UNIVERSITAT DE BARCELONA

Trabajo final de grado

GRADO EN INGENIERÍA INFORMÁTICA

Facultad de Matemáticas e Informática
Universidad de Barcelona

Implementación de una arquitectura end-to-end para la recogida y explotación de datos desde una API pública

Caso de uso - consumo y visualización de datos del COVID-19

Autor: Luis Roset Giménez

Director:	**Dr. Oliver Díaz, Jairo Luzón**
Realizado en:	**Departamento de Matemáticas e Informática**
Barcelona,	**19 de junio de 2021**

Resumen (Español)

Este proyecto muestra el diseño y desarrollo de una arquitectura basada en la tecnología Docker con el objetivo de estudiar su rendimiento mediante métricas y poder generar la visualización de los datos en tiempo real para extraer posibles conclusiones. Para probar la herramienta, se han utilizado datos sobre la incidencia mundial de la COVID-19.

La arquitectura consta de diferentes partes que la configuran y para su implementación se han utilizado las herramientas más actuales y novedosas. Para la capa de recogida de datos se ha utilizado un bus Kafka en streaming partiendo de una API (Application Programming Interface) de datos, para el almacenamiento de los datos se ha utilizado MongoDB como base de datos, para el procesado de datos y su posterior visualización de las métricas se ha utilizado Prometheus junto a Grafana, permitiendo una visualización User Friendly mediante gráficas, y finalmente, para la visualización de los datos, se ha utilizado MongoDB Charts, herramienta que ha permitido la visualización de los datos mediante mapas de calor. Se puede concluir que la arquitectura ha sido diseñada de tal forma que es capaz de gestionar los datos desde cualquier tipo de fuente de datos de forma genérica y totalmente modular y escalable.

Resum (Catalán)

Aquest projecte mostra el disseny i desenvolupament d'una arquitectura basada en la tecnologia Docker amb l'objectiu d'estudiar el seu rendiment mitjançant mètriques i poder generar la visualització de les dades en temps real per extreure possibles conclusions. Per provar l'eina, s'han utilitzat dades sobre la incidencia mundial de la COVID-19.

L'arquitectura consta de diferents parts que la configuren i per a la seva implementació s'han utilitzat les eines més actuals i noves. Per a la capa de recollida de dades s'ha utilitzat un bus Kafka en streaming partint d'una API (Application Programming Interface) de dades, per a l'emmagatzematge de les dades s'ha utilitzat MongoDB com a base de dades, per al processat de dades i la seva posterior visualització de les mètriques s'ha utilitzat Prometheus costat de Grafana, permetent una visualització User friendly mitjançant gràfiques, i finalment, per a la visualització de les dades, s'ha utilitzat MongoDB Charts, eina que ha permès la visualització de les dades mitjançant mapes de calor. Es pot concloure que l'arquitectura ha estat dissenyada de tal manera que és capaç de gestionar les dades des de qualsevol tipus de font de dades de forma genèrica i totalment modular i escalable.

Abstract (Inglés)

This project shows the design and development of an architecture based on Docker technology with the aim of studying its performance through metrics and being able to generate the visualization of the data in real time to extract possible conclusions. To test the tool, data on the global incidence of COVID-19 have been used.

The architecture consists of different parts that configure it and for its implementation the most current and innovative tools have been used. For the data collection layer, a Kafka bus has been used in streaming based on an API (Application Programming Interface) of data, for data storage MongoDB has been used as a database, for data processing and its subsequent Visualization of the metrics, Prometheus has been used together with Grafana, allowing a User Friendly visualization through graphs, and finally, for the visualization of the data, MongoDB Charts has been used, a tool that has allowed the visualization of the data through heat maps. It can be concluded that the architecture has been designed in such a way that it is capable of managing data from any type of data source in a generic and fully modular and scalable way.

Índice

Índice de figuras

1. Introducción

1.1. ¿Qué es una Máquina Virtual?

Los sistemas operativos son los entornos que regulan el acceso de los programas a los recursos de un ordenador. Hoy en dia, las máquinas virtuales se utilizan ampliamente.Entre otras cosas, permiten realizar tareas de forma segura, aislando el acceso a datos y recursos del sistema de software potencialmente malicioso. Las máquinas virtuales están aisladas del resto del sistema; el software dentro de la máquina virtual (guest) no puede alterar al sistema anfitrión (host) Por lo tanto, la implementación de tareas como el acceso a datos infectados con virus y la prueba de sistemas operativos se realizan cada vez más mediante máquinas virtuales.

En definitiva, una máquina virtual es un archivo de un sistema o software que generalmente se denomina invitado, o una imagen que se crea dentro de un entorno informático llamado anfitrión.

Una máquina virtual es capaz de realizar tareas como ejecutar aplicaciones y programas como un sistema independiente, lo que la hace ideal para probar otros sistemas operativos como versiones beta, crear copias de seguridad del sistema operativo y ejecutar software y aplicaciones. Un anfitrión puede tener varias máquinas virtuales ejecutándose en un momento específico. El archivo de registro, el archivo de configuración de NVRAM, el archivo de disco virtual y el archivo de configuración son algunos de los archivos clave que componen una máquina virtual. Otro sector donde las máquinas virtuales son de gran utilidad es la virtualización de servidores. En la virtualización de servidores, un servidor físico se divide en varios servidores únicos y aislados, lo que permite que cada servidor ejecute su sistema operativo de forma independiente. Cada máquina virtual proporciona su hardware virtual, como CPU, memoria, interfaces de red, discos duros y otros dispositivos.

Las máquinas virtuales se dividen ampliamente en dos categorías según su uso:

1. Máquinas virtuales del sistema: una plataforma que permite que varias máquinas virtuales, cada una de las cuales se ejecuta con su copia del sistema operativo, compartan los recursos físicos del sistema anfitrión. El hipervisor, que también es una capa de software, proporciona la técnica de virtualización. El hipervisor, como VMWare o VirtualBox, se ejecuta en la parte superior del sistema operativo o solo en el hardware.

2. Máquinas virtuales de proceso (Process Virtual Machine): proporciona un entorno de programación independiente de la plataforma. La máquina virtual de proceso está

diseñada para ocultar la información del hardware y el sistema operativo subyacentes y permite que el programa se ejecute de la misma manera en cada plataforma dada.

Aunque varias máquinas virtuales que se ejecutan a la vez pueden parecer eficientes, esto puede llevar a un rendimiento inestable. Como el sistema operativo invitado tendría su kernel, conjunto de bibliotecas y dependencias, consumiría una gran parte de los recursos del sistema.

Otros inconvenientes incluyen un hipervisor ineficiente y un tiempo de arranque prolongado. El concepto de contenedorización supera estos defectos y Docker es una de esas plataformas de contenedorización.

1.2. ¿Qué es Docker?

Las organizaciones del mundo actual esperan transformar su negocio digitalmente, lo que es conocido como transformación digital, pero están limitadas por el diverso repertorio de aplicaciones, la nube y las infraestructuras locales. Docker resuelve este inconveniente en todas las organizaciones con una plataforma de contenedores (entornos aislados) que trae aplicaciones y microservicios tradicionales construidos en Windows y Linux en un entorno automatizado y seguro.

Docker es una herramienta de desarrollo de software y una tecnología de virtualización que facilita el desarrollo, la implementación y la administración de aplicaciones mediante el uso de contenedores. Contenedor se refiere a un paquete ligero, independiente y ejecutable de una pieza de software que contiene todas las bibliotecas, archivos de configuración, dependencias y otras partes necesarias para ejecutar la aplicación.

En otras palabras, las aplicaciones se ejecutan de la misma manera independientemente de dónde se encuentren y en qué máquina se estén ejecutando porque el contenedor proporciona el entorno durante todo el ciclo de vida de desarrollo de software de la aplicación. Dado que los contenedores están aislados, brindan seguridad, lo que permite que varios contenedores se ejecuten simultáneamente en el anfitrión. Además, los contenedores son ligeros porque no requieren una carga adicional de hipervisor. Un hipervisor es un sistema operativo invitado como VMWare o VirtualBox. En cambio, los contenedores se ejecutan directamente dentro del kernel de la máquina del anfitrión sinb necesidad de este hipervisor.

Un Docker está formado por dos componentes principales, un contenedor y una imagen. Este contenedor, es ejecutado dentro de una Docker Machine la cual permite administrar los contenedores que se encuentran dentro de ella de una forma más cómoda y eficaz.

Una configuración básica de un Docker consiste, por ejemplo, en realizar un pull de una imagen de un Linux básico que se encuentra en DockerHub, una especie de repositorio para imágenes Docker, y realizar un docker run de dicha imagen. Este comando, generará e inicializará un contenedor, el cual, contendrá el Linux que nos hemos descargado de DockerHub y podremos trabajar directamente con él realizando una conexión vía ssh con el comando *docker exec*.

Como se puede observar en la figura 1 donde se muestra el flujo de ejecución del Docker[13], una vez dentro del contenedor, podemos configurar la imagen, en nuestro caso un Linux, como deseemos. Una vez realizados los cambios, para guardar la configuración en local, ejecutaremos el comando docker commit, el cual generará una imagen con las modificaciones realizadas. Estas modificaciones de la imagen, es decir, del Linux básico, también se pueden realizar mediante un fichero conocido como Dockerfile, el cual, basándose en una imagen local, permite ejecutar comandos e instalar dependencias dentro de la propia imagen sin tener que realizar una conexión ssh con el contenedor de la misma.

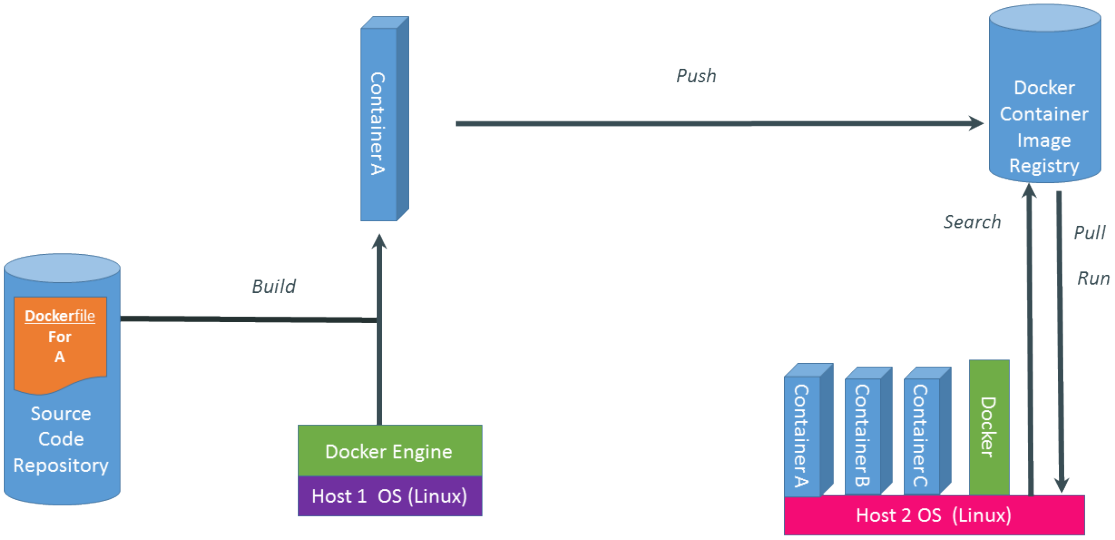

Figura 1: Flujo de ejecución de Docker

La fuente principal de imágenes Docker es DockerHub, una comunidad de usuarios donde se publican modificaciones o imágenes oficiales de diferentes servicios. Una app store de imágenes Docker. El ecosistema de contenedores es único, ya que se basa en un esfuerzo y un compromiso de toda la comunidad con los proyectos de código abierto.

Además, si nos interesa subir nuestra versión de una imagen a DockerHub, podemos

realizarlo mediante un docker push de la misma, al igual que en Github, debemos crearnos un usuario para ello

Los contenedores brindan los siguientes beneficios:

- Recursos de gestión de TI reducidos
- Tamaño reducido de instantáneas
- Aplicaciones más modulares
- Actualizaciones de seguridad reducidas y simplificadas
- Menos código para transferir, migrar y cargar cargas de trabajo

1.3. Máquina Virtual vs Docker

1.3.1. Arquitectura

La principal diferencia radica en su arquitectura, que se muestra a la figura 2 :

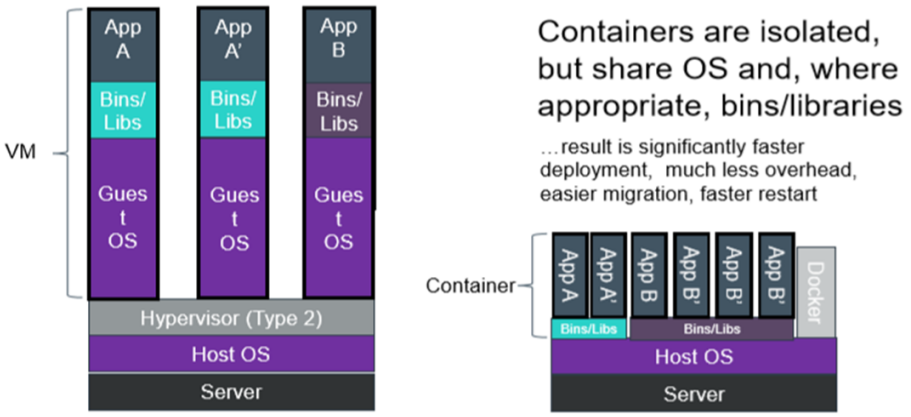

Figura 2: Diferencias de arquitectura entre Máquina Virtual y Docker

Las máquinas virtuales tienen sistema operativo anfitrión y el sistema operativo invitado dentro de cada máquina virtual. El sistema operativo invitado puede ser cualquier sistema operativo, como Linux o Windows, independientemente del sistema operativo anfitrión.

Por el contrario, los contenedores de Docker se alojan en un único servidor físico con un sistema operativo anfitrión, que comparte entre ellos. Compartir el sistema operativo anfitrión entre contenedores los hace ligeros y disminuye el tiempo de arranque. Los contenedores Docker se consideran adecuados para ejecutar varias aplicaciones en un solo kernel del sistema operativo; mientras que, las máquinas virtuales son necesarias si las aplicaciones o servicios deben ejecutarse en diferentes sistemas operativos.

1.3.2. Seguridad

Las máquinas virtuales son independientes con su kernel y características de seguridad. Por lo tanto, las aplicaciones que necesitan más privilegios y seguridad se ejecutan en máquinas virtuales.

Por otro lado, no se recomienda proporcionar acceso root a las aplicaciones y ejecutarlas con instalaciones administrativas en el caso de los contenedores Docker porque los contenedores comparten el kernel del anfitrión y podría dañarlo. La tecnología de contenedores tiene acceso a los subsistemas del kernel; como resultado, una sola aplicación infectada es capaz de piratear todo el sistema anfitrión.

1.3.3. Portabilidad

Las máquinas virtuales están aisladas de su sistema operativo y, por lo tanto, no se transfieren a varias plataformas sin incurrir en problemas de compatibilidad. En el nivel de desarrollo, si se va a probar una aplicación en diferentes plataformas, se deben considerar los contenedores de Docker.

Los paquetes de contenedores de Docker son autónomos y pueden ejecutar aplicaciones en cualquier entorno, y dado que no necesitan un sistema operativo invitado, se pueden migrar fácilmente a diferentes plataformas. Los contenedores Docker se pueden implementar fácilmente en servidores, ya que los contenedores al ser ligeros, se pueden iniciar y detener en muy menos tiempo en comparación con las máquinas virtuales.

1.3.4. Rendimiento

Las máquinas virtuales consumen más recursos que los contenedores Docker, ya que necesitan cargar todo el sistema operativo para iniciarse. La arquitectura ligera de los contenedores Docker consume menos recursos que las máquinas virtuales.

Escalar y duplicar contenedores se puede hacer de forma simple y fácil en comparación con las máquinas virtuales porque no es necesario instalar un sistema operativo en ellas.

La figura 4 presenta un breve resumen sobre las principales diferencias entre ambas tecnologías:

	Docker	Máquina Virtual (VMs)
Tiempo de arranque	Inicia en pocos segundos	Tarda unos minutos en arrancar la VM
Corre sobre	Los Dockers hacen uso del motor de ejecución	Las VMs hacen uso del hypervisor
Eficiencia de memória	No es necesario espacio para virtualizar, por lo tanto menos memoria	Requiere que todo el Sistema Operativo sea cargado antes de arrancar, por lo tanto es menos eficiente
Aislamiento	Propenso a las adversidades ya que no existen disposiciones para los sistemas de aislamiento.	La posibilidad de interferencia es mínima debido al eficiente mecanismo de aislamiento
Deployment	La implementación es fácil, ya que solo se puede usar una imagen en contenedores en todas las plataformas.	El Deployment es más largo en comparación ya que la implementación depende de instancias separadas
Uso	Docker tiene un mecanismo de uso complejo que consta de herramientas administradas tanto por terceros como por Docker	Las herramientas son fáciles de usar y es más simple trabajar con ellas

Figura 3: Comparativa de prestaciones Máquina virtual vs Docker

2. Estado del arte

2.1. Otras tecnologías basadas en contenedores

A parte de Docker, existen múltiples tecnologías basadas en contenedores las cuales presentan interesantes características. Las más destacadas son:

2.1.1. Artifactory Docker Registry

Artifactory Docker Registry[6] es un registro privado seguro que administra imágenes de Docker, brindando acceso a registros de contenedores de Docker remotos con integración para construir ecosistemas.

Permite configurar registros Docker ilimitados, utilizando repositorios Docker locales, remotos y virtuales. Trabajando de forma transparente con el cliente de Docker, administra las imágenes de Docker, que se han creado internamente y se han descargado desde recursos de Docker remotos, como Docker Hub.

Los repositorios locales proporcionan una forma de implementar y alojar imágenes internas de Docker, que luego se pueden compartir entre organizaciones. Los repositorios remotos sirven como un proxy de almacenamiento en caché, un registro administrado en una URL remota, como https://registry-1.docker.io (que es Docker Hub), donde las imágenes de Docker se almacenan en caché bajo demanda. Los repositorios virtuales definidos por artefactos agregan imágenes de repositorios locales y remotos, lo que permite el acceso a imágenes alojadas en repositorios de Docker locales, así como a imágenes remotas, que se envían desde una única URL mediante repositorios de Docker remotos.

Artifactory admite el envío de imágenes de Docker desde un repositorio de Docker en Artifactory a otro. Artifactory también admite las llamadas relevantes de la API de Docker Registry para que pueda usar de forma transparente el cliente de Docker para acceder a las imágenes a través de Artifactory.

2.1.2. LXC (Linux)

LXC[15] es un conjunto de herramientas de gestión de contenedores de bajo nivel que forman parte del proyecto de código abierto LinuxContainers.org. La tecnología fue precursora de Docker y está patrocinada por Canonical, la empresa detrás de Ubuntu.

El objetivo de LXC es proporcionar un entorno de aplicación aislado que se parezca mucho

al de una máquina virtual (VM) en toda regla, pero sin la sobrecarga de ejecutar su propio kernel. LXC también sigue el modelo de proceso Unix, donde no hay un daemon central. En pocas palabras, en lugar de ser administrado por un único programa central, cada contenedor se comporta como si estuviera administrado por un programa separado por derecho propio.

LXC también funciona de manera diferente a Docker en varias otras formas. Por ejemplo, puede ejecutar más de un proceso en un contenedor LXC, mientras que Docker está diseñado para ejecutar un solo proceso en cada contenedor. Sin embargo, Docker es mejor para abstraer recursos y, como resultado, sus contenedores tienden a ser más portátiles que los de LXC.

2.1.3. Contenedores de Windows containers y Hyper-V

Cuando Microsoft lanzó Windows Server 2016, introdujo dos nuevas tecnologías de contenedores, y ambas ofrecen alternativas ligeras a las máquinas virtuales (VM) de Windows en toda regla. El primero, Windows Containers[25], adopta un enfoque de abstracción similar al de Docker. El otro son los contenedores Hyper-V.

Los contenedores Hyper-V están más alineados con el modelo de virtualización de VM, ya que cada uno puede llevar su propio kernel. Esto significa que ofrecen una mayor portabilidad que los contenedores tradicionales, ya que las aplicaciones que se ejecutan dentro de ellos no necesitan ser compatibles con el sistema host. También ofrecen una mayor seguridad como resultado de un mayor aislamiento del sistema operativo host y otros entornos de contenedores. Sin embargo, estos beneficios vienen con una compensación, ya que los contenedores Hyper-V tienen una huella de infraestructura ligeramente mayor que Windows y otros contenedores que dependen de un sistema compartido basado en kernel.

Se pueden administrar contenedores Hyper-V mediante Docker o Windows PowerShell, pero cada entorno invitado debe estar basado en Windows, aunque no necesariamente la misma versión que el sistema operativo host.

2.1.4. rkt

Entre su ecosistema robusto y su fuerte nivel de adopción, rkt[23] (anteriormente conocido como CoreOS Rocket) se ha convertido posiblemente en una de las alternativas más viables a Docker.

Los puntos fuertes de esta tecnología de código abierto son la seguridad y, sobre todo, la

interoperabilidad con otros sistemas. Por ejemplo, puede ejecutar contenedores Docker y usa una arquitectura basada en pod, que funciona directamente con Kubernetes.

Al igual que con LXC, rkt no usa un daemon y, por lo tanto, proporciona un control más detallado sobre sus contenedores a nivel de contenedor individual.

A pesar de sus ventajas, desde que RedHat adquiere CoreOS en 2018, la dirección futura de rkt ha sido cada vez más incierta. Además, en agosto de 2019, la Cloud Native Computing Foundation (CNCF) decidió abandonar su apoyo al proyecto.

2.2. Otras tecnologías complementarias a Docker

2.2.1. Podman

Podman[19] es un motor de contenedores de código abierto, que realiza la misma función que el motor de Docker. Se distingue porque sus características de aislamiento y privilegios de usuario hacen que Podman sea inherentemente más seguro.

Del mismo modo, sus comandos de interfaz de línea de comandos (CLI) son prácticamente idénticos a los que admite la CLI de Docker, con la excepción de que usaría Podman en lugar de la base de Docker. Aunque los comandos CLI de Docker y Podman son similares, es recomendable saber cómo distinguirlos.

Aunque los comandos de la CLI de Docker y Podman son similares, saber cómo diferenciar los dos nos ayudará cuando estemos trabajando con ellos. Docker sigue el modelo cliente/servidor, utilizando un daemon para administrar todos los contenedores bajo su control. Sin embargo, Podman, como rkt y LXC, funciona sin un daemon central. Esto puede mejorar potencialmente la resistencia de cualquier contenedor al eliminar la posibilidad de tener un único punto de fallo. En otras palabras, si el daemon falla, se perderá el control sobre todos los contenedores. Por el contrario, en Podman, los contenedores son entornos autosuficientes y completamente aislados, que pueden administrarse de forma independiente entre sí.

Además, donde Docker otorga permiso de root al usuario del contenedor de forma predeterminada, el acceso no root es estándar en Podman.

2.2.2. runC

runC[24] es un tiempo de ejecución de contenedor de Sistema Operativo universal y ligero. Originalmente era un componente de Docker de bajo nivel, que funcionaba bajo el capó, integrado dentro de la arquitectura de la plataforma. Sin embargo, desde entonces se ha implementado como una herramienta modular independiente.

La idea detrás del lanzamiento era mejorar la portabilidad del contenedor proporcionando un tiempo de ejecución de cada contenedor interoperable y estandarizado que puede funcionar como parte de Docker e independientemente de Docker. Como resultado, runC puede ayudarle a evitar estar fuertemente vinculado a tecnologías, hardware o proveedores de servicios en la nube específicos.

2.2.3. containerd

Con soporte tanto para Linux como para Windows, containerd[8] es básicamente un daemon, que actúa como una interfaz entre el motor del contenedor y los tiempos de ejecución del contenedor.

Proporciona una capa abstracta que facilita la administración de los ciclos de vida de los contenedores, como transferencias de imágenes, ejecuciones de contenedores, funcionalidad y ciertas operaciones de almacenamiento, mediante el uso de solicitudes de API simples. Esto evita la molestia de realizar múltiples llamadas al sistema de bajo nivel. Como esas llamadas al sistema pueden variar de una plataforma a otra, esto también hace que los contenedores sean más portátiles al tiempo que permite que la API permanezca fundamentalmente igual.

Al igual que runC, containerd es otro componente básico del sistema Docker, que se ha convertido en un proyecto independiente de código abierto.

2.3. Actual estado del Docker

Cuando Docker apareció en el año 2013, la popularidad de los contenedores explotó y no es de extrañar que el crecimiento de Docker y el uso de contenedores vayan de la mano. En un inicio, la tecnología utilizaba LXC (LinuX Containers) pero después fueron sustituidos por una librería propia, libcontainer. Además, Docker ofrece un ecosistema completo para el tratamiento de los contenedores.

En el año 2016, se vio la importancia de las aplicaciones basadas en contenedores y los

sistemas se volvieron más complejos y sobre todo, más vulnerables. Esta situación acentuó la importancia de la seguridad en cada una de las partes del desarrollo de aplicaciones, conocido con el nombre de DevSecOps.

En 2017 aparece una nueva tecnología, Kubernetes, con el objetivo principal de ayudar a Docker a gestionar aplicaciones complejas de forma más sencilla, permitiendo a las empresas generar un híbrido entre la nube y los microservicios. Kubernetes es una plataforma portable y extensible de código abierto para administrar cargas de trabajo y servicios. Kubernetes facilita la automatización y la configuración declarativa, tiene un ecosistema grande y en rápido crecimiento y el soporte, las herramientas y los servicios para Kubernetes están ampliamente disponibles. En la DockerCon de Copenhagen, Docker anunció que daría soporte al orquestador de Kubernetes.

En 2018, la contenedorización se convirtió en la base de la infraestructura de software moderna y Kubernetes se utilizó para la mayoría de los proyectos de contenedores empresariales. En 2018, el proyecto Kubernetes en GitHub tuvo más de 1500 colaboradores, con una de las comunidades de código abierto más importantes del momento. La adopción masiva de Kubernetes empujó a los proveedores de almacenamiento en la nube como AWS, Google con GKE (Google Kubernetes Engine), Azure y Oracle con Container Engine para Kubernetes, a ofrecer servicios administrados de Kubernetes. Además, los principales proveedores de software como VMWare, RedHat y Rancher comenzaron a ofrecer plataformas de gestión basadas en Kubernetes. El proveedor de infraestructura VMware avanzó hacia la adopción de Kubernetes cuando, a finales de 2018, anunció que iba a adquirir Heptio, una firma consultora que ayuda a las empresas a implementar y administrar Kubernetes de forma sencilla.

Como se puede observar en la figura 4, hasta este momento, una cuarta parte de las empresas ya había adoptado Docker. A principios de abril de 2018, el 23,4 % de los clientes de Datadog habían adoptado Docker[9], frente al 20,3 por ciento del año anterior. Desde 2015, la proporción de clientes que utilizan Docker ha crecido a un ritmo de entre 3 y 5 puntos.

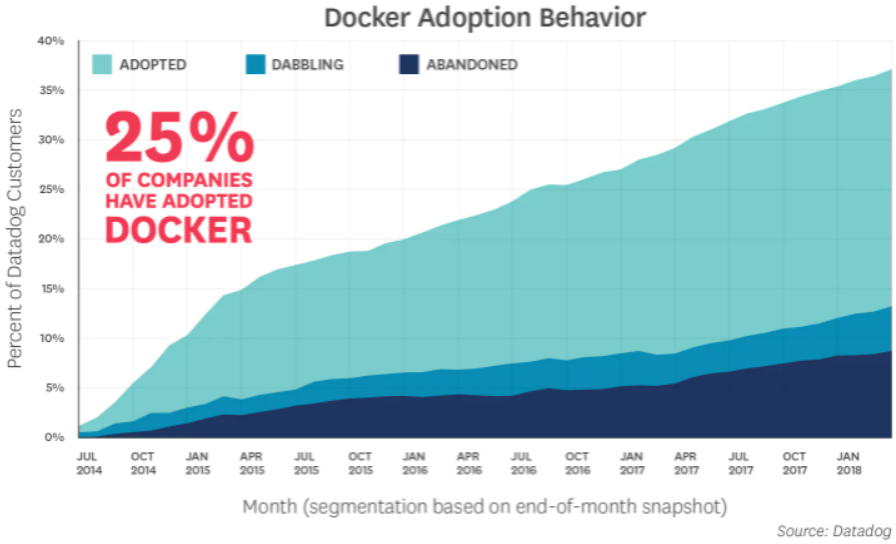

Figura 4: Evolución del uso de Docker hasta el año 2018.
Fuente: Datadog

El año pasado (2020) marcó el comienzo de cambios significativos en el panorama de los contenedores. Los nuevos motores de tiempo de ejecución ahora comenzaron a reemplazar el motor de tiempo de ejecución de Docker, sobre todo containerd, un motor de tiempo de ejecución de contenedor de código abierto, y CRI-O, un motor de tiempo de ejecución ligero para Kubernetes. También se vieron avances en la adopción de tecnologías sin servidor con plataformas como Knative, una plataforma de gestión de cargas de trabajo sin servidor basada en Kubernetes, que ganó terreno en las organizaciones más punteras. Además, grandes compañías como IBM, Google o Amazon, han lanzado soluciones híbridas en la nube basadas en Kubernetes difuminando las líneas tradicionales entre la nube y los entornos locales, ya que ahora se puede administrar clústeres en la nube locales y de un solo proveedor.

Como se observa, la evolución y tendencia de Docker, es fusionarse con Kubernetes para acceder a todas las ventajas de la nube. Generando nuevas tecnologías en la nube como Anthos or Arc, tecnologías que apuntan al futuro de la hyper-abstracción. donde los recursos son desacoplados totalmente de la capa de gestión, similar a como funciona Kubernetes, que desacopla los flujos de trabajo de donde se gestionan. Actualmente en Kubernetes, el nodo master se encuentra en el mismo clúster físico que los trabajadores. Con la hyper-abstracción la gestión de la carga de trabajo se realizará en nodos que pueden distribuirse entre varias infraestructuras informáticas, y el usuario no sabrá ni le

importará dónde se ejecutan físicamente.

El gran éxito y rápido avance del uso de Docker puede atribuirse a la evolución de las siguientes grandes tendencias:

- El auge de la nube. IT busca hacer que las aplicaciones en la nube sean portátiles y escalables al mismo tiempo.
- El auge de DevOps. La contenedorización combina bien con los procesos y herramientas de DevOps. Por lo tanto, aquellos que se están moviendo a DevOps generalmente también se están moviendo a contenedores.
- El auge del uso estratégico de los datos. Algunas de las aplicaciones que se ejecutan en Docker están principalmente orientadas a datos.

3. Objetivos del trabajo

3.1. Resumen

El principal objetivo de este proyecto es la creación e implementación de una arquitectura E2E capaz de consumir datos desde una API genérica de datos, en nuestro caso del COVID-19, y procesarlos en tiempo real, para finalmente, visualizarlos mediante gráficas tanto a nivel de rendimiento como de visualización de los datos con el objetivo de sacar conclusiones sobre la implementación y el rendimiento de la arquitectura generada.

La arquitectura debería ser capaz de trabajar con cualquier tipo de fuente de datos siempre que se procese generando un archivo csv, el cual, independientemente de los datos que contenga, será procesado y se generarán las gráficas correspondientes a dichos datos.

Esta arquitectura cuenta con cinco contenedores conectados entre sí mediante una red local, lo que permite un mejor encapsulamiento y transferencia de los datos entre contenedores. También, la arquitectura debería ser capaz de recoger los datos mediante streaming con un bus Kafka, procesarlos y guardarlos en una base de datos, en este caso, MongoDB, y finalmente, una vez introducidos los valores correctamente en la base de datos, extraer estos valores para su visualización a través de Prometheus y Grafana por la parte de las métricas, y a través de MongoDB Charts por la parte gráfica.

Otro de los objetivos es familiarizarse con la tecnología Docker y las diferentes herramientas que esta nos proporciona para la posterior implementación de los diferentes contenedores que componen la arquitectura. Estos objetivos de trabajo pueden dividirse en:

- Estudio sobre Docker
- Estudio sobre las tecnologías más óptimas para la arquitectura

3.2. Estudio sobre Docker

Para poder realizar este proyecto he realizado un estudio exhaustivo sobre qué son los Docker y las funcionalidades que presentan, todo ello con el objetivo de poder ver si seria capaz de adaptar esta potente tecnología a las necesidades que mi proyecto iría necesitando a medida que avanza.

Como ya he explicado en la Introducción, los Docker son una herramienta muy potente y versátil que nos proporciona un gran conjunto de funcionalidades y ventajas, que se

adaptan a lo que yo buscaba para la arquitectura de este proyecto. Por lo tanto, y debido a la gran cantidad de información y versatilidad de los Docker, me fue sencilla la absorción de los conocimientos básicos para la realización de este proyecto.

Este proceso duró algo más de dos meses, donde paralelamente estuve haciendo pequeñas pruebas para ir familiarizándome con la tecnología y permitirme así, avanzar de una forma más fluida a medida que avanzaba en el proyecto.

En un principio, empecé utilizando la herramienta de Docker Toolbox debido a que mi versión de Windows presentaba incompatibilidades con Docker Desktop. Docker Toolbox es un instalador para la configuración rápida y el inicio de un entorno Docker en sistemas Mac y Windows más antiguos que no cumplen con los requisitos de las nuevas aplicaciones Docker para Mac y Windows. Esta herramienta me permitió de forma rápida configurar mi entorno de contenedores para la realización de las primeras pruebas y poder establecer un primer contacto con la tecnología Docker.

Finalmente, una vez ya tenía control y conocimientos básicos sobre los Docker, instalé la aplicación Docker Desktop, la cual me permitió la posibilidad de gestionar mis imágenes y contenedores de una forma más gráfica y cómoda. Docker Desktop es una aplicación fácil de instalar tanto para entorno Mac o como para entorno Windows que permite crear y compartir aplicaciones y microservicios en contenedores. Docker Desktop incluye Docker Engine, Docker CLI client, Docker Compose, Notary, Kubernetes y Credential Helper. Principalmente en este proyecto he utilizado la herramienta de Docker Compose, es una herramienta para definir y ejecutar aplicaciones Docker de varios contenedores. Con Compose, se utiliza un archivo YAML para configurar los servicios de la aplicación. Después, con un solo comando, se generan e inician todos los servicios desde su el fichero YAML de configuración.

3.3. Estudio sobre las tecnologías más óptimas para la arquitectura

Una vez elegidas las herramientas a utilizar y con el conocimiento básico, llegó el momento de decidir cuáles iban a ser las tecnologías que iba a utilizar en cada componente de la arquitectura.

La arquitectura de este proyecto está configurada por 3 partes principales, primero tenemos la encargada de la gestión de las métricas de los datos, segundo, la encargada del almacenamiento de los datos, es decir, la base de datos, y finalmente, tenemos la encargada del flujo de los datos desde la API hasta la base de datos.

Para la elección de las tecnologías que iba a utilizar en cada una de las partes, estuve realizando un estudio exhaustivo sobre los diferentes rendimientos y prestaciones que me proporcionaban cada una con el fin de conseguir adaptar y acoplar todas de la forma más eficiente posible, consiguiendo así que el rendimiento de la arquitectura como conjunto sea lo mejor posible.

3.4. Tecnologías utilizadas

Esta sección describe los diferentes lenguajes, librerías externas, y tecnologías que han sido utilizadas para desarrollar e implementar la arquitectura.

Docker Desktop

Docker Desktop[10] es una herramienta que nos permite la integración de contenedores de manera local. Es la base de la arquitectura.

DockerHub

DockerHub[12] es un repositorio público en la nube, similar a Github, para distribuir los contenidos. Está mantenido por la propia Docker y hay multitud de imágenes, de carácter gratuito, que se pueden descargar y asi no tener que hacer el trabajo desde cero al poder aprovechar "plantillas".

Python

Pyhton[22] es un lenguaje de programación interpretado, de alto nivel y de propósito general. Sus construcciones de lenguaje y su enfoque orientado a objetos tienen como objetivo ayudar a los programadores a escribir código claro y lógico para proyectos de pequeña y gran escala.

MongoDB

MongoDB[16] es una base de datos no relacional basada en documentos que ofrece una gran escalabilidad, flexibilidad, y un modelo de consultas e indexación avanzado.

MongoDB Charts

MongonDb Charts[17] es una herramienta nativa de MongoDB que permite la visualización de gráficos con el objetivo de hacer más "human friendly" todo tipo de datos.

Apache Kafka

Apache kafka[2] es una plataforma de software de procesamiento de flujo de código abierto desarrollada por Apache Software Foundation, escrita en Scala y Java. Proporciona una plataforma unificada, de alto rendimiento y baja latencia para manejar fuentes de datos en tiempo real.

Prometheus/Grafana

Prometheus[21] es una base de series de tiempo que almacenan datos ordenados cronológicamente y un sistema de monitoreo y alertas. Por otra parte, Grafana[14] es la plataforma de análisis para métricas, que le permite consultar, visualizar, alertar y comprender los datos, sin importar dónde estén almacenadas.

3.5. Tecnologías rechazadas

Docker Engine/Toolbox

Docker Engine y Docker Toolbox[11] son herramientas que nos permiten la integración de contenedores de manera local. Es la base de la arquitectura.

Presto

Presto[20] es un motor de consultas SQL distribuido de alto rendimiento para big data. Su arquitectura permite a los usuarios consultar una variedad de fuentes de datos como Hadoop, AWS S3, Alluxio, MySQL, Cassandra, Kafka y MongoDB. Incluso se pueden consultar datos de varias fuentes de datos dentro de una sola consulta. Presto es un software de código abierto impulsado por la comunidad lanzado bajo la licencia Apache.

La elección de MongoDB en lugar de Presto se debe a las facilidades que MongoDB proporciona para su integración nativa con Kafka mediante la librería pymongo de Python.

Apache Impala

Con Impala[1], puede consultar datos, ya sea almacenados en HDFS o Apache HBase, incluidas las funciones SELECT, JOIN y agregadas, en tiempo real. Además, Impala utiliza los mismos metadatos, sintaxis SQL (Hive SQL), controlador ODBC e interfaz de usuario (Hue Beeswax) que Apache Hive, proporcionando una plataforma familiar y unificada para consultas orientadas por lotes o en tiempo real.

Para evitar la latencia, Impala elude MapReduce para acceder directamente a los datos a través de un motor de consulta distribuido especializado que es muy similar a los que se encuentran en los RDBMS (Relational Database Management System) comerciales paralelos. El resultado es un rendimiento en un orden de magnitud más rápido que Hive, según el tipo de consulta y configuración.

La elección de MongoDB en lugar de Apache Impala se debe a las facilidades que MongoDB proporciona para su integración nativa con Prometheus y por ende con Grafana.

Apache Kudu

En Apache Kudu[3] al igual que SQL, cada tabla tiene una CLAVE PRIMARIA formada por una o más columnas. Puede ser una sola columna, como un identificador de usuario único, o una clave compuesta, como una tupla para una base de datos de series de tiempo de una máquina. Las filas se pueden leer, actualizar o eliminar de manera eficiente mediante su clave primaria.

El modelo de datos simple de Kudu hace que sea muy fácil portar aplicaciones heredadas o construir nuevas: no hay necesidad de preocuparse por cómo codificar sus datos en blobs binarios o darle sentido a una enorme base de datos llena de JSON difícil de interpretar. Las tablas son autodescriptivas, por lo que puede utilizar herramientas estándar como motores SQL o Spark para analizar sus datos.

El rechazo de esta tecnología viene potenciado por el hecho de que no presenta conexión nativa con Prometheus y pese a su gran potencia y versatilidad como base de datos, se ha elegido MongoDB por su integración sencilla y directa con el motor de consultas, Prometheus.

4. Diseño de la arquitectura

La estructura de este proyecto está muy clara, con una evidente diferenciación entre los diferentes componentes que forman parte de la arquitectura y hacen posible la completa funcionalidad de la misma.

Como ya he explicado anteriormente, este proyecto está constituido por cinco contenedores que podemos agrupar en 4 pilares fundamentales, la base de datos MongoDB, donde se almacenan los datos a ser estudiados, las métricas, implementadas mediante las tecnologías Prometheus y Grafana que permiten visualizar de una forma clara y concisa los rendimientos a nivel de base de datos de nuestra arquitectura, el flujo de los datos, implementado con un bus Kafka, que permite la recolecta desde la fuente de datos hasta la inserción en la base de datos de una forma rápida y eficaz, y finalmente, la visualización gráfica, mediante el uso de MongoDB Charts, que permite la representación de los datos de una forma sencilla y agradable para el entendimiento humano.

Una representación en forma de diagrama de la arquitectura sería la siguiente:

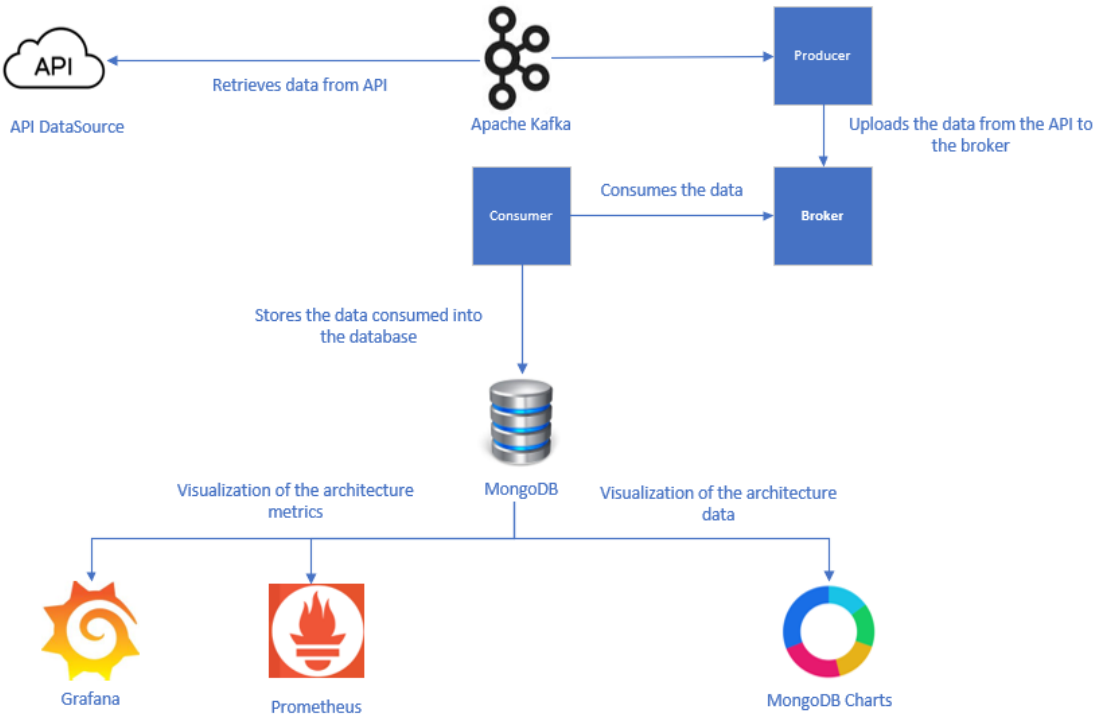

Figura 5: Diagrama de la arquitectura

Date,Country,Confirmed,Recovered,Deaths
2020-01-22,Afghanistan,0,0,0
2020-01-23,Afghanistan,0,0,0
2020-01-24,Afghanistan,0,0,0
2020-01-25,Afghanistan,0,0,0
2020-01-26,Afghanistan,0,0,0
2020-01-27,Afghanistan,0,0,0
2020-01-28,Afghanistan,0,0,0
2020-01-29,Afghanistan,0,0,0
2020-01-30,Afghanistan,0,0,0
2020-01-31,Afghanistan,0,0,0
2020-02-01,Afghanistan,0,0,0
2020-02-02,Afghanistan,0,0,0

Figura 6: Formato de los datos tratados

Estos datos han sido sacados de la API pública que se encuentra en el Github de datasets/covid-19[4]. Se puede observar que está formado por 5 columnas, las cuales nos determinan la fecha en la que se tomaron los datos, el país donde se tomaron, los casos confirmados ese día, los casos recuperados ese día, y finalmente, los fallecimientos del día en cuestión. Esta API contiene los datos especificados anteriormente desde el día 22 de enero de 2020 hasta el día 15 de mayo de 2021. El proceso de actualización de estos datos es tan sencillo como descargar la API de datos actualizada e implantarla como input en el bus Kafka para su procesamiento.

Una vez tenemos definida la estructura de nuestros datos y sabemos con qué cabecera nos va a llegar, es el momento de procesarlos e introducirlos en la base de datos. Para realizar este proceso, se utiliza el ya mencionado varias veces, bus Kafka, el cual nos permite procesar en tiempo real y de forma rápida y eficaz los datos tratados. Su implementación se realiza mediante dos scripts de Python, donde uno se encarga de la gestión de los datos desde la fuente de datos, es el llamado productor y los deja almacenados en el broker, punto común, y el segundo se encarga de la recogida de los datos desde el punto común, el broker, y es el llamado consumidor.

A partir de este punto, el siguiente paso es conectar la base de datos al Prometheus, el cual se conectará al Grafana y permitirá una visualización de las métricas de una forma clara

y concisa, permitiéndonos controlar y analizar el rendimiento de nuestra arquitectura. Esta parte es común independientemente del tipo de datos puesto que se centra en el rendimiento de la arquitectura en sí.

Por otra parte, se encuentra la visualización gráfica, la cual sí es independiente de cada caso de uso, puesto que la visualización de los datos y de qué tipo de datos se visualizan depende de la propia naturaleza de los datos de entrada que se recogen de la API externa.

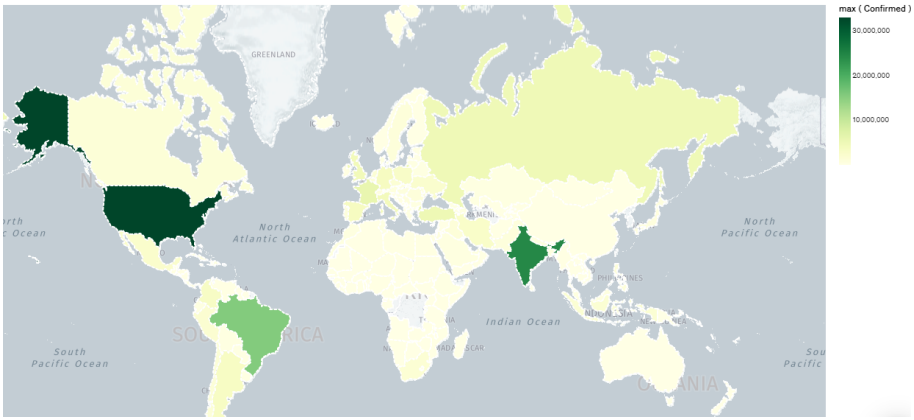

Figura 7: Visualización gráfica de los datos en MongoDB Charts

En la imagen superior se muestra la suma de los casos confirmados de COVID-19 hasta el día 15 de mayo de 2021 por país. Esta representación de los datos permite al usuario poder ver de una forma más clara y amigable una comparación entre países y extraer conclusiones a posteriori. Además, la representación en forma de mapa, hace mucho más cómoda la interacción con los datos recogidos.

5. Implementación

5.1. Contenedor 1 - Prometheus

El primer docker está compuesto por un centos 8, ya que es un sistema operativo ligero y para la implementación de los servicios que quería montar es una gran solución puesto que me proporciona lo básico para poder empezar a instalar las dependencias que me interesan. La configuración del mismo la he realizado mediante un Docker-compose ya que me permite configurar diferentes servicios, es decir, contenedores, al mismo tiempo

Para la implementación de docker del prometheus tuve que configurar un Dockerfile, mediante el cual se hace el build del docker-compose para arrancar el conjunto de los dockers. Además, tuve que configurar diferentes volúmenes(directorios compartidos) para poder realizar tareas como el inicio de servicios por parte del contenedor, o el uso de dbus para la instalación de paquetes. Debido a problemas con el systemd y los cgroups, tuve que crear un directorio dedicado a los cgroups dentro de la docker machine para evitar que el contenedor se quedase "Freezing". La configuración para la creación de este directorio la genero en un pequeño script el cual se encuentra ejecutable dentro de la máquina. Además, para evitar problemas de configuración generados por selinux y por firellad.service, opté por desactivar ambos servicios y configurar el docker sin ellos.

Debido a que el proyecto se hace de forma local, no va a producirse ningún tipo de entrada enemiga desde el exterior y por ello podemos desactivar el firewall. Otro servicio que he desactivado es el kdump, un servicio dedicado a reservar memoria para un posible crash del kernel. En este caso, como podemos pensar, no se producirá ningún caso el crash del mismo debido a que la configuración es propia y sin dependencias externas que puedan llevar a que suceda. En todo caso, siempre podría configurarse en el script de inicio "prometheus.sh", reservando ahí espacio para cuando sucediera.

También, utilizo el puerto 9090 en el contenedor, ya que es el puerto destinado al servicio de Prometheus, mientras que para conectarme desde mi ordenador, es decir, de forma local, utilizo el puerto 9091 para conectarme al mismo, evitando conflictos ya que el puerto 9090 es bastante utilizado. Para conectarnos localmente accedemos a la ip de la máquina virtual docker (se puede encontrar mediante el comando docker machine ip) seguido del puerto asignado al contenedor, en mi caso, el 9091. (http://192.168.99.100:9091)

Para poder dar privilegios al docker, en el fichero "docker-compose.yml" inicializo la variable "privileged:true" con el objetivo de que el contenedor docker sea capaz de utilizar todos los privilegios proporcionados por el sistema anfitrión.

Para la comunicación entre los dockers, creo una red local llamada private_network con

el objetivo de conseguir encapsular de una mejor forma mi arquitectura y por la cual conecto los dockers entre sí para realizar el intercambio de datos y/o información.

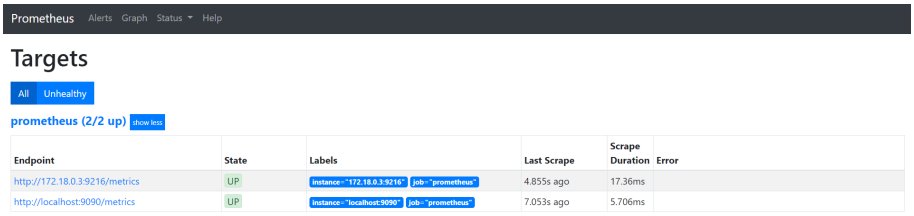

Figura 8: UI de Prometheus con la Base de Datos configurada como target

5.2. Contenedor 2 - MongoDB

En un principio realicé la implementación de la base de datos mediante el servicio Presto, el cual, no es en sí una base de datos, sino que es un intermediario entre la base de datos y el Prometheus. Esto hace que tenga una serie de ventajas como poder conectar a la vez diferentes bases de datos y ejecutarlas en paralelo sobre el servicio Presto. Una vez tuve implementado el servicio, me dispuse a iniciar una base de datos MongoDB dentro del mismo, con el objetivo de que fuese mi base de datos inicial. Esta configuración, aparentemente bastante simple, me llevó a darme cuenta de que no era la mejor idea a nivel arquitectural ya que me generaba conflictos entre el propio MongoDB y el Prometheus. Además, al tener que implementar manualmente la base de datos dentro del Presto, perdía eficiencia y modularidad la arquitectura que estaba generando, característica que me parece esencial para este tipo de proyectos.

A continuación, indagué sobre otra serie de servicios que pudieran suplir al servicio proporcionado por Presto y encontré tecnologías muy interesantes. En primer lugar, ví que a nivel de rapidez el que estaba delante era Impala, un motor de consultas SQL de procesamiento masivo paralelo de código abierto para datos almacenados en un clúster de ordenadores que ejecuta Apache Hadoop. Realicé la implementación del mismo y el posterior testeo y funcionó perfectamente, además, con una pequeña configuración consigues tener un par de nodos trabajando y recibiendo y tratando consultas de manera eficaz y rápida. Además, en ese mismo instante conocí Grafana, un visualizador gráfico muy potente e ilustrativo que me permitiría enriquecer la parte visual de mi arquitectura. Grafana tiene conexión directa con Prometheus por lo que podría conectar este a la base de datos y desde él, conectarme directamente al Grafana. Consiguiendo así que los datos tratados en la base de datos se visualicen en los gráficos proporcionados por Grafana.

El problema surgió cuando quise conectar Impala con Prometheus y por ende con Grafana. No cuenta con conexión nativa con ninguno de los dos y por lo tanto no me permitía trabajar directamente entre servicios. Llegados a este punto, sopesé diferentes opciones, la primera era mirar de encontrar algún plugin instalable dentro del Grafana que tuviera como fuente de datos Impala, el cual, pese a buscar durante unos días no encontré. Mientras buscaba plugins, observé que la base de datos MongoDB contaba con uno nativo y además, contaba con algo llamado MongoDB exporter, que consiste en una configuración que permite que el servicio proporcionado por Mongo sea visible en el exterior, en este caso, visible para mí servicio Prometheus. Esta idea fue la que finalmente escogí e implementé en mi arquitectura.

La implementación de MongoDB fue sencilla, creada dentro de un Docker mediante la configuración del Docker Compose. Conectando los servicios a la red privada local generada por el Docker que contiene Prometheus. Una vez tuve configurado el mongo dentro del Docker y ejecutándose, instalé dentro del MongoDB el exporter con el objetivo de que las consultas pudieran ser vistas y tratadas por Prometheus y por consiguiente por Grafana. Prometheus está configurado para "rastrear targets". Estos targets pueden ser aplicaciones instrumentadas (como aplicaciones Java instrumentadas, por ejemplo), Pushgateway o exportadores. Los exportadores son una forma de vincularse a una entidad existente (una base de datos, un servidor proxy inverso, un servidor de aplicaciones) para exponer métricas a Prometheus. El exportador de MongoDB es uno de ellos. Prometheus se vinculará a los exportadores de MongoDB y almacenará métricas relacionadas en su propio sistema de almacenamiento interno. A partir de ahí, Grafana se vinculará a Prometheus y mostrará métricas en los paneles del tablero de los valores deseados, ganando con ello una mejor visualización de las métricas establecidas y posible extracción de conclusiones a posteriori.

Además, conjuntamente con la base de datos se levanta otro contenedor que contiene Mongo Express, un servicio para la gestión de nuestra base de datos mediante una interfaz gráfica sencilla y fácil de entender. Su objetivo es permitirnos realizar más cómodamente la configuración de la base de datos y tener mayor control sobre ella.

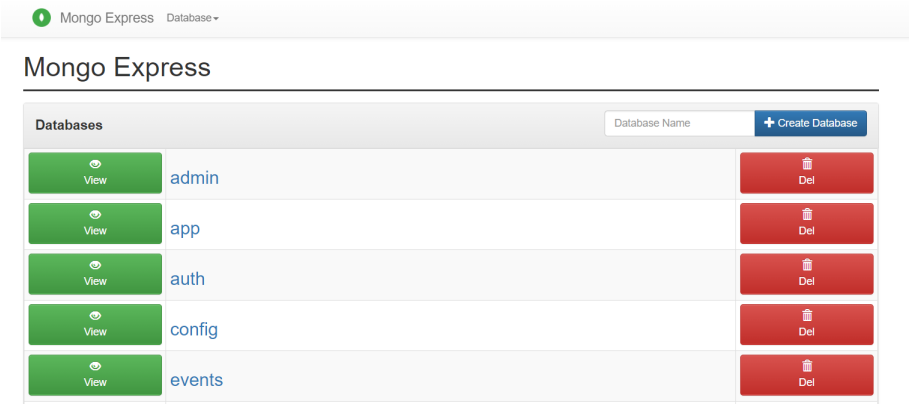

Figura 9: Bases de datos existentes

Como se puede observar en la figura superior, se puede acceder de forma individual a cada base de datos y de esta forma gestionar cualquier tipo de operación dentro de ella, permitiéndonos el borrado, inserción y actualización de los datos de una forma sencilla y rápida.

Este servicio también nos permite tener una visualización del estado del servidor que contiene la base de datos y de esta forma prevenir errores y evitar fallos en el sistema que repercutirían en toda la arquitectura.

Server Status

Hostname	64c7393e3f19	**MongoDB Version**	4.4.1
Uptime	695362 seconds (8 days)	**Server Time**	Mon, 14 Jun 2021 10:51:58 GMT
Current Connections	22	**Available Connections**	51178
Active Clients	2	**Queued Operations**	0
Clients Reading	2	**Clients Writing**	0
Read Lock Queue	0	**Write Lock Queue**	0
Disk Flushes		**Last Flush**	
Time Spent Flushing	ms	**Average Flush Time**	ms
Total Inserts	204817	**Total Queries**	64985
Total Updates	5827	**Total Deletes**	15

Figura 10: Métricas del servidor que contiene la Base de Datos

5.2.1. Problemas solucionados

Para poder realizar la conexión del bus kafka que trata los datos desde un csv generado localmente a partir de una API de datos, en este caso, del Covid-19, he conectado un cliente de mongo a la ip local utilizada por el contenedor del servicio mongo, con el objetivo de hacer posible la inserción de forma directa y en tiempo real por parte del consumidor kafka a nuestra base de datos mongo, que a su vez, envía los datos a medir al Prometheus para su futura visualización gráfica. Para permitir dicha conexión por parte del servicio mongo, he modificado el fichero etc/mongod.conf habilitando el setting bindIpAll: true, abriendo así la conexión desde cualquier IP a mi servicio mongo. permitiendo el tratado de datos dentro de la base de datos.

5.3. Contenedor 3 - Grafana

Con tal de conseguir tener una mejor visualización a nivel de rendimiento, he decidido implementar un contenedor que tenga Grafana. Esta herramienta nos permite conseguir una visualización más gráfica del flujo de nuestra arquitectura, y así, permitirnos captar posibles errores tanto de rendimiento como de tratamiento de los datos. Para poder rastrear en Grafana lo que esta sucediendo en nuestra base de datos, en este caso MongoDB, es necesario establecer como fuente de datos el sistema que se encargue de ellos, en este caso Prometheus. Para ello he seguido la documentación oficial de Prometheus[14], la cual permite de forma sencilla hacer esta configuración.

Previamente ha sido configurado el MongoDB exporter para poder tener conectada la base de datos con Prometheus y que este sea capaz de captar el flujo de datos que está sucediendo en la base de datos.

Una vez sepamos que Prometheus está conectado a nuestra base de datos, el siguiente paso sería establecer como fuente de datos de nuestro Grafana, la url donde estemos exponiendo el Prometheus, con el objetivo de establecer un flujo directo desde la base de datos de MongoDB hasta el Grafana pasando por el Prometheus.

Una vez establecida la conexión, tenemos que importar un gráfico en formato JSON para poder visualizar diferentes métricas, permitiéndonos tener de forma clara y concisa los diferentes sucesos que ocurren en nuestra base de datos en tiempo real.

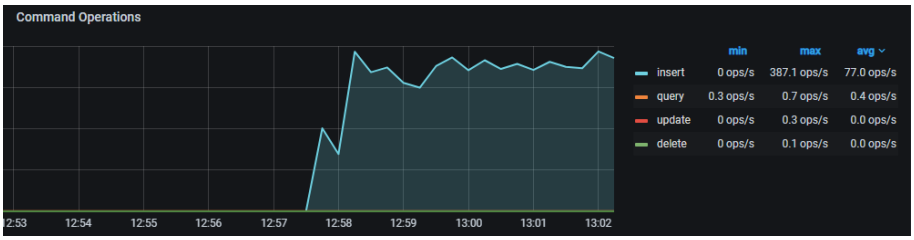

Figura 11: Operaciones CRUD recibidas en la base de datos

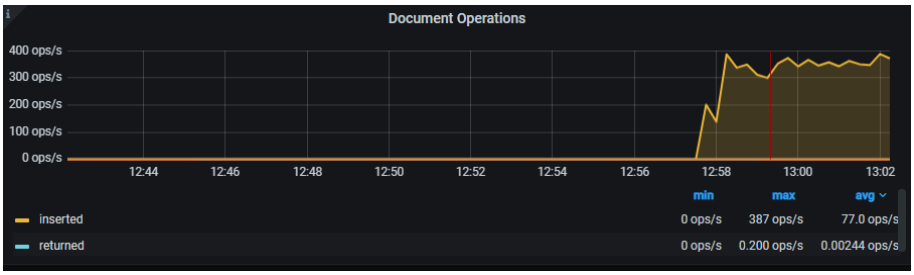

Figura 12: Número de operaciones por segundo recibidas en la base de datos

5.4. Contenedor 4 - Kafka

La rapidez y facilidad de implementación, sumado a la posibilidad de realizar en formato "stream" el flujo de los datos, ha hecho que kafka sea la mejor opción para cubrir el mapeo de los datos desde la API de datos hasta la inserción en la base de datos.

La implementación de kafka consiste en dos contenedores, uno, el zookeeper, es el encargado de:

- Elegir un controlador entre los brokers, encargado de mantener la relación líder-seguidores de todas las particiones.
- Mantener información relativa a los topics: lista de topics existentes, número de particiones, dónde se encuentran las réplicas.
- Mantener una lista de todos los brokers activos en cada cluster.

Y el otro, el servidor kafka, el cual es el encargado de gestionar el flujo de los datos utilizando la gestión productor-consumidor. Para dicha implementación, he creado dos scripts basados en[5] en Python. El primero se encarga de establecer la conexión con el broker, el cual tiene la ip de tu dispositivo local, es decir, la ip local de tu ordenador funcionará como broker en la arquitectura kafka y hace referencia al servidor. Además, este primer script se encarga del procesamiento de los datos que le llegan en formato csv, con el fin de cargarlos en el broker con el formato deseado para que posteriormente el consumidor sea capaz de leerlos correctamente.

```
producer = KafkaProducer(bootstrap_servers='192.168.56.1:9092',
value_serializer=lambda K:dumps(K).encode('utf-8'))

header = ["Date", "Country", "Confirmed", "Recovered", "Deaths"]
with open(args.filename) as file:
    reader = csv.DictReader( file )
    for message in reader:
        row={}
        for field in header:
            row[field]=message[field]
        producer.send(topic, row)
        producer.flush()
        print(row)
sys.exit()
```

Figura 13: Consumo de los datos y envío al broker para futuro consumo e inserción en la base de datos

Una vez se hayan cargado todos los datos en el broker, entra en juego el segundo script. Este script es el encargado del consumo de los datos que han sido cargados previamente por el productor en el broker. Igualmente como ip se utiliza la ip local de nuestro ordenador con la que accederemos a los datos. La función principal de este segundo script es la recogida de los datos del broker, mediante la suscripción a este y su inserción dentro de la base de datos, en este caso MongoDB.

Para ello, hay que importar un módulo llamado MongoClient el cual pertenece a la librería pymongo, el cual nos permite de forma nativa, conectarnos directamente a una base de datos Mongo desde nuestro script de Python. Al tener todos los contenedores en la misma red privada, es posible establecer conexión desde el script de kafka a la base de datos Mongo, permitiéndonos insertar de forma sencilla y rápida todos los datos consumidos desde el broker.

```
    #Connection to MongoDB to insert the values
    client = MongoClient('172.18.0.3', 27017)
    collection = client.metadata.Covid19
    while True:
        try:
            msg = consumer.poll()

            if msg.error():
                raise EopError(msg)
            else:
                message = msg.value()
                value = json.loads(message.decode())
                collection.insert(value)
                print(value)
```

Figura 14: Recogida de datos desde el broker e inserción en la Base de Datos

Como se puede obervar en la figura 14, la inserción de los datos en la base de datos se realiza mediante la conexión a la colección que queremos generar dentro del MongoDB, en mi caso se llamará Covid19. Esta inserción se realiza leyendo línea a línea los valores que se encuentran en el broker Kafka e insertándolas al mismo tiempo en la colección generada. Además se hace un print del valor leído e insertado para tener una mejor visualización del formato de los valores que estan siendo introducidos.

5.5. Contenedor 5 - MongoDB Charts

Finalmente, con el objetivo de tener una visión más gráfica de los datos y poder así, interpretarlos mejor, me decanté por la implementación de un contenedor que implementase MongoDB Charts, el cual, permite conectar como fuente de datos una base de datos Mongo. MongoDB Charts expone una UI en el puerto 80, generando así una forma sencilla de interactuar con nuestra base de datos y generar los respectivos gráficos. La posibilidad de acceder a nuestra base de datos se debe a que, como he comentado anteriormente, todos los contenedores se encuentran en la misma red privada permitiendo la comunicación y transferencia de datos entre todos ellos. Esta configuración se ha hecho basada en el ejemplo del usuario alexion1 en Github[17].

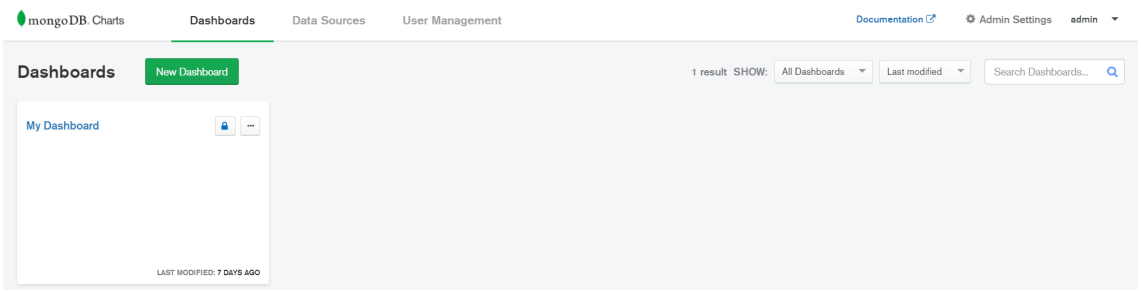

Figura 15: Interfaz gráfica de MongoDB Charts

Una vez nos encontramos en la página principal de la interfaz gráfica, es el momento de generar un gráfico, para ello, el primer paso es establecer una fuente de datos, y en nuestro caso, nuestra base de datos Mongo, utilizando la URI(Uniform Resource Identifier) que hemos expuesto desde el contenedor que contiene la base de datos (mongodb://mongodb_exporter:password@172.18.0.3). Con esto, seremos capaces de acceder a la colección que contiene los datos insertados mediante el bus Kafka.

Además, MongoDB Charts nos permite añadir fuentes de datos desde diferentes direcciones URI con el objetivo de poder unificar en un único gráfico valores recogidos desde diferentes bases de datos. De esta forma, podemos extraer mejores conclusiones sobre los datos y podemos compararlos de una forma más eficaz y rápida.

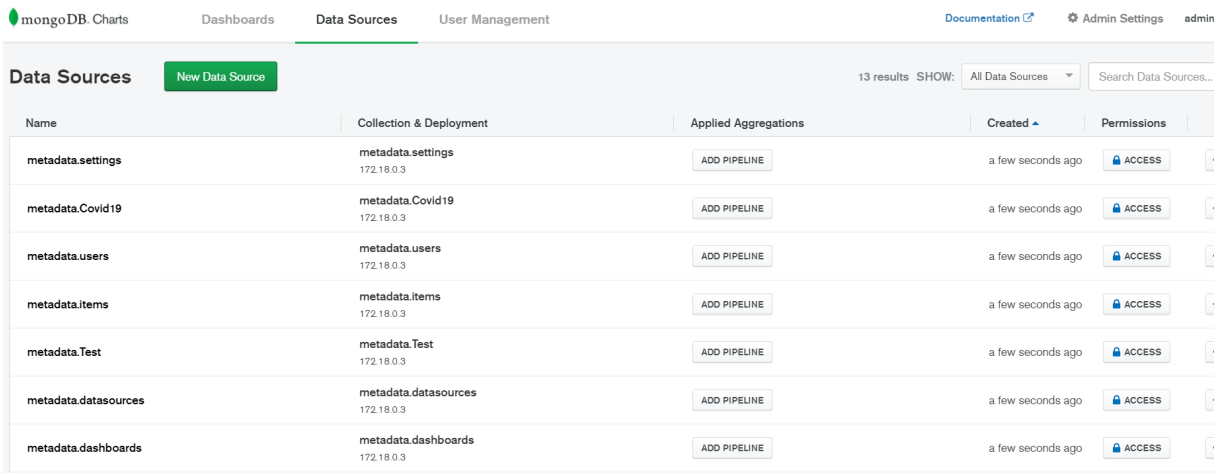

Figura 16: Fuentes de datos

Como se puede observar en la figura 16, una vez establecida la fuente de datos, seremos capaces de visualizar de forma clara las diferentes fuentes con las que podremos trabajar y visualizar sus datos, donde se puede observar que aparece nuestra colección Covid19, la cual contiene todos los datos provenientes de la base de datos e introducidos mediante el bus kafka. Se puede observar que la interfaz es clara y concisa con diferentes columnas explicativas donde cabe destacar el nombre, las colecciones y los permisos, donde podemos gestionar el acceso a cada colección de la forma deseada, permitiéndonos así controlar de forma individual cada una de las fuentes de datos.

La última característica de MongoDB Charts que cabe destacar es la generación del gráfico, esta se nutre de los datos que previamente han sido insertados mediante la URI como fuente de datos y nos permite tener una visualización clara de ellos con diferentes tipos de gráficos, desde diagramas de barras tanto verticales como horizontales hasta mapas de calor.

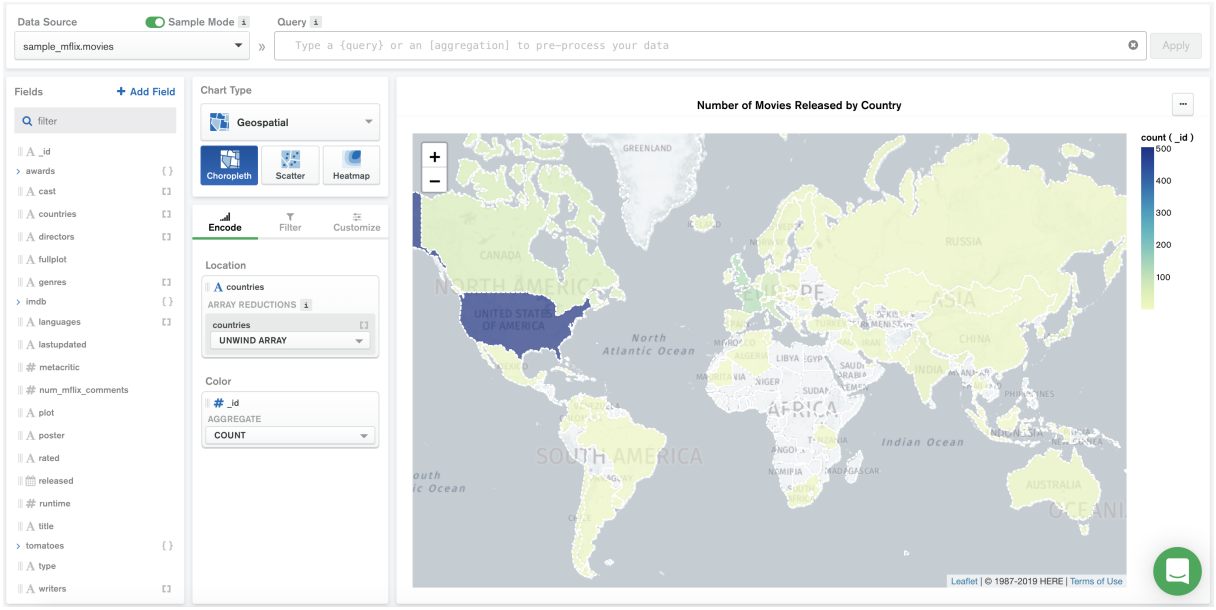

Figura 17: Creación del gráfico a visualizar

La figura 17 muestra la funcionalidad que presenta el MongoDB Charts a la hora de generar un gráfico. Se puede observar que es muy intuitiva y fácil de manejar, con un sistema drag and drop permite de forma sencilla y rápida familiarizarse con su funcionamiento. Me ha permitido generar de forma fácil y cómoda una visualización gráfica y precisa de los datos que han sido recogidos desde la API y enviados a la base de datos y poder así extraer conclusiones más concisas. La configuración del mapa se ha hecho siguiendo la documentación oficial de mongodb[18].

6. Análisis

Como ya se ha explicado anteriormente, la arquitectura cuenta con un conjunto de componentes interconectados entre sí, los cuales permiten que tanto el rendimiento como la modularidad de la arquitectura sean lo mejor posibles. Estos componentes, están interconectados entre sí mediante una red local de tipo bridge generada con el objetivo de conseguir la mejor comunicación entre contenedores para así reducir al máximo los tiempos de conexión entre ellos.

```
version: "3.6"

services:
  prometheus:
    build: .
    image: "centos:centos8"
    container_name: 'prometheus'
    privileged: true
    volumes:
    - ~/configs/prometheus:/configuration
    - /sys/fs/cgroup:/sys/fs/cgroup:ro
    ports:
    - "9091:9090"
    tty: true
    networks:
      - private_network
networks:
  private_network:
    driver: bridge
    external: true
```

Figura 18: Ejemplo de conexión desde el Prometheus a la red privada generada

Como se puede observar en la figura 18, donde se muestra un ejemplo de cómo se conecta el contenedor de Prometheus a la red privada, esta red generada, se llama private_network y es de tipo bridge para dar la posibilidad de comunicación entre los diferentes contenedores y el host. Además, para la mejora del rendimiento de los contenedores, se genera un volumen compartido entre le host y los mismos que involucra a los cgroups(control groups)[7] con el objetivo de aumentar al máximo el rendimiento de la arquitectura puesto que limita los recursos que cada contenedor puede consumir tanto a nivel de memoria como de CPU(Central Process Unit).

El hecho de tener un encapsulamiento de los contenedores dentro de una misma red privada figura 19 aumenta la seguridad de la arquitectura y la hace menos vulnerable ante ataques externos, además, el sistema de gestión mediante las ip de los contenedores reduce todavía más la posibilidad de ser atacado desde el exterior de la red.

```
"Containers": {
    "4d4cf2a332b6415049ad8432dc24e21fed037a6154595e55bf47474e0bdef472": {
        "Name": "kafka_zookeeper-server_1",
        "EndpointID": "4d6b44d37c1dbcf702bf10512f01f33ef76c2219a73e0c6a5c09ff011b6293d7",
        "MacAddress": "02:42:ac:12:00:05",
        "IPv4Address": "172.18.0.5/16",
        "IPv6Address": ""
    },
    "4fa53406c79fc28845b562fa1d78f8e9bdcdd29a26a7b8841e35e680061af4d3": {
        "Name": "mongo_charts",
        "EndpointID": "fbe7e1d56d14157f0750ef5564fb42cf795c534967d75aea16b77a72c943c48e",
        "MacAddress": "02:42:ac:12:00:09",
        "IPv4Address": "172.18.0.9/16",
        "IPv6Address": ""
    },
    "64c7393e3f19e5a479a75535eb46aacd52b4b90a9ef820b6be63d819a7456bce": {
        "Name": "mongodb",
        "EndpointID": "decb2aa2a3b4e50795fe2fa63c2561f12dc25178034c397062245dd454771d06",
        "MacAddress": "02:42:ac:12:00:03",
        "IPv4Address": "172.18.0.3/16",
        "IPv6Address": ""
    },
    "84251d0d63d5374fec3ce8e04550a64dc477a4233d71ca9257df5656f2011e6e": {
        "Name": "kafka_kafka-server1_1",
        "EndpointID": "e9b7f8c1594a88ce11d8585cad5dba21cdf5b9e422873b980c5ab55047ee2e08",
        "MacAddress": "02:42:ac:12:00:06",
        "IPv4Address": "172.18.0.6/16",
        "IPv6Address": ""
    },
    "973d6edd52bcfd0e5a26d6440aaf6003b36b814a2375285c7f55732154ca61c9": {
        "Name": "prometheus",
        "EndpointID": "b0865876dd2b31e3581aa715d09b0ffa6d99029be946d87b7ed0ed4b6dfcace9",
        "MacAddress": "02:42:ac:12:00:02",
        "IPv4Address": "172.18.0.2/16",
        "IPv6Address": ""
    },
    "ebe5f8f27bfa36f9c076ef1d609ade82d1831eab22f9a7efe5f6c18e9c406620": {
        "Name": "mongo_web_ui",
        "EndpointID": "05602a47f122a692f648d2b4e737a28d92634ee5de5aeaeacc15d1c174b9f41a",
        "MacAddress": "02:42:ac:12:00:04",
        "IPv4Address": "172.18.0.4/16",
        "IPv6Address": ""
    }
},
```


Figura 19: Interconexión de los contenedores en una misma red privada

En cuanto a los datos se ha realizado un estudio para poder conseguir el mayor rendimiento y velocidad de consumo. Primero se sopesó el trabajar con ficheros json, los cuales también se consumían haciendo uso de scripts de Python mediante librerías importadas. Pude observar que el rendimiento era óptimo pero que con grandes volúmenes de datos el rendimiento de la arquitectura se quedaba algo corto. Por ello, seguí indagando y comparando diferentes formatos de consumo de datos, donde encontré que el formato csv era el más idóneo para esta arquitectura debido a que yo estaba buscando rapidez y eficiencia, puesto que al tener un formato en columnas y una fácil implementación con Python, me fue muy sencillo implementarlo dentro de la arquitectura. Es cierto que el formato json es más genérico, puesto que al leer de un csv se han de hacer configuraciones en el script de

Python debido a que las cabeceras del fichero csv no son estáticas, pero al ser tan genérico, la lectura de los ficheros por parte de productor de Kafka se hace de forma mucho más lenta.

También, se ha realizado un test de rendimiento para extraer el rendimiento del consumo de los datos que presenta la arquitectura. Este test se ha realizado utilizando un procesador AMD Ryzen 5 3600 con 16GB de memoria RAM. Mediante la importación de la librería timer de pyhton, se ha calculado el tiempo que transcurre desde el inicio hasta el fin del consumo de datos desde la API de datos hasta su inserción en el broker del servidor kafka. La figura 20 presenta la implementación de este test

```
    start = time.perf_counter()

    with open(args.filename) as file:
        reader = csv.DictReader( file )
        for message in reader:
            row={}
            for field in header:
                row[field]=message[field]
            producer.send(topic, row)
            producer.flush()
            print(row)
    stop = time.perf_counter()
    print(f"Data consumed in {stop - start:0.4f} seconds")
```

Figura 20: Configuración para realizar el test de rendimiento de consumo de datos

La figura 21 muestra el tiempo en segundos que tarda la arquitectura en consumir todos los datos provenientes de la API. Se puede observar que realiza la lectura de las 92160 líneas del fichero csv se realiza en tan solo 224.5 segundos, lo que genera una velocidad de lectura de 410.5 líneas por segundo, una velocidad suficientemente rápida para gestionar el consumo de ficheros de datos bastante grandes.

```
{'Date': '2021-05-15', 'Country': 'Zimbabwe', 'Confirmed': '38554', 'Recovered': '36318', 'Deaths': '1582'}
Data consumed in 224.5052 seconds
```

Figura 21: Tiempo transcurrido en segundo del consumo de los datos

7. Conclusiones

Después de realizar la creación y composición de la arquitectura, se pueden sacar en claro diferentes conclusiones diferenciadas en 3 partes.

Por una parte, se encuentra la arquitectura como conjunto de principio a fin, donde podemos concluir que tanto su rendimiento como su modularidad nos permiten que sea utilizada en cualquier tipo de proyecto, independientemente de la fuente y tipo de datos. Esta característica es remarcable y ratifica el haber conseguido uno de los objetivos propuestos para este proyecto. Además hace posible su reusabilidad para futuros proyectos.

En segundo lugar, se encuentra el flujo de los datos, parte fundamental en una arquitectura, donde se prioriza la velocidad y consistencia de los datos. Cabe destacar el gran rendimiento que presenta el bus Kafka, pudiendo procesar de forma paralela el consumo de los datos y su inserción en la base de datos. Podemos concluir que el rendimiento en cuanto al flujo de datos que presenta el bus Kafka ha superado con creces las expectativas iniciales y ha permitido dar un plus de modularidad y rendimiento a la arquitectura.

Y finalmente, se encuentra la parte relacionada con la visualización gráfica, tanto a nivel métrico, donde se observa el rendimiento de la base de datos, como a nivel de visualización de los datos en el mapa, donde se tiene una mejor visualización de los datos insertados. En esta última parte el objetivo a cumplir era tener una clara y correcta visualización de los datos así como poder medir el rendimiento de la base de datos de forma sencilla y con la posibilidad de ver qué y cuántas operaciones se estaba recibiendo. Podemos concluir que ambos objetivos se han cumplido con creces y de forma completa. Además, el cambio a MongoDB Charts para la visualización en el mapa, cosa que en un inicio no estaba planeada, ha brindado al proyecto de una parte visual de los datos mucho más agradable y funcional.

A nivel de caso de uso, podemos pequeñas conclusiones subjetivas sobre la evolución del COVID-19 a lo largo de los meses en todo el mundo, donde claramente se observa que las etapas más duras fueron al inicio de la pandemia en marzo del año 2020, con pequeños repuntes hasta día de hoy. El foco principal de este proyecto es para la arquitectura en sí, dejando un poco en segundo lugar al caso de uso, puesto que por la reusabilidad que la arquitectura presenta, el caso de uso no afecta a su rendimiento.

8. Trabajo futuro

Las posibilidades de ampliación y mejora de este proyecto son bastantes. Por un lado, cabe la posibilidad de poder ampliar el proyecto desde el punto de vista de procesado de datos, donde el bus Kafka permite el consumo de datos de diferentes APIs de forma paralela, lo que permitiría a la arquitectura procesar un mayor número de datos por minuto. Esto se conseguiría mediante el aumento de los brokers que intervienen en la arquitectura, donde cada broker referencia a una APi de datos diferente y contaría con consumidores y productores para poder realizar de forma paralela el consumo e inserción en la base de datos, esto sería de gran utilidad si nuestro proyecto necesitase coger datos desde diferentes fuentes de datos para después unificarlos en la base de datos de la arquitectura, aumentando así la velocidad de consumo de datos.

Por otro lado, también cabría la posibilidad de ampliar la arquitectura mediante el aumento de consumidores, consiguiendo de esta forma un mayor aumento de rendimiento a la hora de insertar los valores en la base de datos en caso de que el número de datos sea muy elevado. Este proceso se realizaría mediante la generación de particiones, las cuales serían consumidas por un único consumidor, para evitar colisiones al consumir los datos. Cada consumidor puede consumir más de una partición pero una partición está asignada únicamente a un consumidor.

Además, también cabe la posibilidad de ampliar la arquitectura en cuanto a almacenamiento se refiere, es decir, presenta la escalabilidad necesaria para poder funcionar con distintas bases de datos MongoDB a la vez, mediante la configuración en los scripts de Python encargados de la inserción de los datos, se conseguiría estar trabajando de forma paralela con diferentes bases de datos.

Otra mejora a implementar en la arquitectura sería en lo que respecta al formato del fichero en el que los datos sonn consumidos puesto que actualmente solo acepta formato csv debido a que se ha buscado velocidad y rendimiento en la arquitectura. Cabe la posibilidad de modificar el script de consumo de datos del Kafka para que sea capaz de entender, por ejemplo, fichero de tipo json, los cuales son muy comunes en cuanto a consumo de datos se refiere. Esta modificación, se conseguiría añadiendo un if condicionante que comprobase la extensión del fichero y según si fuese csv o json gestionar el consumo con unas librerías u otras. Sería un plus a la arquitectura ya que le añadiría modularidad en cuanto al número de APIs que podría atacar y haría la arquitectura todavía más modular y genérica.

9. Apéndice

9.1. Requisitos previos a la instalación

A continuación se exponen los pasos para la generación y configuración de la arquitectura de este proyecto.

Requisitos previos a la instalación:

- Version 3.6 de Python instalada.
- Versión 20.10.5 de Docker instalada, preferiblemente con Docker Desktop para la gestión de los contenedores de forma visual.
- Versión 1.28.5 de docker-compose instalada.
- Virtualización activada en la máquina local para permitir el uso de contenedores.
- Conocimientos básicos sobre Docker.
- Descarga de los ficheros necesarios.

9.2. Instalación y configuración de la arquitectura

Los pasos a seguir para la instalación y configuración de la arquitectura son los siguientes:

1. Carpeta /configs
   - Una vez descargados los ficheros, se ha de colocar la carpeta de configs en C:/Users/¡rootuser¿para poder utilizarlos en los Dockerfiles.
2. Carpeta /containers
   - La segunda carpeta llamada containers, contiene las subcarpetas con los diferentes contenedores. Esta se puede descomprimir donde el usuario desee. Será nuestro espacio de trabajo para la creación de los contenedores.
3. MongoDB
   Una vez tenemos las dos carpetas listas, es momento de empezar con la configuración de los contenedores. El primer contenedor a configurar es el de la base de datos, es decir, el que contendrá MongoDB.

- Para ello entramos en la carpeta /containers/mongo y ejecutamos el comando docker-compose up para levantar el contenedor. observaras que además del contenedor llamado mongodb, también se levanta otro contenedor llamado mongo_web_ui el cual te permitirá de forma más cómoda gestionar el contenido de la base de datos. Una vez tengamos levantados ambos contenedores es momento de configurar el MongoDB.
- Acceder al contenedor mediante el comando docker exec -it mongodb bash y una vez dentro en la carpeta /configuration ejecutaras cada una de las líneas del fichero configuration.sh.
- Una vez hayas hecho este paso, hay que configurar el target del Prometheus para que pueda acceder al mongodb_exporter generado, para ello, se ejecuta el comando docker inspect mongodb y se copia la dirección ip del contenedor en el fichero /configs/prometheus/prometheus.sh en la línea 39, este es el fichero que se ejecuta en la creación del contenedor docker. Esto permitirá al prometheus conectarse directamente a la base de datos MongoDB para el cálculo de las métricas.

4. Prometheus

    - Una vez copiado en el array de targets, procedemos a ejecutar el comando docker compose up desde la carpeta de /containers/prometheus para levantar el Prometheus con el target configurado. Esperamos a que este corriendo y lo comprobamos con el comando docker ps.
    - Una vez tengamos corriendo el contenedor procedemos a realizar la configuración del mismo utilizando el comando docker exec -it prometheus sh /configuration/configuracion_service.sh. Una vez hayamos hecho estos pasos ya tendremos el contenedor con el Prometheus corriendo y conectado a nuestra base de datos. Esto puedo comprobarlo accediendo a la url http://localhost:9091/targets donde te aparecerá como uno de los targets la ip de la base de datos.

5. Grafana

    - El siguiente contenedor a configurar es el Grafana, el cual se encargará de la visualización de las métricas de una forma más visual. Para ellos ejecutamos el comando docker run -d -p 3001:3000 grafana/grafana.
    - Una vez tengamos el contenedor corriendo, accedemos a la UI para su configuración. Para ellos tendremos que acceder a la opción de DataSource y añadiremos una nueva de Prometheus. en la url pondremos la url donde se encuentra nuestro Prometheus, es decir, http://localhost:9091 y pondremos que el acceso es a

través del navegador. Una vez hayamos hecho este paso apretaremos el botón de Save & Test obteniendo un mensaje de éxito.

- A continuación, en la pestaña debajo del buscador, le daremos a import, donde nos llevara a una ventana donde podremos importar un json con nuestro Dashboard, para ello, le damos al botón de "upload file" e importaremos el json que se encuentra en la carpeta de /containers/mongo/dashboards. Una vez hecho, veremos que nos aparecen una serie de gráficas sin datos

6. Kafka

   - El siguiente paso es la configuración del contenedor Kafka, el encargado del consumo y tratado de los datos. Para ello, accederemos a la carpeta /containers/kafka y ejecutaremos el comando docker compose up. Esto nos levantará dos contenedores, el zookeper y el servidor.
   - El primer paso previo a la ejecución de los comandos es la edición del fichero printStream.py en la línea 49. Donde tendremos que poner la ip del MongoDB que obtuvimos anteriormente. Esto permitirá al Kafka conectarse directamente a la base de datos para su inserción. Además, tanto en este fichero como en el fichero sendStream tendremos que modificar la ip del boostrap_server con nuestra ip Ethernet adapter VirtualBox Host-Only Network. Acto seguido, con los dos contenedores activos ,procederemos con la ejecución de los siguientes tres comandos.
   - El primero es docker build -t "kafka". , el cual nos generará una imagen nueva con el contenido de los ficheros de configuración printStream.py y sendStream.py. El segundo es docker run -it –network=private_network kafka python bin/sendStream.py data/countries.csv stream el cual consume los datos del fichero csv(API) y los inserta en el broker para el futuro consumo por parte del consumer kafka. Este proceso puede tardar varios minutos puesto que presenta un gran número de datos y se ha hecho únicamente con un producer.
   - Y por último se ejecuta el comando que hace referencia al producer, el cual coge los datos desde el broker y los inserta en la base de datos ejecutando el comando docker run -it –network=private_network kafka python bin/printStream.py stream. Observamos que la consola se queda a la espera de nuevos inputs, esto significa que a partir de ahora cada vez que el producer envíe datos al broke mediante el stream de datos, este lo escuchará en tiempo real y los procesará para posteriormente insertarlo en la base de datos.

7. MongoDB Charts

   - Finalmente, el último contenedor a configurar es el de los MongoDB Charts, el cual nos permite visualizar de forma gráfica los datos insertados en la base de datos. El primer paso es configurar el fichero /containers/mongodb-charts/charts-monogb-uri, donde cambiaremos la ip con la ip Ethernet adapter VirtualBox Host-Only Network.
   - Una vez hecho esto, desde la carpeta /containers/mongodb-charts ejecutamos el comando docker-compose up. Una vez hecho, tendremos el servicio corriendo en la url localhost:80. A continuación, tenemos que añadir un usuario al servicio, para ello ejecutamos el comando docker exec -it $(docker container ls –filter name=_charts -q) charts-cli add-user –first-name .ªdmin-last-name .ªdmin-email üser@admin.com-password "Password-role ÜserAdmin". Una vez ejecutado recibiremos un mensaje de que el usuario ha sido creado exitosamente y podremos ingresar al servicio utilizando el usuario creado.
   - El siguiente paso es agregar nuestra fuente. Para hacer eso, iremos a la etiqueta Source y añadiremos una fuente con la base de datos que contiene nuestros datos mediante la URI que se encuentra en el fichero /containers/mongodb-charts/charts-monogb-uri y seleccionaremos la colección metadata. Después de eso, generamos un nuevo Dashboard y seleccionaremos la colección que contiene los datos, es decir, metadata.Test. Ahora podremos seleccionar un mapa geoespacial en los tipos de gráficos y agregaremos los países en el tag Location y el valor deseado en el tag Color.
   - Para una correcta visualización de los datos, deberemos cambiar el tipo del dato a visualizar, por ejemplo si queremos ver los casos confirmados, iremos al valor en la parte izquierda, le daremos click izquierdo y cambiaremos el tipo a numérico. Una vez hecho esto, en el tag Color podremos elegir el tipo de aggregate donde seleccionaremos max, visualizando de esta forma los casos de Covid-19 por país hasta la última fecha de recogida de los datos. Además, podemos filtrar por fecha o país en la sección "filter" y añadiendo el campo Date a esta sección.

10. Referencias

[1] *Apache Impala*. URL: https://impala.apache.org/index.html.

[2] *Apache kafka*. URL: https://kafka.apache.org/quickstart.

[3] *Apache kudu*. URL: https://kudu.apache.org/.

[4] *Api data*. URL: https://raw.githubusercontent.com/datasets/covid-19/main/data/countries-aggregated.csv.

[5] *Api scripts*. URL: https://github.com/datasets/covid-19/scripts.

[6] *Artifactory Docker Registry*. URL: https://www.jfrog.com/confluence/display/RTF6X/Docker+Registry.

[7] *cgroups*. URL: https://bobcares.com/blog/docker-performance/.

[8] *containerd*. URL: https://containerd.io/.

[9] *Docker adoption until 2018*. URL: https://www.datadoghq.com/docker-adoption/.

[10] *Docker Desktop*. URL: https://docs.docker.com/docker-for-windows/.

[11] *Docker Toolbox*. URL: http://docs.docker.oeynet.com/toolbox/toolbox_install_windows/.

[12] *DockerHub*. URL: https://hub.docker.com/.

[13] *Flujo de ejecución*. URL: https://astaxie.gitbooks.io/go-system-programming/content/zh/images/docker_flow.png.

[14] *Grafana*. URL: https://prometheus.io/docs/visualization/grafana/.

[15] *LXC*. URL: https://linuxcontainers.org/lxc/introduction/.

[16] *MongoDB*. URL: https://docs.mongodb.com/manual/installation/.

[17] *MongoDB Charts*. URL: https://github.com/alexion1/mongodb_express_charts_docker_compose_file.

[18] *MongoDB Charts Map*. URL: https://docs.mongodb.com/charts/master/chart-type-reference/geo-spatial/.

[19] *Podman*. URL: (http://docs.podman.io/en/latest/).

[20] *Presto*. URL: https://prestodb.io/.

[21] *Prometheus*. URL: https://prometheus.io/docs/introduction/overview/.

[22] *Python*. URL: https://www.python.org/.

[23] *RKT*. URL: https://www.openshift.com/learn/topics/rkt.

[24] *runC*. URL: https://github.com/opencontainers/runc.

[25] *Windows Containers*. URL: https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/.