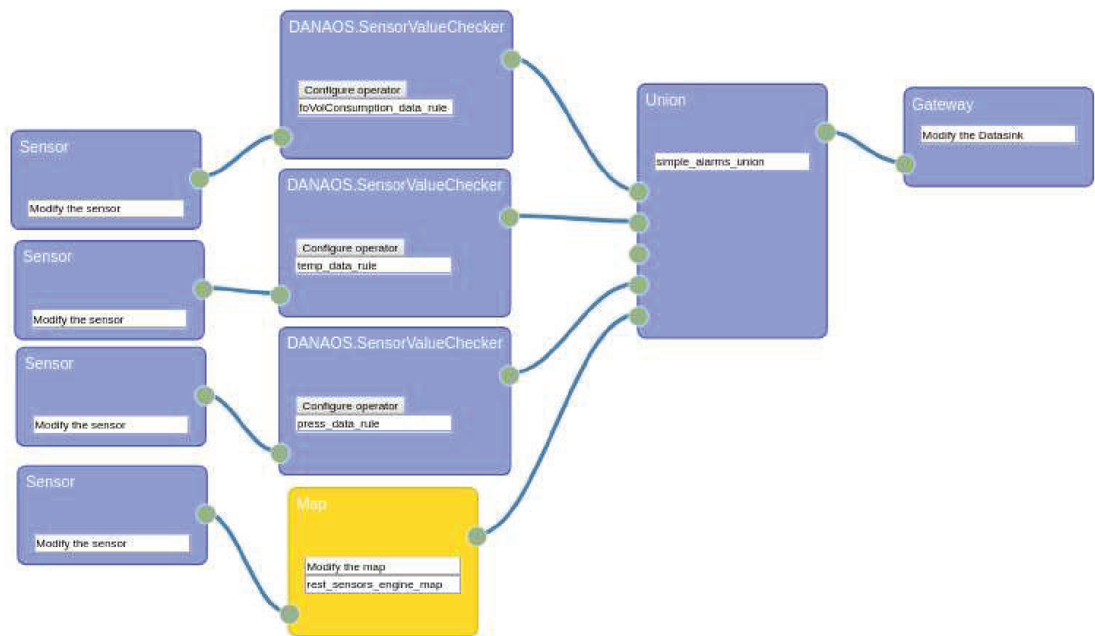


Figura 3. Visualización de una query

Existe un script para instalar todos los componentes mencionados. Una vez instalados, se utiliza un script de despliegue para arrancar de forma coordinada todos los componentes.

1.4 OBJETIVO

El objetivo de este trabajo es diseñar el CEP-HTTPEndPoint, un módulo del CEP que trabaje de forma independiente y que sirva para poder conectar la aplicación web con el CEP y así conseguir que las queries que el usuario ha diseñado utilizando componentes visuales (cajas y flechas) formen una query que el sistema pueda interpretar y ejecutar. La solución debe ser lo más genérica y escalable posible, ya que el número de queries distintas que se pueden llegar a formar es muy elevado. Por otro lado, se deben diseñar las pruebas necesarias para el correcto funcionamiento del componente y su integración con el sistema.

Se pretende que el componente sea capaz de interpretar cualquier combinación de elementos que formen una query en la web, de forma que tanto la *query* más sencilla como la más compleja puedan interpretarse y traducirse para que el CEP sea capaz de entenderla. Además, el servicio debe ser configurable para poder ser desplegado en un sitio distinto al CEP o a la aplicación web, de forma que los tres componentes puedan ejecutarse en diferentes nodos y sean independientes los unos de los otros.

1.5 ESTADO DEL ARTE

1.5.1 API

API o application programming interface (interfaz de programación de aplicaciones), es un conjunto de funciones, procesos o rutinas que ofrecen las bibliotecas de una aplicación software para que dos o más aplicaciones puedan comunicarse, dejando a la aplicación propietaria de la API como una caja negra que procesa las peticiones, pero no muestra cómo está implementada.

Hace años, una aplicación podía haber sido desarrollada en C++ o Java, y ejecutarse bajo Unix, Windows o un computador diferente al resto y no había una forma sencilla de intercomunicar dichas aplicaciones. Gracias al desarrollo de las API, se hizo posible compartir información entre aplicaciones con diferentes arquitecturas, lenguajes de programación, o cualquier sistema operativo, conectadas en red o a través de Internet.

Con la aparición de los sistemas distribuidos, el desarrollo de las API se vio impulsado ya que aligeraba las tareas de los programadores de mover datos de un lado a otro para su procesamiento con distintos tipos de computadora. El Middleware orientado a mensajes (como el IBM MQSeries) fue un avance importante ya que permitía a los diferentes sistemas de una organización comunicarse entre sí, mediante el uso de las API. Además, gracias a la programación orientada a objetos, se vio que cada vez era más cómodo utilizar una API para permitir el acceso remoto a instancias de objetos.

A principios de siglo se dispara el uso de las API web, lo que facilita la comunicación entre máquinas a través de la red. Esto hace que empresas del tipo de Facebook, Google, Amazon o eBay empiecen a utilizar esta tecnología y sirvan de modelos para otras empresas, las cuales empiezan a implementar este tipo de soluciones una vez visto el potencial que tenía.

Un ejemplo de uso es por ejemplo la API de Google Maps. Si una empresa o individuo quiere desarrollar una aplicación de geolocalización por medio de GPS

para mostrarla al usuario, necesita de todos los mapas de calles, carreteras, caminos, etc. Para ello Google dispone de una API web para poder explotarla y recibir mapas de la zona deseada. Esto hace que el desarrollador no necesite crear sus propios mapas, lo que tendría un coste elevadísimo. Con esto accede a los recursos que Google Maps le proporciona para poder desarrollar su aplicación y Google le cobra por ello.

Existen cuatro áreas principales en las que las API han tenido un papel importante:

- E-Commerce
- Social media
- Cloud Computing
- Movilidad

Las API se pueden separar entre locales o remotas. Las API locales suelen usarse para abstraer las capas inferiores y superiores del software y se utilizan a nivel local, es decir, no hacen uso de la red. Un ejemplo de API local es la API de Linux, que permite al programador acceder a los recursos y a las funciones del kernel de Linux. Otro ejemplo de API local, sería la de un teléfono móvil. Si el desarrollador está programando una aplicación en Android y se quiere añadir una función de vibración, bastaría con llamar a la función de vibración de la API del teléfono.

Las API remotas, por el contrario, acceden a recursos o rutinas que se alojan en máquinas distintas y que no tienen por qué estar alojadas en la misma red. La arquitectura más común hoy en día de API remota es la arquitectura REST (REpresentational State Transfer).

1.5.2 REST

REST usa HTTP para obtener datos o generar operaciones sobre datos utilizando el formato XML o JSON, este último más utilizado hoy en día. JSON (JavaScript Object Notation) es un formato de texto muy simple de intercambio de datos basado en JavaScript. Es totalmente independiente al lenguaje de la aplicación que lo utiliza, pero sirve para tener un modelo de como viajan los datos entre usuarios y servicios web, de forma que tanto el cliente como el servidor puedan tener aplicaciones totalmente distintas, en lenguajes distintos. Se caracteriza por tener un conjunto de pares de clave/valor, dentro de los cuales pueden viajar listas/vectores de datos. Es un formato de intercambio de información muy utilizado en la actualidad, gracias a su simplicidad y a lo fácil que resulta explotarlo con las librerías que existen hoy en día. Además, es texto plano, por lo que es legible por una persona y facilita su manipulación.

REST representa una alternativa a los protocolos de intercambio de datos como SOAP que, a pesar de ser muy potente, resulta muy complejo. Algunas características de la arquitectura REST son:

- Protocolo cliente/servidor sin estado: cada petición HTTP contiene toda la información necesaria para ser ejecutada en el servidor, lo que libra al cliente y al servidor de tener que guardar sesión o estado de peticiones anteriores. Esto se caracteriza también por generar respuestas idénticas a peticiones idénticas.

- Se basa en cuatro operaciones básicas: GET para consultar recursos, POST para crear recursos, PUT para modificarlos y DELETE para eliminarlos.
- Cada recurso, está asociado a una única URI (Uniform Resource Identifier) diferente al resto de los recursos y puede generar varios formatos posibles, ya que cada aplicación cliente puede necesitar un formato diferente (xml, json…).

La forma más común hoy en día de acceder a una API remota es a través de servicios web.

1.5.3 Servicios Web

El WC3 (World Wide Web Consortium) define un servicio Web como un sistema software diseñado para soportar interacciones máquina a máquina a través de la red. Es decir, son una forma de intercambiar información entre dos o más aplicaciones software alojadas en diferentes puntos. Se llamaron 'servicios Web' porque fueron diseñados para alojarse en un servidor Web y ser explotados por otros usuarios a través de Internet.

A diferencia del modelo Cliente/Servidor tradicional, los servicios web comparten la lógica del negocio, los datos y los procesos, por medio de una API en la red. Los Web Services no interactúan de forma directa con el usuario/cliente, son una manera de explotar recursos de una forma más libre, con programas ejecutables, páginas Web o aplicaciones mediante IP, puerto y *endpoints*, sin la necesidad de estar sujeto a las restricciones del modelo Cliente/Servidor.

Uno de sus usos más comunes es permitir la comunicación dentro de los diferentes servicios de las empresas y entre las empresas y sus clientes, ya que permiten a las organizaciones intercambiar datos sin necesidad de conocer los detalles de cómo están implementados sus respectivos Sistemas de Información. Dado el nivel de integración que aportan a las aplicaciones, los Web Services son muy populares y han mejorado enormemente los procesos de negocios en las empresas estos últimos años.

Los servicios web no están ligados a ningún lenguaje de programación o sistema operativo. No necesitan usar navegadores web ni el lenguaje de especificación HTML. Son un método estandarizado para integrar aplicaciones WEB y aportar una funcionalidad específica al usuario.

Los servicios REST son servicios web sujetos a las restricciones de la arquitectura REST, tal y como se describe en la Figura 4.

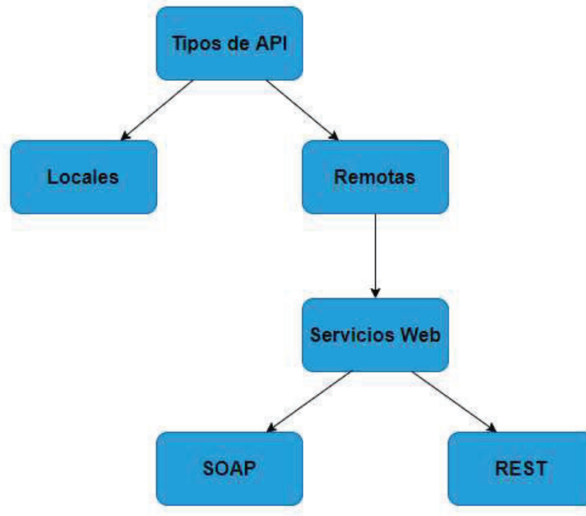


Figura 4. Tipos de API

2 Desarrollo

2.1 PROBLEMA PLANTEADO

La única forma que existe para que un usuario pueda comunicarse con el CEP, es mediante su interfaz Java. Utilizando las librerías del CEP, se debe implementar un cliente Java que cree los json que se mandan al sistema para realizar todas las acciones deseadas: RegisterQuery, DeployQuery… Esto es un impedimento si el usuario no tiene ciertos conocimientos de programación, por lo que se necesita de una solución que simplifique la comunicación cliente-CEP.

Como parte de la solución, se empezó a desarrollar la aplicación web para aportar un componente visual al desarrollo de las queries y facilitar así el trabajo al usuario. El problema entonces era cómo comunicar la aplicación web con el CEP. Se decidió entonces que había que crear un componente que pudiese interpretar peticiones provenientes no solo de la aplicación web, sino de otros posibles clientes o herramientas de visualización. El componente debe poder recibir e interpretar todas las diferentes peticiones y procesarlas para convertirlas en peticiones que el CEP sea capaz de entender. Así mismo, debe poder interpretar las respuestas que le llegan del CEP y mandarlas en un formato adecuado al solicitante inicial (en este caso la aplicación web).

En resumen, se quiere crear un componente que se acople al CEP que pueda recibir solicitudes dirigidas al mismo sin depender del tipo de cliente que implementa la solicitud.

2.2 REQUISITOS

A continuación, se detalla la lista de requisitos que la solución debe contemplar según se acordó con el equipo de desarrollo del departamento.

2.2.1 Requisitos funcionales

La solución tiene que ser capaz de atender las siguientes peticiones REST:

1- Register query: petición POST que recibe una query en formato json y la registra en el CEP. Devuelve la respuesta del CEP. Los posibles operadores que vienen el json son:

   a. map: El operador map convierte cada tupla de entrada en una de salida aplicando transformaciones en los campos.
   b. aggregate: Este operador lee eventos durante una ventana de tiempo o de datos predefinida y los junta mediante funciones de agregado, para devolverlos en la salida.
   c. filter: Este operador filtra las tuplas de entrada, devolviendo únicamente las que satisfacen el predicado definido en el componente.

d. dmux: Envía las tuplas de entrada a todas las salidas si se cumple el predicado definido en el componente.
e. join: El operador Join lee tuplas de dos flujos de entrada. Usando dos ventanas deslizantes, empareja cada tupla de una secuencia con las tuplas mantenidas en la ventana deslizante asociada con la otra secuencia.
f. selfjoin: es un operador join que lee datos del mismo flujo de entrada.
g. splitter: El splitter divide los datos de un flujo de entrada, en dos flujos de salida.
h. union: Este operador une las tuplas de entrada por el campo indicado, el cual debe tener el mismo esquema o tipo de datos, en una sola salida.
i. readsql: Este operador realiza queries mediante lenguaje SQL a bases de datos, conectado con JSBC.
j. storesql: Este operador realiza escrituras mediante lenguaje SQL a bases de datos, conectado con JSBC.

2- Deploy query: petición POST que recibe una descripción del despliegue de la query en formato json para que el CEP lo ejecute en los nodos correspondientes. Devuelve la respuesta del CEP.
3- Retrieve deployed querys IDs: petición GET que envía una solicitud al CEP para saber que querys están desplegadas en ese momento y las retorna. Devuelve la respuesta del CEP.
4- Undeploy query: petición POST que recibe una query y genera un json que manda al CEP para que este pare la ejecución de dicha query. Devuelve la respuesta del CEP.
5- Unregister query: petición POST que recibe una query y comunica al CEP que debe dar de baja dicha query del sistema. Devuelve la respuesta del CEP.
6- Retrieve registered querys: petición GET que solicita al CEP las querys registradas en el sistema y le devuelve la lista recuperada.
7- Status: petición GET que solicita el estado del CEP y lo retorna.

2.2.2 Requisitos técnicos

1- El sistema debe poder arrancarse en un servidor http, concretamente un servidor Jetty para poder mantenerse en paralelo con el proyecto de la web. (Esto ahorrará coste y esfuerzo de mantenimiento a las personas encargadas del proyecto global, una vez este TFG esté finalizado.)

2- Toda comunicación con el CEP debe hacerse utilizando la interfaz Java del cliente del CEP para una correcta comunicación con este.

3- La solución debe ser un módulo del CEP que pueda desplegarse de forma independiente al resto de componentes.

Dado que tanto el CEP como la parte de la aplicación web son proyectos separados que han ido desarrollándose a la vez que el este proyecto, algunos de los requisitos tuvieron que ir adaptándose a los avances o las modificaciones que se producían en los mismos.

Por ejemplo, en un primer momento no se pensó en utilizar un servidor Jetty para la implementación del servicio, pero se decidió utilizar tecnologías comunes para simplificar las tareas de mantenimiento o mejoras del mismo.

2.3 SOLUCIÓN

Se decide desarrollar el componente CEP-HTTPEndPoint que consta de un servicio REST con varias funcionalidades y que es independiente del CEP, de forma que se puede implementar en otros proyectos si fuera necesario.

2.3.1 CEP-HTTPEndPoint

Mediante llamadas REST y utilizando un formato de mensaje como el json, el componente se desentiende de cómo esté implementado el cliente, el cual lo único que tiene que hacer son llamadas a los diferentes recursos del servicio mandando todo lo necesario en un json con una estructura concreta.

El módulo se desarrolla en Java 8 para estar alineado con la versión de Java que utiliza el CEP, debido a que tienen librerías en común. Con esta versión se pueden implementar servicios REST utilizando, en este caso, el framework Jersey.

Jersey es un *framework open source* que se utiliza para la implementación de servicios REST mediante JAX-RS: un api de Java que permite crear servicios web con la ayuda de anotaciones del tipo @Path, @GET, @POST… Lo que aporta sencillez a la hora de implementar un servicio REST. En la figura 3 vemos un ejemplo de función a la que se accede mediante una petición POST al endpoint "/registerQuery" y espera y devuelve un JSON. LA Figura 4 espera una petición GET en el *endpoint* "/status" y devuelve un JSON en la salida.

```
@POST
@Path("/registerQuery")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response registerQueryHandler(String request){
      String result = handler.registerQuery(request);

      return Response.ok(result)
              .header("Access-Control-Allow-Origin", "*")
              .header("Access-Control-Allow-Methods", "POST")
              .header("Access-Control-Allow-Headers", "Content-Type,Authorization")
              .allow("OPTIONS")
              .build();
  }
```

Figura 3. JAX-RS Ejemplo1

```java
@GET
@Path("/status")
@Produces(MediaType.APPLICATION_JSON)
 public Response getStatus() {
    String result = handler.getStatus();

        return Response.ok(result)
                .header("Access-Control-Allow-Origin", "*")
                .header("Access-Control-Allow-Methods", "GET")
                .header("Access-Control-Allow-Headers", "Content-Type,Authorization")
                .allow("OPTIONS")
                .build();
    }
```

Figura 4. JAX-RS Ejemplo2

Para desplegar el componente, se utiliza un servidor Jetty. Es un servidor http y un contenedor de *servlets* de software libre, basado en Java. Se utiliza para crear servidores web de forma fácil y rápida. El funcionamiento de Jetty se basa en la definición y utilización de un conjunto de componentes conectados entre sí. Una colección de conectores, que son los encargados de aceptar conexiones HTTP; un conjunto de *handlers* que manejan las solicitudes provenientes de las conexiones, produciendo respuestas; un *pool* de *threads*, desde los cuales el servidor tomará recursos para realizar el trabajo. Los tiempos de carga suelen ser cortos, ya que usa multiconexión HTTP y su consumo de memoria es muy reducido. Además, puede aumentar de forma dinámica el número de conexiones que tiene abiertas lo que lo hace escalable y perfecto para un entorno distribuido. Jetty permite desplegar varios servicios desarrollados por separado y totalmente independientes los unos de los otros, cada uno con una ruta específica.

El servicio posee varias rutas o *endpoints* para acceder a los recursos que cubren las necesidades funcionales (RegisterQuery, DeployQuery, GetStatus…). Está preparado para recibir peticiones en formato json, que se convierten o '*mapean*' a objetos java compatibles con las clases que utiliza el CEP. El módulo está dividido en tres secciones claramente diferenciables para un mantenimiento más sencillo.

CEPHttpRequestService:

Recibe las peticiones y tiene todos los *endpoints* configurados en ella. Cada uno de estos corresponde a una funcionalidad distinta. Aquí se realiza la primera validación de la petición, comprobando si el JSON que entra está bien formado. Además, se establecen las propiedades del cross-origin resource sharing o CORS, que establece los permisos de acceso a recursos del servidor por parte del usuario o cliente en navegadores. Por defecto, un servidor tiene sus datos restringidos para cualquier petición de un servidor externo. Con esta política se pueden asignar permisos a servidores "amigos", para que puedan realizar consultas a nuestro servicio. Es una medida de seguridad frente a programas que utilizan JavaScript o CSS y cuyas políticas de seguridad son muy deficientes. La información sobre quién tiene acceso al servicio y quién no, va en la cabecera HTTP y viene informado en estas variables:

Para cada *endpoint* se establece el método por el cual se tiene que acceder, un parámetro @Path que indica cómo acceder al recurso, y un formato de entrada y de salida:

```
@POST
@Path("/unregisterQuery")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
```

Figura 5. Ejemplo parámetros endpoint

En el ejemplo de la Figura 5, para acceder a este servicio debería mandarse un JSON válido mediante una petición HTTP con método POST a la *IP:puerto* del servidor con la ruta '/unregisterQuery'. Como respuesta, el servicio responderá con otro JSON con el resultado de la operación, o con un mensaje de error si hubiera ido algo mal o si la petición no se hubiera hecho correctamente.

CEPHandlerManager:

Realiza la función de convertir el JSON de entrada en el objeto java necesario para realizar esta operación. Aquí es donde se controla que la información del mensaje viene bien estructurada e informa de posibles errores en la estructura de los datos de entrada, devolviendo así un mensaje de error al origen avisándole de que lo que ha mandado no tiene un formato válido. Si la estructura de los datos es correcta, la petición avanza.

QueryConstructor:

El *QueryConstructor* recibe la petición, una vez su estructura ha sido validada, y empieza a crear el objeto (operador) que será necesario pasarle al CEP, con la información que nos llega en la petición. En este paso se comprueba que los datos sean válidos, es decir, se comprueban elementos nulos, tipos de datos coherentes, o valores que se salen de los límites establecidos.

Si la petición es válida y todo ha ido bien, se procede a mandarle el objeto java al CEP. Para ello, el componente tiene que tener configurados los parámetros de IP, puerto, etc que necesita para comunicarse con el CEP.

2.3.2 Algoritmos

A continuación, se describen los algoritmos de cada funcionalidad del CEP-HTTPEndPoint.

Algoritmo 1: registerQuery

Recibe una petición HTTP con un json que contiene una query para registrar en el CEP. Se comprueba que tanto el json como la query son válidos y se manda al CEP. Se devuelve la respuesta del CEP.

request: petición HTTP que llega al servicio

```
1.#json <- getData(request)
2.if isJson(json)
3.      if isRegisterQueryFormat(#json)
4.              #jsonCEP <- setStreams(#json)
5.              #jsonCEP <- setOperators(#json)
6.              #respone <- registerQueryCEP(#jsonCEP)
7.      else
8.              #respose <- errorMsg()
9.else
10.     #respose <- errorMsg()
11.return #response
```

Algoritmo 2: unregisterQuery

Recibe una petición HTTP con un json indicando qué query se quiere dar de baja en el CEP. Se comprueba que el json es válido y se manda al CEP. Se devuelve el resultado obtenido por el CEP.

request: petición HTTP que llega al servicio

```
1.#json <- getData(request)
2.if isJson(json)
3.      if isUnregisterQueryFormat(#json)
4               #jsonCEP <- getQueryId(#json)
5.              #respone <- unregisterQueryCEP(#jsonCEP)
6.      else
7.              #respose <- errorMsg()
8.else
9.      #respose <- errorMsg()
10.return #response
```

Algoritmo 3: deployQuery

Recibe una petición HTTP con un json que contiene la query que se quiere desplegar. Se valida el json y se manda al CEP.
Se devuelve el resultado del CEP.

request: petición HTTP que llega al servicio

```
1.#json <- getData(request)
2.if isJson(json)
3.      if isDeployQueryFormat(#json)
4.              #jsonCEP <- getQueryId(#json)
5.              #jsonCEP <- getQueryInstancesConfiguration(#json)
5.              #respone <- deployQueryCEP(#jsonCEP)
6.      else
7.              #respose <- errorMsg()
8.else
9.      #respose <- errorMsg()
10.return #response
```

Algoritmo 4: undeployQuery

Se recibe una petición HTTP con un json indicando qué query se quiere dejar de ejecutar. Se valida el json, se manda al CEP y se devuelve el resultado devuelto por este.

request: petición HTTP que llega al servicio

```
1.#json <- getData(request)
2.if isJson(json)
3.      if isUndeployQueryFormat(#json)
4.              #jsonCEP <- getQueryId(#json)
5.              #respone <- undeployQueryCEP(#jsonCEP)
6.      else
7.              #respose <- errorMsg()
8.else
9.      #respose <- errorMsg()
10.return #response
```

Algoritmo 5: retrieveRegisteredQueriesIds

Se recibe una petición HTTP solicitando las queries registradas en el CEP. Se solicita al CEP y se devuelven como respuesta.

request: petición HTTP que llega al servicio

```
1.#json <- getData(request)
2.#respone <- retrieveRegisteredQueriesIdsCEP()
3.return #response
```

Algoritmo 6: retrieveDeployedQueriesIds

Se recibe una petición HTTP solicitando las queries en ejecución en el CEP. Se solicita al CEP y se devuelven como respuesta.

request: petición HTTP que llega al servicio

1.#json <- getData(request)
2.#respone <- retrieveDeployedQueriesIdsCEP()
3.return #response

Algoritmo 7: getStatus

Se recibe una petición HTTP solicitando el estado del CEP. Se solicita al CEP y se devuelven como respuesta.

request: petición HTTP que llega al servicio

1.#json <- getData(request)
2.#respone <- getStatusCEP()
3.return #response

El interfaz recibe peticiones en formato json, por lo que podría dar servicio a otros clientes que se quisiesen utilizar para mandar/recibir información al CEP. Es decir, es posible que un cliente quisiera utilizar una herramienta visual que ya tiene y adaptarla para generar peticiones al CEP. Para ello solo tendría que modificar su aplicativo para poder realizar las peticiones al interfaz y al ser en formato json sería sencillo de implementar.

Se utilizan las mismas librerías que se usan en el CEP, de forma que, si en algún momento hubiera que realizar algún ajuste en las mismas no tendrían que cambiarse en varios sitios sino en uno solo. Únicamente habría que cambiar la versión de la dependencia del proyecto y volver a desplegar el servicio.

3 Resultados y conclusiones

A continuación, se expondrán las pruebas realizadas, así como los resultados de las mismas y las conclusiones del trabajo realizado.

3.1 Pruebas

Tras realizar el desarrollo, se realizan las pruebas pertinentes para comprobar el correcto funcionamiento del componente. Para ello se despliegan todos los componentes en una máquina del laboratorio de la universidad, ya que tiene la infraestructura necesaria para ejecutar el CEP. Aun así, el CEP puede desplegarse en modo centralizado, lo que permite levantar varias instancias del proceso *InstanceManager* para realzar pruebas en pequeña escala.

3.1.1 Pruebas técnicas

Para poder realizar correctamente las pruebas funcionales, primero hay que comprobar que el sistema está desplegado correctamente.

1- Se instala el CEP, la aplicación web y el CEP-HttpEndPoint mediante los scripts de instalación.

2- Se despliega el CEP, la aplicación web y el CEP-HttpEndPoint mediante los scripts de despliegue.

3.1.2 Pruebas funcionales

Se realizan las siguientes pruebas para comprobar que el componente cumple las expectativas. Las siguientes pruebas se repiten con varios json distintos. Dependiendo de la funcionalidad que se quiere probar, se apunta a un endpoint distinto del servicio:

/registerQuery

- Registrar una query válida con un json válido.
- Registrar una query válida con un json no válido para comprobar el mensaje de error.
- Registrar una query no válida con un json válido para comprobar el mensaje de error.

/unregisterQuery

- Eliminar una query registrada en el sistema mediante un json válido.
- Eliminar una query registrada en el sistema con un json no válido y se comprueba el mensaje de error.

/retreiveRegisteredQueries

- Solicitar las queries que están registradas en el CEP en ese momento.

/deployQuery

- Desplegar una query con un json válido que contenga una query registrada.
- Desplegar una query válida con un json válido que contenga una query que no esté registrada.
- Desplegar una query con un json no válido que contenga una query registrada.

/undeployQuery

- Apagar una query con un json válido que contenga una query en ejecución.
- Apagar una query con un json válido que contenga una query que no esté en ejecución.
- Apagar una query con un json no válido que contenga una query en ejecución.

/retreiveDeployedQueryIds

- Solicitar las queries que están desplegadas (en ejecución) en el CEP en ese momento.

Además de las pruebas anteriores, se realizan pruebas más complejas:

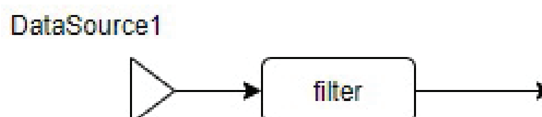
1- Recuperar el estado del CEP sin ninguna query desplegada y comprobar que efectivamente no existen querys registradas.

curl -H "Content-Type: application/json" -X GET
http://localhost:9001/cep-http-service/cep-service/status

El CEP devuelve un json con la información de las instancias que están arrancadas y sus recursos.

➔ {"imStatusJSONS":[{"imIpPort":"localhost:9792","cpu":0.0,"memory":0.0,"status":"Ready","numTuplesIn":0,"numTuplesOut":0,"sqs":null},{"imIpPort":"localhost:9791","cpu":0.0,"memory":0.0,"status":"Ready","numTuplesIn":0,"numTuplesOut":0,"sqs":null}]}

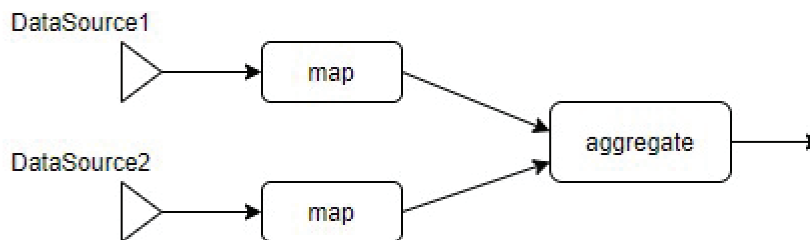
2- Registrar una query sencilla con un solo operador. Se utiliza un operador Filter que escucha de un DataSource. Se envía un json con toda la estructura de la query.



curl -H "Content-Type: application/json" -X POST -d '{JSON_ Filter01}' http://localhost:9001/cep-http-service/cep-service/registerQuery

El CEP devuelve un json con la estructura de la query registrada.

3- Registrar una query compuesta por dos operadores map y un aggregate, cada uno de ellos escuchando de un DataSource distinto. Se envía un json con toda la estructura de la query.



curl -H "Content-Type: application/json" -X POST -d '{JSON_ QueryMMA}' http://localhost:9001/cep-http-service/cep-service/registerQuery

El CEP devuelve un json con la estructura de la query registrada.

4- Solicitar las queries registradas en el CEP y comprobar que las queries anteriores están registradas en el sistema.

curl -H "Content-Type: application/json" -X GET http://localhost:9001/cep-http-service/cep-service/retreiveRegisteredQueries

El CEp devuelve un json con las queries registradas en el sistema.

➔ {"queriesInfo":[{"queryName":"QueryMMA","sqInstances":[{"sqName":"SQ_QueryMMA_0","instances":1},{"sqName":"SQ_QueryMMA_1","instances":1}]},{"queryName":"FilterQuery1","sqInstances":[{"sqName":"SQ_FilterQuery1_0","instances":1}]}]}

5- Eliminar la query FilterQuery1 del registro del CEP. Se envía un json con la información de la query que se desea eliminar.

curl -H "Content-Type: application/json" -X POST -d '{JSON_FilterQuery1}' http://localhost:9001/cep-http-service/cep-service/unregisterQuery

El CEP devuelve un json informando de la query eliminada.

6- Comprobar que la query se ha eliminado correctamente. Se vuelve a solicitar la lista de queries registradas en el CEP para comprobar que la query borrada en el paso anterior ya no existe.

curl -H "Content-Type: application/json" -X GET http://localhost:9001/cep-http-service/cep-service/retreiveRegisteredQueries

Se devuelve un json con la lista de las queries registradas, en el que se aprecia que ya no aparece la query eliminada.

➔ {"queriesInfo":[{"queryName":"QueryMMA","sqInstances":[{"sqName":"SQ_QueryMMA_0","instances":1},{"sqName":"SQ_QueryMMA_1","instances":1}]}

7- Desplegar una query registrada en el sistema. Se envía un json con la información de la query que se quiere desplegar en el CEP.

curl -H "Content-Type: application/json" -X POST -d '{"queryName":"QueryMMA","sqInstances":[{"sqName":"SQ_QueryMMA_0","instances":2},{"sqName":"SQ_QueryMMA_1","instances":1}]}' http://localhost:9001/cep-http-service/cep-service/deployQuery

El CEP devuelve un json informando del correcto despliegue de la query.

➔ {"imId":0,"queryName":"QueryMMA","responseCode":1,"responseDescription":"A query with name QueryMMA06 has been deployed"}

8- Solicitar las queries que se están ejecutando en el sistema:

curl -H "Content-Type: application/json" -X GET http://localhost:9001/cep-http-service/cep-service/retreiveDeployedQueries

El CEP devuelve la lista de queries que se están ejecuntando en el sistema.

➔ [QueryMMA]

3.2 Conclusión

Antes de este trabajo, se empezó a desarrollar una herramienta de visualización para que los usuarios pudieran ver de forma gráfica las queries que crean y que despliegan en el CEP. El CEP-HTTPEndPoint se ha creado para que el CEP y la herramienta de visualización tuvieran un enlace de comunicación para poder mandar y recibir peticiones entre sí. Tras un análisis detallado del problema, se llegó a la conclusión de que un servicio REST era la opción más viable, ya que aporta sencillez a la hora de mandar información de un componente a otro y además se puede utilizar para integrar futuros componentes que necesiten hacer uso de la funcionalidad del CEP.

El desarrollo del proyecto ha sido satisfactorio. Se ha conseguido desarrollar el componente por completo, quedando probada toda su funcionalidad.

4 Bibliografía

CEP:

[1] ‘D4.1 CEP engine architecture.pdf’ (documentación proyecto CEP)

[2] ‘D4.3 CEP Engine Centralized.pdf’ (documentación proyecto CEP)

[3] ‘D4.6 CEP Engine Distributed_v0.8.pdf’ (documentación proyecto CEP)

[4] https://aws.amazon.com/es/streaming-data/

[5] https://www.oracle.com/es/big-data/what-is-big-data.html

[6] https://databricks.com/glossary/complex-event-processing

API:

[7] https://es.wikipedia.org/wiki/Interfaz_de_programaci%C3%B3n_de_aplicaciones

SOAP:

[8] https://www.w3.org/TR/soap/

REST:

[9] https://es.wikipedia.org/wiki/Transferencia_de_Estado_Representacional

[10] https://www.codigonaranja.com/2018/restful-web-service

[11] http://expertojava.ua.es/

JSON:

[12] https://www.json.org/

Flink:

[13] https://flink.apache.org/news/2016/04/06/cep-monitoring.html

YouTube:

[14] ww.youtube.com

Facebook:

[15] www.facebook.com

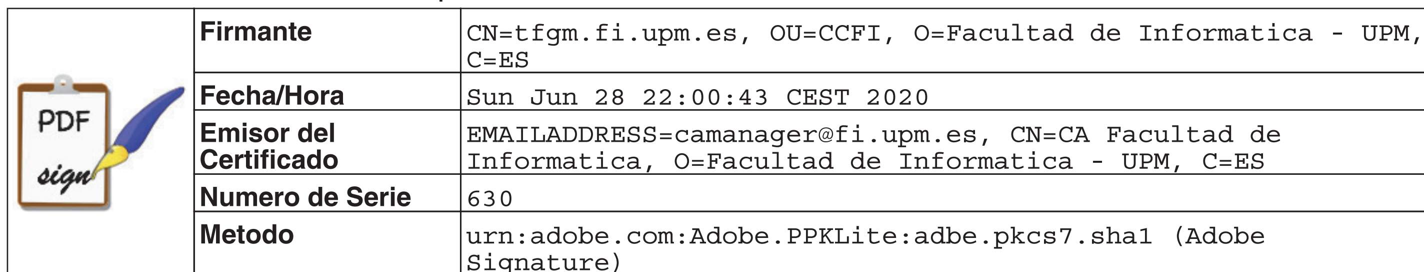
Hadoop:

[16] https://hadoop.apache.org/

Cloudera:

[17] https://es.cloudera.com/

Este documento esta firmado por



	Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Sun Jun 28 22:00:43 CEST 2020
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)