



Análisis de rendimiento de una red de pruebas para entornos IIoT basada en Modbus

PROYECTO

presentado para optar
al Título de Grado en Ingeniería en Sistemas de Telecomunicación por
Mikel Culla Galo
bajo la supervisión de
Santiago Figueroa

Donostia-San Sebastián, septiembre 2020



tecnun
Universidad
de Navarra

Tabla de contenido

Resumen	1
Introducción.....	2
Objetivos	5
Desplegar una red Modbus N a 1 en docker	5
Comprobar el rendimiento de la red.....	5
Crear una red de prueba para un Cyber Range.....	5
Estado del Arte	7
Entorno IIoT, convergencia entre IT y OT	7
Modbus	7
Pymodbus	8
Cyber Range	9
Docker	9
Dockprom.....	10
AWS	11
Diseño	13
Instancia AWS EC2	14
Docker-Compose.....	14
Servidor Modbus	14
Cliente Modbus	15
Arquitectura TLS	16
Implementación	17
Servidor Modbus TCP	17
Cliente Modbus TCP	18
Servidor Modbus TLS	21
Cliente Modbus TLS	24
Docker-Compose	26
Puesta en marcha en la Instancia AWS EC2.....	37
Creación de la Instancia	37
Configuración de la máquina y ejecución	41
Monitorización de la información.....	43
Descripción del proceso de experimentación.....	46
Resultados.....	47

Red Modbus TCP	47
Red Modbus TLS	49
Conclusiones	54
Recomendaciones.....	56
Bibliografía	58

Tabla de Ilustraciones

Ilustración 1 Beneficios del IIoT. Diseño propio, datos de i-Scoop [1].	2
Ilustración 2 Incremento del mercado del IIoT [2].....	3
Ilustración 3 Comparación entre máquinas virtuales y contenedores.....	10
Ilustración 4 Arquitectura del sistema y red.	13
Ilustración 5 Diagrama de flujo de performance.py.....	15
Ilustración 6 Arquitectura de sistema y red TLS	16
Ilustración 7 Generación de los certificados.....	23
Ilustración 8 Lanzar una instancia	37
Ilustración 9 Selección de AMI	38
Ilustración 10 Avanzar en la configuración.....	38
Ilustración 11 Apertura de todo el tráfico	39
Ilustración 12 Generación de par de claves.....	40
Ilustración 13 Conectarse a la instancia	40
Ilustración 14 Nombre de Dominio e IP pública de la instancia.	43
Ilustración 15 Panel Grafana.....	44
Ilustración 16 Dashboards de Grafana.....	44
Ilustración 17 Dashboard de Contenedores Docker.....	45
Ilustración 18 Logs de cliente	46
Ilustración 19 Resultados de Grafana con 1 cliente TCP	47
Ilustración 20 Resultados de Grafana con 4 clientes TCP.....	47
Ilustración 21 Resultados de Grafana con 32 clientes TCP.....	48
Ilustración 22 Resultados de Grafana con 8 clientes TCP.....	48
Ilustración 23 Resultados de Grafana con 16 clientes TCP.....	48
Ilustración 24 Resultados de Grafana con 1 cliente TLS.....	50
Ilustración 25 Resultados de Grafana con 4 clientes TLS	50
Ilustración 26 Resultados de Grafana con 32 clientes TLS	51
Ilustración 27 Resultados de Grafana con 16 clientes TLS	51
Ilustración 28 Resultados de Grafana con 8 clientes TLS	51
Ilustración 29 Sobrecarga del servidor en TLS	53

Índice de Tablas

Tabla 1 Tipos de Instancia T2 de AWS.	12
Tabla 2 Crecimiento de memoria en el servidor TCP	49
Tabla 3 Peticiones por segundo en TCP	49
Tabla 4 Crecimiento de memoria en el servidor TLS	52
Tabla 5 Peticiones por segundo en TLS.....	52

Resumen

La intención de este proyecto es la de probar las capacidades de una red Modbus. Para ello, se ha generado una red de pruebas la cual será la base de un Cyber Range. Para llevar el despliegue virtualizado se ha usado la herramienta Docker. Dentro de la red Modbus existen un conjunto de contenedores que actúan como clientes Modbus (simulando la función de un Scada en una red Modbus real), mientras que otros lo hacen como servidores, simulando una pasarela Modbus (por ejemplo, Moxa Mgate 5111) o un PLC real que soporta Modbus (por ejemplo, Modicon M221). Las interacciones se suceden tal y como lo haría en una red Modbus, cumpliendo las especificaciones del protocolo cuando los contenedores se comunican entre ellos.

Adicionalmente, como parte de la red Modbus, también se han desplegado otros contenedores la función de monitorización y control de esta. Este despliegue incluye herramientas especializadas en la captura de métricas y monitorización, siendo Prometheus la más popular de todas. El objetivo de la monitorización de la red a partir de la obtención de métricas es contar con toda la información relevante para sacar conclusiones válidas.

Posteriormente al despliegue de dicha red se han ejecutado y procesado distintos escenarios variando el número de clientes en los que sobre los cuales se realizan pruebas de rendimiento para estresar el comportamiento del servidor. Estas peticiones permiten comprobar el estado de la red y el rendimiento de los contenedores con especial énfasis en el servidor.

Por un lado, los resultados obtenidos en el proyecto pueden tener relevancia en prácticamente toda la industria ya que los entornos OT utilizan mayormente el protocolo Modbus en sus sistemas. Por otro lado, la creación y despliegue de una red virtual Modbus, útil como arquitectura base para un Cyber Range, abre las puertas de nuevos proyectos en los que se prueben y se lleven al límite la red de otras maneras, por ejemplo, escenarios ataque – defensa dentro de dicha red.

Introducción

Al igual que los dispositivos que usamos en nuestro día a día han tendido a la interconexión, la industria también lo ha hecho. El smartphone, la Smart TV, pulseras inteligentes, y hasta frigoríficos se están conectando a internet y sacando partido de ello gracias a la automatización e información que obtienen de ello, y los dispositivos industriales no iban a ser menos. Atrás quedaron las líneas de producción donde las maquinarias industriales las manejaba una persona a mano, y donde la fábrica estaba totalmente desconectada de la información del mundo exterior. En busca de la eficiencia, y por supuesto, del ahorro de tiempo y dinero, la industria ha ido interconectando todas sus máquinas hasta formar el paradigma IIoT (Industrial Internet of Things).

El IIoT se basa en sus beneficios para extenderse por todas las plantas de fabricación, y es que sus ventajas no son pocas. Como bien se puede ver en la Ilustración 1 del artículo de i-Scoop [1], los beneficios pueden ser de hasta un 47% de mejora en la eficiencia operacional o un 33% de mejora en la productividad.

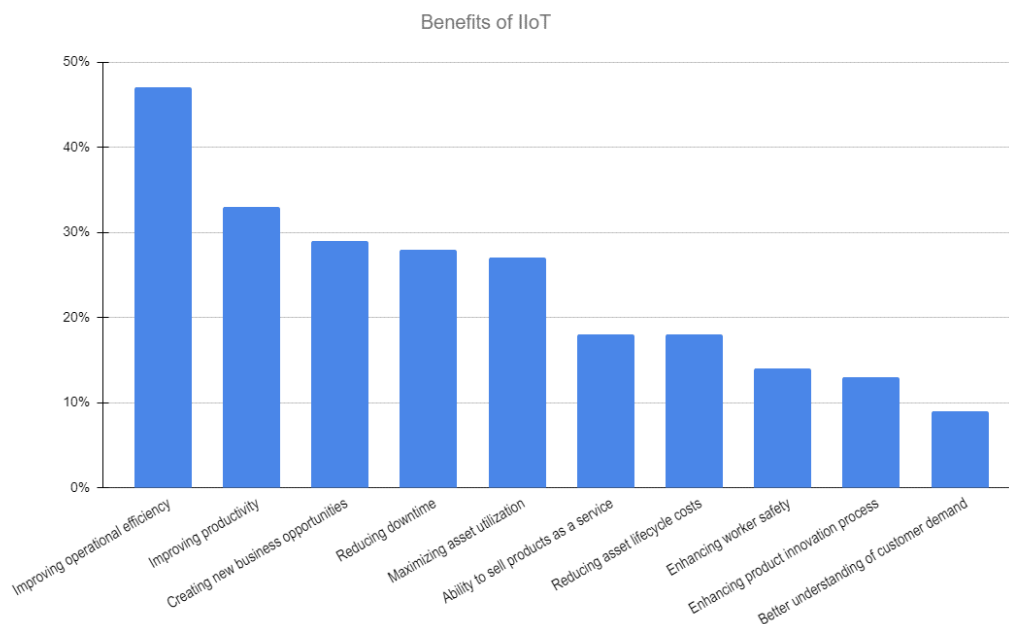


Ilustración 1 Beneficios del IIoT. Diseño propio, datos de i-Scoop [1].

Un sistema IIoT se compone de dispositivos inteligentes, como pueden ser sensores, que se encargan de recopilar información y enviarla a otros dispositivos encargados de almacenarla, analizarla y en caso de ser necesario actuar. Además, el hecho de almacenar la información puede ser

una buena práctica para en un futuro comparar datos y así poder prevenir posibles problemas y actuar de forma preventiva.

Estos beneficios han propiciado que el mercado del IIoT crezca a pasos agigantados como era de esperar y como muestra la Ilustración 2 de Envira [2].



Ilustración 2 Incremento del mercado del IIoT [2].

Contar con flujos de mercado tan sumamente grandes, ha hecho que los ciberdelincuentes hayan puesto su ojo en el IIoT. Pues ellos, en busca de su beneficio propio, pueden generar dinero diseñando ataques, tal y como lo han hecho en algunas empresas con el Ransomware, por ejemplo. Este ataque es un malware que se extiende por el sistema informático de la empresa que va cifrando todos los datos, para luego pedir rescate por la clave de descifrado, básicamente es un secuestro de los datos de la empresa. Los rescates usualmente se piden en criptomonedas como Bitcoin o Monero. Adicionalmente a estos ataques, investigadores de ciberseguridad encuentran vulnerabilidades en dispositivos IIoT, las cuales, de ser explotadas pueden afectar, por ejemplo, la disponibilidad del dispositivo, el cual es un pilar básico para entornos IIoT. Por citar un ejemplo, el investigador independiente Maxim Rupp identificó una vulnerabilidad de bypass de autenticación en los productos MGate de Moxa, que es una pasarela Modbus-serial, la cual fue documentada como CVE-2016-5804¹.

¹ <https://us-cert.cisa.gov/ics/advisories/ICSA-16-196-02>

Ataques y detección de vulnerabilidades como los mencionados, han puesto en alerta a las industrias, y han provocado que comiencen a tomarse la ciberseguridad en serio, revisando sus sistemas y protocolos, así como creando escenarios ataque-defensa sobre una red virtualizada conocidos como Cyber Range.

Objetivos

En este trabajo se diseña e implementa una red Modbus TCP/IP, como red de pruebas para un “Cyber Range”. Para el despliegue satisfactorio de la misma se llevan a cabo pruebas para determinar el rendimiento de esta, de manera que se puedan analizar los resultados y compararlos con dispositivos reales para así poder prever como se comportaría una red real en dichos escenarios. A continuación, analizamos a fondo los objetivos del trabajo:

DESPLEGAR UNA RED MODBUS N A 1 EN DOCKER

El hecho de conseguir una red simulada, capaz de funcionar por si sola de la forma más real posible, ya por sí mismo sería una buena hazaña. Para conseguir el objetivo hay que crear los dispositivos virtuales que deben actuar como hacen los reales en un sistema real. Una vez diseñados se deben conectar y coordinarlos para conseguir así, una simulación lo más veraz posible.

El despliegue de dicha red se hará en Docker, y será de arquitectura 1 a n, siendo 1 el servidor y n los clientes Modbus, cuyo número variará en función de las pruebas que se quieran hacer. Esto se hará de forma que el despliegue y ampliación de la red sea posible y de la forma más automatizada posible.

COMPROBAR EL RENDIMIENTO DE LA RED

Una vez la red esté desplegada, para comprobar cuánto se asemeja a una red real, se harán pruebas de rendimiento y los resultados se analizarán y compararán con el rendimiento esperado de los dispositivos reales típicos en una red Modbus. Estas pruebas exigen cierta flexibilidad y escalabilidad ya que se necesitan desplegar distintos dispositivos. Estas características nos llevan al tercer objetivo, la creación de un de la arquitectura para un Cyber Range.

CREAR UNA RED DE PRUEBA PARA UN CYBER RANGE

La insistencia en la escalabilidad de la red, flexibilidad y su veracidad, es para garantizar la creación de un Cyber Range útil y funcional. Un Cyber Range como el que se pretende crear puede ser una gran herramienta, no solo para las pruebas de rendimiento que se harán en

este trabajo, si no que da alas a otros futuros proyectos concediéndoles un entorno seguro donde experimentar con una red Modbus y sus dispositivos, y les otorga la facilidad para monitorizarlos.

Estado del Arte

Para la correcta consecución de los objetivos definidos anteriormente, se deben definir un conjunto de plataformas y herramientas que son parte constituyente de la red de prueba. Por un lado, hay herramientas que se utilizan para la obtención o facilitación de los objetivos como pueden ser Amazon Web Services (AWS) o Docker, y por otro lado tecnologías de un entorno industrial como Modbus. A continuación, se realiza un breve análisis de cada una de estas en el contexto del paradigma IIoT.

ENTORNO IIOT, CONVERGENCIA ENTRE IT Y OT

Un entorno OT (Operational Technology) se refiere a aquellos sistemas de computación que se usan para gestionar operaciones industriales en vez de operaciones administrativas [3]. Estos sistemas se centran mayormente en el control y monitorización continua de procesos industriales como una línea de producción. Para ello, se usan los PLC(Programable Logic Controllers) que son dispositivos que cumplen esta función y por tanto muy extendidos en la industria.

A diferencia de los entornos IT (Information Technology), donde la prioridad del sistema es el intercambio de la información, los OT donde la información es secundaria y tan solo un accionador de su principal objetivo que es actuar físicamente o producir un cambio, como podría ser por ejemplo, el control operaciones mineras. En los OT, el hardware y software detecta o causa cambios a través de la monitorización y/o control físico de los dispositivos, procesos y eventos en la misión [4].

Hace años la diferenciación entre IT y OT era más clara., ya que los entornos OT se utilizaban con protocolos y software propios, además de que la red estaba por lo general completamente separada de Internet. Pero con el tiempo la industria ha tendido a conectar más sus sistemas y globalizarlos, juntando los entornos OT con los IT y formando de esta manera el IIoT. Sin embargo, la convergencia de estos entornos (OT e IT) conectados a la red global de internet, se han convertido así en un blanco más fácil para ataques cibernéticos. Dotar a la red de mecanismos de simulación que permitan recrear modelos de ataque y defensa sobre esta, es una de las herramientas que pueden garantizar la protección de la misma.

MODBUS

Modbus es uno de los protocolos para entorno OT está diseñado para el control de una red de dispositivos (PLCs). Fue diseñado por Modicon en

1979 y se fue convirtiendo en el protocolo más usado en conexiones de dispositivos electrónicos industriales hasta considerarse un protocolo estándar de facto [5]. Esto fue gracias a su fácil implementación y ser “*royalty-free*”.

Dado el éxito del protocolo, y por otro lado el creciente uso del protocolo TCP/IP en internet (red que en aquella época empezaba a crecer de forma exponencial), no se tardó en combinarlos y formar el protocolo Modbus TCP/IP, el cual definitivamente se ha hecho omnipresente en todos los sistemas industriales, gracias, otra vez, a los beneficios del protocolo Modbus sumados a la conectividad y simplicidad del TCP/IP.

Hay que tener en cuenta que el diseño del protocolo Modbus se hizo pensando en los antiguos entornos OT, desconectados de internet y menos propensos a sufrir ciber ataques, por lo que se centraron en la funcionalidad y operabilidad, dejando de lado la seguridad. Esto junto con el hecho de que este protocolo se utilizase tanto en la industria, ha provocado que sea objeto de diferentes clases de ataques que comprometen tanto la confidencialidad como la integridad y la disponibilidad. Como una contramedida, Modbus Org² ha propuesto la integración del protocolo TLS (Transport Layer Security), para garantizar autenticación y proporcionar un canal cifrado, considerando que TLS se desarrolló para dar continuación al antiguo SSL, se usa en aplicaciones del tipo cliente-servidor de forma que se centra en la autenticación y la privacidad de la información [6]. El estándar Modbus TLS fue lanzado en el año 2018 ([7]) de manera que entidades como Sneider pretenden incluir TLS en las siguientes versiones de PLC³.

PYMODBUS

Pymodbus es una implementación completa del protocolo Modbus en Python y que cumple las especificaciones del estándar de Modbus Org [8]. Aunque Python sea un lenguaje de programación orientado a un tipo de hardware diferente del que se suele usar con Modbus, donde la mayoría de dispositivos suelen ser PLCs, la idea de esta implementación era controlar e inspeccionar los dispositivos de la red de una forma fácil y sencilla [9].

En busca de esa facilidad y sencillez, Python puede ser un gran aliado, ya que el lenguaje que se centra en la fácil legibilidad, su sencillez, y en que sea intuitivo. Estos beneficios permiten controlar todos los aspectos del protocolo fácil, ya que la librería está completamente escrita en Python. Un ejemplo puede ser que, la librería Pymodbus define distintos

² <https://www.modbus.org/>

³ <http://www.enodenetworks.com/assets/docs/ModbusManualDraft.pdf>

clientes y servidores en clases, y por tanto para crear un cliente puede bastar con una sola línea de código. Esto se aplica para cada tipo de uso de Modbus, ya sea Modbus RTU, Modbus TCP, Modbus UDP, o la reciente incorporación del protocolo TLS.

CYBER RANGE

Un Cyber Range es una plataforma virtual que permite simular entornos operativos reales con el propósito de las pruebas y experimentación de técnicas y tácticas de ciberseguridad [10].

Los Cyber-Range se están volviendo populares ya que una práctica habitual de las empresas, en busca de conocer el estado de su ciberseguridad, suele ser emular sus sistemas y dejarse atacar en esa emulación de forma segura y sin poner en riesgo sus sistemas y datos reales.

Para que el Cyber Range sea realmente útil, este debe ser capaz de emular el sistema completo de una empresa (red de prueba). Así que, además de ser un entorno seguro y controlado, el Cyber Range tiene que ser escalable y flexible. Adicionalmente, se encuentra la vinculación con un sistema de ataque-defensa aplicado sobre la red de prueba. Para lograr el diseño e implementación de una red de prueba, una opción deseable es usar contenedores Docker para dicho fin.

DOCKER

Docker es una plataforma para desarrollo, envío y ejecución de aplicaciones [11]. La novedad que trajo Docker fue que, en vez de ejecutar toda una aplicación, o servicio sobre una máquina virtual central, como se muestra en la Ilustración 3, los procesos y sub-servicios se ejecutan de forma independiente. Los contenedores Docker son ejecutados por un solo sistema operativo, evitando así tener que ejecutar distintos sistemas operativos como es necesario en las máquinas virtuales tradicionales, lo cual lo hace ser extremadamente eficiente. En los últimos años esta plataforma se ha hecho muy popular sobre todo en aplicación y hospedajes en web por la capacidad de individualizar cada servicio y recurso y mantenerlos independientes. Esto permite que si uno de esos servicios se suspende (ya sea de forma intencionada o por un fallo o ataque), no llegue a que toda la web y los demás servicios se vean afectados. Adicionalmente, Docker también otorga mayor seguridad que la integración de todos los servicios sobre una máquina virtual, ya que, al ser cada servicio o aplicación independiente del otro, si un atacante toma el control de uno, los otros no tendrán por qué verse afectados.

Aunque la idea inicial de Docker fuese de uso mayoritario en servicios web (microservicios), según fue ganando popularidad, los desarrolladores empezaron a fijarse en ella como una herramienta de experimentación. De esta manera, la aplicación final puede estar dividida en varias tareas y que cada una se ejecute en un contenedor, pero es que, a su vez, cada contenedor se divide en capas las cuales son también revisadas para la optimización y si hay duplicidades, la optimiza al máximo. Esta optimización y eliminación de duplicidades puede ser beneficioso para el desarrollo de una red en la que habrá múltiples clientes ejecutando el mismo código.

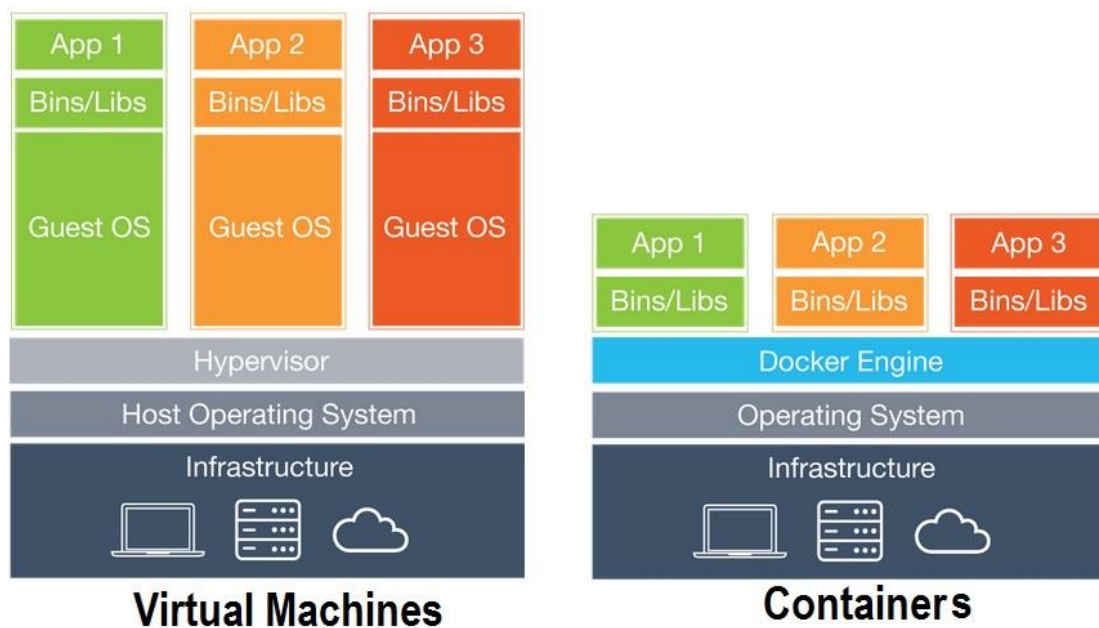


Ilustración 3 Comparación entre máquinas virtuales y contenedores.

DOCKPROM

Dockprom es una solución para monitorizar contenedores Docker desarrollado por Stefan Prodan [12]. Dockprom se encarga de coordinar distintas herramientas de monitorización y hacerlas funcionar en Docker. Entre estas herramientas se encuentran Prometheus, Grafana, cAdvisor, NodeExporter y AlertManager.

Prometheus es una aplicación de software usada para la monitorización y alertas en tiempo real [13]. Fue desarrollado en 2012 en SoundCloud (servicio de distribución de audio y sonido), cuando vieron que sus aplicaciones de monitorización se quedaban cortas. Su desarrollo fue open-source por que se comenzó a usar rápidamente en Docker. A pesar de que por sí solo ya es capaz de mostrar los datos en forma de gráficas, para una mayor comodidad se usa Grafana.

Grafana es un software libre para la visualización de datos. Grafana cuenta con muchos métodos para representar datos métricos, por ello Dockprom Grafana trabaja por encima de Prometheus, obteniendo los datos a mostrar de él, y mostrándolos por el puerto 3000.

cAdvisor es el contenedor encargado de obtener los datos métricos de otros contenedores que más tarde Prometheus recoge. Es un proceso de tipo demonio, es decir, se ejecuta en segundo plano de forma no interactiva [14]. Funciona de forma nativa con contenedores Docker, y colecta la información del uso y el rendimiento de cada contenedor, manteniendo cada proceso y contenedor de forma aislada.

NodeExporter al igual que cAdvisor su función es la de obtención de datos métricos, pero en este caso no son de los contenedores si no que de la máquina huésped, en el caso de este proyecto la máquina EC2 de AWS.

AlertManager gestiona las alertas cuando uno de los datos métricos pasa algún límite impuesto por el usuario. Cuando una de las métricas sobrepasa el límite en Prometheus, AlertManager notifica al usuario de dicho evento ya sea por email u otra aplicación como PagerDuty. Aunque en este caso no se le ha dado uso, ya que se ha limitado directamente la memoria del servidor.

AWS

Amazon Web Services (AWS) es una plataforma de servicios de computación en la nube[15]. Uno de los servicios que ofrece AWS es el EC2 (Amazon Elastic Compute Cloud) en el cual se puede “alquilar” un ordenador virtual que realmente se ejecuta a través de máquinas virtuales en uno de los grandes centros de procesamiento que tiene Amazon.

Dentro de este ordenador virtual el usuario puede elegir el sistema operativo, y características de la máquina que necesita, basado en lo que desee ejecutar. Hay muchos tipos de instancias que se pueden seleccionar, basándose en su arquitectura de CPU, si son Intel o AMD, y por supuesto según su rendimiento, cuantos CPUs, cuanta memoria, rendimiento de red y almacenamiento. Como se puede apreciar en la Tabla 1, hay una gran variedad de máquinas y esta tabla solo muestra las instancias de tipo T2, que son con procesadores Intel Xeon y se centran en un equilibrio de recursos informáticos, memoria y red. Pero realmente hay otros tipos como las A1 por ejemplo que se centran en el sistema de almacenamiento, o las M5n con soporte para redes neuronales. La flexibilidad ofrecida por las instancias es casi total, incluso una vez la instancia ha sido lanzada, existe

la opción de modificar la máquina. Gracias a esta escalabilidad ofrecida, y no tener que realizar una inversión inicial y pagar solamente por el uso, es un servicio que es extremadamente útil para desarrollos y pruebas como las de este proyecto.

Instancia	CPU virtual*	Créditos por hora de CPU	Memoria (GiB)	Almacenamiento	Rendimiento de red
t2.nano	1	3	0,5	Solo EBS	Bajo
t2.micro	1	6	1	Solo EBS	De bajo a moderado
t2.small	1	12	2	Solo EBS	De bajo a moderado
t2.medium	2	24	4	Solo EBS	De bajo a moderado
t2.large	2	36	8	Solo EBS	De bajo a moderado
t2.xlarge	4	54	16	Solo EBS	Moderado
t2.2xlarge	8	81	32	Solo EBS	Moderado

Tabla 1 Tipos de Instancia T2 de AWS.

Diseño

Una vez explicadas ciertas tecnologías y herramientas se ha diseñado un sistema con el propósito de cumplir los objetivos. El esquema de dicho sistema puede verse en la Ilustración 4 inferior donde vemos que está compuesto de varias de las herramientas y tecnologías anteriormente mencionadas.

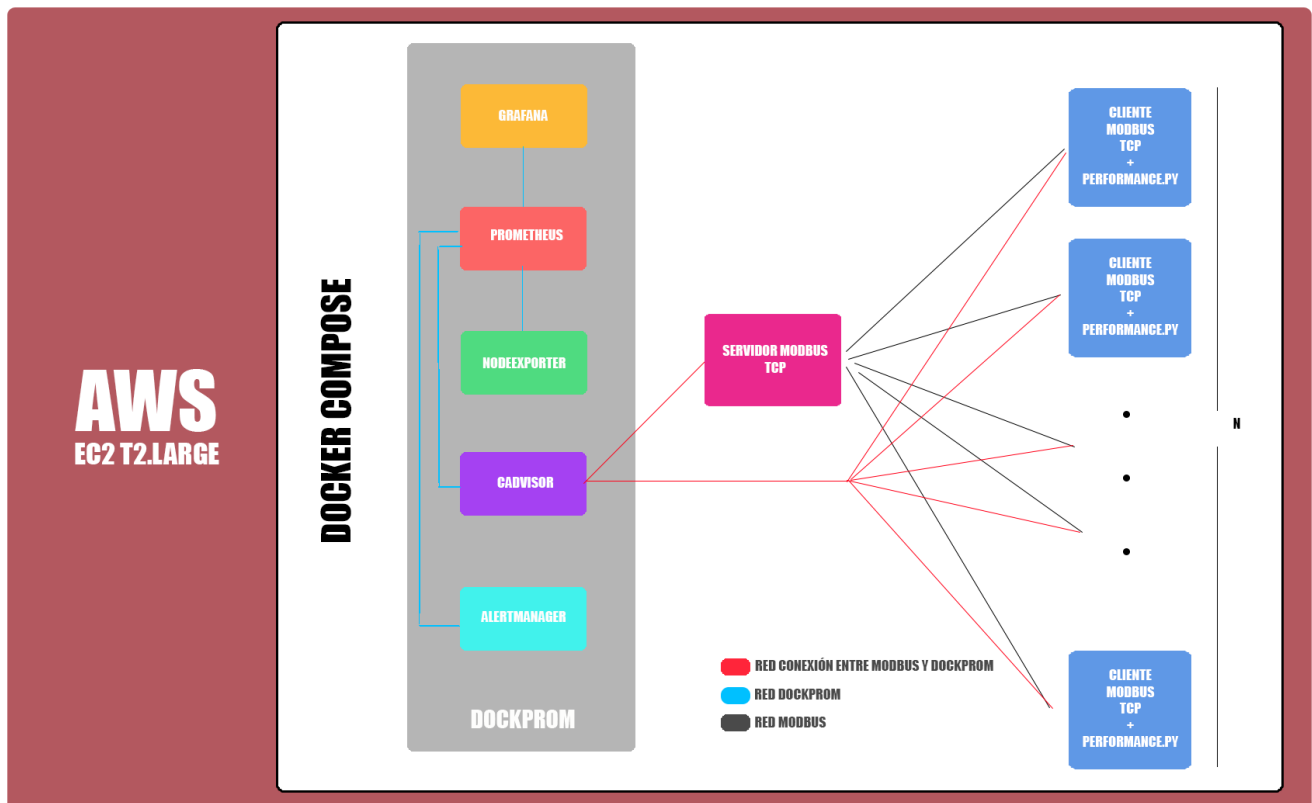


Ilustración 4 Arquitectura del sistema y red.

Como se puede apreciar en el esquema, la conexión de servidor a cliente tiene una relación de N a uno ($n:1$), donde el número de clientes (n) varía según la prueba, desde un cliente a un servidor ($1:1$), hasta 32 clientes a un servidor ($32:1$). Por esto, se hace tanto énfasis en la flexibilidad y escalabilidad del sistema, ya que la carga del sistema puede ser de tan solo 7 u 8 contenedores o hasta casi 40 contenedores docker. Por ello se ha optado por una instancia EC2 de AWS, una tecnología que permite disponer de más capacidad computacional cuando sea necesario.

INSTANCIA AWS EC2

Como ya se ha explicado antes, las instancia EC2 de Amazon dan la flexibilidad necesaria para desplegar el sistema. Inicialmente se utilizó una instancia de tipo t2.micro la cual constaba con 1 CPU y una memoria de 1GB pero resultó no ser suficiente para la prueba con 32 clientes. Por lo que finalmente se optó por una t2.large la cual incorpora 2 CPUs y 8GB de memoria RAM con sistema operativo Amazon Linux. En esta instancia es donde se ejecutan todos los contenedores.

DOCKER-COMPOSE

Docker Compose es un archivo que contiene y organiza todos los contenedores que componen el sistema. Este archivo, de nombre completo "docker-compose.yml" lo ejecuta el Docker Engine que debe estar instalado en la máquina EC2 y en él debe quedar definido que volúmenes se van a necesitar y cuantas redes van a ser creadas. Además de esto hay que especificar para cada contenedor la imagen que utilizará, los comandos a ejecutar dentro de ellos, los puertos a exponer (en caso de ser necesarios), posibles variables de entorno y a que red pertenece.

En el caso de este proyecto, se tomará como base el documento "docker-compose.yml" de Stefan Prodan [12] de base al que se le agregan los contenedores que formarán la red Modbus de un servidor a N clientes que va a ser monitorizada, es decir los contenedores de los servidores y clientes.

SERVIDOR MODBUS

El servidor Modbus TCP es un contenedor que debe estar en la misma red que los otros contenedores de Docker (para así ser monitorizado) y los clientes. En él debe estar instalado tanto Python como Pymodbus.

Obviamente, su función es la de servidor y va a ser objeto de atención ya que es el que mayormente sufre en su rendimiento según el número de clientes aumenta. La monitorización del servidor es crucial si se considera que esta caída de rendimiento puede ser dada tanto por el aumento de clientes como por el estrés introducido en las pruebas de rendimiento. Se debe considerar que en una red industrial real un servidor Modbus se representa a través de una pasarela Modbus (por ejemplo, Moxa Mgate 5111) o un PLC real que soporta Modbus (por ejemplo, Modicon M221).

CLIENTE MODBUS

Al igual que el servidor, el cliente Modbus es también un contenedor que tiene que estar en la red de monitorización definida en el Docker Compose y su función es hacer peticiones al servidor. El archivo que se ejecuta en este contenedor es el “performance.py”, cuyo diagrama se muestra en la Ilustración 5 y representa es un código escrito en Python en el que se crea un cliente Modbus TCP haciendo uso de la librería Pymodbus. que se encarga de introducir estrés en el servidor, midiendo en tiempo la latencia en que el servidor responde a un gran número de peticiones.



Ilustración 5 Diagrama de flujo de performance.py

ARQUITECTURA TLS

También se ha diseñado una arquitectura Modbus TLS mostrado en la Ilustración 6, partiendo de la base de la ya creada con Modbus TCP. El esquema de la arquitectura sería el mismo que en el de anterior caso, pero tanto los contenedores de servidor como los de clientes se modifican. En los servidores se deben generar los certificados necesarios, validarlos, y compartirlos con los clientes, para que éstos se puedan validar ante el servidor. Además de esto, también se debe editar el archivo “performance.py” para que tanto el cliente como el servidor trabajen con el protocolo TLS.

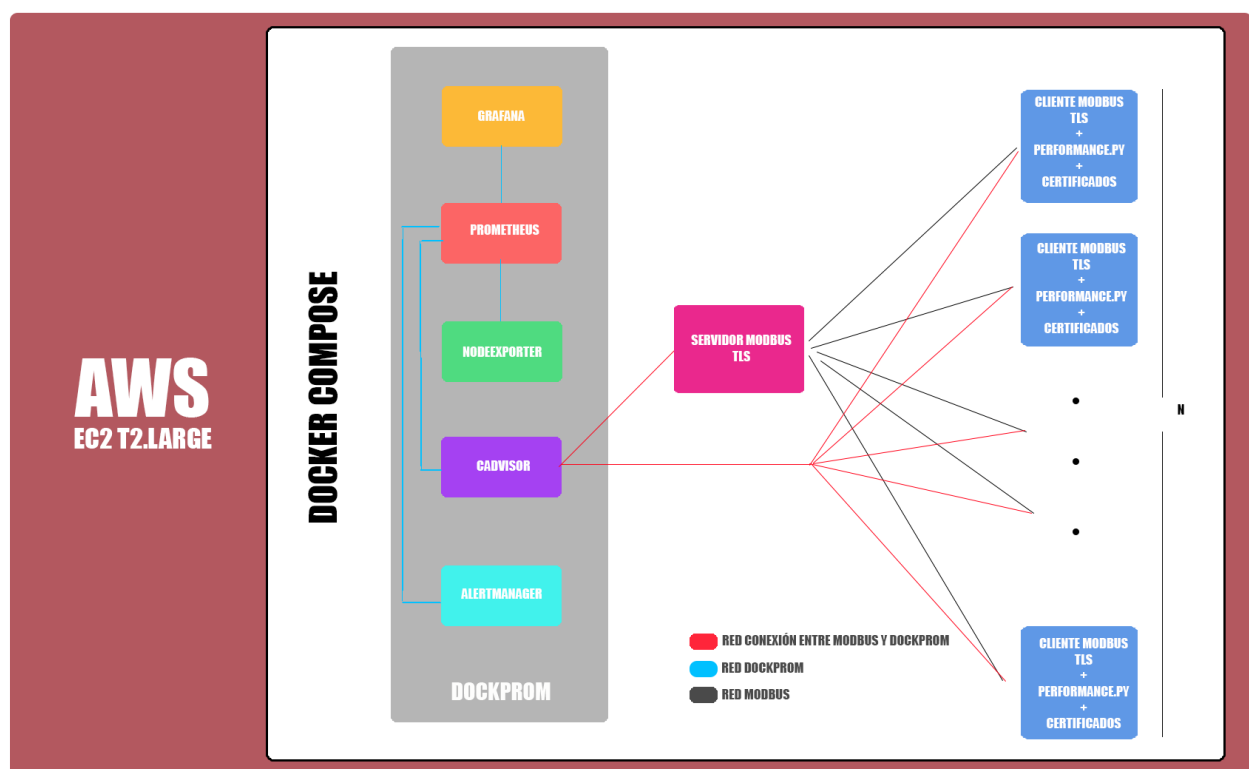


Ilustración 6 Arquitectura de sistema y red TLS

Esta arquitectura va a permitir probar el protocolo TLS sobre Modbus, de manera que se pueda determinar el rendimiento en comparación tanto a los dispositivos reales como a la arquitectura Modbus TCP ya establecida. Esta medición es deseable debido a que TLS representa una sobrecarga para todas las entidades cliente – servidor, que participan en la red, considerando los métodos criptográficos que introduce TLS.

Implementación

Una vez han quedado definidos los elementos que se han de crear para realizar las pruebas de rendimiento, se va a explicar cómo implementarlos y dar las recomendaciones necesarias.

SERVIDOR MODBUS TCP

Para crear el contenedor servidor Modbus (que está disponible en un repositorio de Dockerhub⁴, y por lo tanto ya no hay porque crearlo de nuevo) lo primero que se debe hacer es editar el código del servidor (synchronous_server.py) de ejemplo que hay en la librería Pymodbus [16]. Hemos de cambiar la línea que se muestra en la expresión (1) por la línea que se muestra en la expresión (2) para así aceptar todas las conexiones entrantes.

```
StartTcpServer(context, identity=identity, address=("localhost",  
5020)) (1)
```

```
StartTcpServer(context, identity=identity, address=("0.0.0.0",  
5020)) (2)
```

Por otro lado, creamos un archivo de texto sin ninguna extensión en el que se llame “DockerFile” (ecuación (3)) y escribimos lo siguiente:

```
FROM ubuntu (3)  
RUN apt-get update -y  
RUN apt-get install -y python3.7 \  
&& apt-get install -y python3-pymodbus  
  
ADD synchronous_server.py synchronous_server.py  
CMD ["python3", "synchronous_server.py"]
```

⁴ 19mikel95/pymodmikel:autoserversynchub

Este archivo es para crear la imagen del servidor Modbus TCP, partimos de la imagen Ubuntu, procedemos a actualizarlo, instalamos Python3.7 y Pymodbus. Con esta base añadimos el archivo que habíamos editado y lo guardamos en la imagen con el mismo nombre, para más tarde ejecutarlo en la terminal de Linux o bien en el Command Prompt de Windows.

De esta manera ya tenemos todo listo para montar la imagen desde la consola Windows de la siguiente manera (expresión 4):

```
docker build -t nombredelaimagen D:\Documentos\DocckerFiles (4)
```

Siendo “D:\Documentos\DocckerFiles” la ruta donde se encuentre el archivo DockerFile.

CLIENTE MODBUS TCP

Al igual que con el servidor, para la creación de la imagen del cliente TCP (disponible en Dockerhub⁵) hay que editar en este caso, el archivo performance.py de la librería Pymodbus [9]. Cambiamos la expresión (5) por la expresión (6), debido a que Docker resuelve el nombre del servicio escrito en el Docker Compose (que se enseñará más tarde) como si fuese una dirección, es decir, como un DNS.

```
host = '127.0.0.1' (5)
```

```
host = 'autoserver' (6)
```

También comentamos los logger que dan información extra que no interesa en este proyecto y cerramos la conexión del cliente cliente, como muestra la expresión (7):

⁵ 19mikel95/pymodmikel:autoclientimprove


```
#logger = log_to_stderr()
#logger.setLevel(logging.DEBUG)
#logger.debug("starting worker: %d" % os.getpid())

try:
    count = 0

    #client = ModbusTlsClient(host='autoserver', port=8020, sslctx=None)
    client = ModbusTcpClient(host, port=5020) #TCP
    #client = ModbusSerialClient(method="rtu",
                                # port="/dev/tty0", baudrate=9600)

    while count < cycles:
        with _thread_lock:
            client.read_holding_registers(10, 1, unit=1)
            count += 1

        client.close()
except:
    logger.exception("failed to run test successfully")
#logger.debug("finished worker: %d" % os.getpid())
```

Por último, creamos un ciclo infinito para que se sigan haciendo pruebas indefinidamente y pausando 1 segundo por cada ciclo como muestra la expresión (8).

```
if __name__ == "__main__":  
    args = (host, int(cycles * 1.0 / workers))  
    logger = log_to_stderr()  
    logger.setLevel(logging.DEBUG)  
    logger.propagate = False  
    while True:  
        time.sleep(1)  
        procs = [Worker(target=single_client_test, args=args)  
                 for _ in range(workers)]  
        tic= time.perf_counter()  
        any(p.start() for p in procs) # start the workers  
        any(p.join() for p in procs) # wait for the workers to finish  
        toc= time.perf_counter()  
        logger.debug("%d requests/second" % ((1.0 * cycles) / (toc - tic)))  
        logger.debug("time taken to complete %s cycle by "  
                     "%s workers is %s seconds" % (cycles, workers, toc-tic))
```

(8)

Tras estos cambios procedemos a la construcción del DockerFile, como muestra la expresión (9).

```
FROM ubuntu (9)
RUN apt-get update -y
RUN apt-get install -y python3.7 \
&& apt-get install -y python3-pymodbus

ADD performance.py performance.py
CMD ["python3", "performance.py"]
```

Y ahora sí, al igual que con el servidor construimos la imagen, como muestra la expresión (10).

```
docker build -t nombredelaimagen D:\Documentos\DocckerFiles (10)
```

SERVIDOR MODBUS TLS

Para la imagen del servidor Modbus (disponible en Dockerhub⁶), tendremos que generar certificados para más tarde compartirlos con los clientes para que estos se puedan certificar ante el servidor. Para mantener los certificados, al mismo tiempo que despliega el comando para lanzar los contenedores de forma automática, habrá que construir 2 veces la imagen.

Comenzamos por editar el archivo `synchronous_server.py` que hemos utilizado anteriormente, para cambiar el cliente de TCP a TLS, es decir, de la expresión (11) a la expresión (12).

```
StartTcpServer(context, identity=identity, address=("0.0.0.0", 5020)) (11)
```

```
StartTlsServer(context, identity=identity, address=("0.0.0.0", 8020), sslctx=None, certfile="serv.crt", keyfile="serv.key") (12)
```

⁶ 19mikel95/pymodmikel:autoservtls

Como vemos, se utilizan los archivos “serv.crt” y “serv.key” pero para ello hay que generarlos, de ahí que sea necesario crear un primer DockerFile como el de la expresión (13).

```
FROM ubuntu (13)
RUN apt-get update -y
RUN apt-get install -y python3.7 \
&& apt-get install -y python3-pymodbus
RUN apt-get install -y openssl
RUN apt-get install -y ca-certificates

ADD tlsserver.py tlsserver.py
```

A continuación, se construye la imagen como muestra la expresión (14).

```
docker build -t pruebacertificados D:\Documentos\DockerFiles (14)
```

En este caso el nombre dado a la imagen (“pruebacertificados”) es importante, ya que se parte de esta imagen para la construcción de la imagen final.

Ahora procedemos a la generación de los certificados del servidor, lanzando de la siguiente manera la imagen (expresión 15):

```
docker run -ti pruebacertificados (15)
```

Una vez dentro de la imagen ejecutamos el siguiente comando para generar el archivo serv.key. la clave privada expresión (16):

```
openssl genrsa 2048>serv.key (16)
```

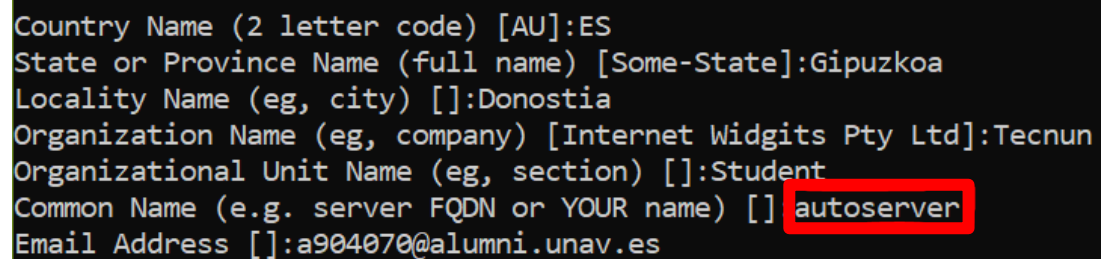
Y procedemos a darle permisos como muestra la expresión (17).

```
chmod 400 serv.key (17)
```

Ahora se genera el certificado (expresión 18) que será compartido con los clientes.

```
openssl req -new -x509 -nodes -sha256 -days 365 -key serv.key -out  
serv.crt (18)
```

Al ejecutar este comando se nos pedirá información a rellenar, donde hay que hacer hincapié en que “Common name” debe ser la dirección a nuestro servidor, y dado que esto se ejecutará en un Docker Compose como ya hemos explicado antes, este debe ser el nombre del servicio del contenedor de nuestro servidor, en nuestro caso “autoserver”, como se muestra en la Ilustración 7.



```
Country Name (2 letter code) [AU]:ES  
State or Province Name (full name) [Some-State]:Gipuzkoa  
Locality Name (eg, city) []:Donostia  
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Tecnun  
Organizational Unit Name (eg, section) []:Student  
Common Name (e.g. server FQDN or YOUR name) [] autoserver  
Email Address []:a904070@alumni.unav.es
```

Ilustración 7 Generación de los certificados

Por un lado, copiamos y guardamos el archivo generado “serv.crt” fuera de la imagen de Docker y por el otro, dentro de la imagen, lo movemos a la ruta mostrada en la expresión (19), la cual representa la instancia por defecto del sistema operativo para almacenar certificados.

```
sudo cp serv.crt /usr/local/share/ca-certificates/ (19)
```

Le damos los permisos al archivo:

```
sudo chmod +x ./serv.crt (20)
```

Finalmente procedemos a actualizar el default CA Store para que reconozca el nuevo certificado agregado, como muestra la expresión (21)

```
sudo update-ca-certificates (21)
```

Ahora ya guardar y cerrar la imagen para crear la imagen definitiva donde el archivo Python se ejecuta automáticamente. El DockerFile sería como muestra la expresión (22).

```
FROM pruebacertificados (22)

RUN apt-get update -y

CMD ["python3", "tlsserver.py"]
```

A continuación, como muestra la expresión (23), se construye la expresión final.

```
docker build -t nombredelaimagen D:\Documentos\DocckerFiles (23)
```

CLIENTE MODBUS TLS

Ahora se procede a crear la imagen del cliente Modbus TLS (disponible en Dockerhub⁷). En este caso, el archivo Python performance.py, sería exactamente igual que en el caso del cliente TCP con los dos únicos cambios de, importar la clase TLS como muestra la expresión (24).

```
from pymodbus.client.sync import ModbusTlsClient (24)
```

Adicionalmente, se cambia la línea de la expresión (25) por la expresión (26).

⁷ 19mikel95/pymodmikel:autoawsclient

```
client = ModbusTcpClient(host, port=5020) #TCP (25)
```

```
client = ModbusTlsClient(host='autoserver', port=8020, sslctx=None) (27)
```

A continuación, se procede a añadir el archivo del certificado previamente generado y actualizarlo, pero lo haremos en el propio DockerFile, como muestra la expresión (28).

```
FROM ubuntu (28)
RUN apt-get update -y
RUN apt-get install -y python3.7 \
&& apt-get install -y python3-pymodbus
RUN apt-get install -y openssl
RUN apt-get install -y ca-certificates

ADD performance.py performance.py
ADD serv.crt /usr/local/share/ca-certificates/serv.crt
CMD ["chmod777", "/usr/local/share/ca-certificates/serv.crt "]
CMD ["sudo update-ca-certificates"]
CMD ["python3", "performance.py"]
```

Finalmente, se construye la imagen, acorde a la expresión (29).

```
docker build -t nombredelaimagen D:\Documentos\DocckerFiles (29)
```

DOCKER-COMPOSE

Si se quiere el archivo “docker-compose.yml” de forma directa, el archivo con el Docker Compose completo con una red Modbus TCP con 32 clientes ya incluidos está disponible en Google Drive⁸. Por otro lado, el archivo para la red MODBUS TLS con 32 clientes, también está disponible en Google Drive⁹. A estos archivos, tan sólo hay que cambiarle el nombre y extensión a “docker-compose.yml”.

En la implementación del archivo “docker-compose.yml” tomaremos de base el archivo “docker-compose.yml” del proyecto Dockprom de Stefan Prodan[12]. Se empieza por definir la versión del Docker Compose junto con las redes y los volúmenes que se van a utilizar como muestra la expresión (30):

```
version: '2.1'
networks:
  monitor-net:
    driver: bridge
volumes:
  prometheus_data: {}
  grafana_data: {}
```

(30)

La versión 2.1 viene definida por Dockprom aunque si a futuro esta se actualiza a la versión 3, sería un paso importante ya que facilitaría la migración de este mismo proyecto hacia Kubernetes, de lo cual ya se hablará en el apartado Conclusiones.

Consecutivamente a la versión, se define una red de nombre “monitor-red” de tipo bridge. Será en esta red, en la que deberemos incluir todos los contenedores que deseemos monitorizar, como los servidores y clientes Modbus. El driver debe ser “bridge” si estamos ejecutando el sistema en AWS aunque si aún se están haciendo pruebas en un ordenador local, se podría utilizar el modo “host” con los consecutivos cambios en los

⁸

https://drive.google.com/file/d/19zb_UmMXjtjBj_rVStC5OPpMXpXDsbZ5/view?usp=sharing

⁹

<https://drive.google.com/file/d/1ZI6HCmXhOfnCWBtXqPvugnFGYNMjWyiJ/view?usp=sharing>

contenedores de los clientes (cambiar la dirección “autoserver” por “localhost”).

Para terminar con la configuración general del Docker Compose, definimos los volúmenes que se van a montar, los cuales son “prometheus_data” y “grafana_data” en los que van archivos de configuración para dichos softwares. Por lo tanto, el archivo “docker-

```
services: (31)

  prometheus:

    image: prom/prometheus:latest

    container_name: prometheus

    volumes:

      - ./prometheus:/etc/prometheus

      - prometheus_data:/prometheus

    command:

      - '--config.file=/etc/prometheus/prometheus.yml'

      - '--storage.tsdb.path=/prometheus'

      - '--web.console.libraries=/etc/prometheus/console_libraries'

      - '--web.console.templates=/etc/prometheus/consoles'

      - '--storage.tsdb.retention.time=200h'

      - '--web.enable-lifecycle'

    restart: unless-stopped

    expose:

      - 9090

    networks:

      - monitor-net

    labels:

      org.label-schema.group: "monitoring"
```

compose.yml” debe estar siempre en la carpeta “dockprom” junto a estas 2 carpetas.

Definimos los servicios, empezando por Prometheus en la expresión (31):

En cada servicio o contenedor se definen: la imagen (de que repositorio de Dockerhub y que versión), el nombre, el cual no es necesario pues puede tomar por defecto el de la imagen, volúmenes para añadir archivos en el caso de que sea necesario, comandos que se ejecutan en el inicio del contenedor, los puertos a exponer y la red a la que pertenecen, como podemos ver en el siguiente contenedor que es AlertManager en la expresión (32):

```
alertmanager: (32)

  image: prom/alertmanager:latest

  container_name: alertmanager

  volumes:

    - ./alertmanager:/etc/alertmanager

  command:

    - '--config.file=/etc/alertmanager/config.yml'

    - '--storage.path=/alertmanager'

  restart: unless-stopped

  expose:

    - 9093

  networks:

    - monitor-net

  labels:

    org.label-schema.group: "monitoring"
```

En el servicio de NodeExporter, al igual que en el de cAdvisor hay una línea en la que se incluye un volumen (expresión (33)):

`- /:/rootfs:ro` **(33)**

Este volumen, puede ocasionar problemas en el caso de que la

`- c:\:/rootfs:ro` **(34)**

máquina huésped sobre la que se ejecuta el Docker Compose sea Windows, para solucionarlos basta con sustituirlo por la expresión (34) para que encuentre la ruta al volumen:

Otro problema que puede surgir, en este caso para máquinas que no sean MacOS se da con el contenedor de cAdvisor, para solucionarlo hay que descomentar una línea (expresión (35)) que viene comentada en el “docker-compose.yml” original de Dockprom:

Estos son los cambios que eran necesarios hacer para el funcionamiento de Dockprom en nuestra máquina Amazon Linux, con lo que los contenedores restantes quedarían de la siguiente forma:

`- /cgroup:/sys/fs/cgroup:ro` **(35)**

- Node Exporter-Expresión (36)
- cAdvisor-Expresión (37)
- PushGateway-Expresion (38)
- Grafana-Expresión (39)
- Caddy-Expresión (40)

nodeexporter:

(36)

image: prom/node-exporter:latest

container_name: nodeexporter

volumes:

- /proc:/host/proc:ro

- /sys:/host/sys:ro

- /:/rootfs:ro

command:

- '--path.procfs=/host/proc'

- '--path.rootfs=/rootfs'

- '--path.sysfs=/host/sys'

- '--collector.filesystem.ignored-mount-points=^/(sys | proc | dev | host | etc)(\$\$ | /)'

restart: unless-stopped

expose:

- 9100

networks:

- monitor-net

labels:

org.label-schema.group: "monitoring"

cadvisor: (37)

image: gcr.io/google-containers/cadvisor:latest

container_name: cadvisor

volumes:

- /:/rootfs:ro
- /var/run:/var/run:rw
- /sys:/sys:ro
- /var/lib/docker:/var/lib/docker:ro
- /cgroup:/sys/fs/cgroup:ro

restart: unless-stopped

expose:

- 8080

networks:

- monitor-net

labels:

org.label-schema.group: "monitoring"

pushgateway: (38)

image: prom/pushgateway:latest

container_name: pushgateway

restart: unless-stopped

expose:

- 9091

networks:

- monitor-net

labels:

org.label-schema.group: "monitoring"

grafana: (39)

image: grafana/grafana:latest

container_name: grafana

volumes:

- grafana_data:/var/lib/grafana
- ./grafana/provisioning:/etc/grafana/provisioning

environment:

- GF_SECURITY_ADMIN_USER=\${ADMIN_USER:-admin}
- GF_SECURITY_ADMIN_PASSWORD=\${ADMIN_PASSWORD:-admin}
- GF_USERS_ALLOW_SIGN_UP=false

restart: unless-stopped

expose:

- 3000

networks:

- monitor-net

labels:

org.label-schema.group: "monitoring"

```
caddy: (40)

  image: stefanprodan/caddy

  container_name: caddy

  ports:

    - "3000:3000"

    - "9090:9090"

    - "9093:9093"

    - "9091:9091"

  volumes:

    - ./caddy:/etc/caddy

  environment:

    - ADMIN_USER=${ADMIN_USER:-admin}

    - ADMIN_PASSWORD=${ADMIN_PASSWORD:-admin}

  restart: unless-stopped

  networks:

    - monitor-net

  labels:

    org.label-schema.group: "monitoring"
```

Con estos contenedores, podríamos lanzar ya el Docker Compose aunque solo podríamos monitorizar los propio sistema de monitorización y la máquina huésped. Tenemos que añadir aún tanto el contenedor del servidor como al menos un cliente.

Para añadir el servidor, al cual hemos llamado “autoserver” tan solo hay que añadir el siguiente servicio, mostrado en la expresión (41):

```

autoserver: (41)
  image: 19mikel95/pymodmikel: autoserversynchub
  container_name: autoserver
  command: sh -c "
    sed -i 's|localhost|0.0.0.0|g' synchronous_client.py;
    python3 synchronous_client.py daemon off
  "
  ports:
    - "8020:8020"
  restart: unless-stopped
  networks:
    - monitor-net
  mem_limit: 256m

```

El nombre de este servicio debe ser igual al que hayamos puesto como dirección del servidor en el archivo “performance.py” de los clientes. Además de esto se agrega un comando para cambiar la dirección de “localhost” a “0.0.0.0” para aceptar todas las conexiones. Se incluye en la red “monitor-net” para que sea posible su monitorización por Dockprom y finalmente se limita la memoria que va a tener el contenedor a 256 MB tomando como referencia el dispositivo Controlador IIOT Modicon M262[17], el cual tiene 192MB de RAM pero hay que tener en cuenta que aunque el contenedor sea extremadamente eficiente, tiene más procesos que el que tendría dicho dispositivo.

Por otro lado, hay que añadir los clientes, y aquí está una de las claves de este proyecto, y es que se pueden añadir tantos clientes como se quiera con solo copiar y pegar en el archivo “docker-compose.yml” tantas veces como queramos, el servicio “clientperf”. Luego se puede añadir un número al nombre para la identificación del cliente, al igual que muestra la expresión (42):


```
clientperfl: (42)
  image: 19mikel95/pymodmikel: autoclientimprove
  container_name: clientperfl
  restart: unless-stopped
  networks:
    - monitor-net
  depends_on:
    - autoserver
  environment:
    - AUTO_SERVER_HOST=autoserver
```

Para la implementación del Docker Compose del sistema Modbus TLS, solo hay que cambiar 2 líneas en todo el archivo, una en el servidor y otra en el cliente. Las líneas de imagen para que descargue los contenedores que habíamos creado anteriormente para la conexión TLS.

Y dichos servicios quedarían como en la expresión (43) para el servidor y la expresión (44) para el cliente:

autoserver: (43)

```
image: 19mikel95/pymodmikel:autoservtls
container_name: autoserver
command: sh -c "
    sed -i 's|localhost|0.0.0.0|g' synchronous_client.py;
    python3 synchronous_client.py daemon off
"
ports:
  - "8020:8020"
restart: unless-stopped
networks:
  - monitor-net
mem_limit: 256m
```

clientperfl: (44)

```
image: 19mikel95/pymodmikel:autoawsclient
container_name: clientperfl
restart: unless-stopped
networks:
  - monitor-net
depends_on:
  - autoserver
environment:
  - AUTO_SERVER_HOST=autoserver
```

PUESTA EN MARCHA EN LA INSTANCIA AWS EC2

Creación de la Instancia

Para lanzar una instancia EC2, lo primero hay que contar con una cuenta de AWS con crédito para ello puede ser recomendable aplicar para una cuenta de estudiante o educador ¹⁰.

Una vez hayamos iniciado sesión, hay que dirigirse a la consola de Amazon EC2 en la cual seleccionamos en el panel EC2 y lanzamos (“Launch Instance”) tal y como se ve en la Ilustración 8.

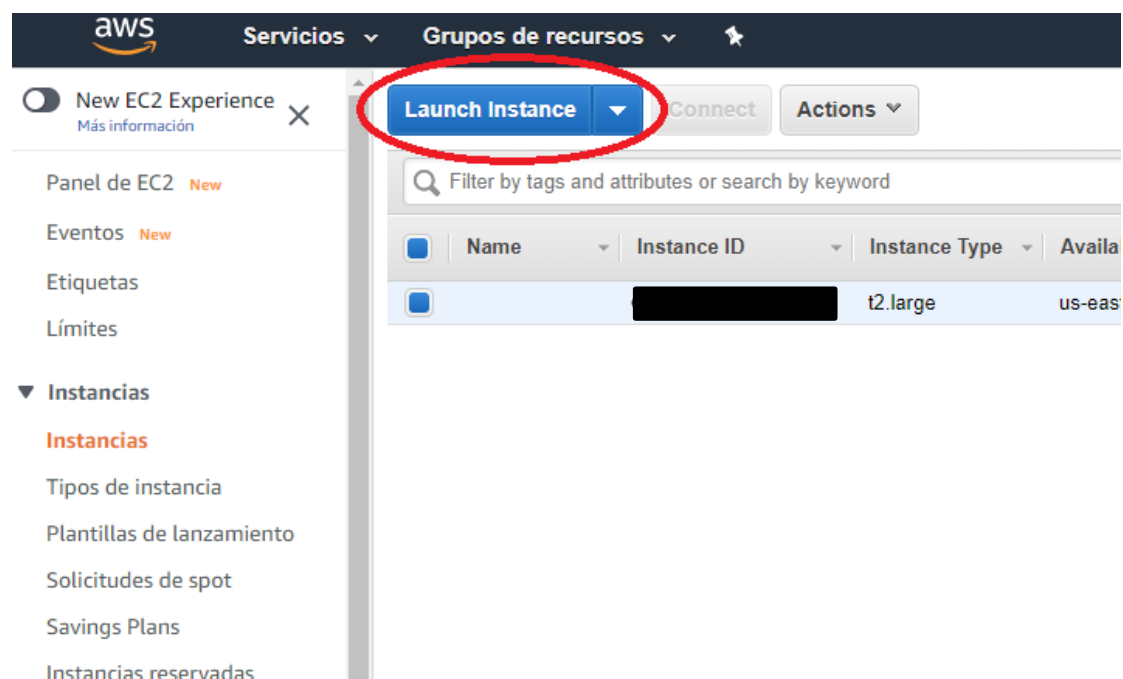


Ilustración 8 Lanzar una instancia

Esto nos llevará al próximo paso que es la selección de la imagen de la máquina, es decir, el sistema operativo que ejecutará nuestro “ordenador en la nube”. Se recomienda usar la Amazon Linux AMI 2018.03.0 (HVM), como se ve en la Ilustración 9, porque es la que se ha usado en el desarrollo de este proyecto, por tener paquetes como Python ya instalados, y aunque teóricamente se pueda ejecutar en otras AMIs puede que haya que hacer algún paso distinto.

¹⁰ <https://aws.amazon.com/es/education/awseducate/>

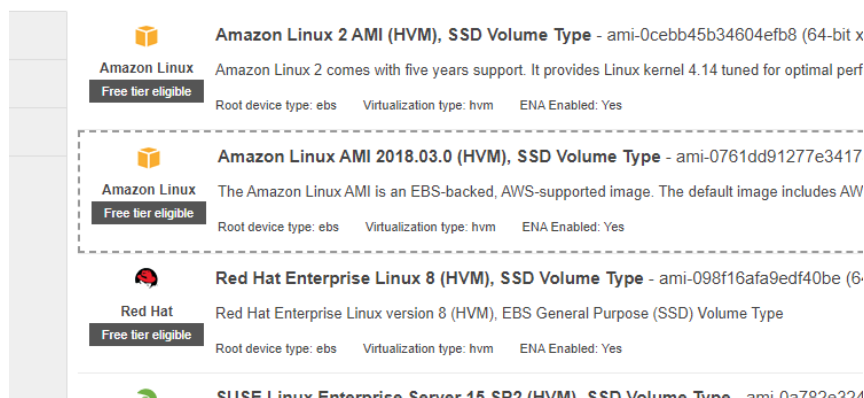


Ilustración 9 Selección de AMI

El siguiente paso es la selección del tipo de instancia, como ya se ha dicho anteriormente, nosotros empezamos con una t2.micro la cual resultó no ser suficiente, y finalmente optamos por una t2.large con 2 CPUs y 8GB de memoria. Veremos una tabla similar a la Tabla 1 mostrada en el apartado de diseño solo que aparecerán muchos más tipos de instancias además de las T2. Sin embargo, es recomendable usar las T2 ya que son las instancias que se centran en el equilibrio de recursos mientras que los demás tipos suele tener ciertas especialidades como para redes neuronales.

Consecutivamente, en vez de seleccionar “Review and Launch” seleccionamos “Next: Configure Instance Details”, como se muestra en la Ilustración 10, y proseguimos a través del botón “Next” hasta llegar al apartado “Configure Security Groups” donde se configura la apertura de puertos.

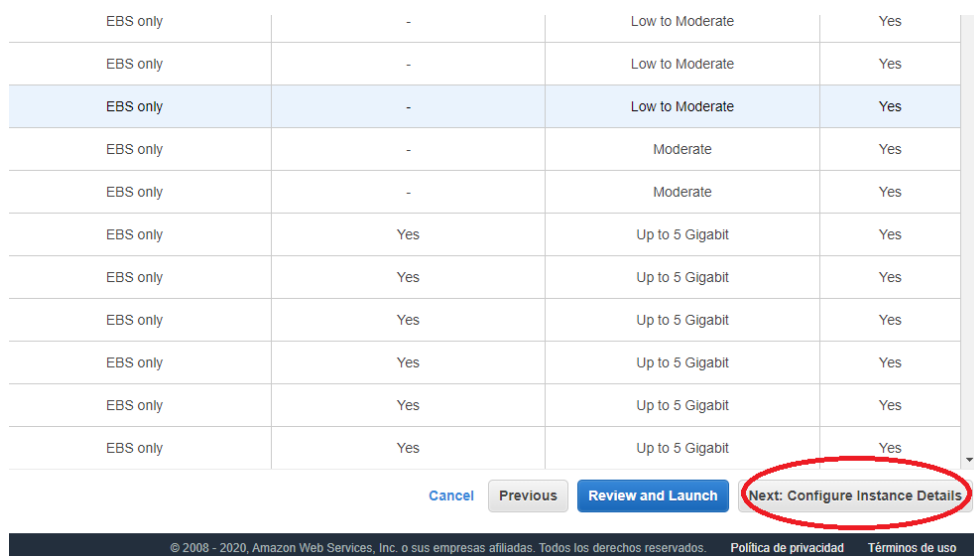
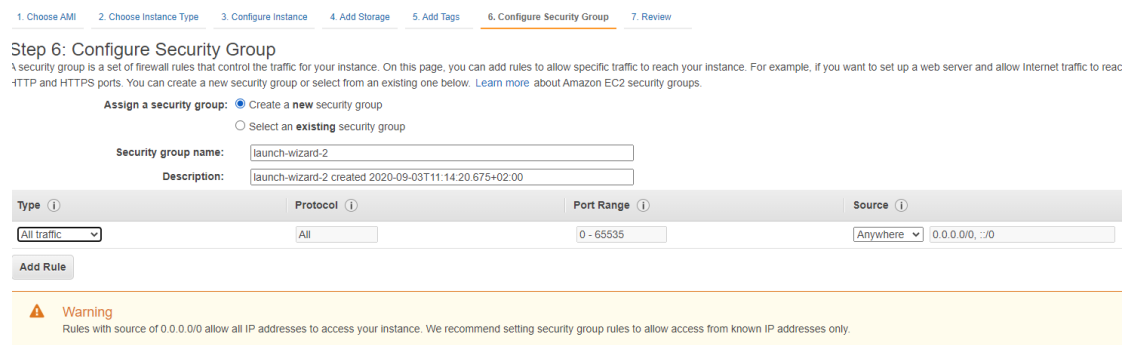


Ilustración 10 Avanzar en la configuración

Cuando lleguemos al apartado de la configuración de la seguridad debemos asegurarnos de abrir por lo menos los puertos TCP utilizados por las herramientas de Dockprom lo cuales son:

- 3000 para Grafana
- 8080 para cAdvisor
- 9100 para NodeExporter
- 9093 para AlertManager
- 9090 para Prometheus
- 9091 para Caddy

Un procedimiento más simple pero menos recomendado desde el punto de vista de seguridad consiste en habilitar todos los puertos como se muestra en la Ilustración 11.



1. Choose AMI 2. Choose Instance Type 3. Configure Instance 4. Add Storage 5. Add Tags 6. Configure Security Group 7. Review

Step 6: Configure Security Group

A security group is a set of firewall rules that control the traffic for your instance. On this page, you can add rules to allow specific traffic to reach your instance. For example, if you want to set up a web server and allow Internet traffic to reach HTTP and HTTPS ports. You can create a new security group or select from an existing one below. [Learn more](#) about Amazon EC2 security groups.

Assign a security group: ☒ Create a new security group ☐ Select an existing security group

Security group name:

Description:

Type	Protocol	Port Range	Source
All traffic	All	0 - 65535	Anywhere (0.0.0.0/0, :::0)

[Add Rule](#)

Warning
Rules with source of 0.0.0.0/0 allow all IP addresses to access your instance. We recommend setting security group rules to allow access from known IP addresses only.

Ilustración 11 Apertura de todo el tráfico

Ahora sí, procedemos a seleccionar “Review and Launch” y tras revisar que todo esté correcto confirmamos el lanzamiento de la instancia. A continuación, se nos pedirá crear un par de claves para conectarnos a nuestra instancia de una forma segura (Ilustración 12), por lo que le ponemos el nombre que queramos y descargarla, y guardarla en un lugar accesible pues se necesita como ya hemos dicho para conectarnos.

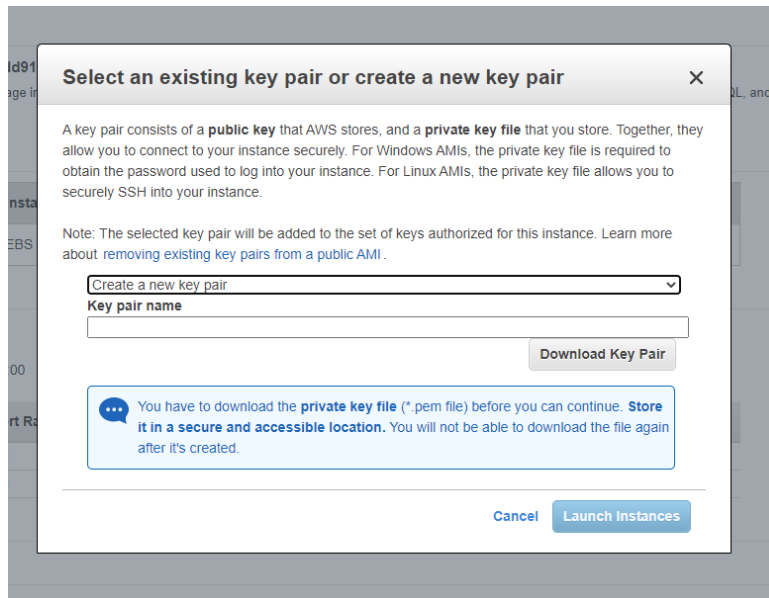


Ilustración 12 Generación de par de claves

Después de este procedimiento, nuestra instancia ya empezará a iniciarse y para conectarnos a ella podemos usar el botón “Ver instancias” o volver a la consola y en el panel EC2 seleccionar “Instancias en ejecución”. Hacemos clic derecho sobre ella y seleccionamos “Connect” tras lo cual se abrirá el cuadro de la Ilustración 13.

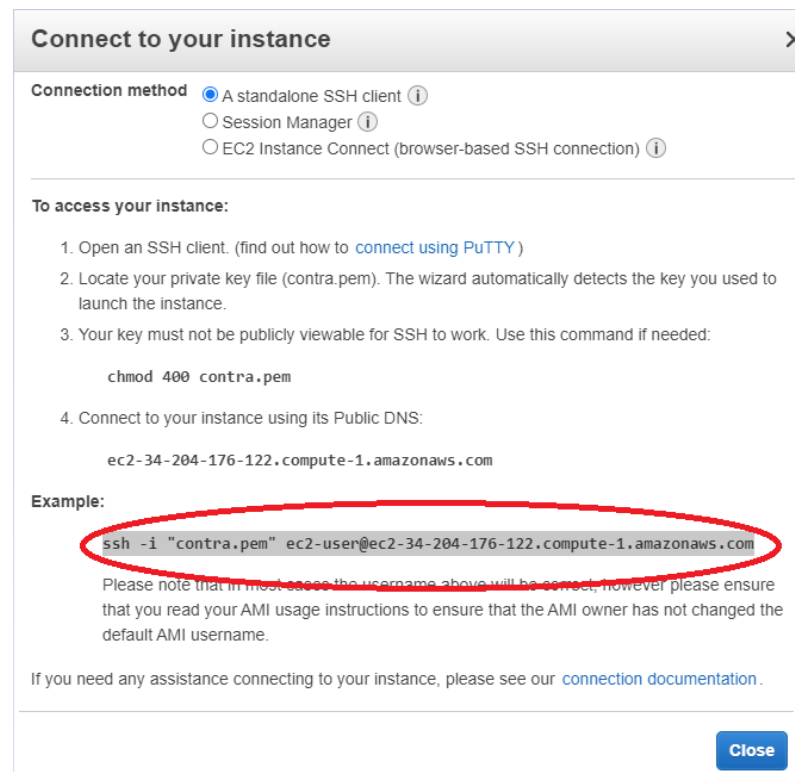


Ilustración 13 Conectarse a la instancia

Configuración de la máquina y ejecución

Una vez desplegada la instancia se procede a la configuración e instalación de los paquetes necesarios para la ejecución del sistema, por lo que copiamos el comando que aparece en “Example” en la Ilustración 13. Conectarse a la instancia y desde una consola en nuestro ordenador lanzamos el comando indicado. Nuestro sistema nos preguntará si estamos seguros de la conexión generada, por lo que aceptamos, escribiendo “yes”, tras lo cual, ya estaremos conectados a nuestra instancia y tenemos una máquina Amazon Linux t2.large a nuestra disposición.

Ejecutamos el comando (expresión (45)) que nos recomienda la propia máquina y procedemos a la actualización del sistema operativo virtual.

```
sudo yum update -y (45)
```

A continuación, se instala Docker a partir del comando de la expresión (46):

```
sudo yum install docker (46)
```

La expresión (47) permite lanzar el servicio Docker y con este el Docker Engine.

```
sudo service docker start (47)
```

Para evitar usar sudo para los comandos Docker, se ejecuta el comando de la expresión (48).

```
sudo usermod -a -G docker ec2-user (48)
```

Finalmente, validamos “docker.sock” ejecutando el comando de la expresión (49).

```
sudo chmod 666 /var/run/docker.sock (49)
```

Ahora que Docker ha sido instalado satisfactoriamente, tenemos que instalar Dockprom¹¹. Para lo cual primero instalamos Git, como muestra la expresión (50).

¹¹ <https://github.com/stefanprodan/dockprom>

```
sudo yum install git (50)
```

A continuación, se descarga Dockprom utilizando la expresión (51).

```
sudo git clone https://github.com/stefanprodan/dockprom (51)
```

Llegados a este punto se accede a la carpeta de Dockprom, a través del comando de la expresión (52).

```
cd dockprom (52)
```

A continuación, se edita el docker-compose.yml usando el comando nano (expresión (53)).

```
nano docker-compose.yml (53)
```

Se procede a sustituir el Docker Compose por defecto de Dockprom por el Docker Compose que nosotros hemos creado y con el número de clientes que queramos. Lo guardamos presionando Ctrl+O, Ctrl+T y seleccionamos “docker-compose.yml”, para salir del editor pulsamos Ctrl+X.

Solo queda un paso antes de lanzar el Docker Compose y empezar a poder realizar los experimentos, que no es otro que la instalación de Docker Compose, como muestran las expresiones (54, 55, 56)

```
sudo curl -L https://github.com/docker/compose/releases/download/1.21.0/docker-  
compose`uname -s`-`uname -m` | sudo tee /usr/local/bin/docker-compose  
> /dev/null (54)
```

```
sudo chmod +x /usr/local/bin/docker-compose (55)
```

```
sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose (56)
```


Tras esto ya estamos listos para ejecutar el Docker Compose a partir de la expresión (57), con lo que inicia la descarga de las imágenes y el despliegue de los contenedores.

`docker-compose up -d (57)`

Monitorización de la información

Como se ha mencionado las entidades que forman Dockprom se encargan de la monitorización de las métricas referentes a la arquitectura de red, para lo cual Grafana es la herramienta más útil para la visualización y captura de dicha información. Adicionalmente, la segunda fuente de información son los logs del archivo `performance.py` de los clientes, que entrega información útil acerca, por ejemplo, de conexiones entrantes.

Para acceder a Grafana necesitamos la IP pública o el nombre de dominio de nuestra instancia, las cuales podemos obtener en el panel EC2 de AWS, seleccionando nuestra instancia aparecerá entonces en la descripción tal y como se enseña en la Ilustración 14.

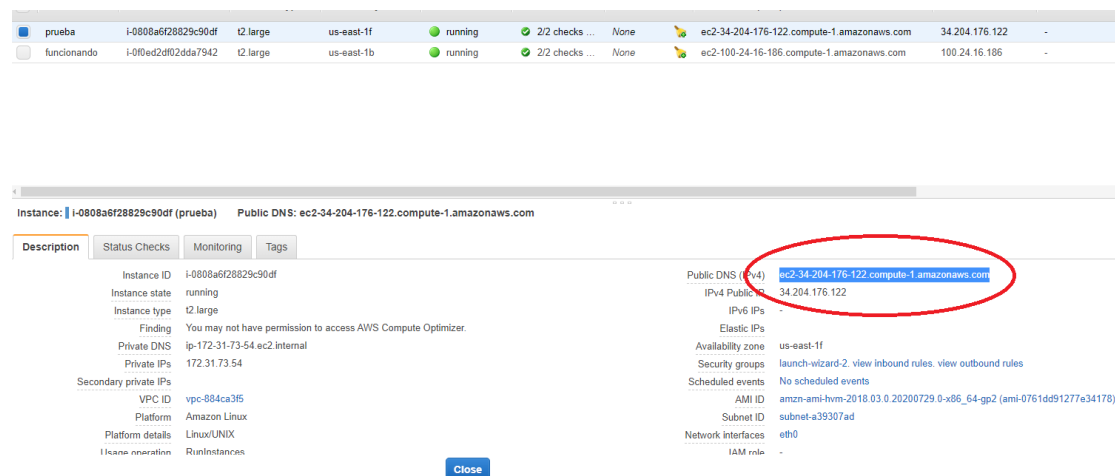


Ilustración 14 Nombre de Dominio e IP pública de la instancia.

Copiamos la dirección y la pegamos en cualquier navegador de la siguiente seguido de “:3000”, indicando que queremos acceder al puerto 3000. Con esto, ya habremos accedido a Grafana el cual nos pedirá usuario y contraseña los cuales son por defecto “admin” y “admin”. Considerando que el despliegue es público, se recomienda cambiar usuario y contraseña.

Una vez en el panel principal de Grafana pulsamos en “Dashboards” tal y como se ve en la Ilustración 15.

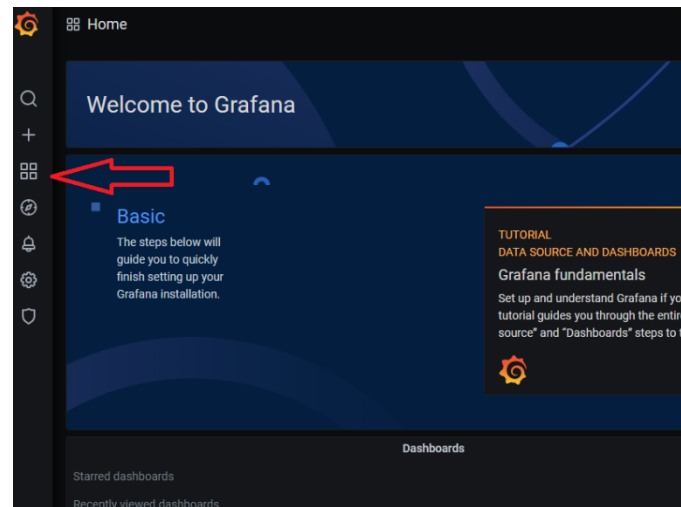


Ilustración 15 Panel Grafana

En Dashboards, seleccionamos Manage lo cual nos llevará al menú de selección de los distintos Dashboard, mostrados en la Ilustración 16. Estos esquemas ya vienen por defecto en la herramienta Dockprom.

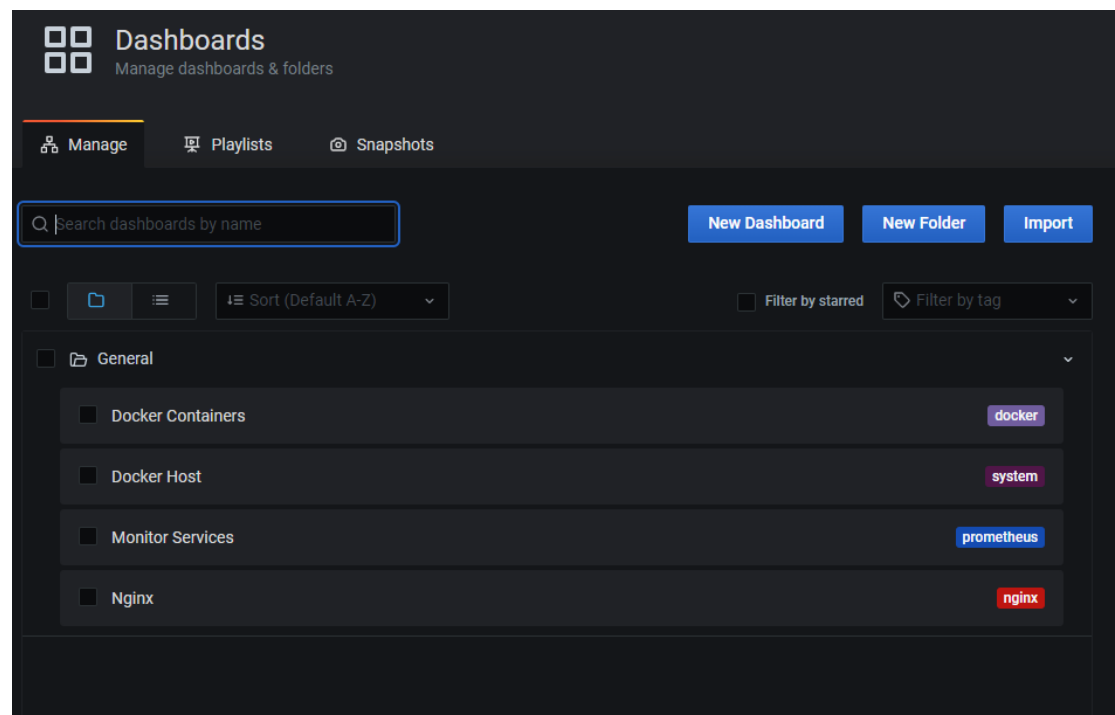


Ilustración 16 Dashboards de Grafana

- Docker Host nos muestra los datos y rendimiento de la máquina EC2 que hayamos seleccionado en Amazon.
- Monitor Services nos muestra la carga de los contenedores del propio sistema de monitorización de Dockprom.

- Nginx en nuestro caso no muestra nada, debido a que no usamos ese servicio y no ha sido configurado.
- Docker Containers es donde está la información que realmente nos interesa pues muestra todos los otros contenedores que no sean de Dockprom.

La Ilustración 17 muestra el “template” que Dockprom provee para analizar Docker Containers.



Ilustración 17 Dashboard de Contenedores Docker

Adicionalmente, para ver las peticiones por segundo que responde el servidor Modbus basta con ejecutar el comando de la expresión (58) en la consola del sistema operativo a partir de la cual nos conectamos a la instancia de AWS.

`docker logs clientperf1` (58)

En el caso de tener más clientes añadidos simplemente cambiamos el número de clientperf al que queramos ver, como en la Ilustración 18.

```

ec2-user@ip-172-31-35-212 dockprom]$ docker logs clientperf5
DEBUG/MainProcess] 99 requests/second
DEBUG/MainProcess] time taken to complete 1000 cycle by 10 workers is 10.005705252000098 seconds
DEBUG/MainProcess] 135 requests/second
DEBUG/MainProcess] time taken to complete 1000 cycle by 10 workers is 7.406405433000145 seconds
DEBUG/MainProcess] 132 requests/second
DEBUG/MainProcess] time taken to complete 1000 cycle by 10 workers is 7.523753128999942 seconds
DEBUG/MainProcess] 148 requests/second
DEBUG/MainProcess] time taken to complete 1000 cycle by 10 workers is 6.7253308109998216 seconds
DEBUG/MainProcess] 134 requests/second
DEBUG/MainProcess] time taken to complete 1000 cycle by 10 workers is 7.407506534000277 seconds
DEBUG/MainProcess] 131 requests/second
DEBUG/MainProcess] time taken to complete 1000 cycle by 10 workers is 7.584014381000088 seconds
DEBUG/MainProcess] 123 requests/second
DEBUG/MainProcess] time taken to complete 1000 cycle by 10 workers is 8.120096446999924 seconds
DEBUG/MainProcess] 145 requests/second
DEBUG/MainProcess] time taken to complete 1000 cycle by 10 workers is 6.88007739700015 seconds
DEBUG/MainProcess] 130 requests/second
DEBUG/MainProcess] time taken to complete 1000 cycle by 10 workers is 7.691797364999729 seconds
DEBUG/MainProcess] 133 requests/second
DEBUG/MainProcess] time taken to complete 1000 cycle by 10 workers is 7.46409182699972 seconds
DEBUG/MainProcess] 141 requests/second

```

Ilustración 18 Logs de cliente

Descripción del proceso de experimentación

Como se ha mencionado anteriormente, uno de los objetivos es hacer pruebas de rendimiento y compararlo con dispositivos reales para ver la veracidad de la simulación. Para ello se han tomado como referencia dispositivos reales como el Controlador IIoT Modicon M262[17], el cual permite determinar que un dispositivos de estas presentaciones tiene alrededor de 256MB de memoria RAM (192MB en este caso concreto) y por otro lado determinamos el número máximo de clientes Modbus TCP que aceptan estos dispositivos, el cual, normalmente llega hasta 32 (varía entre 8 y 32), como es el caso del dispositivo de la serie MGate MB3170/3270 y como muestra su Datasheet[18]. Considerando la información anterior como referencia, a continuación, se describe el proceso de experimentación sobre nuestra red.

Dado la cantidad máxima de clientes soportada por un dispositivo real y que la arquitectura diseñada es de $n:1$, se procede a realizar experimentos variando el número de clientes a 1,4,8,16 y 32. Para ello se edita el Docker Compose en cada ocasión para aumentar en número de clientes. El tiempo de recolección de datos es de 30 minutos en cada caso, desde el lanzamiento del Docker-Compose, para finalmente comparar los resultados. En el caso de la red que soporta Modbus TLS también se procede de igual manera. A partir de esta simulación obtenemos dos tipos de resultados, por un lado, se recupera comportamiento de la simulación de nuestra red y dispositivos, usando los datos de Grafana y por otro, el número de peticiones por segundo usando los logs obtenidos de aplicar a los clientes el comando de la expresión (43).

Resultados

Los resultados de dichos experimentos se van a exponer en este apartado, diferenciando los resultados de la red Modbus TCP de los de la red Modbus TLS.

RED MODBUS TCP

Los resultados del rendimiento y carga del sistema mostrados por Grafana han mostrado como la carga de memoria de los clientes se mantiene estable sobre unos 10MB por cada cliente mientras que la memoria del servidor aumenta de forma lineal según el tiempo que lleve la red activa.

El valor inicial de la memoria requerida por el servidor es inicialmente igual o similar para todos los casos, sin embargo, la pendiente del crecimiento inicial varía según el número de clientes conectados al servidor. A mayor número de clientes, mayor es la velocidad con la que el servidor aumenta su uso de memoria. Esto puede apreciarse en las Ilustración 19, Ilustración 20, Ilustración 21, Ilustración 22, y, Ilustración 23 o resumido en la Tabla 2.



Ilustración 19 Resultados de Grafana con 1 cliente TCP



Ilustración 20 Resultados de Grafana con 4 clientes TCP



Ilustración 22 Resultados de Grafana con 8 clientes TCP



Ilustración 23 Resultados de Grafana con 16 clientes TCP



Ilustración 21 Resultados de Grafana con 32 clientes TCP

No. Clientes	Mínimo	Media	Máximo	Crecimiento
1 Cliente	8,18MB	23,39MB	38,25MB	30,07MB
4 Clientes	12,50MB	31,83MB	49,15MB	36,65MB
8 Clientes	14,23MB	34,91MB	54,09MB	39.86MB
16 Clientes	16,33MB	52,72MB	83,86MB	67.53MB
32 Clientes	21,70MB	61,69MB	99,32MB	77.62MB

Tabla 2 Crecimiento de memoria en el servidor TCP

Vemos por tanto que, aunque el comportamiento de la memoria debido a la optimización de Docker es inicialmente mejor que en los dispositivos reales, ya que con 32 cliente la primera media hora utiliza 61MB de memoria de los 256MB disponibles, cuando un dispositivo real estaría ya muy cerca de su límite de memoria, el aumento lineal de memoria que nos da Docker significa que con 32 clientes el servidor colapsaría por memoria en menos de una hora y media, lo cual no debería de pasar en un dispositivo real.

Por otro lado, gracias los Logs de los clientes podemos apreciar la caída de rendimiento del servidor en la velocidad de resolución de peticiones por segundo mostrados en la Tabla 3.

	Mínimo	Máximo
1 Cliente	1428pet./s	2010pet./s
4 Clientes	132pet./s	510pet./s
8 Clientes	66pet./s	286pet./s
16 Clientes	29pet./s	98pet./s
32 Clientes	38pet./s	44pet./s

Tabla 3 Peticiones por segundo en TCP

RED MODBUS TLS

Recordemos que TLS trabaja en una capa superior al protocolo TCP por lo que propio hecho de utilizarlo significa más carga de procesamiento, debido a las operaciones de cifrado y descifrado aplicado en los participantes. Por ello vemos un notable aumento en la memoria requerida de los clientes que pasa de ser unos 10MB en TCP a unos 21MB como se aprecia en la Ilustración 24.



Ilustración 24 Resultados de Grafana con 1 cliente TLS

Aunque como se puede apreciar en las siguientes Ilustración 25, Ilustración 26, Ilustración 27 e Ilustración 28, este comportamiento tiende a aparecer de modo que en el caso de 4 clientes tan solo uno de ellos usa 20MB y los demás, necesitan los 10MB que usaban los clientes TCP. Y en los casos siguientes directamente ningún cliente parece usar más memoria que en TCP. Aunque el caso de que solo uno de ellos usase más memoria podría ser comprensible dado la eliminación de duplicidades de Docker, la no aparición de dicho fenómeno en los casos con más clientes que 4, no parece tener lógica.



Ilustración 25 Resultados de Grafana con 4 clientes TLS



Ilustración 28 Resultados de Grafana con 8 clientes TLS



Ilustración 27 Resultados de Grafana con 16 clientes TLS



Ilustración 26 Resultados de Grafana con 32 clientes TLS

Por otro lado, en el servidor, vemos (en la Tabla 4) que el comportamiento de la memoria, para un número de clientes bajo, sigue siendo parecido al obtenido en TCP con la diferencia de que el valor inicial, algo más que el doble que en TCP, lo cual como ya hemos mencionado era lo esperado y normal debido a la mayor carga de procesamiento producida por el protocolo TLS. No obstante, vemos que una vez más a partir de los 4 clientes el comportamiento vuelve a ser extraño incluso llegando a decaer con mayor número de clientes. Esto obviamente está relacionado con lo mencionado anteriormente sobre la desaparición de clientes que usan unos 20MB, por lo que podría ser una especie de colapso del sistema.

	Mínimo	Media	Máximo	Crecimiento
1 Cliente	19,77MB	32,00MB	43,84MB	30,07MB
4 Clientes	27,40MB	57,24MB	90,63MB	36,65MB
8 Clientes	26,25MB	46,32MB	63,83MB	39.86MB
16 Clientes	29,03MB	45,32MB	62,56MB	67.53MB
32 Clientes	36,90MB	54,69MB	99,32MB	69,58MB

Tabla 4 Crecimiento de memoria en el servidor TLS

Lo que sí tiene sentido e incluso es aplicable a los dispositivos reales es la caída del rendimiento del servidor en el número de peticiones por segundo mostrado en la Tabla 5.

	Mínimo	Máximo
1 Cliente	839pet./s	922pet./s
4 Clientes	369pet./s	437pet./s
8 Clientes	60pet./s	191pet./s
16 Clientes	22pet./s	83pet./s
32 Clientes	10pet./s	38pet./s

Tabla 5 Peticiones por segundo en TLS

Se aprecia un rendimiento menor que en TCP junto con una mayor estabilidad entre máximos y mínimos, ambos cambios esperados y lógicos. Además, en el caso de 32 clientes, se llega a apreciar la sobrecarga del servidor durante un momento, donde no es capaz de aceptar una conexión de un cliente como se puede apreciar en la Ilustración 29.

```

[DEBUG/MainProcess] 35 requests/second
[DEBUG/MainProcess] time taken to complete 1000 cycle by 10 workers is 28.26242383600038 seconds
[DEBUG/MainProcess] 12 requests/second
[DEBUG/MainProcess] time taken to complete 1000 cycle by 10 workers is 79.63126860699958 seconds
[DEBUG/MainProcess] 10 requests/second
[DEBUG/MainProcess] time taken to complete 1000 cycle by 10 workers is 91.94805216700115 seconds
[DEBUG/MainProcess] 14 requests/second
[DEBUG/MainProcess] time taken to complete 1000 cycle by 10 workers is 67.59918666399972 seconds
[DEBUG/MainProcess] 38 requests/second
[DEBUG/MainProcess] time taken to complete 1000 cycle by 10 workers is 26.236887884999305 seconds
[ERROR/MainProcess] failed to run test successfully
Traceback (most recent call last):
  File "performanceTLS.py", line 74, in single_client_test
    client.read_holding_registers(10, 1, unit=1) #.registers[0] da error por algun motivo(se supone que no tiene atributo registers)
  File "/usr/local/lib/python3.8/dist-packages/pymodbus/client/common.py", line 114, in read_holding_registers
    return self.execute(request)
  File "/usr/local/lib/python3.8/dist-packages/pymodbus/client/sync.py", line 107, in execute
    raise ConnectionException("Failed to connect[%s]" % (self.__str__()))
pymodbus.exceptions.ConnectionException: Modbus Error: [Connection] Failed to connect[ModbusTlsClient(autoserver:8020)]
[DEBUG/MainProcess] 36 requests/second
[DEBUG/MainProcess] time taken to complete 1000 cycle by 10 workers is 27.777717125998606 seconds
[DEBUG/MainProcess] 13 requests/second
[DEBUG/MainProcess] time taken to complete 1000 cycle by 10 workers is 72.06350807799936 seconds
[DEBUG/MainProcess] 10 requests/second
[DEBUG/MainProcess] time taken to complete 1000 cycle by 10 workers is 93.74398726599975 seconds
[DEBUG/MainProcess] 13 requests/second
[DEBUG/MainProcess] time taken to complete 1000 cycle by 10 workers is 72.9443238040003 seconds
[DEBUG/MainProcess] 38 requests/second
[DEBUG/MainProcess] time taken to complete 1000 cycle by 10 workers is 26.0810097829999 seconds
[DEBUG/MainProcess] 36 requests/second
[DEBUG/MainProcess] time taken to complete 1000 cycle by 10 workers is 27.136822796999695 seconds

```

Ilustración 29 Sobrecarga del servidor en TLS

A pesar de los resultados de memoria obtenidos mediante Dockprom en Grafana, este resultado es realmente fiel a un dispositivo real, ya que el servidor (limitado a 256MB de memoria RAM) va perdiendo rendimiento según el número de clientes crece hasta dar el primer fallo en 32 clientes, precisamente el límite de clientes máximos aceptados por los dispositivos reales habituales.

Conclusiones

Echando la vista atrás, para comprobar la obtención de los objetivos vemos que la red Modbus n a 1, ha sido creada con éxito tanto como para Modbus TCP como para Modbus TLS. Se han diseñado e implementado los dispositivos virtuales que forman dicha red, en Python gracias a la librería Pymodbus. Dichos dispositivos se han incluido en imágenes Docker para su posterior ejecución organizada para formar el sistema con Docker Compose. En ambas redes, las conexiones entre los clientes y servidores se ejecutan de forma válida y cumpliendo todas las especificaciones de Modbus.

La comprobación del rendimiento de la red, también se ha podido implementar con Dockprom. La inclusión de nuestra red dentro de la herramienta de monitorización Dockprom ha permitido la comparación con los dispositivos reales. En el caso de TCP hemos visto con Grafana que nuestro sistema se comporta de manera más eficiente que los dispositivos reales, aunque con el tiempo se puede llegar a colapsar debido al crecimiento lineal de la memoria usada por el servidor. Por otro lado, en la red TLS hemos obtenido resultado extraños con Grafana cuando el número de clientes superaba los 4, aunque viendo los Logs de los clientes en TLS, se ha podido ver un comportamiento mucho más normal y cercano a los dispositivos reales que habíamos tomado como referencia.

Por último, la creación de una red de prueba pensada como base de un Cyber Range ha sido conseguida, el sistema que se ha montado, cumple los requisitos de escalabilidad, flexibilidad y seguridad. La red de pruebas constituye un entorno de experimentación totalmente seguro, al estar siendo ejecutado aislado en una máquina virtual de Amazon Web Service Linux. En cuanto a la flexibilidad y escalabilidad, nos los proporcionan Docker Compose y las instancias EC2 de AWS respectivamente. Con tan solo, copiar y pegar unas líneas de código en el archivo “docker-compose.yml”, Docker Compose nos proporciona la posibilidad de cambiar la red al número de clientes que queramos de manera muy rápida y simple. En cuanto a la capacidad de procesamiento, el sistema está alojado en una instancia de 8GB de RAM y 2 CPUs pero AWS nos proporciona la opción de cambiar extremadamente fácil a una máquina más potente como por ejemplo una A14xLarge con 16 CPUs y 32 GB de RAM. Por lo que se puede decir que la red de pruebas para la creación de Cyber Range puede ser efectiva para futuros experimentos e investigaciones en este entorno, como pueden ser más centrados en la ciberseguridad del sistema, por ejemplo.

Recomendaciones

Sobre todo sobre en el apartado de Implementación, se han ido dando ciertas recomendaciones de como proceder pero una recomendación a futuro, que es muy conveniente mencionar, es la inclusión del sistema en Kubernetes.

Kubernetes haría la misma función que hace Docker Compose en nuestro sistema. Es decir, se encarga de gestionar y coordinar los distintos contenedores, organizandolos en una red y desplegándolos. Pero la ventaja de Kubernetes es la facilidad para el despliegue de nuevos contenedores.

En Kubernetes se pueden desplegar contenedores con un simple clic, una vez hayan sido definidos, además el despliegue se hace en tiempo real, sin tener que reiniciar el sistema como se hace con Docker Compose. Sin embargo, Dockprom aún no es compatible con Kubernetes, ya que el software para la traducción de Docker-Compose a Kubernetes, llamado “Kompose”, no trabaja con la versión 2.1 de Docker Compose, que es la que utiliza Dockprom. Por lo tanto, si en un futuro Dockprom se actualiza a la version 3.0 de Docker Compose, por ejemplo, el cambio a Kubernetes, se volvería muy factible y recomendable.

Bibliografía

- [1] i-Scoop, “Business guide to Industrial IoT (Industrial Internet of Things).” .
- [2] Envira, “¿Qué es IIoT, el Internet Industrial de las Cosas?” .
- [3] G. Williamson, “OT, ICS, SCADA – What’s the difference?” 2015.
- [4] Gartner, “IT vs OT industrial internet.” .
- [5] Modbus, “faq @ www.modbus.org.” .
- [6] “The Transport Layer Security (TLS) Protocol Version 1.2.” .
- [7] M. Org, “MODBUS / TCP Security.”
- [8] Modbus, “Modbus Application Protocol Specification V1.1b3,” 2012.
- [9] Riptideio, “Pymodbus.” .
- [10] E. F. Chamorro, “Cyber Range: una capacidad estratégica.” 2016.
- [11] T. DEĞER, “What is Docker?” .
- [12] S. Prodan, “Dockprom.” .
- [13] Prometheus, “What is Prometheus?” .
- [14] K. Shanmugam, “Use Google cAdvisor for monitoring your containers + Docker tutorial.” 2018.
- [15] AWS, “About AWS.” .
- [16] “Pymodbus Synchronous server.” .
- [17] “Controlador IIot Modicon M262.” .
- [18] “MGate MB3170/MB3270 Series.” .