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:

2.4 Aumentar mis 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. Es por ello, que el uso de los sistemas Linux me va a ayudar al manejo de esos sistemas.

2.5 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.

2.6 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.

2.7 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 cuncurrente.

Chapter 3

Estado de la cuestión

Chapter 4

Aspectos metodológicos

4.1 Metodología

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.

4.3 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 ligero, independiente y ejecutable que incluye todo lo necesario para ejecutar una aplicación: código, tiempo de ejecución, herramientas del sistema, bibliotecas del sistema y configuraciones.

Las imágenes de contenedor 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.

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.

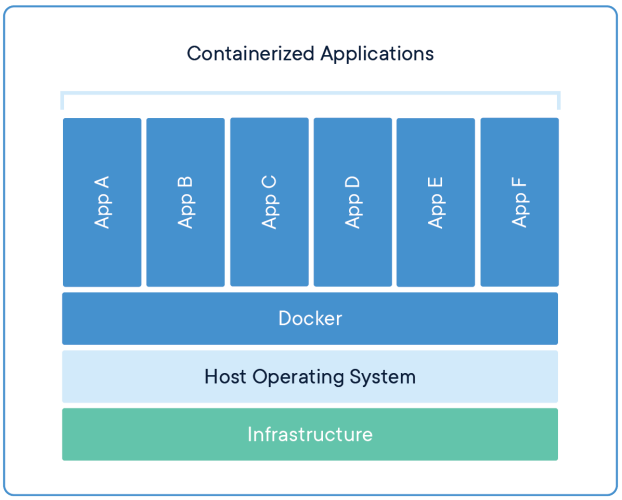

Figure 4.1: Container

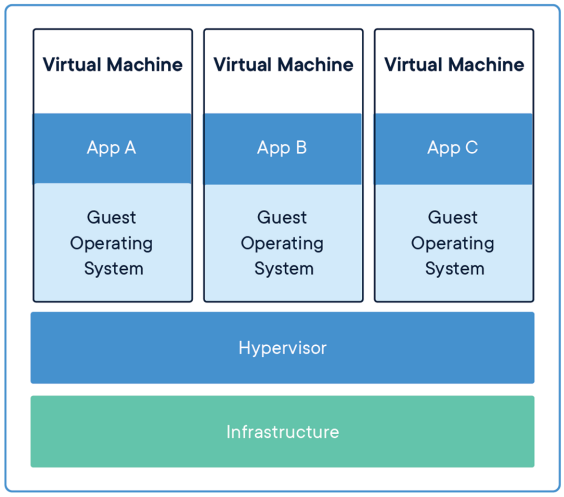

Figure 4.2: VM

4.4 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 seralización 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 codifi-

can 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

4.5 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.

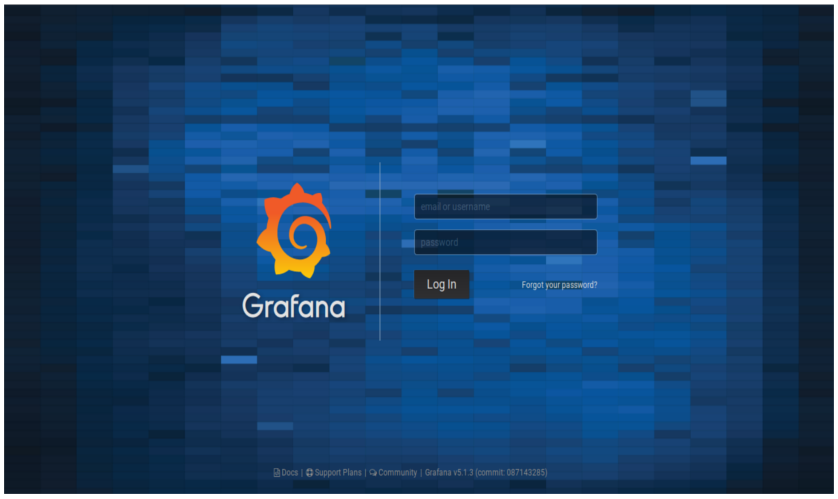

Figure 4.4: Login Grafana

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.

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

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,

Figure 4.5: Logo InfluxDB

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.

4.7 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

4.8 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.

4.9 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.

4.10 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**.

4.11 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

Figure 4.9: Logo bash

y la interfaz de fácil manejo de Pico, pero sin la integración con Pine.

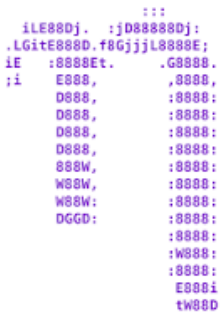

Figure 4.10: Logo nano

4.12 Git

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

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 4.11: Logo git

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 medida de las métricas

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

5.2 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)

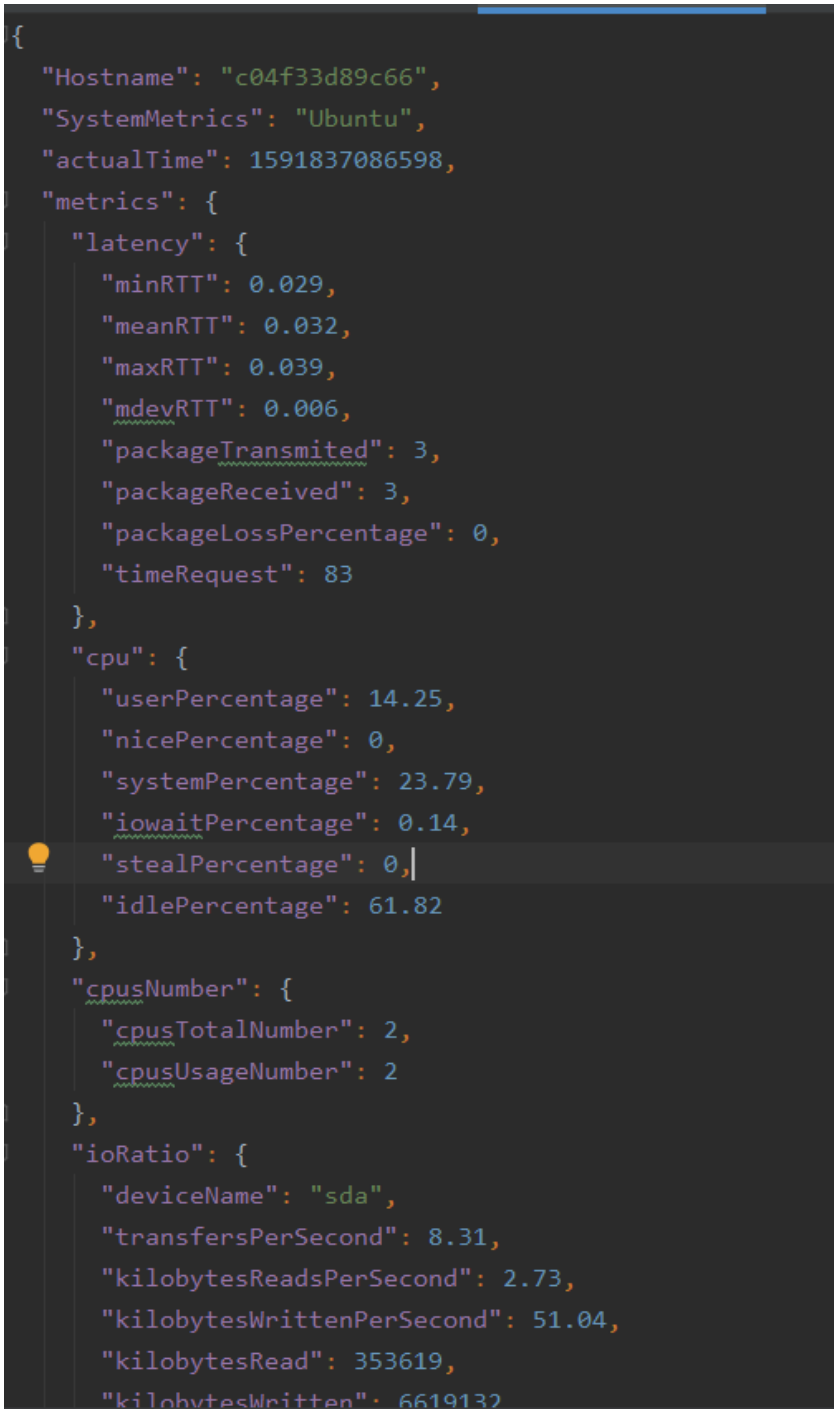

Figure 5.1: Métricas JSON

5.3 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)

5.4 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)

5.5 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)

5.6 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)

5.7 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)

5.8 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)

5.9 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)

5.10 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)

5.11 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)

5.12 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)

5.13 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)

5.14 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.

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

Figure 5.2: Server configuration

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 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.

5.16 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 5.3: 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.

5.17 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 el 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.

Chapter 6

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/

Chapter 7

Anexo

List of Figures