# Acerca del almacenamiento en Google Cloud

En Google Cloud, existen diferentes soluciones para el almacenamiento de datos en la nube. La idea era encontrar una alternativa a Amazon S3, que fundamentalmente consiste en almacenamiento basado en objetos mediante el concepto de “bucket”. Dichos “buckets”, al menos en S3, son contenedores donde se realiza el almacenamiento basado en objetos. Recuerda que los objetos son entidades o archivos, con datos y metadatos desacoplados. En concreto, dichos metadatos pueden ser de usuario, o de sistema.

Así, resulta que en Google Cloud sí que existe algo similar, que es la solución llamada **Cloud Storage** (almacenamiento basado en objetos mediante buckets, que aquí reciben el nombre de **“storage buckets”**. Esta solución, además, tiene caché, y está orientado a “lagos de datos”, justo lo que necesitamos para Raphtory.

En paralelo, existen otros modelos de almacenamiento dentro de Google Cloud:

* **Persistent Disk**: almacenamiento basado en bloques para máquinas virtuales pesadas y contenedores.
* **SSD Local**: similar a lo anterior, pero efímero, conectado de forma local.
* **Filestore**: para archivos, con escalabilidad.
* **Cloud Storage for Firebase**: para datos de aplicaciones móviles.
* **Data Transfer Service**; transferencia rápida de datos. Quizás esto pueda ser también útil para Raphtory.
* **Google Workspace Essentials**: almacenamiento colaborativo.

# Sobre el despliegue en la nube

Por lo que hemos ido comentando en las reuniones de seguimiento, la idea básicamente consistiría en desplegar Raphtory dockerizado en Google Cloud. Recuerda que, en estos momentos, Raphtory se puede lanzar de dos modos:

1. De forma **local**: esto es lo que has hecho en todos los ejemplos que has mirado (GAB y LOTR).
2. Como un conjunto de **contenedores** de Docker: cada una de las entidades que compone Raphtory se ejecuta dentro de un contenedor. Después, este conjunto de contenedores se despliega, conecta, e interactúa mediante Docker Compose. Esta solución no he conseguido que funcione (siguiendo el tutorial de este [repositorio](https://github.com/miratepuffin/Raphtory-Deployment)), pero, como Raphtory se está actualizando a la siguiente versión, y va a cambiar todo el despliegue con Docker, al que le están dando mucha mayor importancia que antiguamente, de momento nos vamos a olvidar de la existencia de esta vía.

Resulta que Google Cloud sí dispone de una herramienta para el lanzamiento de aplicaciones Dockerizadas, el llamado Google Container Engine, o GKE, basado en Kubernetes. Básicamente, se encarga de la gestión y orquestación de contenedores y clústeres en la nube. Google proporciona algunos tutoriales, así que lanzaré la cuenta básica de prueba gratuita (90 días y 300 $), y bien haré este tutorial, bien intentaré desplegar Apache Spark o Flink en el GKE. Por cierto, mi cuenta ya tenía todos los créditos consumidos, tanto los antiguos de otras asignaturas, como los de pruebas, así que, mientras no tenga acceso a la cuenta de investigación de 5000 $, voy a usar la de mi padre, con 300 $ (257 €), y 90 días.

## Primer tutorial

De momento, he hecho un tutorial oficial de Google en el que te explican como crear un clúster de máquinas en Google Cloud con GKE (un solo nodo), y desplegar una aplicación con Kubernetes sobre el clúster.

En ese tutorial vemos cómo se configura Google Cloud para poder usar el GKE, cómo se usan los objetos de Kubernetes para crear y gestionar los recursos de un clúster, etc. En concreto, hay objetos de Despliegue, para desplegar aplicaciones sin estado, como servidores web, y objetos de servicio, que definen reglas y equilibrios de carga para acceder a la aplicación desde Internet.

La idea básica es que creas un objeto de despliegue (con *kubectl create deployment*), con un Pod que ejecuta una imagen de un contenedor (que se indica justo después de lo anterior) con la aplicación que corre.

Después, con *kubectl expose* se crea un servicio que expone la aplicación al tráfico externo. Aquí, además, se indica la creación de un balanceador de carga para el contenedor, con unos flags adicionales para enrutar el tráfico por los puertos deseados.

Con *kubectl get pods*, podemos ver los pods ejecutándose (uno solo, por lo comentado anteriormente), y con *kubectl get service “nombre”*, salen una serie de metadatos, entre ellos, la dirección IP y el puerto con el que se puede acceder a la aplicación desplegada.

Al final, conviene borrar tanto el servicio como el clúster, para evitar que se consuman más recursos.

## Segundo tutorial

El segundo tutorial trata sobre cómo usar BigQuery para almacenar datos, y utilizar Apache Spark en GKE para procesar esos datos.

La parte de BigQuery se relaciona más con teoría de bases de datos, y es una API propia de Google, así que, aunque la he realizado, no la considero de interés.

En la segunda parte del tutorial sí que se lanza una aplicación de Spark que utiliza un conector llamado “spark-bigquery” para ejecutar consultas de SQL contra BigQuery.

Para ello, se crea un clúster predefinido para spark (es un comando propio de la consola de gcloud) con tres nodos. Luego hay una parte muy larga para poder generar las credenciales, roles, y permisos necesarios para que se pueda instalar y ejecutar Spark correctamente.

A continuación, se instala Maven como gestor de paquetes, y se crea un Cloud Storage bucket para almacenar el jar de la aplicación y los resultados del pipeline de Spark. En concretos, se sube a este bucket una aplicación que previamente se ha clonado en local desde un repositorio de Github.

Después, simplemente se descarga Spark, se configuran una serie de propiedades en el fichero correspondiente, y, finalmente, se ejecutan los trabajos de análisis.

Al final, se borra el proyecto para evitar incurrir en gastos adicionales.

# Helm

Helm es una herramienta que permite gestionar aplicaciones de Kubernetes (definición, actualización, instalación, etc.). Parece una especie de gestor de paquetes de Kubernetes, que reciben el nombre de “Charts”.

Así, un **chart** es un empaquetado de toda la información necesaria para crear una instancia de una aplicación de Kubernetes. Además del chart, existe otro concepto importante, el “**config**”, que contiene la información de configuración que se fusiona con el chart para crear el objeto lanzable. De este modo, un lanzamiento (**release**), es una instancia de un chart combinado con un config específico.

## Componentes

Helm tiene dos componentes fundamentales en su arquitectura:

### Cliente Helm

Un cliente de línea de comandos para el usuario final. Permite crear un chart local, gestionar repositorios y lanzamientos, y comunicarse con la librería de Helm.

### Librería Helm

Posee la lógica para ejecutar todas las operaciones de Helm. Es quien se comunica con la API de Kubernetes, para, entre otras cosas, combinar un config y un chart para construir un release, o instalar un chart en Kubernetes.

Esta librería está implementada en Go, y utiliza la librería cliente de Kubernetes para comunicarse con Kubernetes, basada en REST + JSON.

# Amazon Elastic Map Reduce

Amazon EMR es una plataforma basada en clústeres (entendiendo clúster como un conjunto de instancias de EC2), que permite ejecutar tareas o trabajos de análisis de grandes conjuntos de datos, es decir, de Big Data, basándose en Apache Hadoop. Además de Hadoop, también permite el despliegue de stacks de Apache Flink y Spark, entre otros. Tiene mucho uso en aprendizaje automático, indexado de webs, análisis financiero, simulaciones científicas, etc.

Dentro del clúster, la arquitectura se estructura en tres tipos de nodos:

* **Nodo maestro**: gestiona el clúster mediante la ejecución de los componentes software que coordinan la distribución de datos y tareas entre el resto de nodos. También lleva a cabo una monitorización del clúster.
* **Nodo central**: son nodos que llevan a cabo tanto la ejecución de tareas, como el almacenamiento de datos en el HDFS (Hadoop DataFile System, la estructura de almacenado de datos de Hadoop).
* **Nodo de tareas**: similar al anterior, pero solo ejecuta tareas, nada de almacenamiento. De hecho, este tipo de nodo es opcional.

Una vez que lanzas el clúster, y lo tienes corriendo, puedes enviar los diferentes trabajos, mediante distintas alternativas, para ejecutar los análisis sobre los datos, por ejemplo, algoritmos con sus propios pasos.

A raíz de esto, deduzco que Amazon EMR es extremadamente similar a lo que se pretende hacer en este TFM.

Los datos se almacenan como ficheros en un sistema subyacente, como HDFS o EMRFS, basado en S3.

# Spring Boot

Esta parece ser una herramienta cuya finalidad consiste en facilitar el despliegue de aplicaciones, sobre todo en el lado del back-end, integrando todo el servidor de aplicaciones en un .jar. Supongo que esto podría ser útil para lo que dijo Félix, esto es, para facilitar el despliegue de Raphtory, en el sentido de que, si queremos cambiar de grafo, habría que pararlo todo, hacer todos los cambios necesarios y relanzarlo, pero, quizás, con esta herramienta, se podría acelerar el proceso considerablemente.

Sin embargo, como no sabemos si finalmente incluiremos todo esto, y, si lo hacemos, será hacia el final, de momento lo dejo anotado, pero no voy a investigar sobre ello.

# Raphtory 0.4.0

Ya se ha lanzado la siguiente versión de Raphtory. En realidad, la mayoría de los cambios no me son de interés, pues no es el objetivo del TFM conocer al detalle cómo funciona Raphtory, sino como desplegarlo.

## Notas del repositorio

Se ha publicado un registro de cambios, y Félix me ha mandado sus notas con lo más importante.

Actualmente, Raphtory puede desplegarse de dos maneras:

* Single Node: dentro de una misma máquina, Raphtory lanza todos sus componentes (ahora se llaman servicios, y el nombre de “Router” se ha sustituido por “Graph Builder”). Simplemente, es necesario crear un RaphtoryGraph, combinando un Spout y un Graph Builder que encajen con los datos y el grafo definidos. Hecho esto, ahora se pueden lanzar queries (ojo, las View Queries ahora son Point Queries), a través de los métodos proporcionados por el objeto Raphtory Graph.
* Distribuido: Raphtory puede ejecutarse como un sistema distribuido compuesto por una serie de servicios, cada uno de los cuales se ocupa de una de las funciones de Raphtory. Este método todavía no está muy refinado, así que estoy pendiente de concretar una reunión para completar la información. De momento, por lo que he leído, es necesario definir una plantilla propia para los servicios, como, por ejemplo, LOTRDistributed (una clase de Scala, que extiende a com.raphtory.core.build.server.RaphtoryService). Esta clase base sirve como modelo para la creación de todos los componentes, pues dentro se puede especificar las clases para el Spout y el Graph Builder, de forma análoga al single node.

En cuanto a los servicios (componentes) de Raphtory, lo comentado en los análisis sigue siendo cierto, pero hay alguna diferencia, especialmente en el despliegue distribuido, así que los resumo aquí muy brevemente:

* Leader: se encarga de que los diferentes servicios se descubran unos a otros, y de gestionar las comunicaciones.
* Spout: genera trozos de datos de una fuente externa, como un fichero en local, o una cola de Kafka. Actualmente, solo puede haber un Spout, pero en el futuro se podrán añadir más.
* GraphBuilder: parsea trozos de datos para detectar eventos de grafo en un instante específico. En otras palabras, convierte los cambios de los datos en operaciones de actualización para el grafo, es decir, son los antiguos Routers.
* PartitionManager: almacena en memoria un subconjunto, o partición, del grafo, y ejecuta las queries de análisis que son enviados. Es el componentes que requiere más instancias, conforme el tamaño del grafo aumenta.
* queryManager: recibe las llamadas externas para ejecutar queries en un grafo cargado.

Un detalle importantes es que, si Raphtory se ejecuta en modo distribuido, es obligatorio que el líder sea el primer componentes en lanzarse. Además, el resto de servicios han de tener configuradas las propiedades “leaderAddress”, y “leaderPort” (en application.properties), de manera que puedan conectarse con el líder y registrarse cuando se arrancan. La configuración ha de definirse bien en el fichero application.conf en src/resources, bien fijando la propiedad de entorno adecuada.

En modo distribuido, cuando se quiere lanzar un servicio concreto, hay que invocar a la clase main pasando el nombre de dicho servicio como primer argumento. Por ejemplo:

sbt runMain p.k.g.LotrDistributed spout

También en modo distribuido, es necesario que en cada máquina virtual pesada, o contenedor en el que se ejecute un servicio, este se arranque a partir de código base, como, por ejemplo, un .jar generado con sbt. Esto se puede implementar usando tecnologías como Docker, Kubernetes/Helm, Ansible, etc.

Finalmente, añadir que existe una versión de despliegue llamada pseudo distribuida. En este caso, todo se hace en la misma JVM, simplemente se crean diferentes actores de Akka, uno para cada servicio, que escuchan en diferentes puertos locales, emulando así la funcionalidad de Raphtory en Single Node. Hay un ejemplo de esto en test/com.raphtory.RaphtoryPD.

## Repositorio de código

Para ejecutar Raphtory, ya no es necesario modificar las variables de entorno.

Más allá de esto, no acabo de entender demasiado bien qué es lo que se hace. Por lo que veo, tienes que crear una clase de despliegue, que es la ejecutable. Dentro de ella, creas un Spout, y un GraphBuilder (igual que en la versión anterior), y, con estos, construyes el *RaphtoryGraph*. Este se ocupa de todo, y es quien tiene definidas, entre otras cosas, los métodos para ejecutar queries.

El GraphBuilder es el que se encarga de construir el grafo, con todos los métodos para recoger los datos del Spout, y usar los métodos *addVertes*, *addEdge*, etc., para construir el grafo como tal.

Así, la gran duda que me genera todo esto es que veo muy claramente la creción del Spout y del GraphBuilder, pero no del Leader, del PartitionManager, ni del Query Manager. Me da la sensación de que esto sale de la ejecución de Raphtory Graph, que sí necesita que se le pasen el Spout y el GraphBuilder ya creados, pero el resto (PartitionManager, Leader, QueryManager) te los genera automáticamente. De todos modos, igual lo preguntaría.

Más allá de la estructura general de Raphtory, en la que no debería entrar demasiado, lo otro importante es la “API” de los algoritmos, que ha cambiado sustancialmente.

Sin entrar en detalle en qué hacen, que eso ya lo he comentado previamente, la idea ahora es que cada algoritmo extiende de la clase GraphAlgorithm, y sobrescribe el método *algorithm*, que simplemente coge un GraphPerspective, que es el objeto que contiene todos los métodos útiles (filter, step, select, iterate), para ir haciendo todas las operaciones correspondientes al algoritmo.

Por cierto, ya no existe un mecanismo para combinar los resultados de varias particiones, y lo de las tablas es una especie de post análisis de los vértices.

sbt ‘set test in assembly := {}‘ clean assembly 🡪 esto genera un .jar que habrá que meter en la imagen de Docker

Mirar la solución de Google para conectar en Scala.

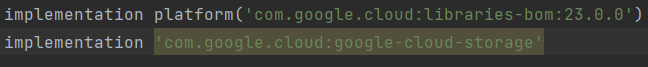
Hacer un pequeño programa para que utilice la API correspondiente de Google (la de Java) para subir un fichero, leerlo, o lo que sea, dentro de una máquina, o contenedor.

# Ejemplo de comunicación con Google Cloud

Una de las tareas consistía en crear un programa de ejemplo para subir un fichero a Google Cloud.

Para ello, básicamente lo que hay que hacer es utilizar la API de almacenamiento que nos proporciona Google. Esta API existe para varios lenguajes de programación, pero no específicamente para Scala, así que he cogido Java.

Para no andar peleando con las dependencias, ni descargando los .jar de la API, he creado con IntellIJ un proyecto de tipo Gradle>Java. Gradle es una especie de gestor de paquetes o dependencias, similar a Maven. En cualquier caso, en el fichero build.gradle, en el campo “dependencies”, hay que añadir las líneas:



Después, para que carguen, hay que hacer File > Invalidate Caches > Invalidate and Restart. Después, he copiado el código de este [ejemplo](https://cloud.google.com/storage/docs/uploading-objects#storage-upload-object-java).

Antes de ejecutar el código, obviamente hay que crear un bucket en el proyecto de Google Cloud. Para ello, en el Panel de Control de Google Cloud Console, te vas a Cloud Storage > Navegador, y sigues unos pasos sencillos para crear el bucket. Ojo, el nombre del bucket ha de ser único a nivel mundial, igual eso te da ruido.

Todavía no puedes ejecutar el código, tienes que configurar la autenticación, siguiendo los pasos de esta [web](https://cloud.google.com/storage/docs/reference/libraries).

# Sobre la documentación de la versión 4

La parte de instalación es exactamente igual. En el apartado en el que se detalla cómo crear un grafo, también es lo mismo que en la versión 3, con la salvedad de que ahora los routers reciben el nombre de Graph Builder, como ya he comentado antes. Sin embargo, han cambiado mucho la parte de los algoritmos, y han añadido una página nueva para el despliegue.

## Escribir algoritmos en Raphtory

Primero hay un breve apartado hablando sobre el paradigma de “Thinking like a vertex”, y las ventajas que representa para la computación distribuida de los vértices, pero esto ya lo he explicado extensivamente en el análisis del artículo de Pregel y el DIT.

Los algoritmos definidos dentro de Raphtory reciben un RaphtoryGraph como entrada, y devuelven una Row por cada vértice del grafo, que contiene el resultado de la ejecución del algoritmo. Para ello, han de ejecutarse tres funciones de forma secuencial.

### step()

Esta recibe una función que se aplica a cada vértice del grafo, y permite que cada vértice mute su estado y envíe mensajes a sus vecinos. Se suele utilizar como una especie de “setup” para el algoritmo definitivo, de manera que los vértices se preparen con un estado por defecto, o se encuentre el subconjunto de nodos a los que se envían los primeros mensajes.

### iterate()

El concepto es similar que el de step(), pero aquí el código que se le pasa se ejecuta repetidamente hasta que se cumple algún criterio, o se llega al máximo de iteraciones. Normalmente se usa el estado de los vértices para decidir si el algoritmo ha convergido (los vértices votan HALT unánimemente). Es destacable que suele haber un flag llamado “executeMessagedOnly”, que hace que solo se ejecuten los vértices que hayan recibido mensajes, aumentando la eficiencia drásticamente, aunque su presencia depende de la naturaleza del algoritmo.

Hablando en plata, esta es la función a la que se le debería pasar el núcleo del algoritmo, después de las inicializaciones de step().

### select()

Simplemente se encarga de asociar un vértice a un objeto Row con los resultados para dicho vértice.

Después, puede haber otras tres funciones, pero que ya no son estrictamente necesarias.

### filter()

Permite reducir la cantidad de datos almacenados. Solo puede aplicarse a la salida del select(). Por ejemplo, se puede usar para recibir tan solo los elementos que hayan recibido una cierta etiqueta.

### explode()

Se utiliza para aumentar la cantidad de filas, o evitar que la salida produzca arrays. Por ejemplo, si select() ha devuelto una lista dentro del Row, con explode() esa lista se puede convertir en objetos independientes.

### writeTo()

Tan solo se ocupa de escribir los resultados en un fichero en el path que se le pasa como argumento.

De este modo, empleando estas funciones, pasándoles el código que hayamos diseñado, se puede implementar el algoritmo de análisis deseado. Además, parece que también existen algunas funciones para proporcionar al desarrollador herramientas de Map-Reduce (filter y explode).

También es notable que ya no existe nada para juntar los resultados de varias particiones, porque se sale del alcance de este proyecto.

### Tipos de algoritmos

En función de la cantidad de pasos para la convergencia, se han clasificado los algoritmos en tres grupos.

#### Algoritmos de cero pasos

Son aquellos que no implican ningún tipo de mensajería entre vértices, con su conocimiento interno es suficiente, con lo que solo es necesario con utilizar select().

#### Algoritmos de un paso

Solo necesitan mandar mensajes una vez, y solo requieren step() y select(), para recibir algo de conocimiento de los vecinos.

#### Algoritmos iterativos

Requieren un número indeterminado de pasos de mensajería, por lo que, además de step() y select(), también se utiliza ahora iterate().

## El algoritmo de LOTR

Esto es exactamente igual que en la versión 3, así que me remito a esas notas. La única diferencia es que ahora a las ViewQueries se les llama PointQueries, pero funcionan exactamente igual.

## Despliegue de Raphtory

El despliegue en Single Node no ha variado respecto a lo ya comentado, así que no tengo más que añadir.

Si se desea desplegar Raphtory en modo distribuido, en vez de utilizar el RaphtoryGraph ya explicado, es necesario emplear RaphtoryService.

Esto lo tienen muy automatizado con Ansible, y mayormente lo han quitado de Github, así que de momento tampoco es que pueda tocarlo mucho, ni, por supuesto, probarlo.

En cualquier caso, tu ejecutable para despliegue distribuido (LOTRDistributed) extiende de RaphtoryService, y, de forma similar a como ocurría con RaphtoryGraph, solo es necesario especificar el Spout y el GraphBuilder.

Sin embargo, para que pueda ejecutarse, es necesario arrancar individualmente los siguientes componentes.

* Leader: descubrimiento y marcas de agua.
* Spout: ingiere los datos.
* GraphBuilder: construye los grafos.
* PartitionManager: almacena el grafo y su historia.
* QueryManager: acepta las queries, y las ejecuta en las particiones.

Cada uno de estos componentes, que se ejecutará en una instancia distinta (puede haber más de uno de ellos, como las particiones, que, por ende, requerirán más máquinas), esperan una serie de variables de entorno:

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

Como vemos, se puede especificar la cantidad de GraphBuilders, y particiones, junto con la cantidad de hebras que pueden correr en cada uno de esos servicios. Además, tienes la dirección IP y puerto de la máquina, y del líder. Esto implica que el líder siempre ha de ser el primer componente en arrancar.

Además, para mejorar el rendimiento, se puede configurar en cada componente unas variables de entorno de Java y Hadoop:



Xms y Xmx son la memoria libre en la caja.



Una vez que ya tienes los servicios y máquinas definidas y preparadas, se pueden ejecutar los diferentes servicios mediante los siguientes comandos:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Se está usando el args(0) de RaphtoryService para saber qué componente se está levantando, así que no lo puedes utilizar para otra cosa.

El YOUR\_CODE.jar es un jar personalizado que puedes tener, por si quieres ejecutar otra cosa al arrancar los servicios, además de Raphtory, pero no es obligatorio.

Lo único que quedaría es lanzar un cliente de Raphtory, que podría tener la siguiente forma:

Captura de pantalla de un celular

Descripción generada automáticamente

Es muy simple, solo hay que pasarle la IP y el puerto del líder, y las queries que quieras mandar. Eso sí, actualmente solo se pueden enviar los algoritmos predefinidos en el .jar de Raphtory, no propios, por cómo funciona Scala.

# Cómo ejecutar un componente de Rapthtory

La idea de esta sección es explicar cómo se puede crear y ejecutar un componente en concreto de Raphtory para el modo distribuido. Para ello, se va a partir del ejemplo de El Señor de los Anillos (LOTR, Lord Of The Rings).

Para crear cualquier componente, tenemos que tener un **proyecto de SBT**, que en su directorio **lib** contenga el **raphtory.jar**.

Así, si queremos hacer la demostración con el LOTR, podemos coger este proyecto de ejemplo, crear un subdirectorio lib en la raíz, y copiar el archivo raphtory.jar dentro.

A continuación, nos iríamos al directorio src/main/scala/com/raphtory/examples/lotr, que es el que contiene los ficheros de código. Aquí dentro, creamos un objeto de Scala, cuyo nombre, para que sea genérico, podría fijarse en Component. Este objeto ha de tener el siguiente código:

Texto

Descripción generada automáticamente

Básicamente, observamos que extiende de la clase abstracta RaphtoryService, y, por ende, ha de implementar sus dos métodos obligatorios, que implican definir un GraphBuilder (en este caso aprovechamos el del ejemplo), y un Spout (aquí es de tipo fichero, o sea, que va a leer de un fichero, por lo que hay que indicarle la ruta y el nombre. En el futuro, espero que sea un spout como el que ya preparé que podía leer y subir cosas de Google Cloud). Esta clase abstracta, obviamente, cuenta con más métodos, pero estos se encuentran pre-compilados, así que entiendo que podemos desentendernos de ellos.

Una vez definido el objeto, nos falta ejecutarlo. Para ello, arrancamos una consola de **sbt**  en la raíz del proyecto, que es donde se encuentra el fichero build.sbt (IntellIJ nos da la opción directamente, pero se puede hacer arrancando un terminal, y ejecutando el comando *sbt* en la raíz). Hecho esto, ejecutamos *compile* para compilar todo el proyecto, el cual debería acabar sin problemas.

Después, lo ejecutamos. El problema que nos encontramos aquí es que el proyecto en sí ya tiene un objeto llamado Runner que extiende de App, y, por ende, si nos limitamos a hacer *run*, va a ser esto lo que va a ejecutar. La solución lógica sería modificar el objeto Runner para que, en vez de crear un RaphtoryGraph, cree un RaphtoryService. Sin embargo, esto es imposible, ya que RaphtoryService es una clase abstracta, y, por ende, no se puede instanciar, te ves obligado a crear un objeto (Component) que extienda de ella.

Por tanto, nos queda la otra solución, que es indicar a SBT que lo que ha de ejecutar es Component, no Runner. Así, el comando que hay que dar dentro de la consola de SBT es :

*runMain com.raphtory.examples.lotr.Component “leader”*

El primer “argumento” del comando es la ruta del objeto que quieres ejecutar. Fíjate que empieza en com, no en la raíz. El segundo argumento es el String que espera el objeto, pues así está implementado RaphtoryService. En este caso, le hemos pasado un String “leader”, que es válido, y permite ejecutar el componente como un Leader. Si quisiéramos arrancar un Spout, el String sería “spout”, y así sucesivamente para los demás componentes (recuerda que el primero que ha de ejecutarse es el líder, para que los demás puedan conectarse a él, y encontrarse los unos a los otros).

Si quisiéramos ejecutar un componente distinto al que ya tenemos arrancado, no basta con abrir otra consola y volver a ejecutar el comando con un String distinto. Ello se debe a que intentaría usar la misma dirección IP y el mismo número de puerto, lo que genera una excepción de Java del tipo “Address already in use”.

A día de hoy, aunque supongo que cambiará en el futuro, la configuración de en qué dirección IP y puerto está el componente, y en qué dirección IP y puerto ha de conectarse (la del líder), se especifica en el fichero “application.conf” dentro del raphtory.jar. Por tanto, si queremos ejecutar otro componente, habría que copiar el proyecto, cambiar este fichero para poner los datos correctos (la edición, de momento, la estoy haciendo de forma manual, con WinRar y un editor de textos para trabajar sobre el .jar), y, en esta copia, arrancar de nuevo SBT y repetir los pasos anteriores.

Si queremos dockerizar esto, con estos pasos debería ser sencillo ejecutar los componentes. Ten en cuenta que el proyecto de ejemplo se lo voy a pasar directamente a la imagen, con lo que no hay que clonar repositorios, ni nada parecido, simplemente descargar las herramientas necesarias, y ejecutarlo.

Suponiendo que en el Dockerfile ya le voy a pasar el proyecto con el código necesario, suponiendo que la imagen base va a ser Linux Ubuntu, los comandos aproximados que habría que incluir en el Dockerfile son los siguientes:

*sudo apt-get install curl*

*curl -s "https://get.sdkman.io" | bash*

*sdk install java $(sdk list java | grep -o "8\.[0-9]\*\.[0-9]\*\.hs-adpt" | head -1)*

*sdk install sbt*

Con esto, se debería instalar Java y SBT, y se debería poder ejecutar el proyecto con lo que he comentado antes.

La duda me viene con la red. Aunque sea en este momento, ¿tengo alguna forma de saber en qué IP se va a desplegar el líder, y que un componente esté en su misma subred? Al ejecutarse dentro del mismo equipo físico, supongo que sí, pero, ¿cómo saber la IP del líder? ¿Y cuándo lo lance a la nube, cómo voy a poder conocer estos datos?

## Acerca de usar el compilador de Scala directamente, o SBT

En el manual que se incluye en el GIT de Raphtory, a la hora de ejecutar cada componente como un servicio, se utiliza directamente el compilador de Scala.

Así, si tenemos un proyecto como el que he ido describiendo en los apartados anteriores, a la hora de ejecutarlo, podemos hacerlo como acabo de describir, usando SBT y especificando con la opción runMain el componente correspondiente, o hacerlo con un comando de Scala, como se ve en la última línea del siguiente script:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

El argumento para invocar el script es el siguiente (para este script en concreto, probablemente el nombre sea algo del estilo builder.py):



Esto veo que puede tener sentido en el caso de que tengamos un proyecto propio, distinto al LOTR, que hayamos empaquetado en un .jar aparte. Sin embargo, dado que mi TFM no va por ahí, y podría utilizar el ejemplo de LOTR para absolutamente todo, veo más lógico seguir con SBT, pues, al fin y al cabo LOTR es un proyecto de SBT.

Por todo esto, me olvidaría completamente de usar los comandos del script, y me limitaría a instalar SBT en los contenedores como ya he ido explicando, y ejecutándolos como sé.

# Ejecutar un componente como un contenedor de Docker

Estas instrucciones parten de que Docker se encuentra instalado en el host. Como estoy usando Windows 11, instalar Docker Desktop es suficiente.

En primer lugar, navegamos al directorio contenedores, donde se encuentra el proyecto base y el Dockerfile (ya está explicado su contenido en el mismo), y ejecutamos:

*docker build -t base\_component .*

Esto crea la imagen, basada en Linux Alpine (es una imagen de OpenJDK con Java 11 ya instalado), y que contiene el proyecto, además de SDKMAN y SBT. A continuación, simplemente hacemos:

*docker run --name leader -ti base\_component /bin/bash -c "sbt 'runMain com.raphtory.examples.lotr.Component leader'"*

Esto levanta un contenedor, y ejecuta el proyecto de Raphtory como un componente aislado.

El siguiente paso es el de usar docker-compose para levantar varios de estos contenedores, y tratar de que se conecten entre sí.

# Raphtory 0.5.0

Para el desarrollo del TFM, finalmente se va a tomar como referencia la versión 0.5.0 de Raphtory. El trabajo previo (ejecución sencilla en 0.3.0., y dockerización en 0.4.0), a priori, se incluirán en la memoria como estado del arte, o trabajo progresivo para llegar hasta el despliegue en Kubernetes, que, realmente, es lo que queda del TFM, junto con la redacción de la propia memoria.

La documentación sobre esta versión, en lugar de en GitHub, se encuentra más actualizada [aquí](https://raphtory.readthedocs.io/en/development/index.html).

## Ejecución sencilla

Como primer paso en esta versión, se va a empezar por ejecutar Raphtory en una sola máquina. En este caso, no se va a optar ni por la ejecución pseudodistribuida, ni por el uso de Docker, ya que esto ya está hecho para otras versiones, y en el TFM la clave es Kubernetes. De hecho, solo hago lo del modo single, por actualizar un poco el código que tengo, y ver los cambios respecto a la versión anterior, pero tampoco aporta mucho.

En la instalación, se recomienda el uso de Java 11 y Scala 12. Mi versión de Java es la 15, y de Scala, la 2.12.4 (creo que la correcta), pero, a priori, no voy a cambiar la versión de Java, pues supongo que con una más moderna también debería funcionar.

La novedad es que ahora hay que instalar además Apache Pulsar. El problema de esto es que parece que no se puede ejecutar directamente en Windows (sí en Linux), y hay que usar Docker como intermediario.

Por ende, lo más cómodo en Windows es arrancar Docker Desktop, y ejecutar en un terminal el siguiente comando:

**docker run -it -p 6650:6650 -p 8080:8080 -p 2181:2181 -v $PWD/data:/pulsar/data apachepulsar/pulsar:2.9.0 bin/pulsar standalone**

Hecho esto, el procedimiento es el de siempre, es decir, descargamos el repositorio de ejemplos, creamos la carpeta lib dentro del ejemplo que queramos, y dentro de ella copiamos el .jar de Raphtory que nos hemos descargado (en este caso, el nombre debe ser el original), y arrancamos un servidor de sbt, compilamos, y ejecutamos.

# Kubernetes

El siguiente paso después de ejecutar Raphtory en modo single, PD, y dockerizado, y tras actualizarlo a la versión 0.5.0, es ejecutar el despliegue en Kubernetes. En la documentación, existe una página enteramente dedicada a ello, pero no está demasiado claro.

En primer lugar, se nos especifican unos requisitos:

1. Tener el .jar de Raphtory (lo tengo).
2. Tener un .jar adicional con clases extra (esto no lo tengo, porque he estado usando SBT para mis proyectos, pero podría generar uno fácilmente, mismamente usando el propio sbt con el comando "sbt package").
3. Un clúster de Kubernetes ejecutándose (esto no lo tengo, pero es que tampoco sé qué requisitos ha de tener dicho clúster, y, por lo que se cuenta más adelante, se despliega después).
4. Acceso al clúster desde el host.
5. Java 11 instalado (creo que yo tengo Java 15, pero, hasta ahora, no ha dado ningún problema).
6. Un clúster de Pulsar accesible desde Kubernetes (supongo que será algo del estilo del contenedor de Docker que uso para lanzarlo).
7. Un controlador nginx (una especie de proxy y balanceador de carga).

## Base del despliegue

Aquí simplemente se nos explica que el despliegue, en realidad, es un conjunto de despliegues, uno por cada componente de Raphtory, con su respectivo Pod configurado mediante las variables que se pasan en el comando de despliegue. De este modo, el despliegue general de Kubernetes genera los siguientes objetos:

1. Un namespace de Kubernetes.
2. Un secret de Kubernetes que contiene detalles del registro.
3. Un despliegue de Kubernetes por cada componente. Esto implica el despliegue del número correspondiente de Pods, que dependerá de la configuración de réplicas (¿cuántas particiones, cuántos builders?)
4. Un service de Kubernetes asociado a cada Pod.
5. Un ingress de Kubernetes asociado a cada service (esto es una "exposición" y control de rutas HTTP y HTTPS desde el exterior hacia los servicios de Kubernetes).

## Despliegue

Obviamente, para ejecutar el despliegue en Kubernetes, hemos de proporcionar imágenes para los Pods. En el estado actual, estas imágenes se construyen en un entorno de Docker.

¿Qué es el script de Pulsar?

Supuestamente, existe un Dockerfile en bin/kubernetes/docker/raphtory que se puede utilizar. La imagen Docker que construyamos ha de contener los .jar (tanto el de Raphtory, como el de las clases extra que hayamos añadido).

También existe un script, en bin/raphtory-docker-image, que se encarga de copiar los .jar que proporciones dentro del contenedor de Docker, y empaquetarlo todo en una imagen Docker desplegable. Incluso tiene una opción para hacer un push de la imagen al registro Docker que utilizará después Kubernetes para hacer el pull de las imágenes. Ojo, para que funcione esta opción, hay que estar logeado en el repositorio de Docker, con:

docker login example.url.com --username “xxxxxx” --password "xxxxxxx" (¿Este es nuestro repositorio en DockerHub, o es algo local?)

Por ende, entiendo que la idea es que, con este script, hacemos:

**./bin/raphtory-docker-image -c build -t tag -r example.url.com/docker/repo -z "/path/to/raphtory/jar/raphtory-fat-jar-0.1.jar /path/to/classes/jar.jar"**

y ya tenemos nuestras imágenes construidas, con todo lo necesario. Luego, hacemos:

**./bin/raphtory-docker-image -c push -t tag -r example.url.com/docker/repo -z "/path/to/raphtory/jar/raphtory-fat-jar-0.1.jar /path/to/classes/jar.jar"**

Con esto pusheamos la imagen, y Kubernetes ya la tiene disponible para desplegarse.

Esta es la teoría, en la práctica, es bastante farragoso el lanzamiento del script.

### Ejecución del script de Docker

En primer lugar, he decidido utilizar un repositorio de DockerHub, de la cuenta que tenía de CDPS, y la aplicación de Ubuntu para lanzar el script. Para, desde el subsistema Linux, acceder a los directorios del anfitrión de Windows, hay que moverse al directorio **/mnt/c**.

Para empezar, he tenido que tocar el Docker Desktop (Configuración > Recursos), para que la aplicación de Ubuntu pudiera ejecutar comandos de Docker. Y es destacable también que hace falta tener permisos de super-usuario para todos los comandos de Docker. Finalmente, creo que no es necesario.

\*Si ya has iniciado sesión en el Docker Desktop, cuando haces “docker login”, ya no es necesario proporcionar credenciales, automáticamente las recoge de Docker Desktop.

Dicho esto, el comando necesario para ejecutar el script con la orden de construir las imágenes es el siguiente (suponiendo que estemos en el directorio raíz de Raphtory, y que tenemos un repositorio en nuestra cuenta de DockerHub llamado “raphtory”):

**sudo ./bin/raphtory-docker-image -c build -t 1.0.0 -r dpisonero -z "../K8s/Raphtory-0.5.0-alpha.jar ../K8s/example-lotr-topic\_2.13-0.5.0-Alpha.jar"**

**./bin/raphtory-docker-image -c build -t 1.0.0 -r dpisonero -z "../JARS/core\_2.13.jar ../JARS/example-lotr\_2.13-0.5.jar"**

Con este comando, se ha creado una imagen llamada “dpisonero/raphrory”. Esto es así porque el script automáticamente añade la coletilla “raphtory” al nombre que nosotros ponemos, así que, si lo que tenemos es un repositorio con la URL dpisonero/raphtory, para no tocar nada del script original, los comandos han de tener estas URLs extrañas.

**sudo ./bin/raphtory-docker-image -c push -t 1.0.0 -r dpisonero -z "../K8s/Raphtory-0.5.0-alpha.jar ../K8s/example-lotr-topic\_2.13-0.5.0-Alpha.jar"**

Posiblemente, metiéndose en el script y tocando cosas, podrá funcionar algún otro comando, pero lo he único con lo que he acertado ha sido con este. Por cierto, la URL definitiva del repositorio es <https://hub.docker.com/repository/docker/dpisonero/raphtory/general>, aunque en los comandos de Docker parece que se está generando esto: docker.io/dpisonero/raphtory:1.0.0

### Lanzamiento de Pulsar

En las últimas actualizaciones de la documentación, parece que han añadido algunas formas nuevas de lanzar Pulsar, usando un script llamado pulsar-local, en la misma ruta que el de Docker.

Para empezar, este script ha de convertirse a un formato apropiado para Unix, haciendo “dos2unix ./bin/pulsar-local”. La opción de lanzarlo en local parece imposible, ya que necesita un paquete de Java que creo no está disponible para Ubuntu 20.04.

Parece que la opción más interesante es hacerlo mediante Docker, con un comando sencillo: **./bin/pulsar-local -d docker -a start -v 2.9.0**

Otra posibilidad es mediante Kubernetes, pero te pide un contexto, así que esperaría a tener lo otro antes de hacer nada.

Finalmente, teniendo en cuenta que voy a usar el despliegue basado en Kubernetes, creo que lo mejor va a ser desplegar Pulsar en Kubernetes, ya que esto me debería garantizar que tiene el acceso desde el clúster. Para ello, hacen falta dos cosas:

1. Tener minikube lanzado, y conocer su “contexto”. Este le podemos ver con **kubectl config get-contexts**.
2. Tener Helm instalado. Tras varios intentos, lo más sencillo es usar un gestor de paquetes para Windows, “choclatey”, que ya he instalado, y hacer choco install kubernetes-helm.

Al final, el script de Pulsar lo estoy lanzando desde Git Bash, ya que si uso el WSL, no ve el contexto del clúster de Kubernetes, lo que me obliga a instalar minikube en el WSL, lo que a su vez da problemas a la hora de lanzarlo por un tema de drivers. Desde Git Bash, al permanecer dentro del host, no hay problema, reconoce todo como si estuviera en un PowerShell.

### Lanzamiento de nginx

Según lo comentado en Slack, nginx ha de desplegarse como un controlador de ingresos. Parece que esto es algo “canónico”, no un invento de ellos, y, de hecho, en la propia documentación de nginx nos explican que simplemente ejecutando

helm upgrade --install ingress-nginx ingress-nginx \

--repo https://kubernetes.github.io/ingress-nginx \

--namespace ingress-nginx --create-namespace

nginx queda desplegado. No sé si con esto es suficiente, porque estar, se ve que está y hace cosas. Me quedan dudas de su conexión con Kubernetes, pero hasta ahora es lo que puedo hacer.

En resumen, lanzas minikube, lanzas Pulsar desde le Git Bash dando el nombre de contexto de minikube, pues ya lo tengo todo instalado, y lanzas nginx con el comando de arriba.

En la web oficial de minikube, también aparece este [tutorial](https://kubernetes.io/docs/tasks/access-application-cluster/ingress-minikube/). En verdad, se reduce a ejecutar “minikube addons enable ingress”.

## Configuración de parámetros

Para lanzar el cluster de Kubernetes, antes de nada, hay que configurar una serie de parámetros. Esto se puede hacer pasando los valores como argumentos al lanzar la clase (algo del estilo "java xxx -algo yyyy -algo2 bbbb), o que sean variables de entorno previamente configuradas. En la documentación viene lo que significa cada una, cuáles son obligatorias, y cuáles opcionales, etc.

Estos parámetros se makean un poco (se quitan prefijos), y se traducen en variables de entorno con nombres estándar dentro de los Pods.

Finalmente, lo único que queda por hacer es ejecutar la clase Deploy para desplegarlo todo. Suponiendo que ya tenemos todo configurado, un ejemplo de comando de despliegue es el siguiente:

**java -cp /path/to/raphtory/jar/raphtory-fat-jar-0.1.jar com.raphtory.deploy.kubernetes.Deploy**

Para pararlo todo, hacemos **java -cp /path/to/raphtory/jar/raphtory-fat-jar-0.1.jar com.raphtory.deploy.kubernetes.Delete**

Los scripts parecen muy chulos, pero habría que dar una vuelta a cómo se usan. Los pasos que yo identifico aquí son:

1) Resolver las dudas con Félix/Slack. HECHO

1) Investigar los scripts y hacer experimentos con ellos. HECHO

2) Enterarme bien de lo del clúster de Kubernetes que se comenta en los prerrequsitos. HECHO

3) Intentar lanzarlo todo en Kubernetes.

En Slack me han confirmado que el clúster que hay que tener *pre-lanzado*, puede ser de cualquier tipo. También parece que se puede utilizar minikube. Así, lo primero que he hecho ha sido instalarlo, y lanzarlo, siguiendo estas [instrucciones](https://minikube.sigs.k8s.io/docs/start/). De primeras, nos dice que le falta un hipervisor. Podemos instalar el de Virtual Box, pero me ha dado problemas, así que he optado por la opción de Docker. Para ello, una vez haya arrancado Docker Desktop, simplemente hacemos:

**minikube start --driver=docker**

Por cierto, parece que kubectl ya estaba instalado. También he instalado **nginx**, y lo he ejecutado. Pensaba que los scripts solo podían lanzarse desde Linux, pero con Windows, usando el Git Bash, también he podido hacerlo. Por dentro parece que todo son comandos de Docker, así que igual sí que puedo ahorrarme el trabajar con máquinas virtuales.

En el script te pide que le des un Docker Registry en remoto. Para ello, parece ser que Google Cloud tiene un Registry que se puede usar. Quizás también se pueda usar algo de Docker Hub. O preguntar algo en Slack. No parece que se pueda usar nada en local.

Parece que se puede usar lo de Ubuntu en Windows, lo único que te encuentras en un directorio raro, habría que mirarlo para llegar a otro lado.

### Variables de entorno

Para lanzar kubernetes, hay que configurar las siguientes variables de entorno:

export RAPHTORY\_DEPLOY\_ID="id"

export RAPHTORY\_DEPLOY\_KUBERNETES\_MASTER\_URL="https://127.0.0.1:60115"

export RAPHTORY\_DEPLOY\_KUBERNETES\_SECRETS\_REGISTRY\_SERVER=https://hub.docker.com/repository/docker/dpisonero/raphtory

export RAPHTORY\_DEPLOY\_KUBERNETES\_SECRETS\_REGISTRY\_USERNAME=dpisonero

export RAPHTORY\_DEPLOY\_KUBERNETES\_SECRETS\_REGISTRY\_PASSWORD=X2BR7PhA3l

export RAPHTORY\_DEPLOY\_KUBERNETES\_SECRETS\_REGISTRY\_EMAIL=david.pisonero.fuentes@alumnos.upm.es

export RAPHTORY\_DEPLOY\_KUBERNETES\_DEPLOYMENTS\_SPOUT\_PODS\_IMAGE=" https://hub.docker.com/repository/docker/dpisonero/raphtory:1.0.0

"

export RAPHTORY\_DEPLOY\_KUBERNETES\_DEPLOYMENTS\_BUILDER\_PODS\_IMAGE=" https://hub.docker.com/repository/docker/dpisonero/raphtory:1.0.0"

export RAPHTORY\_DEPLOY\_KUBERNETES\_DEPLOYMENTS\_PARTITIONMANAGER\_PODS\_IMAGE=" https://hub.docker.com/repository/docker/dpisonero/raphtory:1.0.0"

export RAPHTORY\_DEPLOY\_KUBERNETES\_DEPLOYMENTS\_QUERYMANAGER\_PODS\_IMAGE=" https://hub.docker.com/repository/docker/dpisonero/raphtory:1.0.0"

export RAPHTORY\_DEPLOY\_KUBERNETES\_DEPLOYMENTS\_SPOUT\_PODS\_REPLICAS=1

export RAPHTORY\_DEPLOY\_KUBERNETES\_DEPLOYMENTS\_BUILDER\_PODS\_REPLICAS=2

export RAPHTORY\_DEPLOY\_KUBERNETES\_DEPLOYMENTS\_PARTITIONMANAGER\_PODS\_REPLICAS=2

export RAPHTORY\_DEPLOY\_KUBERNETES\_DEPLOYMENTS\_QUERYMANAGER\_PODS\_REPLICAS=1

java -Draphtory.deploy.id="id" -Draphtory.deploy.kubernetes.master.url="https://127.0.0.1:65025" -Draphtory.deploy.kubernetes.secrets.registry.server="registry.docker.com" -Draphtory.deploy.kubernetes.secrets.registry.username="dpisonero" -Draphtory.deploy.kubernetes.secrets.registry.password="X2BR7PhA3l" -Draphtory.deploy.kubernetes.secrets.registry.email="david.pisonero.fuentes@alumnos.upm.es" -Draphtory.deploy.kubernetes.deployments.all.pods.env.RAPHTORY\_JAVA\_RUN\_CLASS="com.raphtory.develop.EthereumDistributedTest" -Draphtory.deploy.kubernetes.deployments.all.pods.env.RAPHTORY\_PULSAR\_BROKER\_ADDRESS="https://127.0.0.1:6650" -Draphtory.deploy.kubernetes.deployments.all.pods.env.RAPHTORY\_PULSAR\_ADMIN\_ADDRESS="https://127.0.0.1:8080" -Draphtory.deploy.kubernetes.deployments.all.pods.env.RAPHTORY\_ZOOKEEPER\_ADDRESS="https://127.0.0.1:2181" -Draphtory.deploy.kubernetes.deployments.all.pods.env.RAPHTORY\_DEPLOY\_ID="id" -D raphtory.deploy.kubernetes.deployments.all.pods.env.RAPHTORY\_PARTITIONS\_SERVERCOUNT=1 -Draphtory.deploy.kubernetes.deployments.all.pods.env.RAPHTORY\_PARTITIONS\_COUNTPERSERVER=1 -Draphtory.deploy.kubernetes.deployments.builder.pods.env.RAPHTORY\_BUILDERS\_COUNTPERSERVER=1 -Draphtory.deploy.kubernetes.deployments.spout.pods.image="registry.docker.com /raphtory:1.0.0" -Draphtory.deploy.kubernetes.deployments.builder.pods.image="registry.docker.com/raphtory:1.0.0" -Draphtory.deploy.kubernetes.deployments.partitionmanager.pods.image="registry.docker.com/raphtory:1.0.0" -Draphtory.deploy.kubernetes.deployments.querymanager.pods.image="registry.docker.com/raphtory:1.0.0" -Draphtory.deploy.kubernetes.deployments.spout.pods.replicas=1 -Draphtory.deploy.kubernetes.deployments.builder.pods.replicas=2 -Draphtory.deploy.kubernetes.deployments.partitionmanager.pods.replicas=2 -Draphtory.deploy.kubernetes.deployments.querymanager.pods.replicas=1 -cp .\lib\Raphtory-0.5.0-alpha.jar com.raphtory.deploy.kubernetes.Deploy

Si suponemos que podemos hacer todo desde la consola de Google Cloud, el asunto es relativamente sencillo, pues funciona igual que en local. De hecho, muchas herramientas ya vienen instaladas, y es Linux nativo, así que casi es hasta más cómodo. La única duda que me queda es si se tragará bien la URL del clúster, porque no parece ser localhost, (aunque con kubeadm parece que podría cambiarlo). Pulsar sí parece desplegarse en localhost, sin embargo. Hasta que no tenga la versión buena de Raphtory, debería quedarme a la espera para no tener que cambairlo todo.

En cuanto a lo del Spout, quizás lo más sencillo es que, en vez de que el Spout “beba” de un cubo, de forma externa a Raphtory te descargues el fichero de datos, y uses el FileSpout normal y corriente. La salida sí podría escribirse directamente en la nube, ya lo hice en la versión 0.3.0, aunque también podría escribir un fichero en local, y que, de forma externa, lo suba a la nube. Debería discutirlo con Félix.

En el punto en el que estoy ahora, soy capaz de hacer todo el despliegue de los Pods en Kubernetes. El único problema es que los Pods no encuentran la clase que debería ejecutar, y tiene pinta de que es un asunto de rutas, y poco más, porque ya no aparecen errores de la conexión con Pulsar, ni de que falten elementos de configuración, ni nada parecido. En resumen, los requisitos para todo esto son:

* Tener clonado el repositorio de Raphtory, y hacer un checkout a la rama “development”.
* Tener instalado Docker Desktop, y Kubernetes + minikube.
* Configurar minikube para que utilice Docker como driver por defecto.
* Modificar el application.conf del proyecto de Raphtory. Básicamente, hay que copiar los campos “resources” y “affinity”, que vienen en el application.conf del ejemplo de Ethereum, en el application.conf del proyecto raíz.
* Usar el ejemplo de Ethereum, y no el de LoTR, es el único que se ha probado con Kubernetes.
* Asegurarse de que el script “entrypoint.sh” tiene formato LF, y no CRLF. Parece que por defecto Windows te lo convierte a DOS. Si ocurre, en el WSL es suficiente con ejecutar la aplicación dos2unix para poner el formato correcto.
* Tener dos JAR, el de Ethereum (sbt "examplesEthereum/assembly"), y el de Raphtory completo (sbt " core/assembly"). Si hemos modificado el application.conf de Raphtory, el que está en core/main, ambos JAR deberían tener los campos de resources y affinity.

Suponiendo que tenemos todo esto, a grandes rasgos, el procedimiento es el siguiente:

1. Lanzar Docker Desktop.
2. Lanzar minikube.
3. Lanzar Pulsar con el script.
4. Lanzar nginx.
5. Usar el script para generar las imágenes de Docker. En un intérprete de Unix (GitBash), ejecutamos: ./bin/raphtory-docker-image -c build -t 1.0.0 -r dpisonero -z "./core/target/scala-2.13/core-assembly-0.5.jar ./examples/raphtory-example-ethereum/target/scala-2.13/example-ethereum-assembly-0.5.jar"
6. Usar el script para subir las imágenes a DockerHub: ./bin/raphtory-docker-image -c push -t 1.0.0 -r dpisonero -z "./core/target/scala-2.13/core-assembly-0.5.jar ./examples/raphtory-example-ethereum/target/scala-2.13/example-ethereum-assembly-0.5.jar". Esto es muy lento, y quizás sería conveniente migrarlo todo un artefacto de Google Cloud, pero no me apetece volver a investigar cómo debería poner las URLs.
7. Ejecutar el despliegue java -cp .\core\target\scala-2.13\core-assembly-0.5.jar com.raphtory.deployment.kubernetes.Deploy

Finalmente, el lanzamiento en local ha sido descartado, porque parece que Zookeeper da problemas, y porque, realmente, habiendo hecho todo lo de Docker, no aporta nada.

# Despliegue en GKE

Probablemente, el despliegue sea más sencillo y con menos pasos que lo que aquí se recoge, pero, por lo menos, esto se ha comprobado que funciona.

En primer lugar, hay que clonar el repositorio de Raphtory, y hacer un checkout a la rama de fix/documentation-distributed.

A continuación, generamos los dos JAR que va a usar el despliegue. Colocándonos en la raíz del repositorio, hacemos sbt core/assembly y sbt examplesLotr/package. Aunque se puede generar el JAR del ejemplo que contenga todo, haciendo sbt examplesLotr/assembly, se ha comprobado que esto genera problemas de dependencias más adelante, así que se recomienda utilizar los dos JAR de Raphtory, el completo, y el del ejemplo, mucho más reducido.

Después, creamos una imagen de Docker con los dos JAR, y la subimos. Aunque a priori parece buena idea utilizar el Google Artifact Registry, que es mucho más rápido, este trabaja con tokens, y las clases de Scala no están preparadas para este método, solo para duplas usuario/contraseña. Por ende, estamos obligados a usar Docker y DockerHub. Para construir la imagen, en un intérprete de bash colocado en la raíz del repositorio, hacemos:

./bin/raphtory-docker-image -c build -r docker.io/dpisonero -t mytag -z "./core/target/scala-2.13/core-assembly-0.5.jar ./examples/raphtory-example-lotr/target/scala-2.13/example-lotr\_2.13-0.5.jar"

Para subir la imagen a DockerHub, docker push dpisonero/raphtory:mytag.

Antes de construir la imagen, hay que tener cuidado con el formato del script entrypoint.sh, asegurándonos de haberle hecho un dos2unix, porque, si no, cuando se haga el despliegue final de Raphtory, todos los Pods se estrellarán, indicándonos que no encuentra el script, aunque exista.

Este es el momento de saltar a la nube. Así, arrancamos nuestro panel de Google Cloud Console, seleccionamos un proyecto, y creamos un clúster. Mediante la GUI, en el servicio Google Kubernetes Engine (GKE), se puede hacer, pero, si no, en este [tutorial](https://cloud.google.com/kubernetes-engine/docs/deploy-app-cluster#standard), con seguir los primeros pasos, es suficiente. Sí que se han creado tres nodos para el clúster, en lugar de 1, que es la opción por defecto. Como zona de computación, se ha elegido europe-southwest1-a, y como región de computación, europe-southwest1.

Después, todavía en la nube, se ha creado una cuenta de servicio con todos los permisos posibles. Para ello, basta con seguir esta [guía](https://cloud.google.com/iam/docs/creating-managing-service-accounts), que se reduce a ir al servicio IAM > Cuentas de servicio, e ir siguiendo los pasos.

Ahora, volvemos a nuestro equipo, para instalar todas las herramientas de Google Cloud. Félix ha utilizado Debian (nos lo podemos descargar de Microsoft Store), pero me da la sensación de que se podría hacer sin el WSL. En cualquier caso, por el camino, vamos a tener que instalar curl, wget, y Python (usando los comandos típicos de sudo apt install …

Cuando ya tengamos todo instalado, podemos instalar gcloud, usando esta [guía](https://cloud.google.com/sdk/docs/install?hl=es-419#linux). Una vez instalado, lo arrancamos con gcloud init. Se supone que debería autenticarnos en el proceso. Tal vez se queje de que no hay una navegador, y te muestra un comando que has de copiar en una máquina con gcloud instalado, y que sí tenga navegador. Por ello, también he instalado gcloud en el host, para obtener la respuesta a ese comando, y copiarla en el Debian. Por el camino, también te debería haber obligado a seleccionar un proyecto, que, obviamente, debería ser aquel en el que has creado el clúster de tres nodos. Si no, con el comando gcloud config set project Project\_id, deberías poder hacerlo.

El siguiente paso es obtener las credenciales para interactuar con el clúster. Para ello, lo primero es instalar el componente kubectl de gcloud. Simplemente, en nuestra consola con gcloud inicializado, hacemos gcloud components install kubectl. Hecho esto, las credenciales se obtienen con gcloud container clusters get-credentials raphtory-cluster. Si ahora hacemos kubectl get nodes, veremos que nos enseña nuestros tres nodos en la nube, es decir, que a partir de ahora nuestros comandos kubectl van contra el clúster del proyecto en Google Cloud.

El siguiente paso es el despliegue de Pulsar. Para ello, hay que usar el script de pulsar-local, con la opción de minikube, poniendo el nombre del clúster (se puede extraer fácilmente haciendo kubectl config get-contexts -o name. Ojo, para usar el script, también hay que usar Helm, que en Debian, por defecto, no viene instalado. Podemos seguir el tutorial de la [web](https://helm.sh/docs/intro/install/). Además, hay que hacer unas pequeñas modificaciones en el script, consistentes en sustituir ${PULSAR\_HELM\_CHART\_VERSION} por 2.7.2, y cambiar $PULSAR\_VERSION por 2.9.0. Es posible que también debas hacer un dos2unix al script, al ejecutarlo desde Debian. Ahora, el script debería ejecutar correctamente, y deberíamos ver en Google Cloud que tenemos Pods asociados a Pulsar funcionando correctamente (5 servicios, todos en verde, aunque al principio veas cosas con Warnings y demás).

Ojo, el despliegue de Pulsar es un poco puñetero, y parece que, una vez lo has lanzado, si lo paras y lo vuelves a ejecutar, ya te va a dar problemas con los sockets, aunque te salga todo en verde. En estos casos, lo más cómodo es directamente reiniciar el ordenador.

Dicho esto, para lanzar Raphtory en la nube, hay que ejecutar con Python el script setup-kubetory.py, desde Debian. Lo único que hace es dar valores a las variables de entorno necesarias, y ejecutar el despliegue. Aquí parece que hay que estar correctamente logeado en Docker y en Google Cloud. Por ello, conviene repetir docker login --username "dpisonero" y gcloud container clusters get-credentials raphcluster. Hay que revisar el script, para asegurarse de que la IP seleccionada es la de nuestro clúster, y que el nombre de usuario, contraseña, y correo de DockerHub son los nuestros. Una vez nos hayamos asegurado de todo esto, simplemente ejecutamos python3 bin/setup-kubetory.py, y se realiza el despliegue. Tanto en Lens como en el panel de Google Cloud, deberíamos ver todos los Pods en verde, y con logs indicando que todo funciona bien, que está a la espera de empezar a ejecutar lo que se le envíe.

Para poder probarlo, de momento lo más sencillo es entrar en el contenedor “batch-partitionmanager”, y ejecutar la clase LOTRClient. Para ello, te puedes copiar los jars, que están en /raphtory/jars, en un nuevo directorio /jars2, te sitúas dentro, y ejecutas java -cp core-assembly-0.5.jar:example-lotr\_2.13-0.5.jar com.raphtory.examples.lotrTopic.LOTRClient

Ahora, si nos vamos al directorio /tmp/raphtory, dentro se habrá creado una carpeta con el nombre del algoritmo, y los datos resultantes.