U-tad

Facultad de Ingeniería de software

MACHMON - Monitorización de sistemas

Proyecto Fin de Carrera

Autor:
Javier Lima

Junio 2020

Contents

Chapter 1

Abstract

This TFG covers a distributed system for monitoring systems (machines). The main objective of the project is to develop a distributed application for the collection of different states of distributed systems. A Server - Workers architecture is proposed, so that the Server runs on one machine, and the Workers run on the machines to be monitored.

The project also requires a communication protocol that has to be design between the different pairs of the application (Server and Workers), abstracting the peculiarities of each system to be monitored. Once the data has been collected, the Server cleans it, enriches it and stores it, so the "UI"(user interface), in this case Grafana, can use the stored data to display graphs of them.

Python is used as the main language, at the same time bash is used to collect the metrics and protobuffer for the communications protocol. As the main tool for managing the software life cycle, git and GitHub are used.

Chapter 2

Introducción

2.1 Justificación y contexto

Actualmente, la monitorización de los ordenadores, es común en todas las empresas con sistemas propios que manejen datos de forma digital, es un paso más en el control de los sistemas, al final acabas aprovechando más los recursos, prevees posibles incidencias, mejoras la experiencia del cliente y ahorras tiempo y dinero.

Los recursos de los que dispones en un ordenador son muy amplios, el tener controlado en todo momento dichos recursos te permite tomar decisiones con mucha más seguridad. Pongamos un ejemplo, imaginémonos un sistema donde corren distintas aplicaciones, ¿cómo sabríamos si es viable introducir otra aplicación dentro del sistema?, a priori no lo sabes, introducirías la aplicación dentro del sistema hasta que en algún momento deje de funcionar. En cambio, si desde un principio supiésemos la salud del sistema, cuanto porcentaje de cpu está gastando o cuanta memoria RAM queda libre para que trabajen paralelamente, podríamos adelantarnos a posibles incidencias y no saturar los sistemas de recursos o por el contrario darnos cuenta que deberíamos aprovechar más los recursos del sistema.

Tener monitorizado tus propios sistemas aporta tranquilidad, es un plus que cualquier cliente se sentería satisfecho, no solo hay que conformarse con elaborar una aplicación y ponerla a funcionar, hay que dejar preparadas ayudas que nos aporten valor por si ocurre alguna incidencia. No es fácil arreglar posibles fallos que pueda tener una aplicación, sobre todo porque es posible que ni si quiera sea fallo de la aplicación, ante eso hay que bajar un nivel y entender que está ocurriendo dentro del sistema.

2.2 Planteamiento del problema

Todo este tiempo en la universidad hemos desarrollado código para generar valor con aplicacones o scripts, pero ¿cómo medimos el rendimiento del sistema realmente?. Imaginemos que tenemos varias aplicaciones funcionando en una máquina, ¿como podríamos saber si es posible meter una nueva aplicación dentro del sistema?. Machmon, es un proyecto para profundizar en los sistemas y así poder saber la salud de ellos, consiste en recoger métricas que nos aporten información sobre la salud de los ordenadores (como la cpu, memoria ram, disco, etc) y representarlas gráficamente para sacar conclusiones sobre sus estados.

2.3 Objetivos del trabajo

Como propósito este trabajo tendrá unos objetivos a cumplir:

2.3.1 Aumentar los conocientos de sistemas linux

Linux es uno de los sistemas más populares entre los desarrolladores, la consola que tiene facilita mucho la operación de esas máquinas, además de que en el mundo laboral son indispensables. Es por ello, que el uso de los sistemas Linux es fundamental.

2.3.2 Utilizar tecnologías nuevas

La programación es muy revolucionaria, en cualquier momento aparece una aplicación nueva capaz de dejar obsoleta a otra. El estar puesto en las nuevas tecnologías nos facilita el trabajo, nos permite ver las utilidades de cada herramienta y los beneficios que nos pueda aportar.

Este trabajo tiene un fin, obtener la salud de los sistemas monitorizados, pero en el camino nos encontremos dificultades que seguramente una herramienta nos las solucione.

2.3.3 Modelo abstracto

Es un objetivo un poco pretencioso, trata de ser capaz de recibir métricas de cualquier tipo de sistema, que exista una entidad capaz de controlar los mensajes recibidos y almacenarlos para su tratamiento, es decir, que si en algún momento decido introducir una nueva máquina en el sistema sea capaz de entender los mensajes también y almacenarlos.

2.3.4 Concurrencia

Es un objetivo fundamental, es necesario tener en cuenta posibles mensajes que lleguen a la vez. La concurrencia nos soluciona el objetivo, tendremos que ser capaces de elaborar un sistema concurrente.

Chapter 3

Estado de la cuestión

En esta sección se analiza el estado del arte para las diferentes piezas requeridas del proyecto, además de las herramientas existentes como los trabajos relacionados.

3.1 Marco teórico del trabajo

El marco teórico del trabajo describe las necesidades que dispone el trabajo, así como las nociones técnicas o científicas implicadas en el trabajo.

3.1.1 Base de datos

Una base de datos es un conjunto de datos pertenecientes a un mismo contexto y almacenados sistemáticamente para su posterior uso. En este sentido; una biblioteca puede considerarse una base de datos compuesta en su mayoría por documentos y textos impresos en papel e indexados para su consulta.

Tras entender esto, necesitamos ir un paso más allá; nuestro sistema debe de describir el estado de los sistemas a tiempo real, es por ello que necesitamos controlar los tiempos de ingesta en la base de datos, es decir saber en todo momento cuando se ha recibido el dato para que a la hora de interpretarlo sea cómodo e intuitivo.

Según los conocimientos adquiridos una base de datos en SQL sería capaz de resolver nuestro problema, solo sería necesario estructurar la base de datos en función de los tiempos en los que lo reciba, pero nos obligaría a

tener presente los timestamp de entrada. Ante ello, SQL nos exige un esfuerzo adicional a la hora de desarrollarlo, por tanto es más recomendable escoger otro tipo de bases de datos.

Figure 3.1: Tipos de bases de datos

Existen distintos tipos de base de datos, en la figura 3.1 se aprecian varios tipos como: distribuidas, orientadas a grafos, de red, deductivas, de series temporales, relacionales, no relacionales, etc; tras un ínfimo entendimiento sobre la utilidad de ellas, se identifica que la que más nos conviene es una base de datos de series temporales, para que cada ingesta esté controlada por su tiempo de entrada y así poder visualizar con mayor facilidad los campos de la base de datos.

3.1.2 Master/Workers (Server/Clientes)

La arquitectura cliente-servidor es un modelo de diseño de software en el que las tareas se reparten entre los proveedores de recursos o servicios, llamados servidores, y los demandantes, llamados clientes. Un cliente realiza peticiones a otro programa, el servidor, quien le da respuesta. Esta idea también se puede aplicar a programas que se ejecutan sobre un solo ordenador, aunque es más ventajosa en un sistema operativo multiusuario distribuido a través de una red de ordenadores.

La idea principal es desarrollar un modelo con una estructura compuesta por Master/Workers siendo utilizadas las mismas metodologías que se usan en el presente. Dicho modelo debe incluir:

- Un servidor capaz de recibir multiples peticiones y saber tratarlas (balanceo de carga).
- Almacenamiento de la información.
- Comunicaciones.
- Disponibilidad.
- Generación de logs.
- Escalabilidad.
- Flexibilidad de configuraciones.
- Control de seguridad para empezar la comunicación con los clientes.
- Control de que clientes sean capaces de conectarse al servidor.

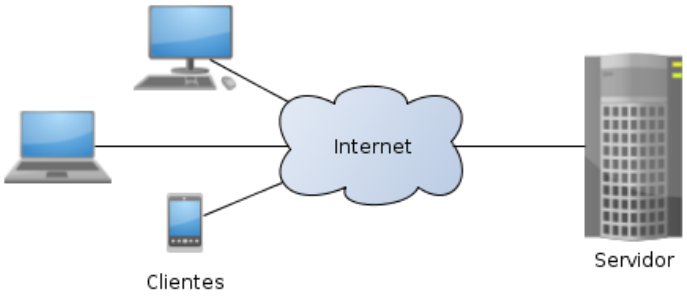

Figure 3.2: Estructura cliente-servidor

3.1.3 Recopilación de métricas

Actualmente existen infinidad de herramientas capaces de describir como es la salud de nuestros ordenadores, un ejemplo de ello son los plugins de telegraf que recogen métricas de todo tipo: cpu, disco, memoria RAM, latencia, temperatura y muchas más. Estas herramientas funcionan de la misma manera, preguntando al ordenador, algunas con comandos más complejos y otros más sencillos, pero viendo la esencia que tienen tan parecida sería posible desarrollar un script que recoja únicamente las métricas que necesitemos o incluso ir añadiendo nuevas métricas en un futuro.

3.1.4 Generación de gráficas

Un aspecto fundamental es la visualización de las métricas que recojamos, si en un principio existían infinidad de herramientas para obtener las métricas, estamos ante lo mismo o más herramientas para poder visualizarlas. Cabe destacar que sería posible diseñar una página web capaz de representarlas a tiempo real, pero con tantas herramientas que dan la mayoría hecho no sale rentable invertir el tiempo en desarrollarlo, obviamente si merece la pena intentarlo, pero en el mercado laboral ya se miden las métricas con el uso de herramientas existentes así que adaptarnos será lo más útil.

A continuación nombraré distintas herramientas, sus puntos fuertes y cuál de ellas disponen de mayor apoyo de la comunidad.

1. Cyfe:
   De las más completas para crearte tu cuadro de mando avanzado. Puedes conectar un montón de fuentes diferentes y con una interfaz intuitiva. La posibilidad de redimensionar widgets rápidamente le da un plus de personalización bastante útil.

2. Klipfolio:
   Es una herramienta analítica que da la posibilidad de ver una comparación de datos mensual para saber lo que se podría corregir.

3. Clicdata:
   Clicdata es mucho más que un dashboard. Quizá tenga algunas limitaciones como cuadro de mandos general, pero su capacidad para resumir y crear grandes infografias de datos, es un añadido que pocas herramientas tienen. Algo malo que tiene es su limitación de espacio, su versión gratuita solo dispone de 1 GB.

4. Qlik:
   Qlik es otra de las herramientas de creación de dashboards más recientes que tenemos. Pese a que anteriormente se la conocía como QlikView, la nueva versión es mucho más intuitiva, ofreciendo la posibilidad de medir la experiencia de usuario en un dashboard muy visual y ameno, adaptándose a todo tipo de segmentos.

5. Sweetspot Intelligence:
   Es una de las que mejor se han adaptado al cambio, puesto que cuenta con una versión mobile first, e incluso una para los SmartWatch, permitiendo búsquedas de KPIs por voz. Por otro lado, con Sweetspot pode-

mos programar actualizaciones de estado e interacciones sin necesidad de estar conectados a Internet.

6. DashThis:
   DashThis es problablemente la herramienta más amena visualmente, siendo a su vez una de las más completas, ofreciendo datos de una forma muy completa y avanzada. Se actualiza periódicamente de forma automática, con su posterior envío por correo, para que no tener que estar consultando continuamente la evolución de nuestros principales KPIs.

7. Grafana:
   Es una plataforma de análisis para la generación de dashboards. Permite consultar, visualizar, alertar y comprender las métricas sin importar dónde estén almacenadas. Su creación de paneles fomenta una cultura basada en datos, además que es la más confiada por la comunidad.

3.2 Trabajos relacionados

Chapter 4

Aspectos metodológicos

4.1 Metodología

El conjunto de procedimientos racionales utilizados para alcanzar el objetivo del trabajo rige una investigación científica y una planificación de fases para el desarrollo del trabajo. En esta primera fase, se necesita un estudio de los conceptos que nos ayude a alcanzar nuestro objetivo. Se trata de barajar las posibilidades existentes para desarrollar cada cada una de las fases. Dichas fases se nombran a continuación:

- Investigación.
- Recopilación de métricas.
- Desarrollo de servidores y clientes.
- Ingesta en base de datos.
- Generación de gráficas.
- Documentación.

De forma excepcional, cabe destacar que como punto de continuidad y como procedimiento para llevar un control del trabajo de fin de grado, se ha realizado el desarrollo bajo la inclusión de la metodología agile "Scrum". Scrum es un marco de trabajo para desarrollo ágil de software que se ha expandido a otras industrias. Es un proceso en el que se aplican de manera regular un conjunto de buenas prácticas para trabajar colaborativamente, en equipo y obtener el mejor resultado posible de proyectos. Aunque sea una metodología colaborativa, se ha realizado individualmente, se han organizado reuniones a modo de reviews, entregas en forma de fin de sprints y planings

para priorizar las tareas más importantes.

4.2 Tecnologías empleadas

En la actualidad hay multitud de herramientas, nos facilitan el trabajo si las conocemos y las utilizamos según el propósito de su existencia. Aunque se nombren muchas herramientas a continuación, existen muchas otras que también podían haber sido de utilidad, un ejemplo de ello es "Prometheus" que no se llega a utilizar en este trabajo, pero su función de recoger métricas se hubiese adaptado muy bien, además de que en el mundo laboral se utiliza con frecuencia.

A continuación se mostrarán todas las tecnologías utilizadas.

4.2.1 Docker

Un contenedor es una unidad de software estándar que empaqueta el código y todas sus dependencias para que la aplicación se ejecute de manera rápida y confiable de un entorno informático a otro. Una imagen de contenedor Docker es un paquete de software, independiente, gestionable y que incluye todo lo necesario para ejecutar una aplicación: código, tiempo de ejecución, herramientas del sistema, bibliotecas del sistema y configuraciones.

Las imágenes de dcoker se convierten en contenedores en tiempo de ejecución y, en el caso de los contenedores Docker, las imágenes se convierten en contenedores cuando se ejecutan en Docker Engine. Disponible para aplicaciones basadas en Linux y Windows, el software en contenedores siempre se ejecutará igual, independientemente de la infraestructura. Los contenedores aíslan el software de su entorno y aseguran que funcione de manera uniforme a pesar de las diferencias, por ejemplo, entre el desarrollo y la puesta en escena.

La tecnología de contenedores Docker se lanzó en 2013 como un motor Docker de código abierto. Aprovechó los conceptos informáticos existentes en torno a los contenedores y específicamente en el mundo de Linux, primitivas conocidas como cgroups y espacios de nombres. La tecnología de Docker es única porque se centra en los requisitos de los desarrolladores y operadores de sistemas para separar las dependencias de las aplicaciones de la infraestructura. El éxito en el mundo de Linux impulsó una asociación con Microsoft

que llevó los contenedores Docker y su funcionalidad a Windows Server.

Comparación entre contenedores y máquinas virtuales
Los contenedores y las máquinas virtuales tienen beneficios similares de aislamiento y asignación de recursos, pero funcionan de manera diferente porque los contenedores virtualizan el sistema operativo en lugar del hardware. Los contenedores son más portátiles y eficientes.

Los contenedores son una abstracción en la capa de la aplicación que agrupa el código y las dependencias juntas. Se pueden ejecutar varios contenedores en la misma máquina y compartir el núcleo del sistema operativo con otros contenedores, cada uno de los cuales se ejecuta como procesos aislados en el espacio del usuario. Los contenedores ocupan menos espacio que las máquinas virtuales (las imágenes de los contenedores suelen tener un tamaño de decenas de MB), pueden manejar más aplicaciones y requieren menos máquinas virtuales y sistemas operativos.

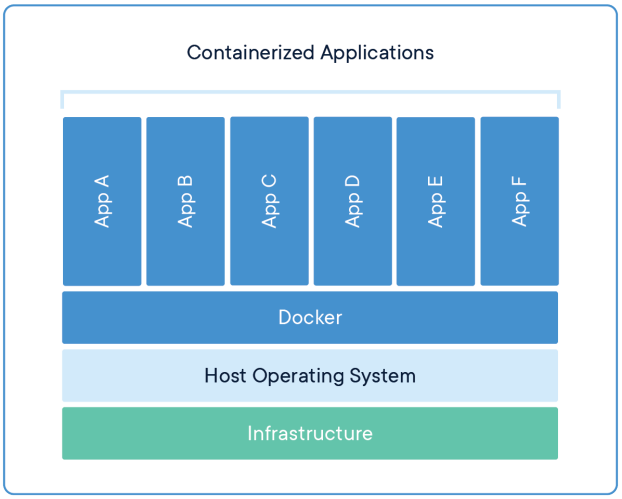

Figure 4.1: Container

En cambio, las máquinas virtuales (VM) son una abstracción del hardware físico que convierte un servidor en muchos servidores. El hipervisor permite que varias máquinas virtuales se ejecuten en una sola máquina. Cada VM incluye una copia completa de un sistema operativo, la aplicación, los binarios y bibliotecas necesarias, que ocupan decenas de GB. Las máquinas virtuales también pueden ser lentas para arrancar.

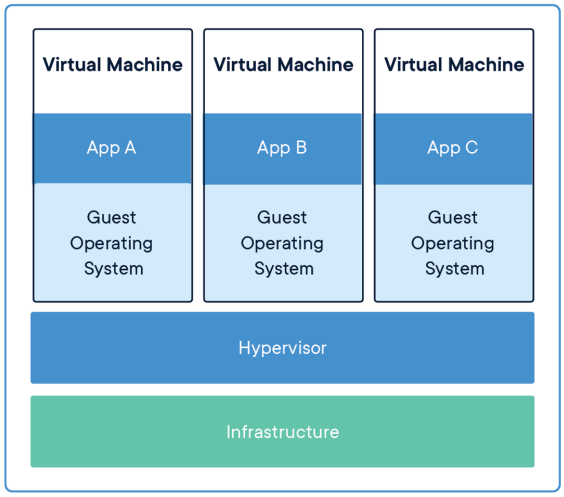

Figure 4.2: VM

4.2.2 Protocol buffers

Protobuffer diseñado internamente por Google, es una herramienta que nos permite generar protocolos de comunicaciones o estructuras de mensajes para facilitar al usuario las comunicaciones, a través de la serialización de mensajes. La serialización es un proceso de codificación de un objeto en un medio de almacenamiento, como puede ser una archivo o un buffer en memoria, en ocasiones para transmitirlo a través de una conexión de red o para preservarlo entre ejecuciones de un programa. La serie de bytes que codifican el estado del objeto tras la serialización puede ser usada para crear un nuevo objeto, idéntico al original, tras aplicar el proceso inverso de deserialización. Además, dispone de un compilador que te permite una vez hecho el esquema de la estructura de datos codificarlo en distintos lenguajes; C++, Java, python, Objective-C y con la versión proto3 se puede trabajar con: Dart, Go, Ruby y C.

Según la página oficial de Google, los buffers de protocolo son el mecanismo extensible de lenguaje neutral, plataforma neutral para serializar datos estructurados; como en XML, pero más pequeño, más rápido y más simple. Uno define cómo desea que se organicen los datos una vez, luego puede usar un código fuente generado especial para escribir y leer fácilmente sus datos estructurados hacia y desde una variedad de flujos de datos y usando una variedad de idiomas.

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
}
```

Figure 4.3: Ejemplo Protobuffer

4.2.3 Grafana

El proyecto Grafana fue iniciado por Torkel Ödegaard en 2014 y en los últimos años se ha convertido en uno de los proyectos de código abierto más populares en GitHub. Le permite consultar, visualizar y alertar sobre métricas y registros sin importar dónde estén almacenados.

Grafana tiene un modelo de fuente de datos conectable y viene con un amplio soporte para muchas de las bases de datos de series de tiempo más populares como Graphite, Prometheus, Elasticsearch, OpenTSDB e InfluxDB. También tiene soporte incorporado para proveedores de monitoreo en la nube como Google Stackdriver, Amazon Cloudwatch, Microsoft Azure y bases de datos SQL como MySQL y Postgres. Grafana es la única herramienta que puede combinar datos de tantos lugares en un solo tablero.

Grafana Labs se enorgullece de liderar el desarrollo del proyecto Grafana, fomentando una comunidad próspera y asegurando que los clientes de Grafana Labs reciban el soporte y las características de Grafana que necesitan.

4.2.4 InfluxDB

InfluxDB es una base de datos de series temporales desarrollada por InfluxData. Está escrito en Go y optimizado para el almacenamiento rápido, la alta disponibilidad y la recuperación de datos de series de tiempo en campos como monitoreo de operaciones, métricas de aplicaciones, datos de sensores de Internet de las cosas y análisis en tiempo real. También tiene soporte para procesar datos y ser capaz de hacer agrupaciones, reducciones, añadir cam-

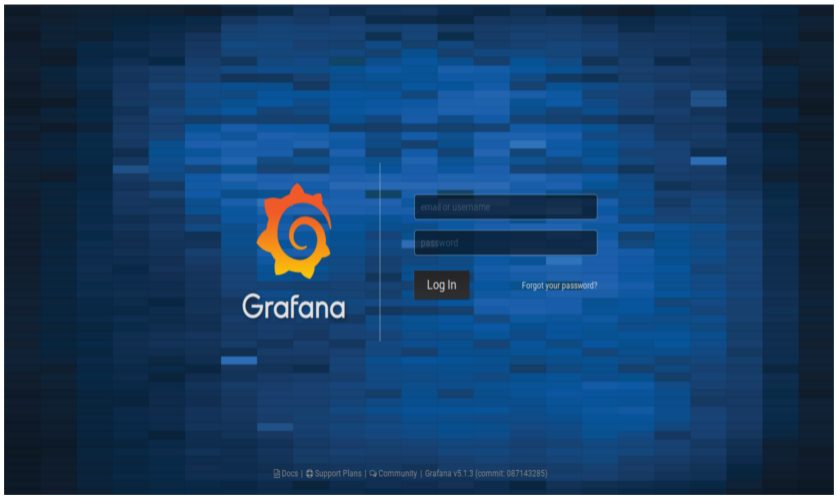

Figure 4.4: Login Grafana

pos tras realizar cálculos, para resumir es capaz de hacer transformaciones sobre los datos obtenidos de las métricas a tiempo real.

Figure 4.5: Logo InfluxDB

InfluxDB no tiene dependencias externas y proporciona un lenguaje similar a SQL, escuchando en el puerto 8086, con funciones centradas en el tiempo, incorporadas para consultar una estructura de datos compuesta de medidas, series y puntos. Cada punto consta de varios pares clave-valor, por un lado el conjunto de campos y por el otro una marca de tiempo (un timestamp). Cuando se agrupan por un conjunto de pares clave-valor llamado conjunto de etiquetas, definen una serie. Finalmente, las series se agrupan por un identificador de cadena para formar una medida. Dicha medida es el jugo de la aplicación, gracias a esas medidas obtenidas a tiempo real podríamos hacer estudios sobre ellas y sacarle el máximo potencial.

Los valores pueden ser enteros de 64 bits, puntos flotantes de 64 bits, cadenas y booleanos. Los puntos se indexan por su tiempo y conjunto de etiquetas. Las políticas de retención se definen en una medición y controlan cómo se disminuyen y eliminan los datos. Por último, las consultas continuas se ejecutan periódicamente, almacenando los resultados en una medición objetivo.

Según Capital One, un holding bancario de Estados Unidos, InfluxDB es una base de datos de lectura y escritura de alta velocidad. Los datos se escriben en tiempo real, puede leer en tiempo real, y cuando lo está leyendo, se puede aplicar su modelo de aprendizaje automático. Entonces, en tiempo real, se puede pronosticar y detectar anomalías.

4.2.5 PyCharm

Es un IDE de python desarrollado por programadores y para programadores creado por Jet Brains, una herramienta que te facilita programar a través de: finalización de código inteligente, detección de errores de código sobre la marcha y posibles arreglos, navegación simple en proyectos, integración continua con Git y mucho más.

Figure 4.6: Logo Pycharm

4.2.6 Python

Python es un lenguaje de programación interpretado cuya filosofía hace hincapié en la legibilidad de su código. Creado por Guido van Rossum y lanzado por primera vez en 1991, 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 a pequeña y gran escala. Admite múltiples paradigmas de programación , incluida la programación estructurada, orientada a objetos y funcional.

Figure 4.7: Logo Python

Es administrado por la Python Software Foundation y posee una licencia de código abierto, denominada Python Software Foundation License.

4.2.7 Ubuntu

Ubuntu es un sistema operativo de software libre y código abierto, una distribución de Linux basada en Debian. Actualmente corre en computadores de escritorio y servidores. Además, está orientado al usuario promedio, con un fuerte enfoque en la facilidad de uso y en mejorar la experiencia del usuario.

Figure 4.8: Logo ubuntu

Ubuntu incluye lo mejor en traducción e infraestructura de accesibilidad que la comunidad de Software Libre tiene para ofrecer, para que Ubuntu pueda ser utilizado por la mayor cantidad de personas posible. Dicho sistema es gratuito, no hay una tarifa adicional para la "edición empresarial",

ya que está totalmente comrometido con los principios de desarrollo de código abierto; alentan a las personas a usar software de código abierto, mejorarlo y transmitirlo.

4.2.8 Bash

Es un lenguaje de comandos y shell de Unix, te permite programar sentencias que causan acciones. En este caso lee y ejecuta comandos desde un archivo, llamado script. Cuando se inicia Bash, ejecuta los comandos en una variedad de archivos de puntos, es similar a los comandos de script de shell Bash, que tienen permiso de ejecución habilitado y una directiva de intérprete como cabecera #**!/bin/bash**.

Figure 4.9: Logo bash

4.2.9 Nano

Nano es un editor de texto para sistemas Unix basado en curses, que es una biblioteca para el control de terminales sobre sistemas de tipo Unix, posibilitando la construcción de una interfaz para el usuario, para aplicaciones ejecutadas en un terminal. Esta escrito en C y es un clon de Pico, el editor del cliente de correo electrónico Pine. Nano trata de emular la funcionalidad y la interfaz de fácil manejo de Pico, pero sin la integración con Pine.

4.2.10 Git

Git es un software de control de versiones, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas

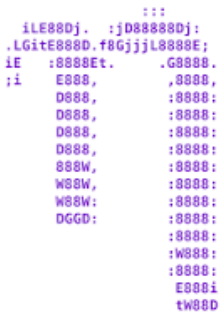

Figure 4.10: Logo nano

tienen un gran número de archivos de código fuente. Permite que varias personas puedan trabajar en el mismo proyecto sin pisarse los unos en los otros, además del control que te aporta sobre tu código, es capaz de volver a versiones anteriores del código.

Figure 4.11: Logo git

La comunidad utiliza con frecuencia git, hay varias variantes como gitlab o github que trabajan internamente con git, en este caso se ha utilizado github para el proceso de construcción del código.

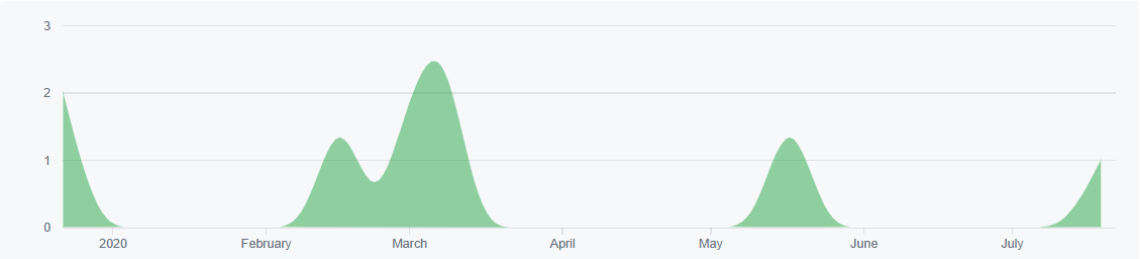

Figure 4.12: Gráfica del desarrollo del trabajo en github

4.2.11 Visual Studio Code

Visual Studio Code es un editor de código fuente desarrollado por Microsoft para Windows, Linux y macOS. Incluye soporte para la depuración, control integrado de Git, resaltado de sintaxis, finalización inteligente de código, fragmentos y refactorización de código. También es personalizable, por lo que los usuarios pueden cambiar el tema del editor, los atajos de teclado y las preferencias. Es gratuito y de código abierto, aunque la descarga oficial está bajo software privativo e incluye características personalizadas por Microsoft. Visual Studio Code se basa en Electron, un framework que se utiliza para implementar Chromium y Node.js como aplicaciones para escritorio, que se ejecuta en el motor de diseño Blink.

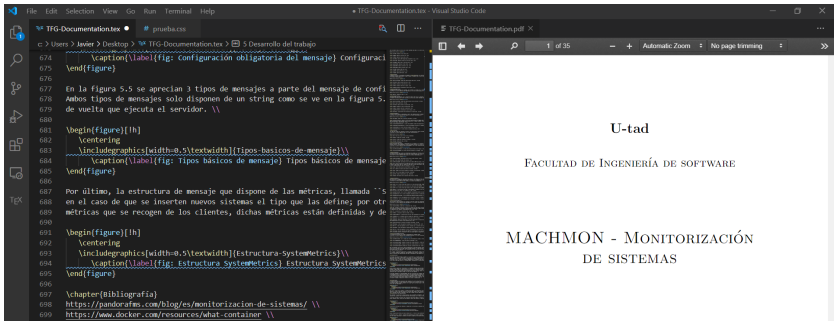

Figure 4.13: Ejemplo Visual Studio Code

Se ha utilizado este editor de texto para realizar la documentación por su fácil manejo y por todas las ayudas que aporta.

4.2.12 LaTeX

LaTeX es un sistema de composición de textos, orientado a la creación de documentos escritos que presenten una alta calidad tipográfica. Por sus características y posibilidades, es usado de forma especialmente intensa en la generación de artículos, libros científicos que incluyen, entre otros elementos, expresiones matemáticas, artículos académicos, tesis y libros técnicos, dado que la calidad tipográfica de los documentos realizados en LaTeX, se considera adecuada a las necesidades de una editorial científica de primera línea, muchas de las cuales ya lo emplean.

```latex
%Preambulo
\documentclass[ spanish, a4paper, 12pt, oneside]{report}
\usepackage{hyphenat}
\usepackage{graphicx}

\graphicspath{ {TFGImages/} }
% Cuerpo
\title{\Huge MACHMON - Monitorización de sistemas }
\author{Javier Lima}
\date{Junio 2020}

\begin{document}

\begin{titlepage}
    \centering
    {\bfseries\LARGE U-tad \par}
    \vspace{1cm}
    {\scshape\Large Facultad de Ingeniería de software \par}
    \vspace{3cm}
    {\scshape\Huge MACHMON - Monitorización de sistemas \par}
    \vspace{3cm}
    {\itshape\Large Proyecto Fin de Carrera \par}
    \vfill
    {\Large Autor: \par}
    {\Large Javier Lima \par}
    \vfill
    {\Large Junio 2020 \par}
\end{titlepage}
```

Figure 4.14: Ejemplo LaTeX

Chapter 5

Desarrollo del trabajo

Para empezar, es necesario tener claro la aspiración del trabajo, saber la salud de los sistemas, es por ello por lo que necesitamos obtener las métricas de los sistemas. Para ello se ha desarrollado un script en bash capaz de obtener métricas de todo tipo: cpu, RAM, disco, etc e introducirlas en un archivo JSON.

A continuación se mostrarán todas las métricas recogidas detalladamente.

5.1 Estándares de métricas

- Memoria: kilobyte unidad utilizada para todas las métricas.
  - Tiempo: milisegundos(preguntar por mbps).
  - Temperatura: grados celcius.

5.1.1 cpusNumber

Recogeremos el número de cpus tanto en uso como totales y así poder hacer un seguimiento al rendimiento, obtendremos los elementos con los comandos nproc y lscpu. La métrica va a estar nombrada como cpusNumber y va a estar constituida por:

Nombre de la métrica: **cpusNumber**
- cpusTotalNumber: número de cpus totales.(int)
- cpusUsageNumber: número de cpus en uso. (int)

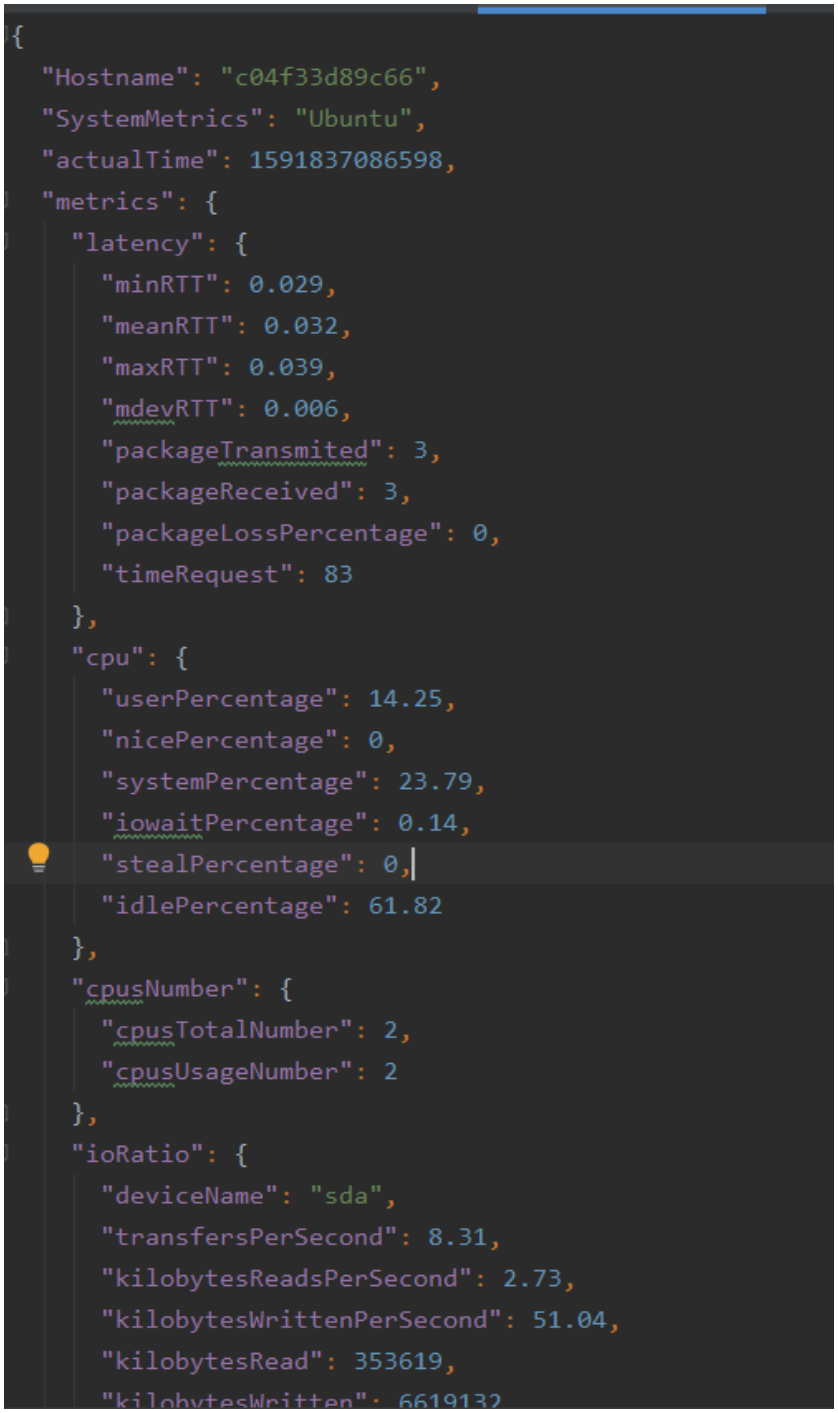

Figure 5.1: Métricas JSON

5.1.2 cpu

Recogeremos los porcentajes de uso de la cpu, lo obtendremos con el comando "iostat -c" (que hay que instalarse). La métrica va a estar nombrada como cpu y va a estar constituida por:

Nombre de la métrica: **cpu**
- userPercentage: porcentaje de uso de cpu que se produjo durante la ejecución a nivel de usuario. (float)
- nicePercentage: porcentaje de uso de cpu que se produjo al ejecutar a nivel de usuario con buena prioridad. (float)
- systemPercentage: porcentaje de uso de cpu que se produjo durante la ejecución a nivel de sistema. (float)
- iowaitPercentage: porcentaje de tiempo que la CPU o las CPUs estuvieron inactivas durante las cuales el sistema tuvo una solicitud I/O de disco pendiente. (float)
- stealPercentage: porcentaje de tiempo que la CPU virtual o las CPUs pasaron en espera involuntaria mientras el hipervisor daba servicio a otro procesador virtual. (float)
- idlePercentage: porcentaje de tiempo que la CPU o las CPU estuvieron inactivas y el sistema no tenía una solicitud de I/O de disco pendiente. (float)

5.1.3 mem

La memoria RAM la obtenemos del comando free, nos centraremos tanto en la memoria RAM como la memoria swap. La métrica va a estar nombrada como mem y va a estar constituida por:

Nombre de la métrica: **mem**
- totalMem: memoria física total.(int)
- usedMem: memoria física en uso. (int)
- freeMem: memoria física libre. (int)
- sharedMem: memoria RAM compartida actualmente en uso. (int)
- buffersMem: memoria actual del búffer de caché. (int)
- cachedMem: memoria de la caché de disco. (int)
- swapTotalMem: memoria virtual total. (int)
- swapUsedMem: memoria virtual en uso. (int)
- swapFreeMem: memoria virtual libre. (int)
- totalRAM: memoria RAM total.(int)
- usedRAM: memoria RAM en uso. (int)
- freeRAM: memoria RAM libre. (int)

5.1.4 disk

La memoria del disco la obtenemos del comando df --total, nos centraremos en el cálculo total y dejamos a un lado el resto sistema de ficheros, en un futuro se podría añadir el desglose del cálculo total. La métrica va a estar nombrada disk y va a estar constituida por:

Nombre de la métrica: **disk**
- identificatorName: nombre de identificación del disco. (string)
- totalDisk: memoria total del disco. (int)
- usedDisk: memoria en uso del disco. (int)
- freeDisk: memoria libre en el disco. (int)
- usagePercentDisk: porcentaje de uso del disco. (int)

5.1.5 temperature

La temperatura del sistema está mockeada, está seteada a 27 grados más un random de 0 a 11 grados La métrica va a estar nombrada como temperature y va a estar constituida por:

Nombre de la métrica: **temperature**
- degrees: temperatura del pc. (int)

5.1.6 partitions

Las particiones las obtenemos con el comando df memoria del disco la obtenemos del comando df, nos centraremos en la carpeta raíz utilizando el elemento /dev/sda1 para filtrar con grep. La métrica va a estar nombrada como partitions y va a estar constituida por:

Nombre de la métrica: **partitions**
- identificatorName: nombre de identificación de la partición. (string)
- type: tipo de partición. (string)
- totalDisk: memoria total de la partición. (int)
- usedDisk: memoria en uso de la partición. (int)
- freeDisk: memoria libre de la partición. (int)
- usagePercentDisk: porcentaje de uso del disco. (int)
- mountPoint: localización del direcotorio de la partición. (string)

5.1.7 ioRatio

El ratio de IO del disco lo obtendremos a partir del comando iostat, de donde filtraremos la salida para quedarnos con el disco principal sda. La métrica va a estar nombrada como ioRatio y va a estar constituida por:

Nombre de la métrica: **ioRatio**
- deviceName: nombre de la partición de memoria. (string)
- transfersPerSecond: indica el número de transferencias por segundo que se emitieron al dispositivo. Una transferencia es una solicitud I/0 al dispositivo, se pueden combinar varias solicitudes lógicas en una sola solicitud porque es de tamaño indeterminado. (float)
- kilobytesReadsPerSecond: el número de kilobytes leídos del dispositivo por segundo. (float)
- kilobytesWrittenPerSecond: el número de kilobytes escritos en el dispositivo por segundo. (float)
- kilobytesRead: El número total de kilobytes leídos. (int)
- kilobytesWritten: El número total de kilobytes escritos. (int)

5.1.8 logs

Los logs del sistema los leeremos de archivo /var/log/syslog en nuestro sistema ubuntu. La métrica va a estar nombrada logs y va a estar constituida por:

Nombre de la métrica: **logs**
- date: fecha en la que se hizo el log. (datetime)
- nameHost: nombre del sistema. (string)
- process?: nombre del proceso del sitema??. (string)?
- message: mensaje del log. (string)

5.1.9 processesNumber

Número de procesos del sistema, lo obtendremos con el comando ps, el argumento -a nos devuelve solo los procesos activos, nos guardaremos el número de procesos activos y el número de procesos totales La métrica va a estar nombrada como processesNumber y va a estar constituida por:

Nombre de la métrica: **processesNumber**
- activeProcessesNumber: el número de procesos activos de la máquina. (int)

- totalProcessesNumber: el número total de procesos de la máquina. (int)

5.1.10 process

La tabla de procesos del sistema la obtendremos del comando ps, filtramos el comando para que nos devuelva las entradas que nos interesan, que son las que tienen carga de cpu, es decir aquellas que tengan distinto de 0 el porcentaje de cpu. La métrica va a estar nombrada como process y va a estar constituida por:

Nombre de la métrica: **process**
- usedPercentageCpu: porcentaje de uso de la cpu del proceso. (float)
- pid: número identificador del proceso. (int)
- usedPercentageMem: porcentaje de uso de la memoria RAM del proceso. (float)
- nice: número que define la prioridad del proceso. Esta prioridad se llama Niceness en Linux, y tiene un valor entre -20 y 19. Cuanto más bajo sea el índice de Niceness, mayor será una prioridad dada a esa tarea. El valor predeterminado de todos los procesos es 0. (int)
- group: nombre del grupo al que pertenece el proceso. (string)
- user: nombre del usuario que ejecutó el proceso. (string)
- state: estado en el que se encuentra el proceso (R en ejecución, S dormido, T detenido, X muerto). (string)
- start: hora a la que empezó el proceso. (datetime)
- cpuTime: tiempo de ejecución en cpu. (datetime)
- command: descripción de la ejecución del proceso. (string)

5.1.11 latency

La latencia es el tiempo que lleva enviar una señal más el tiempo que lleva recibir un acuse de recibo de esa señal. Para calcularla utilizamos el comando ping en localhost con 3 paquetes y nos quedamos con el rtt. La métrica va a estar nombrada como latency y va a estar constituida por:

Nombre de la métrica: **latency**
- minRTT: es el número mínimo que tardó una de las peticiones ECHO_REQUEST. (float)
- meanRTT: es la media de lo que tardaron las peticiones ECHO_REQUEST. (float)
- maxRTT: es el número máximo que tardó una de las peticiones ECHO_REQUEST. (float)

- mdevRTT: es la desviación estándar, esencialmente un promedio de cuán lejos está cada RTT de ping de la RTT media. Cuanto más alto es el número, más variable es el RTT. (float)

- packageTransmited: es el número de paquetes transmitidos durante la prueba de latencia. (int)

- packageReceived: es el número de paquetes recibidos durante la prueba de latencia. (int)

- packageLossPercentage: es el porcentage de paquetes perdidos durante la prueba de latencia. (float)

- timeRequest: son los milisegundos que se han tardado en hacer la prueba de latencia. (int)

- clientServer: son los milisegundos que ha tardado el servidor en recibir las métricas. (int)

5.1.12 networkMetrics

Las métricas de red que disponen las tarjetas de red del sistema las obtendremos con el comando ifconfig. La métrica va a estar nombrada como netWorkMetrics y va a estar constituida por:

Nombre de la métrica: **networkMetrics**

- networkCardName: nombre de la tarjeta de red. (string)

- MTU: (unidad de transmisión máxima) es el tamaño de cada paquete recibido por la tarjeta de red. El valor de MTU para todos los dispositivos Ethernet de manera predeterminada se establece en 1500. Establecer un valor más alto podría poner en peligro la fragmentación del paquete o desbordamientos del búfer. (int)

- IP: ip visible de la tarjeta de red. (string)

- netMask: la máscara de red del sistema. (string)

- broadcastAddress: la dirección que se usará para representar las transmisiones a la red. (string)

- IPv6Address: es la etiqueta numérica usada para identificar la interfaz de red en una red IPv6. (string)

- MACAddress: es la dirección MAC de la tarjeta de red. (string)

- txQueueLen: denota la longitud de la cola de transmisión del dispositivo. Por lo general, lo configura en valores más pequeños para dispositivos más lentos con una latencia alta. (int)

- connectionProtocol: protocolo de conexión utilizado. (string)

- RXPackages: número de paquetes recibidos por la interfaz de red. (int)

- RXErrors: número de paquetes con error recibidos por la interfaz de red. (int)

- TXPackages: número de paquetes transmitidos por la interfaz de red. (int)

- TXErrors: número de paquetes con error transmitidos por la interfaz de red. (int)

- collisions: el valor de este campo debería ser idealmente 0. Si tiene un valor mayor que 0, podría significar que los paquetes están colisionando mientras atraviesan la red, una señal segura de congestión de la red. (int)

5.1.13 systemAdditionalInfo

La información adicional del sistema la obtendremos del comando uptime. La métrica va a estar nombrada como systemAdditionalInfo y va a estar constituida por:

Nombre de la métrica: **systemAdditionalInfo**

- systemRunningTime: es el tiempo que lleva funcionando el sistema. (string)

- usersLoggedOnNumber: número de usuarios conectados. (int)

- systemLoadAverage1M: promedio de carga del sistema durante el último minuto. (float)

- systemLoadAverage5M: promedio de carga del sistema durante los últimos 5 minutos. (float)

- systemLoadAverage15M: promedio de carga del sistema durante los últimos 15 minutos. (float)

Una vez obtenidas las métricas, hay que almacenarlas en una base de datos. Para ello, hay que plantearse el cómo hacerlo, ¿qué forma sería la más eficiente? ¿quá cada sistema se pueda conectar a la base de datos para almacenarlas o solo una entidad es la encargada de introducirlas?. Por seguridad y por acoplamiento es mejor que solo una entidad sea capaz de almacenar las métricas, independientemente del tipo de sistema que sea.

Por lo tanto, es necesario desarrollar un sistema cliente-servidor, donde el servidor se encargué de almacenar las métricas de los sistemas y el cliente recogerlas y mandarlas. Indirectamente estos tipos de sistemas te obligan a gestionar las comunicaciones entre ellos, así que es necesario hacer un protocolo de comunicaciones para que se entiendan en todo momento el servidor y los clientes.

5.2 Script en bash

Es un script capaz de recoger todas las métricas anteriormente nombradas en el apartado 5.1 “Estándares de métricas”. Bash te permite programar en función de comandos, el uso de esos comandos nos obliga a seguir su orden y sintaxis, es por ello que con ciertos comandos, se ejecuta un filtrado para solo obtener las variables que necesitamos o como en todos los comandos eliminar aquello que nos de ruido con la ayuda de “grep”, “tail”, “sed”, “awk”, “cut” o “tr”.

```
#!/bin/bash
mainDisk=(`df --total | grep 'total'`)
mem=(`free --total | sed "1d"`)
date=`date +%s%3N`
times=(`uptime | sed 's/,/./g'`)
systemRunningTime=`uptime -s`
cpu=(`iostat -c | tail -n 4`)
ioRatio=(`iostat -d | grep "sda" | sed 's/,/./g'`)
cpusUsageNumber=`lscpu --extended -b | sed "1d" | wc -l`
cpusTotalNumber=`nproc --all`
#!logs=`cat /var/log/syslog`
disks=(`df -T | sed "1d"`)
disksEntries=`df -T | sed "1d" | wc -l`
temperature=`echo $(((RANDOM%11)+27))`
processesTable=(`ps -e -o pcpu,pid,pmem,nice,state,group,user,start,cputime,args --sort pcpu | sed -e 's/$/ .endline/' | sed '/^ 0.0 /d' | s
processesTableEntries=`ps -e -o pcpu,pid,pmem,nice,state,group,user,start,cputime,args --sort pcpu | sed '/^ 0.0 /d' | sed "1d" | wc -l`
activeProcessesNumber=`ps -a | wc -l`
totalProcessesNumber=`ps -ax | wc -l`
networkCards=(`ifconfig`)
latency=`ping 127.0.0.1 -c 3 | tail -n 2`
latencyPackageStadistics=($latency)
minRTT=`echo $latency | grep rtt | awk -F '/' '{print $4}' | cut -d '=' -f 2 | tr -d ' '`
meanRTT=`echo $latency | grep rtt | awk -F'/' '{print $5}'`
maxRTT=`echo $latency | grep rtt | awk -F'/' '{print $6}'`
mdevRTT=`echo $latency | grep rtt | awk -F '/' '{print $7}' | cut -d ' ' -f 1 | tr -d ' '`
host=`hostname`
```

Figure 5.2: Comandos script en bash

Una vez se han ejecutado los comandos que aparecen en la figura 5.2, se ejecuta un proceso de preparación de variables para su posterior inserción en un JSON, como se puede observar en la figura 5.3.

Por último, una vez ya filtrado los comandos y preparadas las variables con un orden preestablecido, el script se encarga de insertar todo las métricas recogidas en un JSON, como se puede observar en la figura 5.4.

```
countEntries=0
partitions='['
iterator=0
while [ $countEntries -lt $disksEntries ];do
  for e in 0 1 2 3 4 5 6;
  do
    case "$e" in 0)
      partitions=''${partitions}'{"identificatorName":"'${disks[$iterator]}'",'
    ;;
    1)
                        partitions=''${partitions}' "type":"'${disks[$iterator]}'",'
                ;;
    2)
                        partitions=''${partitions}' "totalDisk":'${disks[$iterator]}','
                ;;
    3)
                        partitions=''${partitions}' "usedDisk":'${disks[$iterator]}','
                ;;
    4)
                        partitions=''${partitions}' "freeDisk":'${disks[$iterator]}','
                ;;
    5)
                        partitions=''${partitions}' "usagePercentDisk":'${disks[$iterator]%\%}','
                ;;
    6)
                        partitions=''${partitions}' "mountPoint":"'${disks[$iterator]}'"},'
                ;;
    *)
      echo "ERROR: La métrica partitions no dispone de tantos parámetros"
    ;;
    esac
    ((iterator++))
  done;
  ((countEntries++))
done;
```

Figure 5.3: Preparación de variables

5.3 Servidor

El servidor está desarrollado en python, en la versión 3.7. En primer lugar, el servidor se encarga de setear su propia configuración según un json que permite actualizar la dirección IP, el puerto o el tamaño del buffer para el control de las comunicaciones, también te permite controlar que sistemas puden comunicarse con el servidor, un plus de seguridad que solo deja conectarse con máquinas ya conocidas, además setea la contraseña por la cuál se conectan los clientes al servidor y por el último los formatos elegidos para el sistema de logs que dispone el servidor, el sistema de logs es simplemente para tener un control de errores, en el caso de que algo falle en el sistema

Figure 5.4: Inserción de métricas en un JSON

quedará reflejado en un archivo aparte llamado SystemMetrics-Server.log.

Dicho servidor siempre está funcionando, en el momento que haya sido ejecutado entrará en un bucle de donde no podrá salir hasta que un usuario pare la ejecución. En primer lugar prepara el socket por el cuál permitirá la comunicación y una vez hecho eso entra en un bucle infinito con el siguiente algoritmo; aceptamos una comunicación entrante (a través de una línea bloqueante que nos la da la líbrería "socket") y se la derivamos a un thread, con esto permitimos que haya concurrencia y en el caso de que otro cliente quiera comunicarse con el servidor podrá dicho servidor lanzar otro thread para que se comuniquen. Además cada conexión entrante tiene un timeout, es decir solo podrán comunicarse por un período de tiempo, en este caso 120 segundos.

El servidor dispone de 3 funcionalidades; que solo la utilizan los threads. Cuando un thread es creado se pone a la escucha de nuevos mensajes, en donde solo hay 3 posibles mensajes: un mensaje de respuesta que da el servidor para avisar a los clientes de que la primera comunicación a ido bien, un segundo mensaje que contiene las métricas, por lo tanto cada thread trata dichas métricas y las almacena en una base de datos de series de tiempo llamada InfluxDB y por último un mensaje para cercionarse de que se cierran las

Figure 5.5: Server configuration

conexiones con los clientes. Siempre ocurre el mismo algoritmo, el cliente en primera instancia se dispone a comunicarse con el servidor, le manda un mensaje con la contraseña anteriormente vista y si coincide el servidor le manda un respuesta de que todo ha ido bien, el cliente al recibir dicho mensaje le manda las métricas al servidor, una vez recibidas el servidor las parsea y las almacena en InfluxDB con un timestamp, por último si todo ha ido correctamente el cliente le manda un mensaje al servidor pàra cerrar la comunicación.

5.4 Cliente

El cliente sigue la misma estructura del servidor está escrito en python en la versión 3.7, se inicia y se setea su propia configuración. Dispone de un JSON donde lee en que ip se encuentra el servidor, la ip propia donde poder comunicarse, el puerto, el tamaño del buffer, la contraseña para interactuar con el servidor, los formatos elegidos para el sistema de logs y a diferencia de la configuración del servidor éste dispone de dos parámetros más: que tipo de sistema es(debian, windows, ubuntu, etc) y que tipo de métricas enviará, estos últimos parámetros son para llevar un control de que sistema es cada uno para que en un futuro si quisiéramos filtrar por los sistemas de un solo tipo pudiésemos.

Figure 5.6: Client configuration

Una vez el cliente se ha seteado, se prepara para empezar la comunicación con el servidor. En primer lugar, le manda un mensaje con la contraseña y se pone a la espera de la respuesta del servidor, si todo ha ido bien ejecuta un script en bash que calcula las métricas del sistema y las introduce en un JSON, una vez generado el JSON lee las métricas del archivo y crea el mensaje con ellas para mandarlas al servidor. Por último, corta la comunicación con el servidor para que la próxima vez que vaya a mandar métricas empiece el algoritmo desde cero.

5.5 Generación de logs

Por defecto, los sistemas operativos crean múltiples logs en los que se registran y clasifican diferentes tipos de procesos. Los sistemas realizan protocolos sobre eventos de aplicaciones (programas), eventos del sistema, eventos relacionados con la seguridad, eventos de configuración y eventos reenviados. Consultar la información contenida en un log puede ayudar a un administrador a solucionar problemas.

Para llevar un mayor control sobre nuestra aplicación se han generado archivos de logs tanto para el cliente como para el servidor, esos logs son informativos, son escrituras que hacen en un fichero cuando ocurre un acción importante en el sistema. Ambos ficheros de logs siguen la misma estructura,

como se aprecia en la figura 5.7, está compuesto por la hora en la que ocurre, el tipo de información que es, INFO O ERROR, sus nombres son bastantes descriptivos, uno describe lo que va pasando en la aplicación y el otro informa de un error y por último un mensaje descriptivo sobre lo ocurrido. Todos esos logs se van escribiendo en un fichero que podemos nombrarlo como queramos.

```
"logging": {
  "format": "'%(asctime)-15s - %(levelname)s - %(message)s'",
  "name": "/logs/SystemMetrics_Client.log"
}
```

Figure 5.7: Formato de los logs

Por el lado del cliente, solo es posible que ocurra un error, que no sea capaz de conectarse con el servidor, en cambio en los logs del servidor se puede observar el porqué ha fallado, es decir los logs del cliente son mera información descriptiva, en cambio en los del servidor se va especificando que fallos pueden ocasionarse. En este sistema disponemos de cinco posibles fallos:

- Que no se haya podido insertar en la base de datos.
- Que no exista conexión previa con el cliente, es decir que haya pasado por alto un paso.
- Que el host que se intente conectar no lo tenga contemplado el servidor, que no esté en los hosts permitidos.
- Que se mande un mensaje de establecer conexión cuando ya están conectados.
- Que la clave de seguridad al empezar una comunicación sea incorrecta.

```
'2020-06-01 21:30:07,059 - INFO - [*] Iniciando el cliente: '
'2020-06-01 21:30:07,060 - INFO - [*] Estableciendo conexion con: 192.168.1.114:5005'
'2020-06-01 21:30:07,062 - INFO - [*] Recibiendo respuesta del: 192.168.1.114:5005'
'2020-06-01 21:30:09,213 - INFO - [*] Ejecutando script para obtener las métricas'
'2020-06-01 21:30:12,225 - INFO - [*] Seteando el mensaje con las métricas a enviar'
'2020-06-01 21:30:12,238 - INFO - [*] Enviando metricas a: 192.168.1.114:5005'
'2020-06-01 21:30:17,244 - INFO - [*] Finalizando conexión con: 192.168.1.114:5005'
```

Figure 5.8: Logs del cliente

```
'2020-06-06 14:09:36,149 - INFO - [*] Iniciando el servidor: '
'2020-06-06 14:09:36,150 - INFO - [*] Esperando mensajes en 192.168.1.114:5005'
'2020-06-06 14:10:02,265 - INFO - [*] Conexión establecida con: <socket.socket fd=728, family=A
'2020-06-06 14:10:02,266 - INFO - [*] Esperando mensajes en 192.168.1.114:5005'
'2020-06-06 14:10:03,269 - INFO - [*] Mensaje recibido: config {
  Hostname: "raspberrypi"
  message_type: START_COMMUNICATION
  ip: "192.168.1.108"
  port: 5005
}
startCommunication {
  password: "P0t4t0 c4n 1 c0nn3ct?"
}
'
'2020-06-06 14:10:03,271 - INFO - [*] Conexion establecida con 192.168.1.108:5005, raspberrypi'
'2020-06-06 14:10:06,272 - INFO - [*] Mensaje de confirmación enviado a 192.168.1.108:5005, ras
'2020-06-06 14:10:09,021 - INFO - [*] Conexión establecida con: <socket.socket fd=760, family=A
'2020-06-06 14:10:09,022 - INFO - [*] Esperando mensajes en 192.168.1.114:5005'
'2020-06-06 14:10:09,022 - INFO - [*] Mensaje recibido: config {
  Hostname: "c04f33d89c66"
  message_type: START_COMMUNICATION
  ip: "172.20.0.2"
  port: 5005
}
startCommunication {
  password: "P0t4t0 c4n 1 c0nn3ct?"
}
'
'2020-06-06 14:10:09,023 - INFO - [*] Conexion establecida con 172.20.0.2:5005, c04f33d89c66'
'2020-06-06 14:10:10,304 - INFO - [*] Mensaje recibido: config {
  Hostname: "raspberrypi"
  message_type: METRICS
  ip: "192.168.1.108"
  port: 5005
}
data {
```

Figure 5.9: Logs del servidor

5.6 Protocolo de comunicaciones

Para entender el protocolo de comunicaciones, hay que entender protobuffer, una herramienta que permite programar los mensajes tal y como quieras. Protobuffer te permite setear atributos en las cabeceras, que él por detrás se encarga de comprimirlos y generar una estructura de datos serializada, por lo tanto una vez llegue el mensaje a su destino solo habría que deserializarlo para obtener cada métrica.

La versión utilizada es proto3, que a diferencia de proto2 cambia la sintaxis y los campos siempre son opcionales a la hora de generar la estructura del mensaje, no permite poner campos requeridos, Google decidió eliminarlo para evitar posibles fallos. Ante ello solo queda el control por parte de la aplicación, es decir, cuando se mande un mensaje se asegure de que ciertos campos hayan sido rellenados previamente.

La estructura principal del mensaje se compone de 4 estructuras de mensajes distintos, de donde solo 1 es obligatoria, la configuración.

Dicha configuración define que tipo de mensaje será, a través de una enumeración que mostrará si es un ACK, si es un mensaje para cerrar la

```
syntax = "proto3";

package TFG;
```

Figure 5.10: ProtocolBuffer version

```
message Message{
  Config config = 1;
  StartCommunication startCommunication = 2;
  SystemMetric data = 3;
  ACK ack = 4;
}
```

Figure 5.11: Estructura mensaje

comunicación, en caso contrario para crearla o un mensaje con las métricas del sistema .

El mensaje de tipo “Config”, dispone del hostname de quién manda el mensaje, la ip y el puerto por el cuál se comunican. La enumeración “Message type” identifica que tipo de mensaje es, por lo tanto aunque no ponga que sean campos requeridos es necesario rellenarlo siempre que se manden mensajes entre sistemas.

En la figura 5.5 se aprecian 3 tipos de mensajes a parte del mensaje de configuración obligatorio, dejando para el final el mensaje de tipo “SystemMetric”, se proseguirá explicar los tipos de mensajes “ACK” y “StartCommunication”. Ambos tipos de mensajes solo disponen de un string como se ve en la figura 5.7, en el caso de “StartCommunication” la contraseña por la cuál va a permitir empezar la interacción con el servidor y en el caso de “ACK” el mensaje de vuelta que ejecuta el servidor.

Por último, la estructura de mensaje que dispone de las métricas, llamada “SystemMetric” que se puede observar en la figura 5.11. Esta estructura de mensaje dispone de un string que define el tipo de métricas que recibe: ubuntu, raspbian y en el caso de que se inserten nuevos sistemas el tipo que las define; por otro lado un timestamp de la hora a la que se mandó el mensaje, dicho timestamp es obligatorio con el trato de InfluxDB, la base

```
message Config {
  string Hostname = 1;

  enum Message_Type {
    CLOSE_COMMUNICATION = 0;
    METRICS = 1;
    START_COMMUNICATION = 2;
    ACK = 3;
  }

  Message_Type message_type = 2;
  string ip = 3;
  int32 port = 4;
}
```

Figure 5.12: Configuración obligatoria del mensaje

```
message StartCommunication {
  string password = 1;
}

message ACK {
  string response = 1;
}
```

Figure 5.13: Tipos básicos de mensaje

de datos de series temporales y finalmente todas las métricas que se recogen de los clientes, dichas métricas están definidas y detalladas en el apartado 5.1 “Estándares de métricas”. En el caso de que una métrica no se pueda recoger, se seteará al valor 0 por defecto.

5.7 Generación de contenedores

Una vez desarrollado los puntos claves como el cliente y el servidor, pasamos a la virtualización de los mismos, esto nos permite abstraernos en la capa de la aplicación que agrupa el código y las dependencias juntas. Se pueden ejecutar varios contenedores en la misma máquina y compartir el núcleo del sistema operativo con otros contenedores, cada uno de los cuales se ejecuta

```
message SystemMetric {
  string SystemMetrics = 1;
  int64 actualTime = 2;

  message latency {
    float minRTT = 1;
    float meanRTT = 2;
    float maxRTT = 3;
    float mdevRTT = 4;
    int32 packageTransmited = 5;
    int32 packageReceived = 6;
    float packageLossPercentage = 7;
    int32 timeRequest = 8;
    int32 clientServer = 9;
  }

  message cpu {
    float userPercentage = 1;
    float nicePercentage = 2;
    float systemPercentage = 3;
    float iowaitPercentage = 4;
    float stealPercentage = 5;
    float idlePercentage = 6;
  }
```

Figure 5.14: Estructura SystemMetrics

como procesos aislados en el espacio del usuario.

Para ello, generamos un archivo llamado "docker-compose.yml", dicho archivo es una herramienta que permite simplificar el uso de Docker. A partir de archivos YAML es mas sencillo crear contendores, conectarlos, habilitar puertos, volumenes, etc. En vez de utilizar Docker via una serie inmemorizable de comandos bash y scripts, Docker Compose te permite mediante archivos YAML para poder instruir al Docker Engine a realizar tareas, programáticamente. Y esta es la clave, la facilidad para dar una serie de instrucciones, y luego repetirlas en diferentes ambientes.

El archivo YAML dispone de varios servicios, por lo tanto, vamos a subdividirlo en secciones para hablar sobre cada una de ellos.

5.7.1 Docker del servidor

Como se puede apreciar en la figura 5.12, el servidor dispone de unas configuraciones base que se nombraran a continuación.

```
server_metrics:
  restart: always
  build:
    context: .
    dockerfile: ./config/server/Dockerfile
  container_name: server_metrics
  entrypoint:
      - /entrypoint.sh
  working_dir: /server_metrics/
  volumes:
    - ./server_metrics:/server_metrics
  depends_on:
    - influxdb
```

Figure 5.15: Configuración del servidor en docker

Este docker se llama "server-metrics" y siempre que ocurra un error interno se reiniciará el docker, esto nos permite evitar tener que solucionar posibles fallos que pueda dar la aplicación, además la imagen de docker se va a construir en función de un Dockerfile (un archivo de configuración interno de Docker) como se puede observar en la figura 5.13, se trata de una configuración básica por no decir sencilla, solo se encarga de generar el entorno de python donde va a trabajar, copiar archivos de configuración a la raíz de la máquina, instalar las dependencias de las librerías de python, que en este caso son las librería de influx y de ProtocolBuffer, y actualizar la máquina con un update.

La imagen docker del servidor necesita que le especifiquemos un "entrypoint", que simplemente es un script en bash para explicarle el proceso o los procesos que se va a encargar de hacer, en este caso simplemente ejecuta el script del servidor (python server.py) que está escrito en python. Asimismo, especificamos donde queremos que trabaje la máquina, es decir, en que car-

```
# Use an official node runtime as a parent image
FROM python:3.6
ENV PYTHONUNBUFFERED 1
COPY ./config/server/entrypoint.sh .
COPY /config/server/requirements.txt .
RUN pip3 install --upgrade pip -r requirements.txt
RUN apt-get update
```

Figure 5.16: Dockerfile del servidor

peta, la carpeta "server-metrics" dispone de todos los archivos necesarios para que pueda trabajar el servidor sin que ocurra fallos, dichos archivos se pueden observar en la figura XXXXXXXXX.

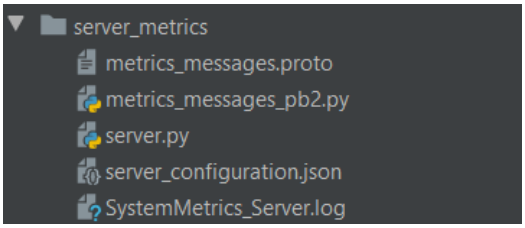

Figure 5.17: Archivos del servidor

Especificamos que este contenedor depende del de "influxdb" ya que no puede inicializarse este docker primero, generaría problemas de sincronización, el servidor se encarga de insertar las métricas en la base de datos, si la base de datos no está levantada aún ocasionará fallos. Para concluir, en este docker se especifica un volumen, cuando un contenedor es borrado, toda la información contenida en él, desaparece, dejamos de tener almacenamiento persistente en nuestros contenedores, es decir, que no se elimine al borrar el contenedor, es necesario utilizar volúmenes de datos, dichos datos en un servidor no aportan mucho, pero a la hora de recoger los logs nos interesa que no se borren.

5.7.2 Docker del cliente

Como se observa en la figura XXXXXXXXX, el cliente dispone de unas configuraciones base que se nombraran a continuación.

```yaml
client_cron_metrics:
  restart: always
  build:
    context: .
    dockerfile: ./config/client/Dockerfile
  container_name: client_cron_metrics
  entrypoint:
      - /entrypoint.sh
  working_dir: /client_cron_metrics/
  volumes:
    - ./client_cron_metrics:/client_cron_metrics
  depends_on:
    - server_metrics
```

Figure 5.18: Configuración del cliente en docker

Este docker se llama "client-cron-metrics" y siempre que ocurra un error interno se reiniciará el docker, esto nos permite evitar tener que solucionar posibles fallos que pueda dar la aplicación, además la imagen de docker se va a construir en función de un Dockerfile (un archivo de configuración interno de Docker) como se puede observar en la figura XXXXXXXXX, se trata de una configuración básica solo se encarga de generar el entorno de python donde va a trabajar, copiar archivos de configuración a la raíz de la máquina, instalar las dependencias de las librerías de python, que en este caso es la librería de ProtocolBuffer, instalar un par de progamas que nos ayudan a recoger métricas, y actualizar la máquina con un update.

```
# Use an official node runtime as a parent image
FROM python:3.6
ENV PYTHONUNBUFFERED 1
COPY ./config/client/entrypoint.sh .
COPY /config/client/requirements.txt .
RUN pip3 install --upgrade pip -r requirements.txt
RUN apt-get update
RUN apt-get install jq -y
RUN apt-get install net-tools
RUN apt-get install sysstat -y
```

Figure 5.19: Dockerfile del cliente

La imagen docker del cliente necesita que le especifiquemos un "entry-point", que simplemente es un script en bash para explicarle el proceso o los procesos que se va a encargar de hacer, en este caso simplemente ejecuta el script del cliente (python client.py) que está escrito en python. Asimismo, especificamos donde queremos que trabaje la máquina, es decir, en que carpeta, la carpeta "client-cron-metrics" dispone de todos los archivos necesarios para que pueda trabajar el servidor sin que ocurra fallos, dichos archivos se pueden observar en la figura XXXXXXXXX.

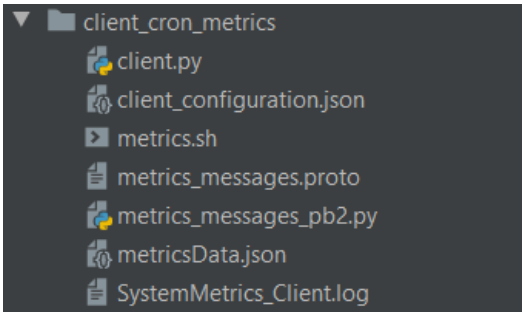

Figure 5.20: Archivos del cliente

Especificamos que este contenedor depende del de "server-metrics" ya que no puede inicializarse este docker primero, generaría problemas de comunicación, el cliente se comunica con el servidor, si mandamos un mensaje desde el cliente antes de que se inicie el servidor ocurriría un timeout y perderíamos esa primera comunicación. Para concluir, en este docker se especifica un volumen, cuando un contenedor es borrado, toda la información contenida en él, desaparece, dejamos de tener almacenamiento persistente en nuestros contenedores, es decir, que no se elimine al borrar el contenedor, es necesario utilizar volúmenes de datos, dichos datos en un cliente no aportan mucho, pero a la hora de recoger los logs nos interesa que no se borren.

5.7.3 Docker de la base de datos

Como se observa en la figura XXXXXXXXX, la base de datos dispone de unas configuraciones base que se nombraran a continuación.

El docker de InfluxDB dispone de las configuraciones más básicas, se especifica la versión de la imagen que queremos tener, en este caso la última, como solo queremos almacenar datos la versión no resulta tan importante

Figure 5.21: Configuración de la base de datos en docker

como lo podría ser si fuésemos a generar campos calculados a partir de variables. Asimismo, le ponemos un nombre al contenedor y un hostname, este hostname no nos servirá para comunicarnos desde el servidor ya que en la configuración le hemos dicho que pertenece a una red interna, por lo que si quisiéramos comunicarnos con la base de datos tiene que ser a través de la red que se llama monitoring por el puerto que seteamos por defecto el 8086, de dicha red hablaremos más adelante. En el caso de que falle la base de datos, esta se reiniciará como método de contención ya que le pusimos un restart "on-failure". Además, se han seteado unas variables de entorno para garantizar el acceso a la base de datos como se observa en la imagen. Por último, la base de datos necesita un volumen para no perder ningún dato, la persistencia en las bases de datos es fundamental y más cuando disponemos de ingestas cada dos minutos.

Cabe destacar que el script del proceso del servidor es el que se encarga de generar la política de retención en la primera inserción que se haga, en este caso mensual, cada 30 días se van eliminando los datos recogidos, en el caso de que queramos aumentar o disminuir el tiempo de retención se podrían

crear nuevas políticas.

```
client = InfluxDBClient(host='178.17.22.1', port=8086, database='metrics')
if {'name': 'metrics'} not in client.get_list_database():
    client.create_database('metrics')
    client.create_retention_policy("Monthly", "4w", "2", database='metrics')
    client.create_user("admin", "admin1234", admin=True)
```

Figure 5.22: Configuración de la base de datos en su inicio

5.7.4 Docker del visualizador de gráficas

Como se observa en la figura XXXXXXXXX, el visualizador de gráficas dispone de unas configuraciones base que se nombraran a continuación.

```
grafana:
  image: grafana/grafana:7.0.3
  container_name: grafana
  restart: on-failure
  ports:
    - 4000:3000
  networks:
    - monitoring
  volumes:
    - grafana-volume:/var/lib/grafana
```

Figure 5.23: Configuración de Grafana en docker

El docker de Grafana dispone de las configuración más básica, se especifica la versión de la imagen que queremos tener. Asimismo, le ponemos un nombre al contenedor y en el caso de que falle Grafana, esta se reiniciará como método de contención ya que le pusimos un restart "on-failure". Grafana se aloja en una red interna llamada monitoring, la cuál se hablará en este mismo apartado en una subsección distinta, además hay una conversión de puertos, Grafana por defecto se setea en el puerto 3000 pero se ha hecho una conversión al puerto 4000 ya que el 3000 ya estaba siendo utilizado por otra

aplicación. Por último, se dispone de un volúmen para mantener la persistencia, puede parecer inútil en esta aplicación, pero si quisiéramos cambiar la versión de Grafana por una más reciente o estable perderíamos todos los dashboards realizados.

5.7.5 Volúmenes

Docker como servicio gestiona por si solo la persistencia de la información de una manera muy simple, cada imágen y contenedor que tengamos en nuestro equipo se conservarán hasta que los borremos. En cada contenedor creado tendremos asociado un "volúmen" en nuestro equipo real donde se almacenarán los ficheros que copiemos a ese contenedor. Existen tres maneras distintas de gestionar espacios para contenedores, dos de ellas serán persistentes (volume o bind volume) y la tercera no (TMPFS temporal file system). Al decir que "no es persistente" queremos decir que cuando un contenedor se apague todos los ficheros almacenados en esa carpeta se borrarán y en cada corrida del contenedor comenzaremos con ese espacio vacío.

Un volúmen es la manera sencilla y predefinida para almacenar todos los ficheros (salvo unas pocas excepciones) de un contenedor, usará el espacio de nuestro equipo real y en "/var/lib/docker/volumes" creará una carpeta para cada contenedor. En este trabajo se harán dos volúmenes como recursos externos, lo que significa que ya se ha definido en Docker, en este caso mediante la consola de comandos con un create volume, como se observa en la figura XXXXXXXXXXX.

5.7.6 Redes internas

De forma predeterminada, Docker-Compose configura una única red para su aplicación. Cada contenedor que de un servicio se une a la red predeterminada y es accesible para otros contenedores en esa red y detectable por ellos con un nombre de host idéntico al nombre del contenedor.
En lugar de simplemente usar la red de aplicaciones predeterminada, se puede especificar las propias redes con la palabra clave "networks" que abstrae la red predeterminada a una de nivel superior .Esto permite crear topologías más complejas y especificar opciones y controladores de red personalizados. También puede usarse para conectar servicios a redes creadas externamente que no son administradas por Docker-Compose.

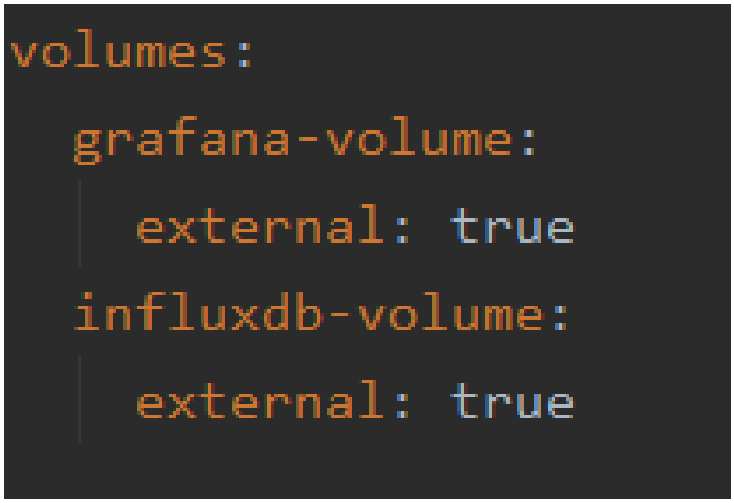

Figure 5.24: Configuración de los volúmenes en docker

En este caso como se ve en la figura XXXXXXXXX, se ha creado una red interna llamada monitoring que sirva como puente entre los contenedores, configurada con una máscara de red constituida por un rango de 24 que implica que puedan conectarse 255 máquinas.

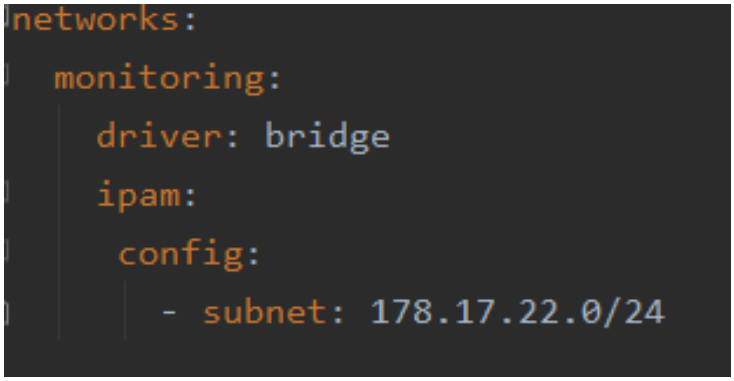

Figure 5.25: Configuración de las redes en docker

5.8 Estudio de gráficas

Chapter 6

Conclusiones

Chapter 7

Bibliografía

https://pandorafms.com/blog/es/monitorizacion-de-sistemas/
https://www.docker.com/resources/what-container
https://developers.google.com/protocol-buffers
https://medium.com/jmtorres/protocol-buffers-f5b266783652
https://grafana.com/oss/grafana/
https://en.wikipedia.org/wiki/InfluxDB
https://www.influxdata.com/
https://www.jetbrains.com/es-es/pycharm/
https://help.ubuntu.com/lts/installation-guide/s390x/ch01s01.html
http://metodos.fam.cie.uva.es/ latex/apuntes/apuntes3.pdf
https://docs.latexbase.com/symbols
https://www.comoinstalarlinux.com/como-usar-el-editor-nano-linux/
https://es.wikipedia.org/wiki/Curses
https://git-scm.com/
https://code.visualstudio.com/
https://es.wikipedia.org/wiki/LaTeX
https://www.ticportal.es/glosario-tic/base-datos-database
https://www.40defiebre.com/mejores-herramientas-dashboard-analitica-web
https://dockertips.com/utilizando-docker-compose
https://dockertips.com/volumenes
https://docs.docker.com/compose/networking/

Chapter 8

Anexo

List of Figures