Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Emulador de comunicaciones de sistemas de control industriales

Autor: Juan José Rosendo Alonso

Tutor: Antonio Estepa Alonso

Dpto. Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2025

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Emulador de comunicaciones de sistemas de control industriales

Autor:

Juan José Rosendo Alonso

Tutor:

Antonio Estepa Alonso

Doctor

Dpto. Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2025

Trabajo Fin de Grado: Emulador de comunicaciones de sistemas de control industriales

Autor: Juan José Rosendo Alonso
Tutor: Antonio Estepa Alonso

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

A lo largo de estos años de formación, he tenido la fortuna de contar con personas increíbles, tanto quienes se han sumado recientemente como quienes llevan tiempo a mi lado. Este éxito no es solo mío, sino también de ellos:

A mis compañeros de clase. Gracias por la ayuda, la compañía y los buenos momentos.

A mi familia. Gracias por el apoyo incondicional, el sacrificio y los ánimos.

A mis profesores. Gracias por el conocimiento, la dedicación y la inspiración.

A mi tutor. Gracias por la paciencia, el tiempo y la confianza.

A mi pareja. Gracias por el cariño, la comprensión y el apoyo constante.

Gracias de corazón a todos los que habéis participado en esta etapa. Sin vosotros, no habría sido posible.

Juan José Rosendo Alonso
Sevilla, 2024

Resumen

Este Trabajo de Fin de Grado se centra en el desarrollo de un emulador de tráfico Modbus, una herramienta diseñada para simular la comunicación entre dispositivos industriales. El principal objetivo es proporcionar un entorno seguro y práctico que permita probar, depurar y analizar sistemas basados en el protocolo Modbus sin necesidad de interactuar con equipos reales, evitando así riesgos asociados a su manipulación.

El proyecto abarca desde el diseño de una arquitectura basada en contenedores Docker hasta la implementación de una interfaz gráfica intuitiva. El sistema utiliza el patrón Modelo-Vista-Controlador (MVC) y está compuesto por tres partes principales: Core, que gestiona la emulación mediante Docker y Tcpdump; Backend, encargado de coordinar las acciones del sistema; y Frontend, que ofrece al usuario la posibilidad de configurar escenarios de red de manera interactiva.

La solución desarrollada permite emular múltiples dispositivos Modbus TCP, configurados como maestros o esclavos, en una red virtual. Se destacan funcionalidades como la validación de configuraciones de red, la personalización del tráfico emulado y la captura de paquetes en formato PCAP para su análisis posterior. Además, el sistema facilita la creación, carga, guardado y ejecución de escenarios de red personalizados.

Este emulador no solo tiene aplicaciones prácticas en la seguridad y análisis de redes industriales, sino que también puede ser utilizado en entornos educativos y en la generación de datasets para entrenar algoritmos de inteligencia artificial destinados a la detección de ciberataques.

Abstract

This Bachelor's Thesis focuses on the development of a Modbus traffic emulator, a tool designed to simulate communication between industrial devices. The main objective is to provide a safe and practical environment that allows for testing, debugging, and analyzing systems based on the Modbus protocol without the need to interact with real equipment, thus avoiding risks associated with its handling.

The project spans from the design of a container-based architecture using Docker to the implementation of an intuitive graphical interface. The system uses the Model-View-Controller (MVC) pattern and is composed of three main parts: Core, which manages emulation using Docker and Tcpdump; Backend, responsible for coordinating system actions; and Frontend, which offers the user the ability to configure network scenarios interactively.

The developed solution allows for the emulation of multiple Modbus TCP devices, configured as masters or slaves, in a virtual network. Key features include network configuration validation, customization of emulated traffic, and packet capture in PCAP format for subsequent analysis. Additionally, the system facilitates the creation, loading, saving, and execution of custom network scenarios.

This emulator not only has practical applications in the security and analysis of industrial networks but can also be used in educational environments and in the generation of datasets to train artificial intelligence algorithms aimed at detecting cyberattacks.

Índice

Resumen III
Abstract V

1 Introducción **1**
1.1 Motivación 1
1.2 Objetivos 1

2 Conocimientos previos **3**
2.1 Modbus 3
2.2 Docker 4
2.2.1 Docker Compose 5
Services 5
Networks 6
Volumes 6
Ejemplo 6
2.3 Patrón MVC 7
2.4 REST API 7
2.4.1 Ejemplo práctico de REST API 8
2.4.2 Errores comunes y desafíos en REST API 8

3 Estado del Arte **11**
3.1 Simulación vs. Emulación 11
3.2 Software de Emulación 11
3.3 Trabajos Relacionados 12

4 Diseño e Implementación de la Solución **13**
4.1 Selección de Componentes 13
4.1.1 Core 13
4.1.2 Backend 13
Endpoints de la REST API 14
4.1.3 Frontend 14
4.2 Entorno de Desarrollo 15
4.3 Arquitectura de la Solución 15
4.4 Core 16
4.4.1 Docker Compose 16
Networks 16
Services 17
4.4.2 Docker Images 18
Dockerfiles 18
Scripts 19

4.4.3 Módulo docker_compose_generator.py 20
Clase DockerComposeGenerator 20
Métodos de la Clase 20
Ejemplo de Uso 21
4.4.4 Módulo scenario_config_generator.py 21
Clase ScenarioConfigGenerator 22
Métodos de la Clase 22
Ejemplo de Uso 22
4.4.5 Módulo runner.py 22
start 22
stop 23
status 23
4.5 Backend 23
4.5.1 Módulo web.py 23
GET /api/networks/ 24
POST /api/networks/ 24
GET /api/networks/name 24
PUT /api/networks/name 25
GET /api/run/ 25
POST /api/run/name 25
DELETE /api/run/ 25
GET /index.html 25
GET /networks/id 26
4.5.2 Módulo cytoscape_adapter.py 26
Función validate_cytoscape_scenario 26
Función parse_cytoscape_json 26
Función generate_network_nodes 27
4.5.3 Módulo scenario_handler.py 28
Función save_scenario 28
Función get_cytoscape_scenario 28
Función get_python_scenario 28
4.6 Frontend 28
4.6.1 Script index.js 28
4.6.2 Script network.js 28
Elementos visibles 28
Acciones de Usuario 30
Configuración de nodos 30
4.6.3 Configuración de los mensajes 31
4.6.4 Confirmación de Guardado 31
4.6.5 Ejecución 32
4.7 Casos de Uso 32
4.7.1 Creación de un Nuevo Escenario 32
4.7.2 Carga de un Escenario 32
4.7.3 Guardado de un Escenario Cargado 32
4.7.4 Emulación del Escenario Cargado 33
4.7.5 Detención de la Emulación 33

5 Puesta en Ejecución y Ejemplos de Uso 37
5.1 Instalación 37
5.1.1 Obtención del código fuente 37
5.1.2 Instalación de dependencias 37
5.1.3 Ejecución del proyecto 37
5.2 Pruebas 37
5.2.1 Pruebas unitarias 38

Docker Compose Generator 38
Comunicación Maestro - Esclavo 38
5.3 Ejemplos de uso 38
5.3.1 Creación de Escenario 38
5.3.2 Carga de Escenario 39
5.3.3 Pantalla de red y utilidades 39
5.3.4 Configuración de Maestros 40
5.3.5 Configuración de Esclavos 40
Registros 41
Identidad 41
5.3.6 Configuración de Mensajes 42
Maestro Insistente 42
Maestro Relajado 42
5.3.7 Guardado de escenario 43
5.3.8 Ejecución de la emulación 43
5.3.9 Resultados 43

6 Conclusiones 47
6.1 Usos derivados 47
6.1.1 Emulación de Ataques 47
6.1.2 Enseñanza 47
6.2 Reflexión Final 47

7 Limitaciones y Líneas Futuras 49
7.1 Incrementar cantidad de protocolos 49
7.2 Retroalimentación durante la ejecución 49
7.3 Velocidad de enlace no infinita 49
7.4 Dependencia de Docker 50

Índice de Figuras 51
Índice de Tablas 53
Índice de Códigos 55

1 Introducción

1.1 Motivación

En el entorno de los sistemas industriales modernos, la ciberseguridad ha adquirido una importancia crítica debido al creciente número de amenazas a los que se enfrentan las infraestructuras de control. El protocolo Modbus, ampliamente utilizado en sistemas Industrial Control System (ICS), fue diseñado originalmente sin medidas de seguridad intrínsecas, lo que lo hace vulnerable a una variedad de ataques. Estos ataques comprometen los tres atributos más importantes de la información: integridad, disponibilidad y confidencialidad[1]. En particular, comprometer la información de estas comunicaciones pone en riesgo infraestructuras críticas como lo es la red eléctrica de España.

La necesidad de contar con herramientas que faciliten la labor de los profesionales de la ciberseguridad es vital. Aquí es donde entra la emulación de tráfico realista, donde se puede evaluar la seguridad de una red ante ciberataques para posteriormente prepararse adecuadamente. Este proyecto busca llenar ese vacío, proporcionando un emulador que no solo facilite el análisis de una red, sino que también permita realizar pruebas prácticas y realistas en un entorno seguro y controlado.

La motivación última de un proyecto de estas características es la de generar *datasets* que permitan entrenar algoritmos de inteligencia artificial para la detección de ciberataques en sistemas ICS.

1.2 Objetivos

Siendo extensa el área de la ciberseguridad, se ha tratado de llegar a un compromiso donde el alcance y la facilidad de uso no se vean comprometidos. Por ello, se han establecido los siguientes objetivos:

1. Desarollar un software que emule de forma precisa la comunicación entre múltiples dispositivos Modbus en su implementación Transmission Control Protocol (TCP).
2. Desarrollar una interfaz de usuario que permita la configuración gráfica de los escenarios sin conocimiento de la tecnología subyacente.

En el proyecto se han desarrollado más funcionalidades que las propuestas inicialmente, descritas en el capítulo de resultados.

2 Conocimientos previos

Un proyecto de estas características tiene una alta carga de conocimientos previos tanto en el ámbito de la gestión de servicios como en el de la programación. En este capítulo se expondrán los conceptos básicos necesarios para entender el proyecto.

2.1 Modbus

Modbus es un protocolo de comunicación originado en el año 1979 por Modicon (actualmente parte de Schneider Electric), diseñado inicialmente para la automatización industrial y la interconexión de dispositivos en sistemas de control distribuido. Su ámbito de aplicación reside en la gestión de dispositivos en fábricas y plantas industriales.

A lo largo de su historia, se han propuesto diversas soluciones para mejorar la baja seguridad inherente al protocolo, como la implementación de capas de cifrado y autenticación. Sin embargo, debido a la filosofía "si funciona no lo toques" y a la necesidad de mantener la compatibilidad con equipos existentes, estas soluciones no están en funcionamiento. [2]

Modbus tiene tres variantes principales: Modbus Remote Terminal Unit (RTU), Modbus TCP, y Modbus User Datagram Protocol (UDP), cada una adaptada a diferentes entornos de comunicación. Modbus RTU utiliza una comunicación serie, mientras que Modbus TCP y Modbus UDP operan sobre redes Internet Protocol (IP), permitiendo una integración más sencilla en redes modernas.[3]

Modbus se basa en un paradigma de comunicación maestro-esclavo, donde un dispositivo maestro (generalmente un controlador) envía comandos a uno o varios dispositivos esclavos (sensores, actuadores, etc.), que responden a estas solicitudes de acuerdo con su función específica.[4]

En este paradigma el maestro es el que inicia la comunicación, y el esclavo responde a las solicitudes del maestro. En algunos protocolos, como es el caso de DNP3 o IEC 60870-5-104 (comúnmente conocido como IEC 104), se permite que además el esclavo actualice al maestro del estado de sus registros sin una solicitud explícita previa.

La estructura de la cabecera de Modbus se muestra en la Figura 2.1, donde se destacan los campos de la cabecera fijos. Los campos variables dependen del Function Code (FC) que se esté utilizando, añadiendo si lo necesita un campo que referencie un valor (1 Byte) o una dirección de registro (2 Bytes). Dado que el objetivo de este apartado es introducir la cabecera y no explicarla en profundidad se dejan de lado el resto de campos dependientes del FC que puedan aparecer en la cabecera.

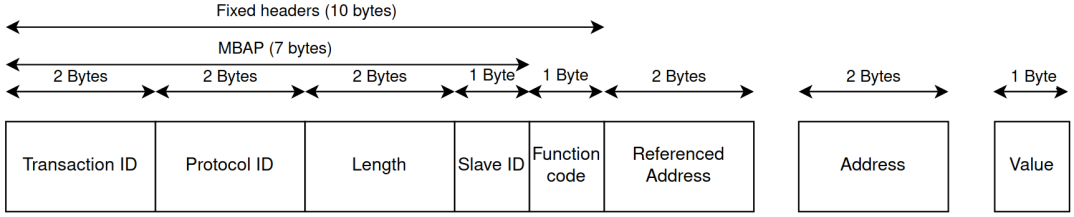

Figura 2.1 Campos de la cabecera del protocolo Modbus.

Existen diversos FC implementados en el protocolo Modbus, cada uno con una función específica. En la Tabla 2.1 se muestran algunos de los FC más comunes.

Tabla 2.1 Códigos de función Modbus.

Function Code	Nombre	Descripción
01	*Read Coils*	Lee el estado de las bobinas en el rango especificado.
02	*Read Discrete Inputs*	Lee el estado de las entradas digitales en el rango especificado.
03	*Read Holding Registers*	Lee el contenido de los registros de retención.
04	*Read Input Registers*	Lee el contenido de los registros de entrada.
05	*Write Single Coil*	Escribe un valor en una sola bobina.
06	*Write Single Register*	Escribe un valor en un solo registro.
15	*Write Multiple Coils*	Escribe valores en múltiples bobinas.
16	*Write Multiple Registers*	Escribe valores en múltiples registros.
43	*Read Device Information*	Lee información específica del dispositivo, como identificadores.

El protocolo define cuatro tipos de registros, los cuales tienen un espacio de direcciones distinto y viene dado por el FC, esto es, dos registros pueden estar en la misma dirección mientras sean de distinto tipo.

Los cuatro tipos de registros se diferencian por su función y acceso:

- **Coil (Bobina):** Registros de tipo bit diseñados para operaciones de salida. Permiten lectura (FC 1) y escritura (FC 5 para un bit o FC 15 para múltiples bits). Se utilizan típicamente para controlar actuadores como relés o válvulas.
- **Discrete Input (Entrada Discreta):** Registros de solo lectura (FC 2) que representan estados binarios de entradas físicas, como sensores digitales o interruptores. No permiten escritura.
- **Holding Register (Registro de Retención):** Registros de 16 bits para almacenar datos numéricos. Son accesibles para lectura (FC 3) y escritura (FC 6 para un registro o FC 16 para múltiples). Se emplean en parámetros configurables como setpoints o valores de control.
- **Input Register (Registro de Entrada):** Registros de 16 bits de solo lectura (FC 4), usados para datos analógicos como mediciones de temperatura, presión o valores de sensores que no requieren escritura.

Esta división garantiza coherencia funcional: los registros *Input* (2 y 4) son de solo lectura para monitorización, mientras los *Holding* y *Coils* (3 y 1) permiten configuración activa. La tabla 2.2 resume los FCs asociados a cada tipo de registro.

Tabla 2.2 Tipos de Registros en Modbus.

Tipo	Tipo Dato	FC lectura	FC escritura
Coil	Bit	1	5, 15
Discrete Input	Bit	2	-
Holding Register	16-bit Word	3	6, 16
Input Register	16-bit Word	4	-

Un aspecto crítico es la codificación de datos en los registros de 16 bits. Mientras los *Holding* e *Input Registers* almacenan valores en formato binario (enteros sin signo, complemento a dos o punto flotante según estándar IEC 61131-3), los *Coils* y *Discrete Inputs* usan valores booleanos (0x0000=Falso, 0xFF00=Verdadero). Esta estructura permite interoperabilidad entre dispositivos mediante una semántica predecible.

2.2 Docker

Docker es una tecnología desarrollada por Docker Inc. en el año 2013 cuyo fin es facilitar la creación, despliegue y ejecución de aplicaciones mediante contenedores. Los contenedores permiten a los desarrolladores empaquetar una aplicación con todas sus dependencias en un entorno aislado y consistente, mejorando la

portabilidad y la eficiencia.

En este contexto se habla de imágenes, que son plantillas inmutables a partir de las cuales se instancian los contenedores. Una semejanza rápida sería comparar las imágenes y contenedores a programas y procesos, respectivamente: un contenedor es una imagen en ejecución. Para la gestión de imágenes contamos con los comandos principales mostrados en la tabla 2.3.

Tabla 2.3 Comandos comunes para la gestión de imágenes en Docker.

Comando	Uso
`docker pull`	Descargar una imagen desde un repositorio
`docker build`	Construir una imagen a partir de un Dockerfile
`docker images`	Listar todas las imágenes locales
`docker rmi`	Eliminar una imagen local

Para el aprovisionamiento de imágenes se puede o bien reusar una ya existente o bien crearla. La creación de una imagen personalizada se realiza mediante un Dockerfile, que es un archivo de texto con instrucciones que describen cómo construir la imagen. Las instrucciones más importantes que se pueden usar dentro de un Dockerfile están recogidos en la tabla 2.4.

Tabla 2.4 Instrucciones comunes de Dockerfile.

Instrucción	Uso
`FROM`	Especificar la imagen base para la construcción
`RUN`	Ejecutar comandos en el contenedor durante la construcción
`COPY`	Copiar archivos/directorios al sistema de archivos del contenedor
`CMD`	Especificar el comando por defecto a ejecutar
`EXPOSE`	Declarar el puerto en el que la aplicación escuchará
`ENV`	Establecer variables de entorno

Un Docker Registry es un servicio centralizado que permite almacenar, gestionar y distribuir imágenes Docker. Estas imágenes pueden ser públicas o privadas, dependiendo de las necesidades del proyecto. El registro más conocido es Docker Hub, que ofrece una gran variedad de imágenes preconstruidas listas para su uso. Sin embargo, también es posible configurar registros privados, como Amazon Elastic Container Registry (ECR) o Azure Container Registry (ACR), para mantener un control más estricto sobre el acceso y la distribución de las imágenes. Los registros facilitan la colaboración entre equipos al proporcionar un repositorio centralizado desde donde se pueden descargar las imágenes necesarias, y además, soportan procesos de CI/CD al integrarse fácilmente en los flujos de desarrollo. A modo de resumen se incluye la figura 2.2

2.2.1 Docker Compose

Docker Compose es una herramienta de Docker que facilita la gestión de los contenedores. Mediante un archivo YAML Ain't Markup Language (YAML) de nombre por defecto *docker-compose.yml*, se puede especificar cómo deben ser configurados y relacionados varios servicios, simplificando el proceso de despliegue y gestión de aplicaciones complejas.

YAML es un formato de serialización de datos fácil de leer y escribir. En este caso, el YAML deberá tener obligatoriamente un campo *services* y opcionalmente si así lo requiere, otro de *networks* y *volumes*.

Services

El bloque *services* define los distintos servicios que forman la aplicación. Cada servicio representa un contenedor que puede incluir configuraciones como la imagen a usar, variables de entorno, puertos expuestos, volúmenes, y comandos a ejecutar.

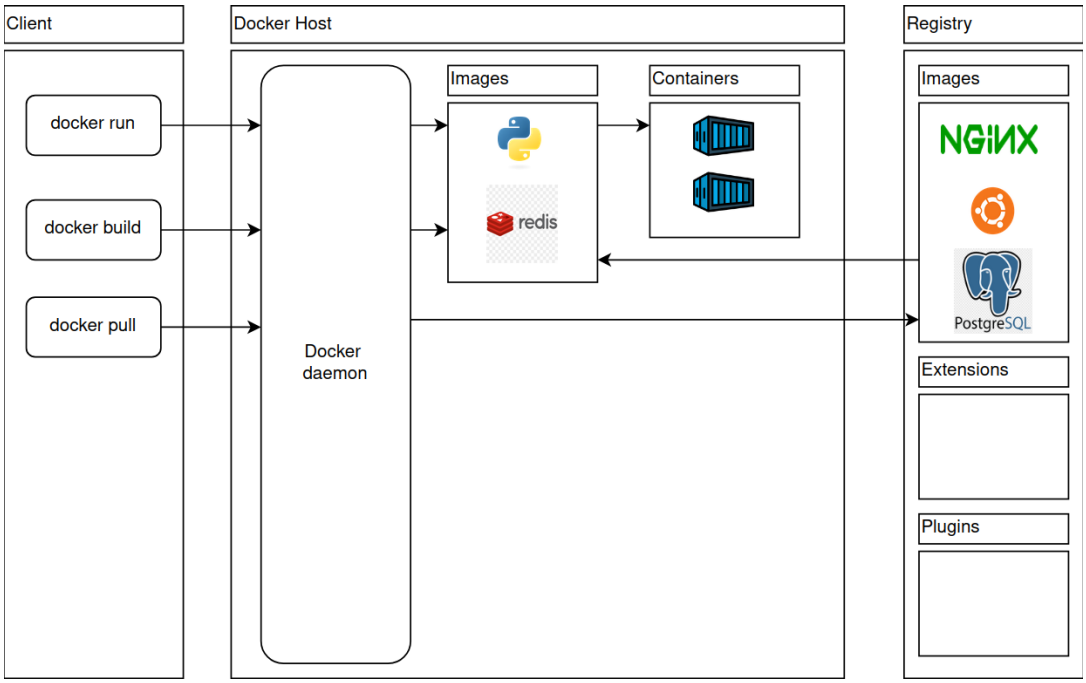

Figura 2.2 Arquitectura de Docker.

Networks

El bloque *networks* permite definir redes personalizadas para facilitar la comunicación entre servicios de forma aislada. Esto ofrece un mayor control sobre la conectividad entre los contenedores y permite crear entornos más seguros y organizados.

Volumes

El bloque *volumes* permite definir volúmenes que pueden ser utilizados por los servicios para almacenar datos de manera persistente. Esto es útil cuando es necesario mantener datos más allá del ciclo de vida de un contenedor. Los volúmenes pueden ser configurados para mapear una parte específica del sistema de archivos del anfitrión a una ubicación dentro del contenedor, asegurando que los datos permanezcan accesibles incluso si los contenedores se eliminan o reinician.

Ejemplo

Un ejemplo de un *docker-compose.yml* con dos servicios con poca configuración sería el descrito en 2.1. Aquí se definen dos servicios *web* y *db*. El primero usa la imagen *nginx:latest* y expone el puerto 80 del contenedor en el puerto 8080 del anfitrión. El segundo usa la imagen *postgres:latest* y expone el puerto 5432 del contenedor en el puerto 5432 del anfitrión. Además, se definen variables de entorno para configurar la base de datos.

Código 2.1 Ejemplo básico docker-compose.yml.

```
services :
web:
  image: nginx: latest
  ports :
    - "8080:80"

db:
  image: postgres : latest
  environment:
    POSTGRES_USER: exampleuser
    POSTGRES_PASSWORD: examplepass
    POSTGRES_DB: exampledb
  ports :
    - "5432:5432"
```

2.3 Patrón MVC

El patrón Modelo-Vista-Controlador (MVC) es una arquitectura de software utilizada ampliamente en el desarrollo de aplicaciones web, móviles y de escritorio [5]. Su principal objetivo es organizar el código en tres componentes claramente diferenciados: Modelo, Vista y Controlador. Esto promueve la separación de responsabilidades, permitiendo mejorar la modularidad, escalabilidad, mantenimiento y reutilización del código.

Cada uno de estos componentes desempeña un rol fundamental dentro del patrón:

1. **Modelo**: Representa los datos y la lógica de negocio de la aplicación. Se encarga de realizar cálculos, gestionar la persistencia de datos (por ejemplo, mediante acceso a bases de datos) y procesar las reglas de negocio. El Modelo debe estar desacoplado de los detalles específicos de presentación y enfocarse exclusivamente en la funcionalidad de la aplicación.
2. **Vista**: Es la capa responsable de la presentación de los datos y de la interacción con el usuario. Muestra la información proveniente del modelo a través de interfaces gráficas o visuales, ya sea en navegadores web, aplicaciones de escritorio o dispositivos móviles. La Vista debe estar diseñada para minimizar los problemas de integración y facilitar la definición de la interfaz de usuario dentro del sistema.
3. **Controlador**: Actúa como intermediario entre la Vista y el Modelo. Procesa las peticiones del usuario, las interpreta y las traduce en acciones que interactúan con el Modelo. Posteriormente, el Controlador toma la respuesta del Modelo y la pasa a la Vista, asegurando que los datos sean presentados correctamente.

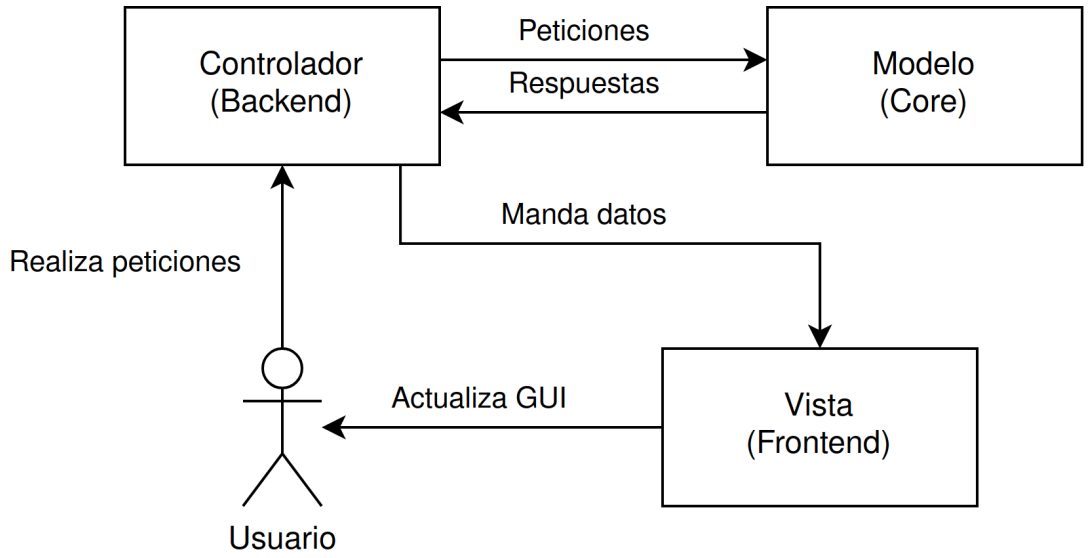

Figura 2.3 Aplicación del patrón Modelo-Vista-Controlador.

2.4 REST API

En el contexto de aplicaciones web modernas, Representational State Transfer (REST) es un estilo arquitectónico ampliamente adoptado para diseñar servicios web que permitan la comunicación entre sistemas distribuidos. REST no es un protocolo, sino un conjunto de buenas prácticas que promueven la uniformidad en la comunicación entre componentes.[6]

REST se basa en el uso de estándares web, como Hypertext Transfer Protocol (HTTP) o Constrained Application Protocol (CoAP), para la transferencia de recursos identificados por Uniform Resource Identifier (URI)s. Cada recurso es representado en formatos estandarizados como JavaScript Object Notation (JSON) o Extensible Markup Language (XML), facilitando la interoperabilidad entre sistemas heterogéneos. Las principales características de una REST Application Programming Interface (API) son:

- **Cliente-Servidor**: La arquitectura se divide en dos partes independientes, el cliente y el servidor, que interactúan a través de una interfaz bien definida.

- **Interfaz uniforme**: Los recursos son accedidos a través de un conjunto de operaciones Create, Read, Update, Delete (CRUD).
- **Sin estado**: Cada petición al servidor debe contener toda la información necesaria para procesarla, sin depender de un estado almacenado entre peticiones.

El diseño de una REST API HTTP efectiva está guiado por buenas prácticas que incluyen:[7]

- **Diseño orientado a recursos**: Cada URI debe representar un recurso específico, nombrado con sustantivos (por ejemplo, `/usuarios` o `/pedidos`).
- **Uso adecuado de los métodos HTTP**: Como se detalla en la tabla 2.5, los métodos HTTP están asociados a operaciones CRUD estandarizadas.
- **Manejo de códigos de estado HTTP**: Las respuestas deben incluir códigos de estado como `200 OK`, `404 Not Found` o `500 Internal Server Error` para informar al cliente sobre el resultado de la operación.

Tabla 2.5 Equivalencia entre operaciones CRUD y métodos HTTP.

Operación CRUD	Método HTTP	Uso
Create	POST	Crea un nuevo recurso
Read	GET	Recupera información de un recurso existente
Update	PUT	Modifica un recurso existente
Delete	DELETE	Elimina un recurso existente

En el contexto del patrón MVC, las REST APIs suelen desempeñar un papel fundamental al actuar como interfaz entre el Modelo y la Vista. Por ejemplo, cuando un usuario interactúa con la interfaz de una aplicación web, las peticiones generadas por la Vista (normalmente utilizando Asynchronous JavaScript And XML (AJAX) o similar) son enviadas al Controlador. Este, a su vez, interactúa con el Modelo para obtener o modificar datos y devuelve una respuesta estructurada (en formato JSON o XML) que es procesada y mostrada por la Vista.

2.4.1 Ejemplo práctico de REST API

Un ejemplo típico de REST API para la gestión de usuarios podría incluir un endpoint para obtener información de un usuario específico:

```
GET /usuarios/123 HTTP/1.1
Host: api.ejemplo.com
Authorization: Bearer token123

HTTP/1.1 200 OK
Content-Type: application/json

{
    "id": 123,
    "nombre": "Juan Pérez",
    "email": "juan.perez@example.com"
}
```

En este ejemplo, el cliente realiza una petición GET al recurso `/usuarios/123` para obtener información de un usuario con el identificador 123. El servidor responde con un código de estado 200 y un cuerpo en formato JSON que contiene los datos solicitados.

2.4.2 Errores comunes y desafíos en REST API

A pesar de su simplicidad, el diseño de una REST API presenta varios retos:

- **Uso incorrecto de métodos HTTP**: Es frecuente observar endpoints que no respetan las convenciones REST, como `/obtenerUsuario` en lugar de `/usuarios`.

- **Endpoints poco intuitivos**: Diseñar URLs claras y orientadas a recursos puede ser un desafío, especialmente en aplicaciones complejas.
- **Autenticación y autorización**: La gestión de permisos a través de sistemas como OAuth2 o tokens JSON Web Token (JWT) requiere una implementación cuidadosa.

Abordar estos problemas es fundamental para garantizar que las APIs sean fáciles de usar, mantener y escalar. Además, las buenas prácticas en el diseño REST contribuyen a alinear el desarrollo con los principios del patrón MVC, promoviendo la modularidad y la claridad en la arquitectura del sistema.

3 Estado del Arte

3.1 Simulación vs. Emulación

La emulación y la simulación son dos enfoques distintos para la creación de tráfico de red. En este apartado se analizan las diferencias entre ambos métodos y se justifica la elección de la emulación para este proyecto.

La distinción inicial a considerar es la diferencia entre emulación y simulación. La emulación implica replicar fielmente el comportamiento de un sistema real, mientras que la simulación busca modelar dicho comportamiento sin reproducirlo de manera exacta. Según la Real Academia Española (RAE), una emulación es "imitar las acciones de otro procurando igualarlas e incluso excederlas"[1], mientras que una simulación se define como "representar algo, fingiendo o imitando lo que no es"[2].

Ambos enfoques tienen sus ventajas y limitaciones [8]. La emulación permite obtener resultados muy precisos, replicando de forma fiel las condiciones de un sistema real. Esto es particularmente útil en contextos donde la exactitud es fundamental, como en la generación de tráfico de red. No obstante, la emulación suele requerir un mayor consumo de recursos.

Por otro lado, la simulación es más eficiente en términos de recursos, lo que la convierte en una opción adecuada para escenarios que implican largos periodos de análisis. Sin embargo, su principal inconveniente radica en la dificultad de desarrollar modelos suficientemente detallados, lo que puede comprometer la fidelidad de los resultados.

Considerando la importancia de la precisión en el análisis de tráfico de red, la emulación se presenta como la opción más adecuada para este proyecto. Aunque la simulación podría ser una alternativa viable, la emulación garantiza resultados más fiables y útiles para el análisis.

3.2 Software de Emulación

Tal como se muestra en la tabla 3.1, las opciones más destacadas para la emulación son *Docker* y *GNS3*. En el caso de *GNS3*, se trata de un software enfocado en la emulación de dispositivos de red. Sin embargo, presenta tres limitaciones principales:

- Está diseñado principalmente para simular equipos de red, no dispositivos finales.
- La personalización de dispositivos finales requiere una virtualización completa de un sistema operativo o el uso de Docker integrado con GNS3.
- Existe una pérdida de funcionalidad y falta de documentación sobre la integración entre Docker y GNS3.

En cambio, Docker es una herramienta de virtualización ligera con capacidades básicas de red. Su diseño orientado a la virtualización ligera permite un control detallado sobre los recursos y el comportamiento de los contenedores. Además, cuenta con una amplia documentación y soporte de la comunidad, lo que facilita el desarrollo del proyecto.

[1] https://dle.rae.es/emular
[2] https://dle.rae.es/simular

Tabla 3.1 Comparativa entre Software.

Software	Ámbito	Precisión	Eficiencia	Personalización
QEMU	Emulación de hardware	Muy alta	Moderada	Alta
Docker	Virtualización de contenedores	Alta	Alta	Muy alta
Mininet	Emulación de redes	Alta	Alta	Moderada
OMNeT++	Simulación de redes	Moderada	Muy alta	Alta
GNS3	Emulación de dispositivos de red	Alta	Moderada	Alta

La configuración de Docker mediante línea de comandos puede resultar tediosa y compleja, especialmente cuando se gestionan múltiples contenedores. Para abordar este desafío, existen las herramientas *Docker Compose* y *Docker Swarm*, las cuales simplifican esta tarea.

Docker Compose destaca como una solución ampliamente utilizada para configurar múltiples contenedores en una misma máquina. Permite definir configuraciones en un archivo de texto local, lo que facilita la creación y administración de entornos complejos de forma reproducible y sencilla.

Docker Swarm es un orquestador de tráfico de red que permite la gestión de múltiples contenedores en entornos distribuidos. Esta herramienta ofrece más posibilidades que Docker Compose, sin embargo, dado que no estamos en un entorno distribuido, no se ha considerado necesario su uso.

3.3 Trabajos Relacionados

La idea de un software capaz de replicar tráfico no es algo novedoso, remontándose a los primeros años de vida del protocolo TCP [9] para fabricar sintéticamente paquetes de tráfico. El estudio de las dos corrientes en cuanto a la fabricación de tráfico, simulación y emulación, tiene sus inicios a comienzos de los 2000s [8]. En la actualidad existen múltiples herramientas que permiten la creación de tráfico de red. A continuación se presentan algunas de las más relevantes.

MiniCPS [10] es una herramienta para la emulación de sistemas ciberfísicos. Está desarrollado en Mininet, un emulador de red que usa mecanismos de virtualización ligera empleados en los sistemas operativos GNU/Linux, como los *namespaces* y la virtualización de interfaces de red. MiniCPS completa la funcionalidad de Mininet añadiendo componentes como Programmable Logic Controller (PLC) y Human-Machine Interface (HMI).

SCADASim [11] es una herramienta de código abierto basada en OMNET++, un simulador de eventos discretos. Ofrece módulos que representan dispositivos empleados en sistemas SCADA, como PLC y RTU, además de los protocolos utilizados para la comunicación entre estos dispositivos. También permite la integración con aplicaciones y equipos externos.

ICSSIM [12] permite la emulación de HMI, PLC y procesos industriales como componentes de *Hardware in the Loop* en entornos virtuales separados. Cada componente de simulación cuenta con direcciones IP privadas, controladores de red configurables y comunicación independiente con otros componentes. El marco soporta dos entornos de virtualización: Docker y GNS3. Este trabajo es el más cercano al nuestro en cuanto a que usa la misma tecnología subyacente, Docker, y usa el mismo lenguaje de programación dentro de los contenedores, Python. Sin embargo, el proyecto tiene una baja flexibilidad a la hora de configurar aspectos específicos de los nodos como las direcciones IP o Media Access Control (MAC). Además, usa la imagen de Docker de Ubuntu, lo cual tiene un mayor consumo de disco y memoria.

TinyICS [13] es una herramienta de código abierto basado en NS-3, un simulador de eventos discretos. Este proyecto permite la configuración de los equipos por código, lo cual permite una mayor flexibilidad en la configuración de los nodos. Sin embargo, la configuración de los nodos es más compleja que en ICSSIM, ya que requiere la codificación de un archivo de código en C++.

Este proyecto, ICSCommEmulator, usa la tecnología de Docker para emular el tráfico de red en sistemas de control industrial. El proyecto se centra en la facilidad de uso y la flexibilidad, permitiendo generar tráfico con cualquier dirección IP y MAC. Esta herramienta está equipada para definir y configurar nodos del protocolo Modbus: registros, qué mensajes mandan, identidad... Además, se ha diseñado para ser fácilmente extensible a otros protocolos, lo que lo convierte en un proyecto fácilmente modificable para necesidades no cubiertas.

4 Diseño e Implementación de la Solución

En este capítulo se describe en detalle el diseño y la implementación de la solución propuesta. Tal como se expuso en el Capítulo 1, el objetivo principal de este proyecto es desarrollar un emulador de tráfico ICS, acompañado de una interfaz gráfica que sea intuitiva para el usuario.

4.1 Selección de Componentes

Al diseñar un proyecto de esta índole, lo primero es identificar la funcionalidad mínima. Recurriendo al patrón MVC ya explicado en Sección 2.3 se vislumbra como buena opción desarrollar tres componentes. Estos componentes reciben el nombre de Modelo, Vista y Controlador, comúnmente conocidos como Core, Frontend y Backend respectivamente.

4.1.1 Core

Empezando desde el final, se pretende generar un fichero de captura de tráfico *pcap*, por lo que se necesitará *Tcpdump* para lograrlo.

Para conseguir ese tráfico se necesita poder emular elementos que generen ese tráfico. Para ello nos encontramos con *Docker*, una herramienta de virtualización ligera que permite la creación de contenedores. Estos contenedores serán los encargados de emular los dispositivos que generen el tráfico.

La gestión de Docker por terminal es tediosa y complicada, por lo que se opta por usar *Docker Compose*, una herramienta que permite gestionar múltiples contenedores de forma sencilla.

Docker usa imágenes para crear contenedores, por lo que se necesitarán imágenes de los dispositivos a emular. Para ello se ha buscado en *Docker Hub*, resultando en que no hay imágenes de modbus bien comentadas, configurables y ligeras. Es por ello que se ha optado por hacer las imágenes desde cero.

Como resulta lógico en un desarrollo multicomponente, se ha escogido un lenguaje de programación fácil de usar y depurar como lo es *Python*. Por ello se ha escogido *Pymodbus* como librería para emular Modbus.

Para configurar cada contenedor de forma independiente, disponemos de tres opciones: *Bases de Datos*, *Variables de Entorno* o *Ficheros de Configuración*. Se ha optado por la última opción de ellas, ya que es la más sencilla y la que menos problemas puede dar.

Para gestionar la configuración contamos con el módulo que se encarga de la gestión del *docker-compose.yml*, que se va a llamar *docker-compose-generator.py*, el encargado de los ficheros de configuración de los contenedores, que se llamará *scenario-config-generator.py*, y el que gestione el entorno de ejecución, que se llamará *runner.py*.

4.1.2 Backend

Puesto que ya requerimos de *Python* para el core, lo razonable es mantener la homogeneidad y evitar complicar las APIs entre componentes. Por ello, se ha optado por usar *Flask*, un framework web en Python que simplifica la creación de aplicaciones web y la gestión de solicitudes HTTP.

Los servicios mínimos que tiene que ofrecerle el backend al frontend son:

- Crear un nuevo escenario.

- Cargar un escenario previamente guardado.
- Guardar el escenario cargado.
- Emular el escenario cargado.
- Detener la emulación actual.

La solución adoptada para los endpoints concretos se resume en la tabla 4.1, que aunque es una simplificación porque no habla de los parámetros a mandar ni los devueltos, es suficente para entender la funcionalidad mínima que debe ofreecer el backend.

Endpoints de la REST API

Tabla 4.1 Endpoints de la REST API.

Método	Endpoint	Descripción
GET	/api/networks/	Lista los escenarios creados.
POST	/api/networks/	Crea un nuevo escenario.
GET	/api/networks/{name}	Obtiene la configuración de un escenario.
PUT	/api/networks/{name}	Actualiza o valida un escenario existente.
GET	/api/run/	Muestra el estado de la simulación.
POST	/api/run/{name}	Inicia la simulación de un escenario.
DELETE	/api/run/	Detiene la simulación en ejecución.
GET	/index.html	Renderiza la página principal.
GET	/networks/{id}	Renderiza la página de un escenario.

Tras inspeccionar la funcionalidad mínima necesaria, se dividirá en tres ficheros. El primero, *web.py*, se encargará de gestionar las rutas de la REST API y coordinar los componentes de la aplicación. El segundo, *scenario-handler.py*, se encargará de gestionar las entradas y salidas relacionadas con los archivos del sistema. Por último, el tercero, *cytoscape-adapter.py*, se encargará de convertir los datos del grafo interactivo de Cytoscape.js a un formato procesable por el sistema.

Dado que trabajar con el formato YAML es más cómodo que JSON, se ha optado por usar este formato para los ficheros de configuración del sistema.

4.1.3 Frontend

Tras haber identificado la funcionalidad mínima que debe ofrecer el backend, continuamos con el diseño del frontend. Como estamos en una aplicación web, la vista estará compuesta principalmente por JavaScript.

Puesto que el usuario debe poder cargar o crear un escenario y luego poder modificarlo, se van a necesitar dos pantallas distintas.

- */index.html*: Pantalla principal, donde se podrá crear un nuevo escenario o cargar uno previamente guardado.
- */network/{id}*: Pantalla donde se renderizará el escenario y se podrá modificar de forma interactiva.

Una necesidad clara es la de ofrecer una interfaz donde el usuario vea la configuración del escenario de forma clara y concisa. Es por eso que se ha optado por usar una librería de JavaScript, *Cytoscape.js*, que facilita la resolución del problema, puesto que ayuda a la representación de los nodos y las conexiones de red mediante un grafo interactivo.

Si inspeccionamos qué funcionalidades se necesitan en el frontend, nos encontramos con que el proyecto es de mayor envergadura de lo que pudiera parecer en principio. A continuación se muestra una lista de funcionalidades, cada una compleja y que supone un reto en sí misma:

- Validar entradas de usuario.
- Gestión de nodos.
- Modificar la configuración de red de los nodos manteniendo consistente el escenario.
- Especificar qué mensajes manda qué maestro a qué esclavo.

4.2 Entorno de Desarrollo

Es importante indicar el entorno de desarrollo en el que se ha llevado a cabo el proyecto para garantizar la reproducibilidad y la compatibilidad con futuras versiones. El desarrollo se ha realizado en *Ubuntu 22.04.5 LTS* con kernel *6.8.0-45-generic*. Se empleó *Visual Studio Code* (1.85.1), junto a *Python 3.10.12* y *JavaScript ES11*. Las versiones de herramientas y librerías se resumen en las tablas 4.2 y 4.3.

Tabla 4.2 Librerías Usadas en el Proyecto.

Librería	Lenguaje	Versión
Flask	Python	3.0.3
Pandas	Python	2.2.3
Pymodbus	Python	3.7.2
Pytest	Python	8.3.3
Waitress	Python	3.0.0
Cytoscape.js	JavaScript	3.18.1
ipaddr.js	JavaScript	2.2.0

Tabla 4.3 Herramientas Externas Usadas en el Proyecto.

Herramienta	Versión
Docker Cli	27.3.1
Docker Engine	27.3.1
Docker Compose	2.29.7
Tcpdump	4.99.1
Libcap	1.10.1
OpenSSL	3.0.2

4.3 Arquitectura de la Solución

Teniendo ya claro cómo va a ser el escenario, se deja un diagrama de paquetes que ejemplifique cómo es la estructura de directorios del sistema. En la figura 4.1 se puede ver cómo se ha estructurado el proyecto. Cabe destacar que al diagrama le falta otra parte, referente a los ficheros de configuración específicos de cada contenedor. Esa se detallará en una futura sección.

La arquitectura del sistema se ha diseñado siguiendo un enfoque modular, lo que facilita la escalabilidad y el mantenimiento del proyecto. A continuación, se describen los principales componentes y su funcionalidad:

- **Protocolos de Comunicación**: El sistema soporta la emulación de protocolos industriales, como Modbus, tanto en modo maestro como esclavo. Los archivos `master.py` y `slave.py` implementan las funcionalidades correspondientes.
- **Dockerización**: Para garantizar la portabilidad y consistencia del entorno de ejecución, se han creado archivos `Dockerfile` que permiten la generación de imágenes Docker. Además, el módulo `docker_compose_generator.py` facilita la creación de archivos `docker-compose.yml` para la orquestación de múltiples contenedores.
- **Interfaz Web**: La interfaz gráfica del sistema se implementa mediante un servidor web en `web.py`, que utiliza plantillas HTML ubicadas en el directorio `templates`. Los recursos estáticos, como hojas de estilo CSS, scripts JavaScript e imágenes, se almacenan en el directorio `static`.
- **Gestión de Escenarios**: El directorio `scenarios` contiene diferentes configuraciones de escenarios de emulación. Los archivos `config.json` y `config.yml` contienen la información del escenario. El módulo `scenario_handler.py` se encarga de gestión de estos escenarios.
- **Núcleo del Emulador**: El archivo `main.py` actúa como punto de entrada principal del sistema, mientras que `runner.py` coordina la ejecución de los escenarios y la interacción entre los diferentes componentes.
- **Adaptadores y Utilidades**: El módulo `cytoscape_adapter.py` proporciona integración con la librería Cytoscape para la visualización de redes. El código fuente principal del proyecto se encuentra

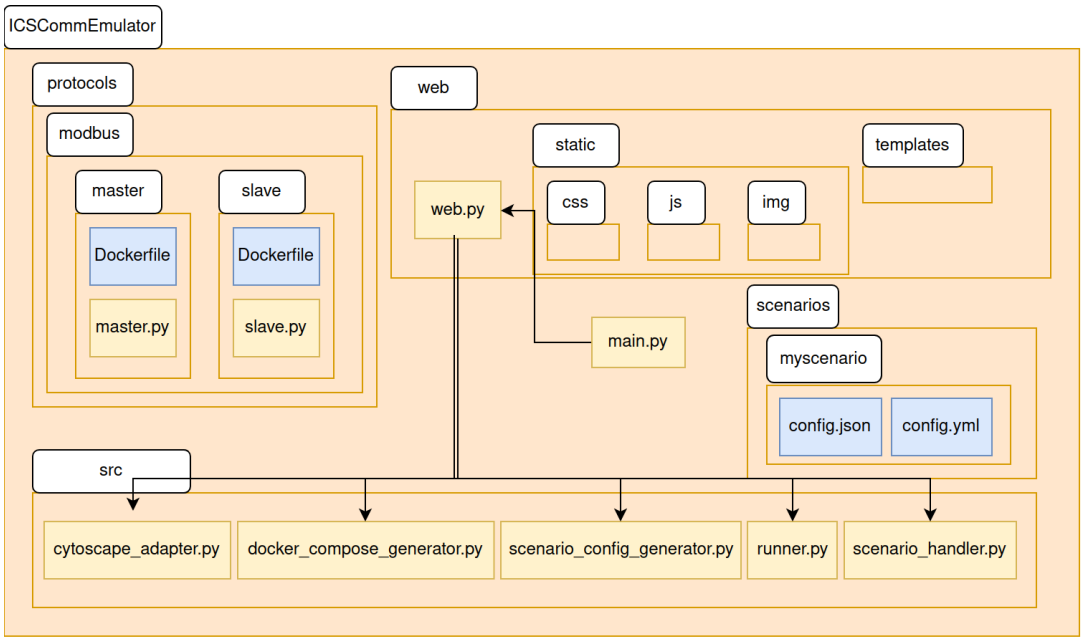

Figura 4.1 Diagrama de componentes: Backend y Core.

en el directorio `src`.

Esta estructura modular no solo permite una fácil extensión del sistema para soportar nuevos protocolos y escenarios, sino que también facilita la implementación y el uso en diversos entornos gracias a la combinación de Docker y una interfaz web intuitiva.

4.4 Core

El Core es el componente más importante del sistema, ya que es el encargado de dar la funcionalidad del sistema. En este desarrollo se le ha dado mucha importancia a la claridad y a la modularidad del código, lo cual ha facilitado la codificación y depuración de errores.

Con la estructura identificada, la implementación se ve enormemente facilitada. Esta vez vamos a empezar desde el principio, especificando la implementación y justificando las decisiones tomadas.

4.4.1 Docker Compose

El primer paso es definir el formato que tendrá el fichero *docker-compose.yml* de nuestro proyecto. A grandes rasgos consta de dos secciones relevantes para este proyecto: *networks* y *services* Ahora se va a entrar en detalle en cada sección.

Networks

Este bloque tiene como función ubicar a todos los contenedores en la misma subred para que cuando se capture el tráfico, tenga las direcciones MAC e IP correctas.

Un ejemplo de definición sería el expuesto en el listado 4.1.

Código 4.1 Ejemplo de network.

```
networks:
  icscommemulator:
    ipam:
      config:
      - subnet: 192.168.45.0/24
    name: icscommemulator
```

La configuración es bastante directa e intuitiva. *Networks* es la palabra reservada en el *docker-compose.yml* que sirve para comenzar la lista de redes definidas.

En este caso solo hay una con el nombre *icscommemulator*, que se guardará con dicho nombre. *Ipam* (IP Address Management)[14], especifica que la subred que se quiere es la 192.168.45.0/24.

Services

Este es el bloque principal de un fichero *docker-compose.yml*. En él se definen los servicios que se van a lanzar. En nuestro caso, los contenedores que emularán los dispositivos Modbus. Cada servicio tiene un nombre, que será el identificador del nodo, y una serie de configuraciones.

En nuestro caso hace falta configurar todos los contenedores de forma específica. Se muestra la configuración de un nodo maestro en el listado 4.2.

Código 4.2 Ejemplo de nodo maestro.

```
modbus_master_0:
  build:
    context: ./protocols/modbus/master
    dockerfile: Dockerfile.master
  container_name: modbus_master_container_0
  depends_on:
    modbus_slave_0:
      condition: service_healthy
    modbus_slave_1:
      condition: service_healthy
  environment:
  - PYTHONUNBUFFERED=1
  image: modbus_master_image
  networks:
    icscommemulator:
      ipv4_address: 192.168.45.10
      mac_address: E4:A8:DF:D1:11:0A
  volumes:
  - /tmp/ICSCommEmulator/masters/0/master.csv:/app/master.csv:ro
```

En el listado se identifican muchos campos que se deben explicar:

- **modbus_master_0**: Este es el nombre del servicio. En este caso, se trata del nodo maestro Modbus.
- **build**:
  - **context**: Especifica el directorio de contexto para la construcción de la imagen Docker. En este caso, es `./protocols/modbus/master`.
  - **dockerfile**: Indica el archivo Dockerfile a utilizar para construir la imagen. Aquí se usa `Dockerfile.master`.
- **container_name**: Define el nombre del contenedor. En este ejemplo, es `modbus_master_container_0`.
- **depends_on**: Especifica las dependencias del servicio. El nodo maestro depende de los nodos esclavos `modbus_slave_0` y `modbus_slave_1`, y solo se iniciará cuando estos estén en un estado saludable (`service_healthy`).
- **environment**: Define las variables de entorno para el contenedor. Aquí se establece `PYTHONUNBUFFERED=1`, lo que desactiva el almacenamiento en búfer de la salida estándar de Python.
- **image**: Especifica la imagen Docker a utilizar. En este caso, es `modbus_master_image`.
- **networks**: Configura la red a la que se conectará el contenedor. En este ejemplo, se conecta a la red `icscommemulator` con una dirección IPv4 específica (`192.168.45.10`) y una dirección MAC (`E4:A8:DF:D1:11:0A`).
- **volumes**: Monta un volumen en el contenedor. En este caso, se monta el archivo de configuración del nodo maestro.

También contamos con la configuración de un esclavo, la cual se muestra en en el listado 4.3

Código 4.3 Ejemplo de nodo esclavo.

```
modbus_slave_0:
```

```
    build:
      context: ./protocols/modbus/slave
      dockerfile: Dockerfile.slave
    container_name: modbus_slave_container_0
    environment:
    - PYTHONUNBUFFERED=1
    expose:
    - '502'
    healthcheck:
      interval: 10s
      retries: 3
      start_period: 10s
      test:
      - CMD-SHELL
      - test -f /app/app_running.lock
      timeout: 5s
    image: modbus_slave_image
    networks:
      icscommemulator:
        ipv4_address: 192.168.45.101
        mac_address: E4:A8:DF:D1:11:1B
    volumes:
    - /tmp/ICSCommEmulator/slaves/0/slave.yaml:/app/slave.yaml:ro
```

Las diferencias con el nodo maestro son:

- **depends_on**: El nodo esclavo no tiene dependencias especificadas, mientras que el nodo maestro depende de todos los nodos esclavos.
- **expose**: El nodo esclavo expone el puerto 502, puerto estándar de Mobus, lo que permite que otros servicios se comuniquen con él a través de este puerto. El nodo maestro no tiene esta configuración.
- **healthcheck**: El nodo esclavo incluye una configuración de verificación de salud (`healthcheck`) que define cómo y cuándo se verifica el estado del contenedor. Esta configuración incluye:
  - **interval**: El intervalo de tiempo entre verificaciones, en este caso, 10 segundos.
  - **retries**: El número de intentos fallidos antes de considerar que el contenedor no está saludable, aquí son 3 intentos.
  - **start_period**: El período de tiempo durante el cual las verificaciones fallidas no cuentan hacia el límite de intentos, en este caso, 10 segundos.
  - **test**: El comando que se ejecuta para verificar el estado del contenedor. Aquí se verifica la existencia del archivo `/app/app_running.lock`.
  - **timeout**: El tiempo máximo que se espera para que el comando de verificación se complete, en este caso, 5 segundos.
- **volumes**: El nodo esclavo monta un archivo de configuración diferente (`slave.yaml`) en comparación con el nodo maestro (`master.csv`).

4.4.2 Docker Images

En la sección anterior se ha explicado cuál va a ser el formato del docker-compose. Aquí se va a entrar en detalle en las dos imágenes desarrolladas en el proyecto.

Como se observa en la figura 4.1, tenemos un directorio donde se encuentra el protocolo Modbus. Dentro de este directorio, se encuentran dos subdirectorios, uno para el maestro y otro para el esclavo. En cada uno de estos directorios se encuentra un Dockerfile que se encarga de construir la imagen y un script principal.

Dockerfiles

A fin de tener la mayor cantidad de contenedores corriendo, estos Dockerfiles se han diseñado para generar imágenes del menor tamaño posible, por lo que se han seguido las recomendaciones de Docker para ello. Se ha minimizado el espacio ocupado en disco por las imágenes, siendo de 52.7MB y 55.3MB para el maestro y el esclavo, respectivamente. Esto es inferior a otras imágenes que se pueden encontrar en Docker Hub, con tamaños de entre 60 y 100 MB. En el listado 4.4 se muestra el Dockerfile del nodo maestro y en el listado 4.5 el del nodo esclavo.

Código 4.4 Dockerfile del maestro.

```
# Use an official Python runtime as a parent image
FROM python:3.10.0-alpine

# Set the working directory in the container to /app
WORKDIR /app

# Add the main script into the container at /app
COPY ./master.py /app

# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir pymodbus

# Run master.py when the container launches
ENTRYPOINT ["python", "master.py"]
```

Código 4.5 Dockerfile del esclavo.

```
# Use an official Python runtime as a parent image
FROM python:3.10.0-alpine

# Set the working directory in the container to /app
WORKDIR /app

# Add the main script into the container at /app
COPY ./slave.py /app

# Install any needed packages
RUN pip install --no-cache-dir pymodbus pyyaml

# Run slave.py when the container launches
ENTRYPOINT ["python", "slave.py"]
```

Ambas imágenes son muy parecidas, ya que solo se diferencian en el script que se copia y en las librerías que se instalan. En el caso del nodo maestro, solo se necesita la librería *pymodbus*, mientras que en el nodo esclavo se necesita también *pyyaml*.

Scripts

El script del maestro es muy sencillo, ya que solo se encarga de enviar mensajes Modbus. Esos mensajes están descritos en un archivo CSV que se monta en el contenedor. En el listado 4.6 se muestra un ejemplo de este archivo.

Código 4.6 Ejemplo de master.csv.

```
count,function_code,interval,ip,port,recurrent,slave_id,start_address,timestamp,values
3,3,5,192.168.45.100,502,True,1,0,0,[]
2,1,4,192.168.45.100,502,True,1,0,1,[]
1,3,5,192.168.45.101,502,True,2,0,0,[]
3,1,4,192.168.45.101,502,True,2,0,1,[]
```

El script del esclavo requiere algo más de configuración, ya que se tiene que configurar la respuesta a los mensajes del maestro. Estos mensajes están descritos en un archivo YAML que se monta en el contenedor. En el listado 4.7 se muestra un ejemplo de este archivo.

Código 4.7 Ejemplo de slave.yaml.

```
  coils:
  type: sequential
  values:
  - 1
  - 0
  - 1
discrete_inputs:
  type: sequential
  values: ''
holding_registers:
  type: sequential
  values:
  - 3
identity:
  major_minor_revision: '2.5'
  model_name: MagicMaster 3000
```

```
    product_code: TF98765
    product_name: Controlador de Automatizacin Mgica
    user_application_name: Control de Procesos Mgicos
    vendor_name: TechnoFantasy Inc.
    vendor_url: https :// www.technofantasy.com
  input_registers :
    type: sequential
    values: ''
ip: 192.168.45.101
mac: E4:A8:DF:D1:11:1B
port: 502
slave_id: 2
```

El script diferencia entre registros definidos de forma secuencial o dispersa. En el caso de los registros secuenciales, se especifica el valor de cada uno de ellos. En el caso de los registros dispersos, se crea un diccionario "dirección:valor".

La librería utilizada para gestionar los mensajes de Modbus en Python (Pymodbus) presenta un comportamiento indeseado al definir los registros en el esclavo: los índices empiezan en 1 en lugar de en 0. Es decir, cuando se guarda un registro con índice N (posición N) en Python, lo estará guardando en el índice N (posición N-1) en Pymodbus. Al recuperarlos recupera lo esperado (si se le pide M, recupera M+1) De este modo, a la hora de guardar los registros hay que tener en cuenta este detalle e incrementar en uno la dirección de los registros.

4.4.3 Módulo docker_compose_generator.py

Este módulo tiene un papel muy importante, que es pasar de la configuración del escenario, descrito más adelante en 4.5.2, a un fichero *docker-compose.yml*. Para ello se ha considerado apropiada una programación orientada a objetos en este módulo.

La interfaz con el módulo es muy sencilla, ya que solo se necesita crear una instancia de la clase *DockerComposeGenerator* y llamar al método `parse` con la configuración del escenario. A continuación, se detallan los componentes y funcionalidades clave del módulo:

Clase DockerComposeGenerator

La clase `DockerComposeGenerator` es la encargada de generar y validar archivos de configuración de Docker Compose. Los atributos principales de la clase incluyen:

- `services` (dict): Almacena la configuración de los servicios.
- `networks` (dict): Almacena la configuración de las redes.
- `protocol` (str): Nombre del protocolo utilizado en la configuración de Docker Compose.
- `ip_base` (ipaddress.IPv4Address): Dirección IP base para la red.
- `path` (str): Ruta al archivo Docker Compose.
- `config_path` (str): Ruta a los archivos de configuración.
- `last_ip` (ipaddress.IPv4Address): Última dirección IP asignada para la asignación dinámica.

Métodos de la Clase

- `__init__`: Inicializa la clase con el protocolo, la ruta del archivo y la ruta de configuración.
- `add_network`: Añade una red a la configuración de Docker Compose.
- `add_node`: Añade un nodo (servicio) a la configuración de Docker Compose.
- `generate`: Genera el archivo YAML de Docker Compose.
- `validate`: Valida el archivo de Docker Compose generado.
- `validate_file`: Método estático para validar un archivo de Docker Compose dado.
- `_validate_file`: Método estático auxiliar para validar un archivo de Docker Compose utilizando la CLI de Docker Compose.
- `parse`: Genera una configuración de Docker Compose basada en el escenario proporcionado.
- `get_dependencies`: Obtiene las dependencias para los nodos maestros.
- `is_master`: Verifica si un nodo es un nodo maestro.

- `is_slave`: Verifica si un nodo es un nodo esclavo.

Ejemplo de Uso

A continuación, se muestra un ejemplo de cómo utilizar la clase `DockerComposeGenerator` para generar un archivo *docker-compose.yml* a partir de una configuración de escenario:

Código 4.8 Ejemplo de uso de docker-compose-generator.py.

```
if __name__ == "__main__":
  scenario = {
      "protocol": "modbus",
      "ip_network": "172.28.0.0/16",
      "nodes": [
          {"role": "master", "ip": "172.28.0.2"},
          {"role": "slave", "ip": "172.28.0.3"},
          {"role": "slave", "ip": "172.28.0.4"},
      ],
  }
  generator = DockerComposeGenerator(
      "modbus", "docker-compose.yml", "/tmp/ICSCommEmulator"
  )
  generator.parse(scenario)
```

Este ejemplo crea una instancia de `DockerComposeGenerator` con el protocolo "modbus", la ruta del archivo *docker-compose.yml* y la ruta de configuración */tmp/ICSCommEmulator*. Luego, llama al método `parse` con la configuración del escenario para generar el archivo de Docker Compose.

4.4.4 Módulo scenario_config_generator.py

En el diseño se identificó la necesidad de gestionar los archivos de configuración de los nodos maestros y esclavos. Para ello, se ha desarrollado el módulo `scenario-config-generator.py`, que se encarga de generar y validar estos archivos.

Tras estudiar las posibilidades, se consideró que se debía usar una estructura de directorios bajo */tmp/ICS-CommEmulator* para almacenar los archivos de configuración. En este directorio se crean subdirectorios para cada nodo, donde se almacenan los archivos de configuración específicos.

Cuando se mencionó la figura 4.1, se dijo que faltaba una parte referente a los ficheros de configuración específicos de cada contenedor. Esta sección se encarga de explicar cómo se han implementado. Para ello en la figura 4.2 se especifica la estructura de directorios generada por el módulo.

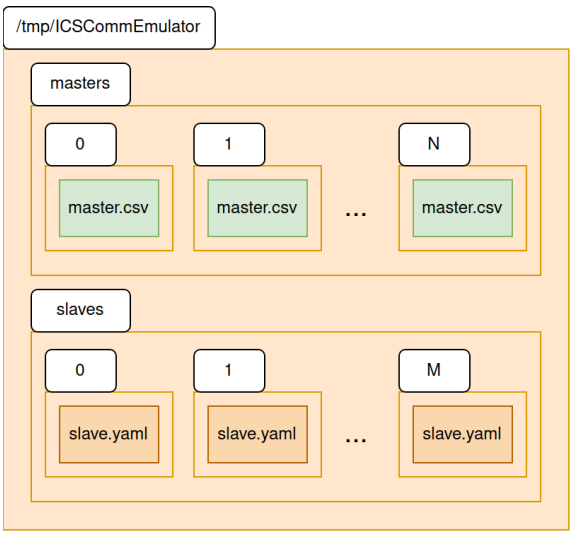

Figura 4.2 Estructura de directorios de /tmp/ICSCommEmulator.

Para el uso de este módulo, se ha desarrollado la clase `ScenarioConfigGenerator`, que se encarga de generar los archivos de configuración para los nodos maestros y esclavos. A continuación, se detallan los componentes y funcionalidades clave del módulo:

Clase ScenarioConfigGenerator

La clase `ScenarioConfigGenerator` es la encargada de generar los archivos de configuración para los nodos maestros y esclavos. Los atributos principales de la clase incluyen:

- `scenario` (dict): Diccionario que contiene la configuración del escenario.
- `config_path` (str): Ruta a los archivos de configuración.

Métodos de la Clase

- `__init__`: Inicializa la clase con el escenario y la ruta de configuración.
- `_convert_to_int`: Método estático para convertir una cadena a un entero si es posible.
- `_craft_master`: Crea archivos de configuración para los nodos maestros.
- `_craft_slave`: Crea archivos de configuración para los nodos esclavos.
- `clean`: Limpia la ruta de configuración eliminando los archivos existentes.
- `generate`: Genera los archivos de configuración para el escenario.

Ejemplo de Uso

A continuación, se muestra un ejemplo de cómo utilizar la clase `ScenarioConfigGenerator` para generar archivos de configuración a partir de una configuración de escenario:

Código 4.9 Ejemplo de uso de scenario-config-generator.py.

```
scenario = {
"nodes": [
{"role": "master", "messages": [...]},
{"role": "slave", ...}
]
}
generator = ScenarioConfigGenerator(scenario, "/tmp/ICSCommEmulator")
generator.generate()
```

Este ejemplo crea una instancia de `ScenarioConfigGenerator` con la configuración del escenario y la ruta de configuración. Luego, llama al método `generate` para generar los archivos de configuración.

Para evitar repetición en la documentación, el formato que ha de tener cada nodo es un diccionario con el formato expuesto de cada archivo de configuración. En el caso del nodo maestro, código 4.6, y en el del esclavo, código 4.7.

4.4.5 Módulo runner.py

Aunque no se haya dicho de forma explícita, se necesita un script que sirva de interfaz para la ejecución de Docker Compose y Tcpdump de forma coordinada. De esta necesidad nace el módulo `runner.py`.

Este módulo usa una clase singleton, ScenarioRunner, la cual solo permite que haya un escenario corriendo a la vez. Se ha decidido que solo se pueda ejecutar un escenario a la vez para evitar problemas de concurrencia.

Este módulo tiene tres funciones que actúan de interfaz para el resto de módulos.

start

En esta función asegura que solo se ejecutará el escenario propuesto si no se está ejecutando ningún escenario. Recibe como parámetro la ruta hacia el fichero docker compose, el tiempo de simulación y la ruta hacia el fichero de salida.

Con estos parámetros configura ScenarioRunner y lo corre. Para no bloquear el hilo que llama a esta función, se lanza otro hilo donde correrá la emulación.

Para lanzar el escenario se realizan las siguientes acciones:

1. Asegurar seguridad de hilos con un candado.
2. Marcar que se está corriendo un escenario.
3. Lanzar Docker Compose.
4. Lanzar Tcpdump en la interfaz de Docker.
5. Esperar a que la emulación termine.
6. Parar Docker Compose.
7. Parar Tcpdump.

Debe añadirse que Docker Compose borra los contenedores que crea pero no borra la red. Es por eso que se ha añadido una función que limpia la red creada.

Por último, Docker cursa tráfico adicional por las redes, como el protocolo Multicast DNS (mDNS) o Simple Service Discovery Protocol (SSDP). mDNS es un servicio diseñado para llevar a cabo la resolución de nombres en redes más pequeñas[15]. SSDP es un protocolo que sirve para la búsqueda de dispositivos Universal Plug and Play (UPnP) en una red. Utiliza UDP en unicast o multicast en el puerto 1900 para anunciar los servicios de un dispositivo[16]. Dado que nuestro objetivo es el realismo de la red, se ha configurado Tcpdump para que no escuche esos protocolos.

stop

Esta función solo se llama si el usuario pide desde la interfaz gráfica detener el escenario. La función fuerza la detención del escenario que se esté ejecutando, por lo que no recibe parámetros. El proceso es primero parar Docker Compose y luego Tcpdump.

status

En la configuración de Tcpdump se especfició que se guardase el tráfico en el fichero de salida periódicamente. De este modo se puede devolver información relevante para saber el estado de la emulación como el tamaño del pcap.

En este caso se han considerado necesarios los campos:

- Tiempo transcurrido.
- Tiempo total de emulación.
- Tamaño actual del pcap.
- Si el escenario está corriendo.

4.5 Backend

El Backend es el componente que coordina la vista con el modelo, es decir, la interfaz gráfica con el núcleo de la funcionalidad. Dado que este fue el segundo componente en ser codificado, se mantuvo el espíritu de modularidad anteriormente expuesto en el core.

4.5.1 Módulo web.py

Este módulo es el encargado de crear la aplicación web que ofrece sus servicios tanto de REST API como de interfaz gráfica. Para ello, como se ha mencionado anteriormente, se ha usado *Flask*, un framework minimalista de Python.

Anteriormente ya se nombraron los distintos endpoints, aquí se explicará en detalle cómo se han implementado.

Lo primero es entender la estructura de directorios que genera *Flask*. Como se puede ver en la figura 4.1, hay una carpeta *web* en la raíz del proyecto, donde se encuentran los ficheros de la aplicación web.

La subcarpeta *templates* es una carpeta reservada por el framework para colocar los ficheros Hypertext Markup Language (HTML). Esto es así porque *Flask* permite la renderización del HTML con código Python dentro[1], similar a *JavaServer Pages (JSP)*, visto a lo largo de la carrera.

También contamos con la subcarpeta *static*, donde se encuentran el código JavaScript, el CSS y las imágenes utilizadas. Como se ha mencionado anteriormente, se han usado las librerías *Cytoscape.js* y *ipaddr.js*

[1] https://www.geeksforgeeks.org/flask-rendering-templates/

en el Frontend. Al ser archivos de JavaScript que carga el usuario junto al HTML, para permitir que la aplicación pueda correr sin necesidad de acceso a internet, se han descargado y situado en la carpeta asignada para JavaScript.

A continuación se describen brevemente los endpoints desarrollados por este proyecto, ya vistos en la tabla 4.1.

GET /api/networks/

- Descripción: Devuelve una lista de redes disponibles en el sistema.
- Parámetros: Ninguno.
- Respuesta:
  - `200 OK`: Lista de redes. Ejemplo:
    ```
    [
      "demo",
      "prueba"
    ]
    ```
  - `500 Internal Server Error`: Error inesperado en el servidor.
- Funcionamiento: Este endpoint usa el módulo *scenario_handler.py* para saber todos los escenarios que existen.

POST /api/networks/

- Descripción: Crea un escenario con los parámetros especificados.
- Parámetros:
  - `projectName`: Nombre del proyecto.
  - `ipSubrange`: Rango IP de los nodos en el proyecto.
  - `protocol`: Protocolo que tendrá el escenario.
  - `masterNodes`: Número de maestros inicialmente en el escenario.
  - `slaveNodes`: Número de esclavos inicialmente en el escenario.
- Respuesta:
  - `200 OK`: Escenario creado con éxito.
  - `400 Bad Request`: El escenario tiene parámetros inválidos.
  - `500 Internal Server Error`: Error inesperado en el servidor.
- Funcionamiento: El escenario comprueba secuencialmente si:
  - El escenario no existe.
  - El valor de maestros o esclavos es correcto.
  - Están todos los parámetros en la petición.
  - El rango IP es válido.

  Una vez comprobado, crea el escenario usando el módulo *cytoscape_adapter.py* para la creación de la representación de la red y el módulo *scenario_handler.py* para guardarlo.

GET /api/networks/name

- Descripción: Devuelve la representación JSON del proyecto.
- Parámetros:
  - `name`: Nombre del proyecto.
- Respuesta:
  - `200 OK`: Escenario en representación JSON de *Cytoscape.js*.
  - `404 Not Found`: No se ha encontrado el escenario.
  - `500 Internal Server Error`: Error inesperado en el servidor.
- Funcionamiento: Este endpoint consulta la base de datos para obtener todas las redes disponibles. Si se proporciona un parámetro de filtro, las redes se filtrarán en consecuencia.

PUT /api/networks/name

- Descripción: Actualiza la configuración de un escenario.
- Parámetros: El escenario en la representación JSON que es capaz de generar y restaurar *Cytoscape.js*.
- Respuesta:
  - `200 OK`: Escenario actualizado con éxito.
  - `400 Bad Request`: Escenario inválido.
  - `500 Internal Server Error`: Error inesperado en el servidor.
- Funcionamiento: Se valida el escenario usando el módulo *cytoscape_adapter.py* y posteriormente se guarda el escenario usando el módulo *scenario_handler.py*.

GET /api/run/

- Descripción: Obtiene las métricas de la ejecución actual.
- Parámetros: Ninguno
- Respuesta:
  - `200 OK`: Métricas de la ejecución. Ejemplo:
    ```
    [
      "elapsed_seconds": 2,
      "total_seconds": 15,
      "pcap_size": 3093,  # bytes
      "running": true,
    ]
    ```
  - `500 Internal Server Error`: Error inesperado en el servidor.
- Funcionamiento: Llama a la api ofrecida por *runner.py* y devuelve sus resultados.

POST /api/run/name

- Descripción: Arranca el escenario especificado.
- Parámetros:
  - `name`: Nombre del proyecto.
- Respuesta:
  - `200 OK`: Información de la ejecucición:
    ```
    [
      "message": "Scenario running",
      "simulation_time": simulation_time,
      "file_path": file_path,
    ]
    ```
  - `404 Not Found`: Escenario no encontrado.
  - `500 Internal Server Error`: Error inesperado del servidor.
- Funcionamiento: Usando los módulos *docker_compose_generator.py*, *scenario_config_generator.py* y *runner.py*, se crea el escenario, se generan los ficheros de configuración y se arranca la emulación.

DELETE /api/run/

- Descripción: Fuerza la detención del escenario actual.
- Parámetros: Ninguno.
- Respuesta:
  - `200 OK`: OK.
  - `500 Internal Server Error`: Error inesperado en el servidor.
- Funcionamiento: Usa la api de *runner.py* para detener los recursos utilizados por la emulación, como Docker Compose y Tcpdump.

GET /index.html

- Descripción: Devuelve la página principal.

- Parámetros: Ninguno.
- Respuesta:
  - `200 OK`: OK.
- Funcionamiento: Devuelve el archivo HTML principal.

GET /networks/id

- Descripción: Devuelve el escenario especificado.
- Parámetros:
  - `id`: Id del proyecto (usualmente el nombre).
- Respuesta:
  - `200 OK`: OK.
  - `404 Not Found`: No se ha encontrado el escenario.
- Funcionamiento: Devuelve el HTML con la información del escenario.

4.5.2 Módulo cytoscape_adapter.py

El proyecto necesita una forma de convertir la representación de la red de *Cytoscape.js* a un formato que pueda ser procesado por el sistema. Para ello, se ha desarrollado el módulo `cytoscape-adapter.py`.

Este módulo responde ante el problema de "qué pasaría si quisiésemos cambiar el frontend". Con carácter general no se debe hacer optimización preventiva, pero en este caso se han probado distintas librerías para el frontend hasta encontrar la definitiva. Durante ese periodo de prueba, se ha mantenido la funcionalidad del sistema gracias a este módulo.

La interfaz del módulo es sencilla y consta de un conjunto de funciones que se detallarán a continuación.

Función validate_cytoscape_scenario

Esta función se encarga de validar la representación de la red de *Cytoscape.js*. Recibe el JSON de *Cytoscape.js* y el nivel de exhaustividad con el que se quiere realizar la validación (hasta error o solo warning).

La función termina devolviendo una lista con todos los errores o avisos generados al validar la representación de la red. Si la lista está vacía es que no ha detectado nada. Por mantenibilidad, este código de validación es análogo al implementado en la validación del lado del cliente.

Las comprobaciones se pueden apreciar en la tabla 4.4.

Tabla 4.4 Validaciones realizadas para un escenario.

Severidad	Validación
ERROR	Los nodos tienen IDs duplicados. Nota: no confundir id y nombre.
ERROR	Hay enlaces entre nodos con el mismo rol.
ERROR	La IP de los nodos no está en el rango de la red.
WARNING	Un esclavo determinado no tiene ningún registro definido.
WARNING	La comunicación entre dos nodos no tiene mensajes.
WARNING	Un nodo no tiene enlaces.

Función parse_cytoscape_json

Esta función recibe un diccionario obtenido del JSON que representa el escenario y lo convierte en la representación final de dicho escenario.

Aquí no se realiza la validación del escenario, por lo que es necesario llamar previamente a la función *validate_cytoscape_scenario* descrita en la Subsubsección 4.5.2.

La representación final del escenario está en YAML y en el Código 4.10 tenemos un ejemplo.

Adicionalmente, los esclavos cuentan con un campo llamado *identity*. En este campo el usuario podría haber introducido valores inválidos cuyo manejo hiciese el código más complejo de lo necesario. Por ello, se sanea la entrada con la expresión regular *r"[a-zA-Z0-9\s.,/:_-]"*, la cual permite letras mayúsculas y minúsculas, números y los caracteres especiales *.,/:_-*.

Código 4.10 Ejemplo de configuración YAML.

```yaml
ip_network: 192.168.1.0/24
nodes:
- id: master_0
  ip: 192.168.1.2
  messages:
  - count: 1
    function_code: 1
    interval: null
    ip: 192.168.1.4
    port: 502
    recurrent: false
    slave_id: 1
    start_address: 0
    timestamp: 0
    values: []
  - count: 1
    function_code: 1
    interval: 1
    ip: 192.168.1.4
    port: 502
    recurrent: true
    slave_id: 1
    start_address: 1
    timestamp: 2
    values: []
  name: master_0
  role: master
- coils:
    type: sequential
    values:
    - 1
    - 0
  comment: ''
  discrete_inputs:
    type: sequential
    values: ''
  holding_registers:
    type: sequential
    values: ''
  id: slave_0
  identity:
    major_minor_revision: ''
    model_name: ''
    product_code: ''
    product_name: ''
    user_application_name: ''
    vendor_name: ''
    vendor_url: ''
  input_registers:
    type: sequential
    values: ''
  ip: 192.168.1.4
  name: slave_0
  port: 502
  role: slave
  slave_id: 1
protocol: modbus
```

Función generate_network_nodes

Esta función es la encargada de generar la representación que posteriormente procesará *Cytoscape.js*. Recibe el protocolo a usar, rango ip, número de esclavos y de maestros, y devuelve un diccionario con la representación de la red válida para *Cytoscape.js*.

Dicho diccionario se guarda a formato JSON con la función *save_scenario* del módulo *scenario_handler.py*. En este documento no se lista ningún fichero JSON por el formato poco compacto que tiene.

4.5.3 Módulo scenario_handler.py

Este módulo tiene un objeto muy importante, que es el de abstraer al resto del sistema de dónde están los ficheros de configuración y cómo se llaman. A continuación se describen las funcione sdel módulo.

Función save_scenario

Se le pasa como argumento el nombre del escenario y los datos del escenario antes de depurar. Dichos datos se guardan en JSON y su versión saneada en YAML. Para dicho saneamiento se hace uso de la función *parse_cytoscape_json* del módulo *cytoscape-adapter.py*.

Como se observa en la figura 4.1, se guardan dichos archivos en la carpeta *scenarios/nombre escenario*.

Función get_cytoscape_scenario

Esta función recibe el nombre del escenario y devuelve los datos de configuración del JSON del escenario.

Función get_python_scenario

Esta función recibe el nombre del escenario y devuelve los datos de configuración del YAML del escenario.

4.6 Frontend

Se cosidera Frontend a los dos scripts de JavaScript que se corren en el lado del cliente: *index.js* y *network.js*.

4.6.1 Script index.js

El archivo *index.js* gestiona la interfaz de usuario para la configuración y carga de escenarios de red. Incluye funciones para alternar entre carga de escenario o formulario de creación, validar y enviar datos del escenario al backend, y manejar la interacción con los elementos de la interfaz.

Para la creación de un escenario nos encontramos con un formulario con los campos descritos en 4.5.

Tabla 4.5 Campos de creación de un escenario.

Nombre del campo	Valor esperado	Verificado
Nombre del Proyecto	Cadena de texto	No
Subred IP	Formato CIDR válido (ej. 192.168.100.0/24)	Sí
Protocolo del Escenario	Selección de una lista desplegable	Sí
Número de nodos maestro	Entero positivo	Sí
Número de nodos esclavo	Entero positivo	Sí

Se implementan validaciones para asegurar que los datos introducidos en la creación de escenario, como subredes IP y nodos, sean correctos. Para la validación de las subredes IP se utiliza la librería *ipaddr.js*. Una vez validados, se envían a */api/networks* mediante una petición POST.

Además, permite obtener y listar escenarios existentes desde el backend con una solicitud GET a */api/networks*.

Para facilitar la explicación se incluyen las figuras 4.3 y 4.4, las cuales son diagramas de secuencia de la creación y de la carga de los escenarios, respectivamente.

4.6.2 Script network.js

El archivo *network.js* tiene muchas funcionalidades. A grandes rasgos se encarga de la gestión de la representación de la red usando la librería *Cytoscape.js* y de la interacción con el usuario.

A continuación se irá detallando el funcionamiento completo. Para no repetir capturas, se ha decidido mostrarlas en la sección 5.

Elementos visibles

La interfaz gráfica se comopone de varios elementos visibles en todo momento y otros que son visibles momentáneamente. Los elementos que siempre son visibles son los siguientes:

- Instrucciones básicas para el uso de la aplicación.

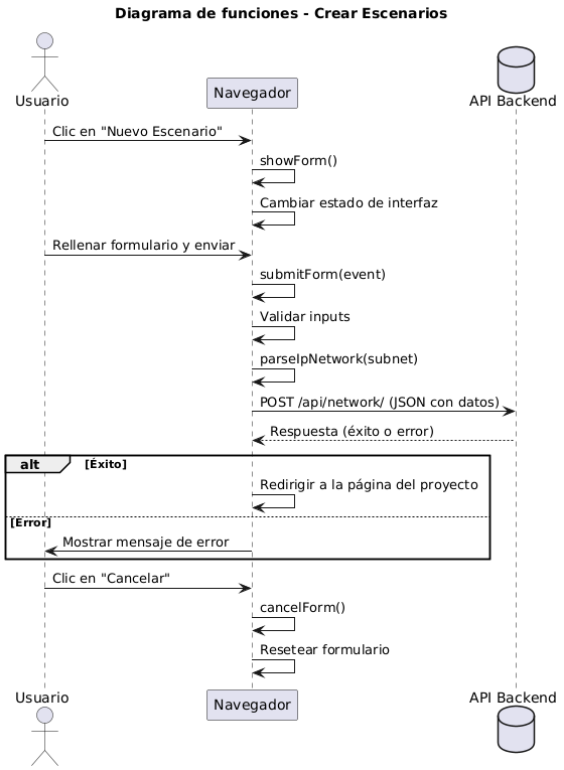

Figura 4.3 Diagrama de Secuencia para Crear un Escenario.

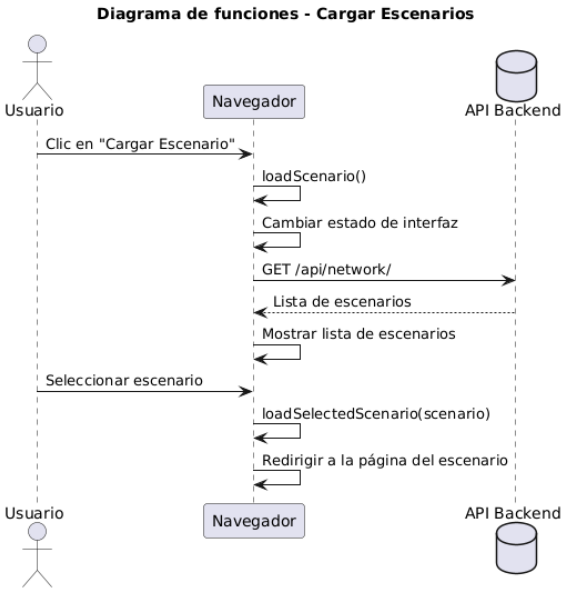

Figura 4.4 Diagrama de Secuencia para Cargar un Escenario.

- Botones de zoom in y zoom out.
- Botones de ejecutar y guardar.
- Lienzo con nodos.

El elemento más importante de los mencionados es el lienzo, al que el usuario dedicará la mayoría de su atención.

Al interactuar con él, irá abriendo distintas pestañas, las cuales son:

- `Configuración de los nodos`: Sirve para modificar atributos de los nodos (direcciónes MAC e IP, nombre, rol...).
  - `Configuración de los registros` (solo esclavos): Para configurar los registros de los esclavos existe una pestaña que se abre a partir de la genérica.
  - `Configuración de la identidad` (solo esclavos): En esta pestaña se puede configurar el campo identidad de un esclavo.
- `Configuración de los mensajes`: Aquí se pueden configurar los mensajes que mandará un maestro a un esclavo.
- `Confirmación de guardado`: Al guardar en caso de que existan anomalías en el escenario salta una pestaña requiriendo confirmación.
- `Solicitud de datos para la ejecución`: Antes de que se ejecute el escenario se necesita especificar la duración de la emulación.
- `Estado de la ejecución`: Durante la ejecución del escenario se nos presenta una pestaña donde iremos recibiendo información de la ejecución.

Acciones de Usuario

El usuario cuenta con varias acciones que puede realizar en la interfaz gráfica. Dado que el objetivo inicial era realizar una interfaz fácilmente utilizable, se han añadido las siguientes acciones:

- `Control + Z`: Permite rebobinar la última acción realizada.
- `Control + Y`: Permite deshacer una rebobinación.
- `Escape`: Cierra la pestaña que esté abierta o deselecciona el elemento seleccionado.
- `Pulsación corta`: Selecciona el elemento pulsado. En caso de que hubiese un nodo seleccionado y se esté pulsando sobre otro, los enlaza.
- `Pulsación larga`: Si se hace encima de un elemento abre su pestaña de configuración, en caso contrario, genera un nuevo nodo en esa ubicación.

Configuración de nodos

Al realizar una pulsación larga sobre un nodo se abre su pestaña de configuración. En esa pestaña de configuración nos encontramos con múltiples campos, los cuales dependen de si el nodo tiene rol de maestro o esclavo. En la tabla 4.6 se detallan los campos que se pueden encontrar en la configuración de cualquier nodo. Adicionalmente, en la tabla 4.7 nos encontramos con la configuración específica de un esclavo.

Tabla 4.6 Campos de configuración de un nodo.

Nombre del campo	Valor esperado	Verificado
Nombre	Cadena de texto	No
Comentario	Comentario opcional	No
Rol	Master o Slave	Sí
IP	Dirección IPv4 válida dentro del rango de red	Sí
MAC	Dirección MAC válida (formato XX:XX:XX:XX:XX:XX)	Sí

Tabla 4.7 Campos de configuración específicos de un esclavo.

Nombre del campo	Valor esperado	Verificado
Puerto	Entero entre 0 y 65535	Sí
ID Esclavo	Entero positivo único entre esclavos	Sí
Registros	Configurado con al menos un registro	No
Identidad	Configurado correctamente	No

Configuración de los registros

En la configuración de los registros se pueden configurar los registros de un esclavo. Para configurar los registros contamos con dos opciones, definir los registros de forma dispersa o de forma secuencial. Esto es para dar flexibilidad en la configuración de los registros.

Un ejemplo de configuración secuencial sería: "1, 0, 1, 0, 1, 1, 1" y el mismo ejemplo con una configuración dispersa sería: "0: 1, 1: 0, 2: 1, 3: 0, 4: 1, 5: 1, 6: 1", a fin de dar flexibilidad en dicho proceso. Esta decisión de aceptar ambos formatos viene tras una investigación de mercado, viendo que los dispositivos Modbus pueden tener ambos formatos.

Además, como ya se vio en la tabla 2.2, contamos con cuatro tipo de registros. Cada uno de ellos tiene un tipo de dato distinto, por lo que recae en el usuario el introducir los valores correctos.

Identidad

La identidad de un esclavo es un campo opcional. En caso de que se quiera introducir, se cuenta con los campos del estándar en su versión expandida:

- Versión del dispositivo.
- Nombre del modelo.
- Código del producto.
- Nombre del producto.
- Nombre de la aplicación.
- Nombre del vendedor.
- URL del vendedor.

Cabe destacar que se esperan cadenas de texto y no se valida ninguna en el lado del cliente. En el lado del servidor se sanean los caracteres para que sean válidos.

Este campo solo se usa en los mensajes con FC 43, por lo que no es necesario rellenarlo si no se va a usar.

4.6.3 Configuración de los mensajes

Para crear un enlace entre dos nodos se necesita realizar una pulsación corta sobre dos nodos con roles distintos. Posteriormente, para abrir la pestaña de configuración, se realiza una pulsación larga sobre el enlace.

En la pestaña de configuración de mensajes nos encontramos con una tabla donde podemos definir más mensajes y configurar los previamente creados. En la tabla 4.8 se detallan los campos que se pueden encontrar en la configuración de los mensajes.

Tabla 4.8 Validaciones realizadas para los parámetros de comunicación.

Nombre	Tipo de dato	Valor esperado
Marca de tiempo	Entero	Mayor o igual a 0
Periódico	Booleano	`True` o `False`
Intervalo	Entero	Mayor a 0
Function Code	Entero	1, 2, 3, 4, 5, 6, 15, 16 o 43
Dirección de Comienzo	Entero	Mayor o igual a 0
Cuenta	Entero	Mayor a 0
Valor(es)	Lista de enteros	Lista válida de valores asociados a registros

La idea detrás de la configuración escogida es que se pueda configurar cuándo se manda un mensaje esporádico y cuándo se manda un mensaje periódico y con qué frecuencia.

Cabe destacar que el sistema permite que se manden mensajes a registros no definidos, generando así las excepciones de Modbus.

Para más detalle acerca de los FCs se ruega consultar la tabla 2.1.

4.6.4 Confirmación de Guardado

Al pulsar el botón de guardar se valida la configuración del escenario antes de mandársela al servidor. Ambos lados, cliente y servidor, realizan una validación de los mismos campos. Las validaciones se encuentran en la tabla 4.4.

A diferencia de lo que hace el servidor, el Frontend sí informa la usuario de la existencia de anomalías, ya sea una alerta o un error.

En caso de error no se le permite al usuario continuar, ya que daría lugar a una configuración inválida. En caso de que sean alertas se le permite continuar.

4.6.5 Ejecución

Para comenzar la ejecución se le pide al usuario que introduzca el tiempo que durará la emulación. Esto es así porque al tener la posibilidad de mandar mensajes periódicos, no se puede determinar un tiempo de emulación.

Una vez comenzada la ejecución el script le pide información acerca de la ejecución al servidor y con ello se muestra en una pestaña la siguiente información:

- Tiempo actual de emulación.
- Tiempo total de emulación.
- Tamaño del pcap en bytes.

4.7 Casos de Uso

Dadas las funciones principales, tenemos los siguientes casos de uso:

- Creación de un nuevo escenario.
- Carga de un escenario previamente guardado.
- Guardado del escenario cargado.
- Emulación del escenario cargado.
- Detención de la emulación actual.

4.7.1 Creación de un Nuevo Escenario

La creación de un escenario es una funcionalidad esencial en el proyecto. El usuario completa un formulario con los campos descritos en la tabla 4.9. Al confirmar, el escenario se guarda el ficheor y se redirige a la págin web del escenario automáticamente. La lógica del proceso está representada en el diagrama de la figura 4.5.

Tabla 4.9 Campos requeridos para la creación de un escenario.

Campo	Tipo	Descripción
Nombre del proyecto	Cadena de texto	Identificador único del escenario
Subred IP	Formato IP CIDR	Rango de direcciones IP para el escenario
Protocolo	Cadena de texto	Campo reservado para compatibilidad futura
Número de maestros	Número entero	Cantidad inicial de nodos maestros
Número de esclavos	Número entero	Cantidad inicial de nodos esclavos

4.7.2 Carga de un Escenario

Esta funcionalidad permite al usuario cargar un escenario previamente guardado desde la interfaz gráfica. Al seleccionar la opción correspondiente, se muestra una lista de escenarios disponibles. Tras elegir uno, este se carga automáticamente. El diagrama de la figura 4.6 detalla el proceso.

4.7.3 Guardado de un Escenario Cargado

Una vez cargado un escenario, el usuario puede modificarlo y guardarlo. La información del escenario (nodos, configuraciones, posiciones, nombres, etc.) se almacena en archivos cuyo formato depende de las librerías utilizadas en el frontend. Aunque el guardado está influenciado por estas herramientas, la lógica subyacente es independiente, como se muestra en la figura 4.7.

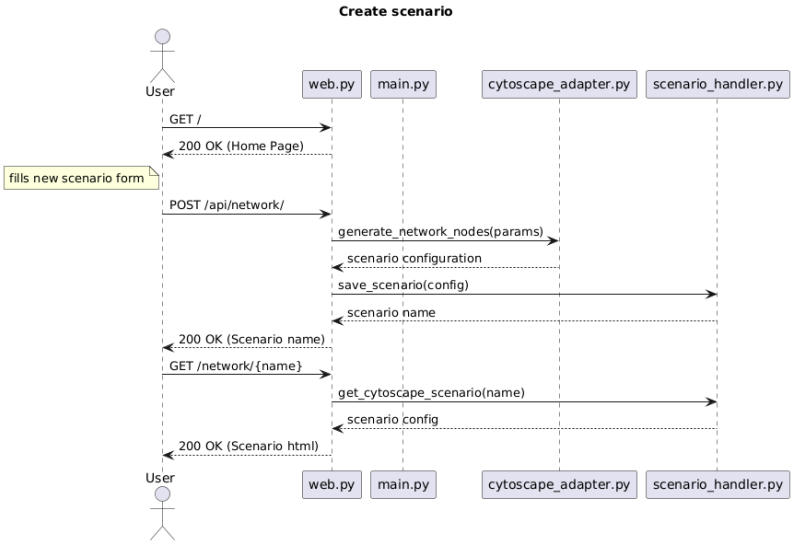

Figura 4.5 Diagrama de secuencia para la creación de un escenario.

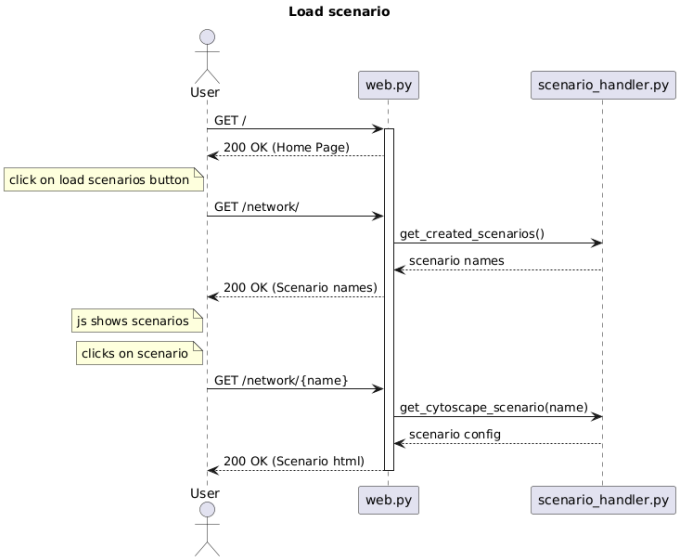

Figura 4.6 Diagrama de secuencia para la carga de un escenario.

4.7.4 Emulación del Escenario Cargado

Una vez guardado el escenario, el usuario puede iniciar su emulación, para lo que especifica la duración del proceso. La lógica de esta funcionalidad se divide en dos etapas, ilustradas en las figuras 4.8 y 4.9.

4.7.5 Detención de la Emulación

Por último, un escenario en ejecución se puede detener en cualquier momento. La figura 4.10 muestra el proceso de detención de la emulación.

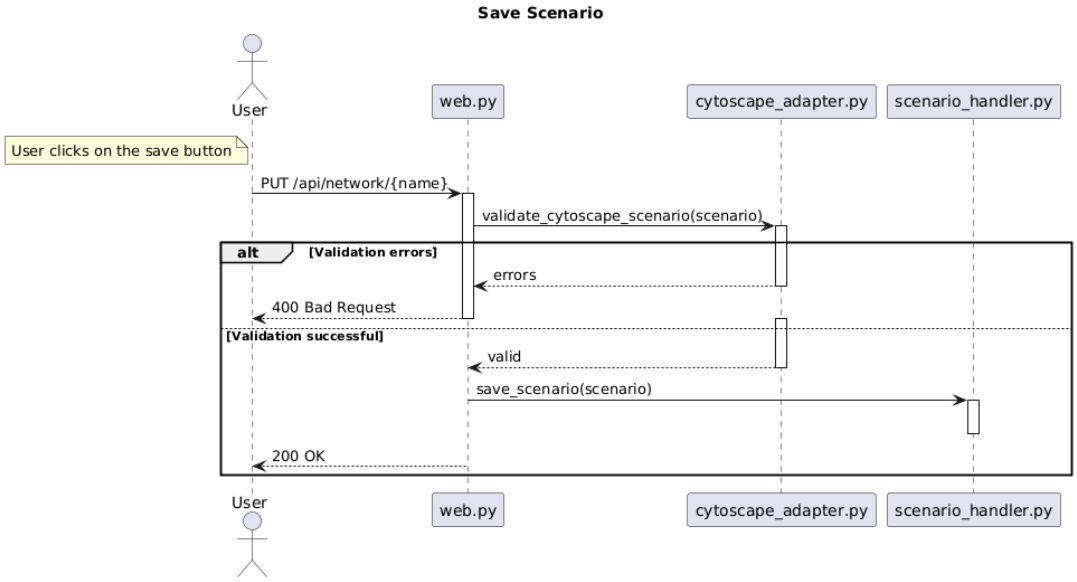

Figura 4.7 Diagrama de secuencia para el guardado de un escenario.

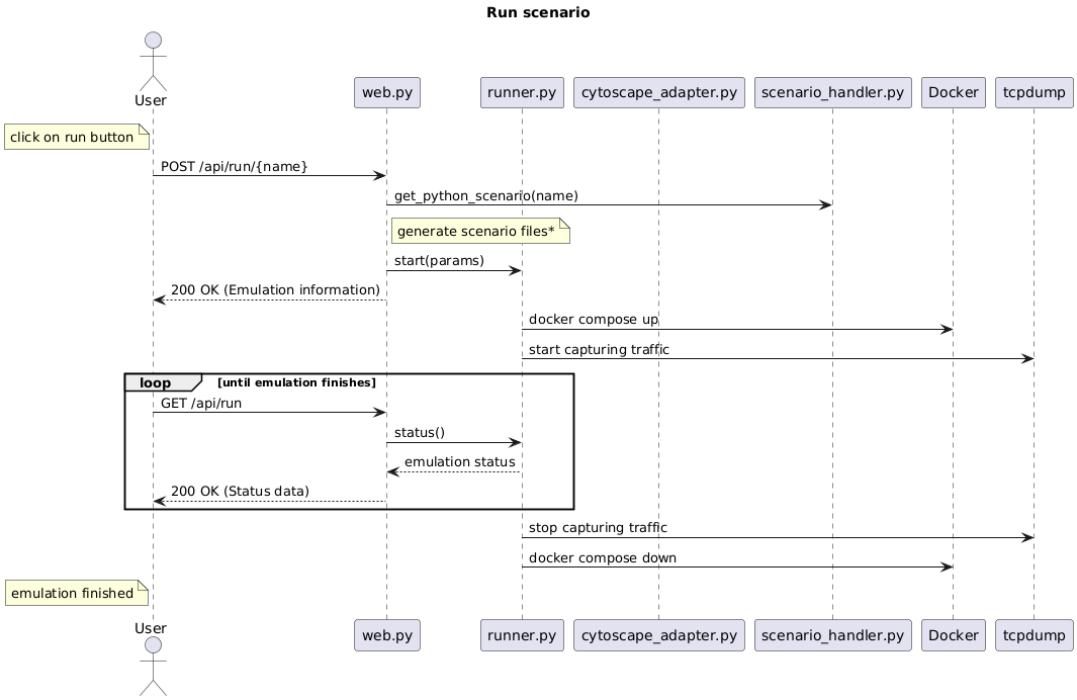

Figura 4.8 Diagrama de secuencia para la ejecución de un escenario.

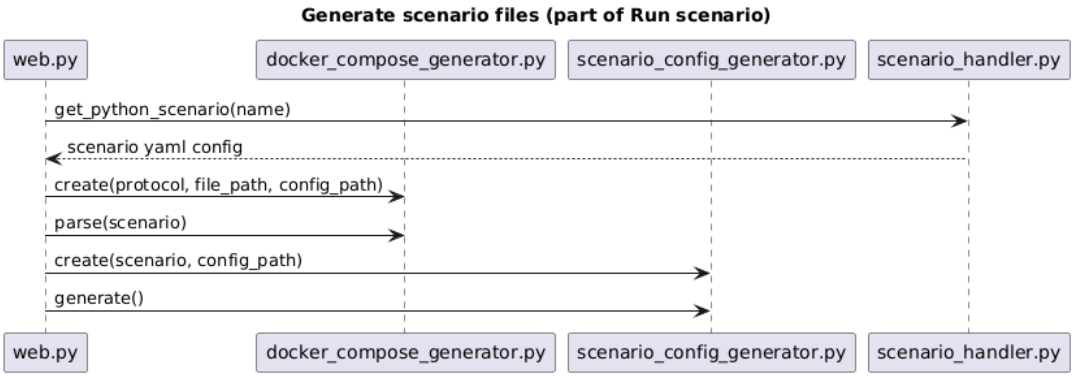

Figura 4.9 Diagrama de secuencia para la generación de archivos necesarios para la emulación.

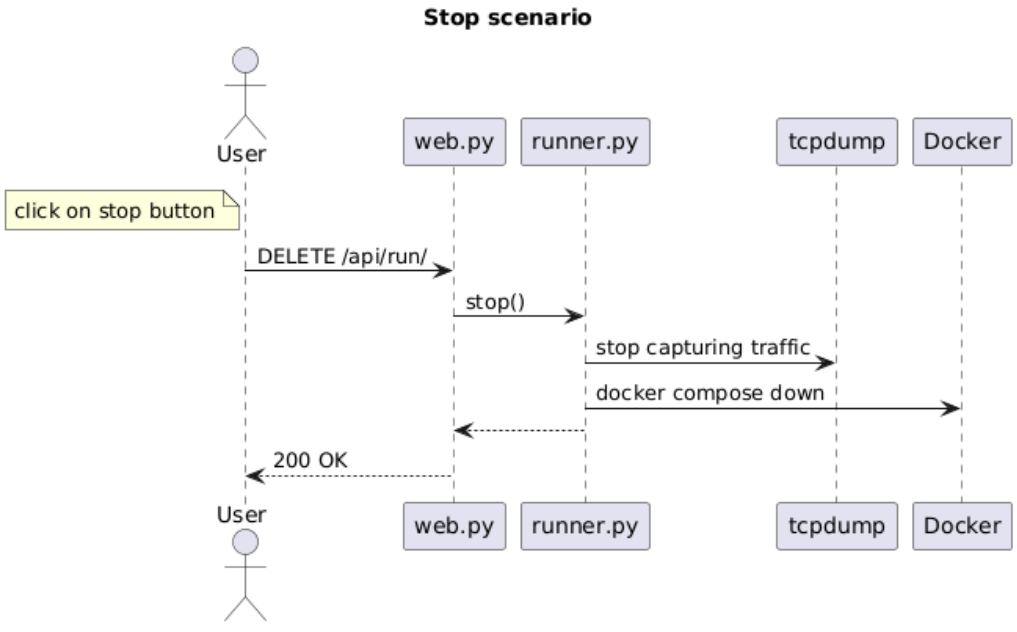

Figura 4.10 Diagrama de secuencia para la detención de la emulación.

5 Puesta en Ejecución y Ejemplos de Uso

5.1 Instalación

5.1.1 Obtención del código fuente

El primer paso es obtener el código fuente, que aunque puede ser obtenido de múltiples formas, la más sencilla es descargarlo desde el repositorio de GitHub:

```
curl -L0 \
https://api.github.com/repos/Nelnitorian/icscommemulator/tarball \
-o icscommemulator.tar.gz
```

Descomprimimos el archivo descargado:

```
tar -xvzf icscommemulator.tar.gz
```

5.1.2 Instalación de dependencias

Para instalar las dependencias necesarias se recomienda instalar mediante el script *install.sh*, el cual instala:

- La última versión de Docker y Docker Compose.
- Tcpdump.
- Python3.
- Dependencias de librerías de Python, que se incluyen en el archivo requirements.txt.

5.1.3 Ejecución del proyecto

Para ejecutar el proyecto, se debe ejecutar el script de python *main.py*. Para ello, ejecutamos en la terminal:

```
python3 main.py
```

Esto nos abrirá un servidor http en 127.0.0.1:8080, al que podremos acceder desde un navegador web entrando en la dirección http://127.0.0.1:8080.

5.2 Pruebas

Las pruebas son esenciales en cualquier proyecto porque garantizan la funcionalidad y la calidad del sistema desarrollado. Al probar el código, se minimizan errores que podrían comprometer el rendimiento o la experiencia del usuario, especialmente en proyectos con múltiples componentes interdependientes. Además, las pruebas son fundamentales para verificar que los requisitos iniciales se cumplan y que los cambios futuros no introduzcan problemas inesperados, asegurando la sostenibilidad del proyecto a largo plazo.

En algunos casos, realizar pruebas exhaustivas en todos los componentes puede ser innecesario y poco eficiente, especialmente si el proyecto tiene recursos limitados o plazos ajustados. Focalizar las pruebas únicamente en los componentes más complejos o críticos del sistema puede ser suficiente para garantizar la

estabilidad general. Al centrarse en estos puntos, se detectan los posibles fallos de mayor impacto sin invertir un tiempo excesivo en partes que son más simples o menos propensas a errores. Este enfoque balancea el costo y el beneficio, asegurando una validación adecuada del sistema sin comprometer la viabilidad del proyecto.

Puesto que el diseño del proyecto se hizo con la idea de ser modular, durante una ejecución del código principal es fácilmente identificable dónde está el fallo. Esto ha reducido los requisitos de pruebas unitarias, ya que la modularidad del código permite una depuración más sencilla.

5.2.1 Pruebas unitarias

En el proyecto se ha identificado como necesario testear los componentes sobre los que se tiene menos control y podrían dificultar la depuración. En este caso se han realizado pruebas unitarias sobre el generador de ficheros de configuración de Docker Compose y la comunicación entre un maestro y un esclavo.

Estas pruebas están alojadas en el directorio *tests/* y se ejecutan con el comando *python3 -m pytest*.

Docker Compose Generator

En esta prueba unitaria se usa el generador de Docker Composes para crear un archivo de configuración y añadirle nodos de forma controlada.

Posteriormente, se comprueba que en el fichero están los nodos generados correctamente.

Finalmente, se le pide a Docker Compose que use su validador para terminar la prueba.

Comunicación Maestro - Esclavo

En esta prueba unitaria se lanza el maestro escrito en *protocols/modbus/master/master.py* y el esclavo de *protocols/modbus/slave/slave.py*.

Posteriormente se comprueba que el esclavo inicia y responde correctamente a los mensajes.

```
============================= test session starts ==============================
platform linux -- Python 3.10.12, pytest-8.3.3, pluggy-1.5.0
rootdir: /home/nelio/uni/icscommemulator
collected 2 items

tests/docker_compose_generator_test.py .                                 [ 50%]
tests/modbus_test.py .                                                   [100%]

============================== 2 passed in 2.14s ===============================
```

Figura 5.1 Ejecución exitosa de los tests unitarios.

5.3 Ejemplos de uso

En este apartado presenta la interfaz gráfica así como algunos resultados interesantes. Previamente habrá sido necesario seguir los pasos descritos en sección 5.1.

Para acompañar la demostración se ha creado un vídeo donde se prepara el mismo escenario. El vídeo se puede encontrar en https://drive.google.com/file/d/1VYJP0eNjIukOdCXdtUinA2j5hwsCZtj4/.

5.3.1 Creación de Escenario

Para crear un escenario nos dirigimos a http://127.0.0.1:8080 y pulsamos en el botón de crear escenario.

Como ya se ha visto en 4.6.1, nos encontramos con un formulario en el que tendremos que especificar algunos datos para la creación del escenario.

Para el escenario de prueba vamos a crear dos esclavos y dos maestros en la subred *192.168.45.0/24*. Llamaremos al escenario *demo*.

De esta forma nos queda el formulario como se muestra en 5.2.

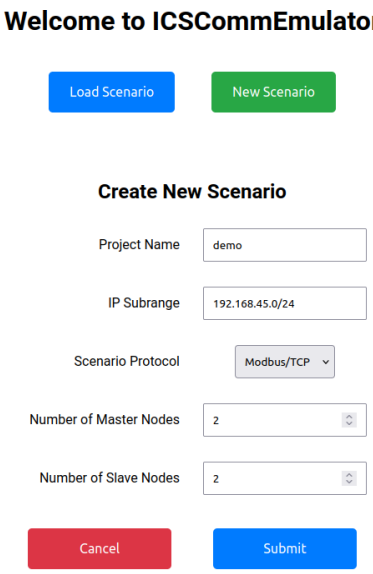

Figura 5.2 Formulario de creación de escenario.

5.3.2 Carga de Escenario

Tras crear el escenario en el apartado anterior se nos redirigió automáticamente hacia http://127.0.0.1:8080/network/demo. Si accidentalmente hubiéramos cerrado la pestaña del navegador y quisiéramos volver a abrirla, nos dirigiríamos a http://127.0.0.1:8080, y esta vez pulsaríamos en el botón de cargar escenario.

De esta forma nos salen todos los escenarios creados y podemos seleccionar el que queramos cargar, como se muestra en la figura 5.3.

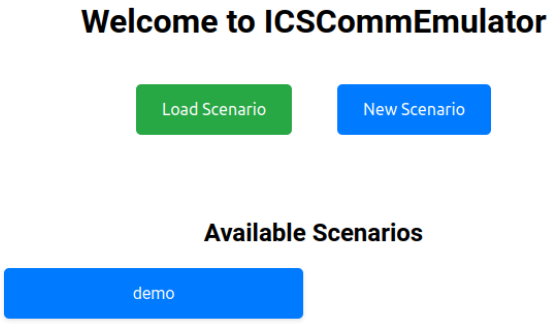

Figura 5.3 Formulario de carga de escenario.

5.3.3 Pantalla de red y utilidades

En el momento de o bien cargar o bien crear nuestro escenario llamado *demo*, se redirige automáticamente a http://127.0.0.1:8080/network/demo.

Aquí nos encontramos con múltiples elementos relevantes, los cuales se describen en 4.6.2.

En la figura 5.4 se muestran en azul las instrucciones básicas para el uso de la aplicación. Además, en verde se muestran los botones de zoom in y zoom out. En amarillo se muestran los botones de ejecutar y guardar. Por último, nos encontramos con un lienzo con nodos.

La funcionalidad del lienzo se irá detallando en las siguientes secciones.

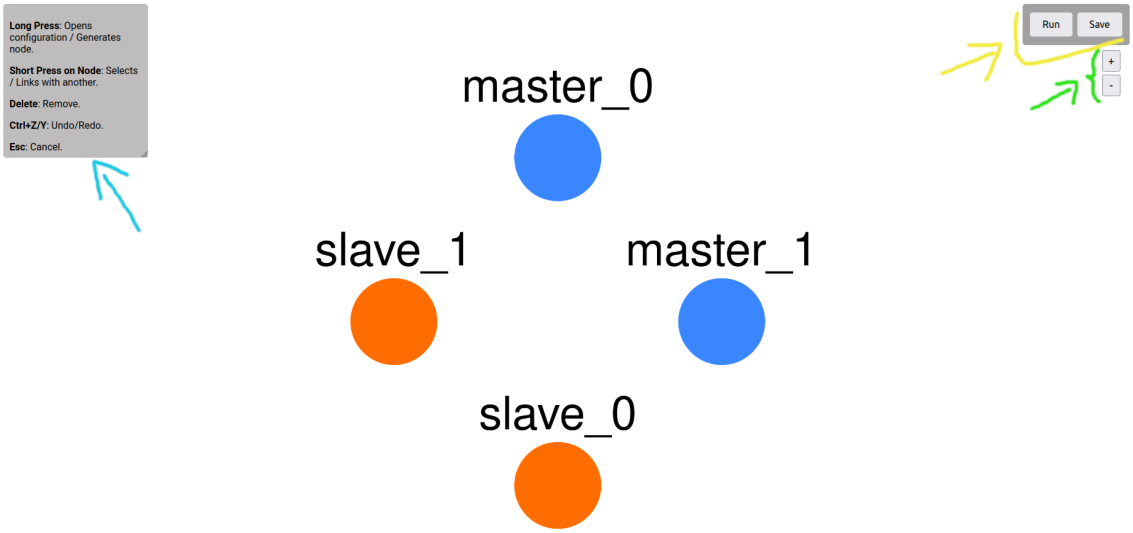

Figura 5.4 Pantalla de red.

5.3.4 Configuración de Maestros

Todos los nodos tienen una configuración común, la cual se compone de un nombre, comentario, rol, ip y mac, tal y como se vio en 4.6.2. Los nodos se guardan su configuración automáticamente al cerrar la pestaña.

Los nodos vienen preconfigurados con un nombre y una ip, pero se pueden modificar a través de la interfaz gráfica.

En este caso vamos a modificar los maestros para que uno se llame "maestro_insistente" y otro "maestro_-relajado". Los nombres no afectan a la emulación y se pueden repetir.

Además, para hacer más interesante la configuración de los maestros vamos a cambiarles la dirección IP para que el maestro insistente esté en *192.168.45.10* y el relajado en *192.168.45.11*.

Para ilustrar las virtudes del proyecto, se cambiará la MAC del maestro insistente a *e4:a8:df:d1:11:0a* y la del maestro relajado a *e4:a8:df:d1:11:0b*.

Figura 5.5 Configuración de los maestros.

5.3.5 Configuración de Esclavos

Los esclavos tienen más campos de configuración que los maestros porque estamos ante un nodo más complejo.

En este ejemplo de uso nombramos a los esclavos *esclavo1* y *esclavo2* y cambiamos sus respectivas IPs a *192.168.45.100* y *192.168.45.101*, así como sus respectivas MACs a *e4:a8:df:d1:11:1a* y *e4:a8:df:d1:11:1b*.

Yendo a la configuración específica del protocolo, vamos a dejar el puerto *502* para facilitar la decodificación del protocolo de Wireshark.

En cuanto al ID de esclavo, vamos a asignarle a cada esclavo su ID correspondiente, *1* y *2*.

Registros

Como ya se vio en 2.1, hay cuatro tipo de registros. Podemos definir los registros utilizados de dos formas distintas: dispersa o secuencial. Dado que el modo de uso más habitual es el sequencial, vamos a usar este.

Para esta demostración vamos a usar *Holding Registers* y *Coils*. Ambos esclavos van a definir algunos registros de ambos tipos.

En cuanto al *esclavo1*, para los *Holding Registers* vamos a definir los primeros tres registros (0-2) con los valores 0, 1, 2. Para los *Coils* vamos a definir los primeros dos registros (0-1) con los valores 0, 1.

En cuanto al *esclavo2*, para los *Holding Registers* vamos a definir únicamente el primer registro (0) con el valor 3. Para los *Coils* vamos a definir los primeros tres registros (0-2) con los valores 1, 0, 1.

De este modo las configuraciones quedan como se muestra en las figuras 5.6 y 5.7.

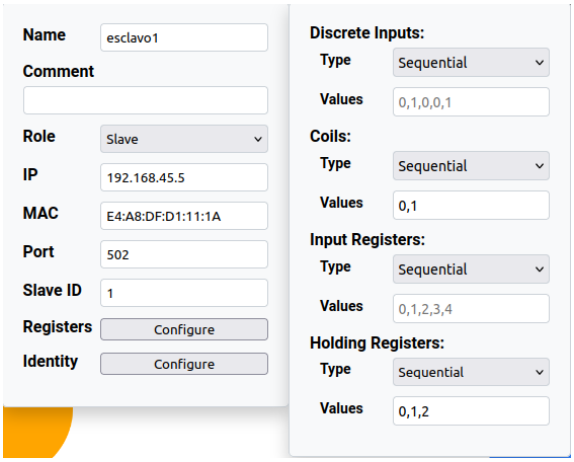

Figura 5.6 Configuración de esclavo1.

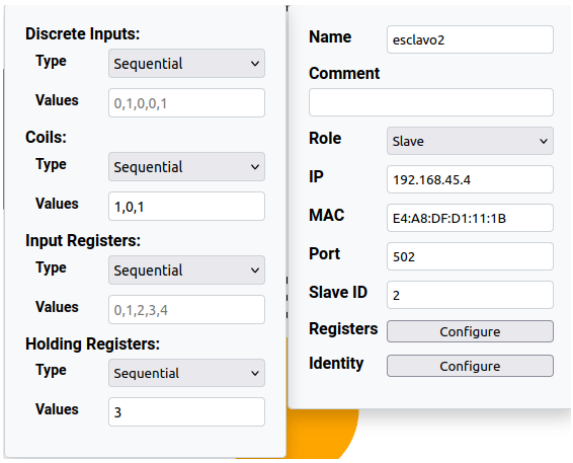

Figura 5.7 Configuración de esclavo2.

Identidad

Como ya se vio en 4.6.2, el último apartado de configuración de los esclavos es la identidad. En este apartado se configura el modelo, la versión y el fabricante del esclavo.

Para hacer una demostración completa de todas las capacidades del software, se le va a configurar al *esclavo2* la identidad. Se le configurará con los siguientes datos:

- Nombre del Vendedor: TechnoFantasy Inc.
- Código de Producto: TF98765
- Revisión: 2.5

- URL de Vendedor: *https://www.technofantasy.com*
- Nombre de Producto: Controlador de Automatización Mágica
- Nombre de Modelo: MagicMaster 3000
- Nombre de Aplicación de Usuario: Control de Procesos Mágicos

5.3.6 Configuración de Mensajes

Los mensajes, como se vio en 4.6.3, tienen muchos campos. Para esta demostración vamos a configurar mensajes desde el maestro insistente a los dos esclavos y desde el maestro relajado únicamente al *esclavo2*.

Maestro Insistente

Para el maestro insistente vamos a configurar mensajes periódicos que pregunten por todos los registros *Holding Registers* de ambos esclavos cada 5 segundos, empezando en el segundo 0.

También, vamos a configurar mensajes periódicos que pregunten por todos los registros *Coils* de ambos esclavos cada 4 segundos empezando en el segundo 1.

Esta configuración se consigue creando dos enlaces, uno por cada esclavo. En cada enlace añadimos dos mensajes, uno para los *Holding Registers* y otro para los *Coils*.

La configuración del enlace con el *esclavo1* se muestra en la figura 5.8. La del enlace con el *esclavo2* en la figura 5.9.

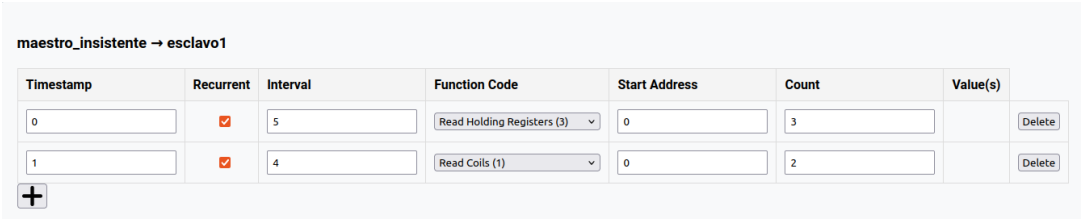

Figura 5.8 Configuración de mensajes del maestro insistente al esclavo1.

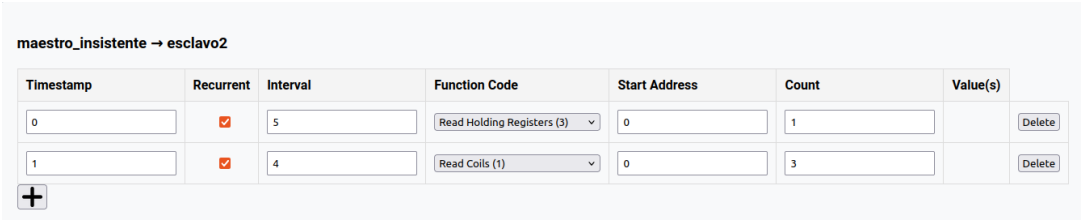

Figura 5.9 Configuración de mensajes del maestro insistente al esclavo2.

Maestro Relajado

El maestro relajado lo único que hará será preguntarle al *esclavo2* por su identidad en el segundo 7 de la emulación.

Para ello creamos un enlace entre el maestro y el *esclavo2* y añadimos un mensaje que pregunte por la identidad. La configuración será como se muestra en la figura 5.10.

Figura 5.10 Configuración de mensajes del maestro relajado al esclavo2.

5.3.7 Guardado de escenario

Para ejecutar el escenario lo primero que haremos será guardar el escenario. Para ello pulsamos en el botón de guardar en la parte superior derecha de la pantalla.

Esto hará una comprobación del escenario, ya descrita en 4.6.4. En caso de haber algún error, se mostrará un mensaje de error y no se guardará el escenario. En caso de que haya avisos, se le informará al usuario y, en caso de que reafirme su intención, se le permitirá guardar el escenario.

En la figura 5.11 se observa el mensaje de éxito al guardar el escenario.

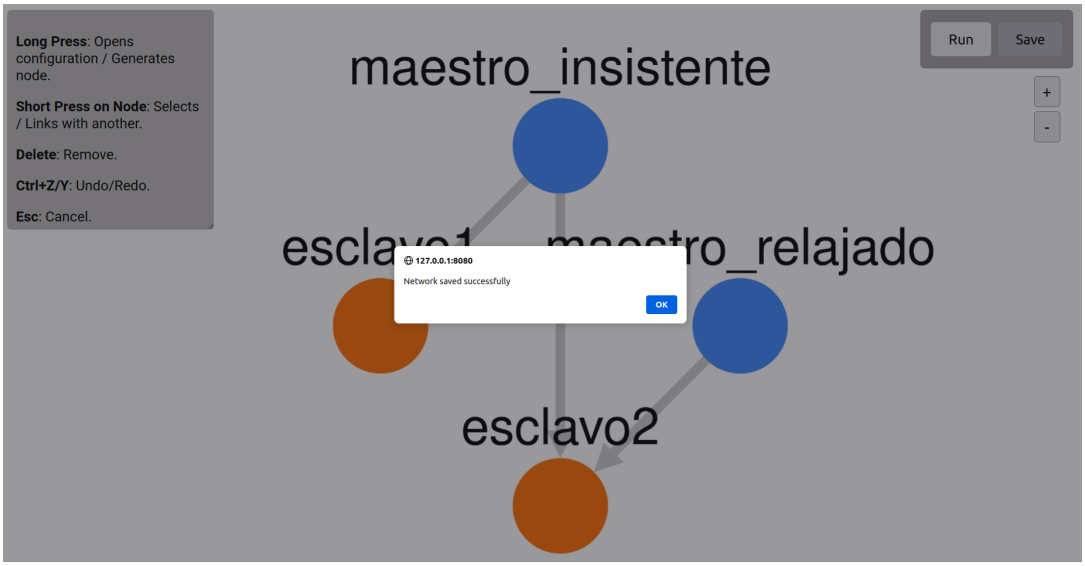

Figura 5.11 Mensaje de éxito al guardar el escenario.

5.3.8 Ejecución de la emulación

Una vez guardado el escenario podremos darle al botón de ejecutar. Esto lanzará un cuadro donde se nos pedirá la duración de la emulación en segundos, como se muestra en la figura 5.12. Para esta demostración vamos a emular 15 segundos.

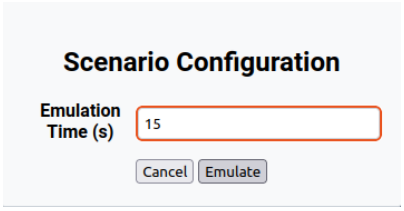

Figura 5.12 Parámetros de ejecución de la emulación.

Tras darle al botón de iniciar, se nos redirigirá a una página donde se nos mostrará el progreso de la emulación, como se muestra en la figura 5.13.

5.3.9 Resultados

Al terminar la emulación se nos avisa con un recuadro donde se nos informa dónde se ha guardado el archivo de tráfico, como se muestra en la figura 5.14.

Para finalizar buscamos el archivo de tráfico en la ruta especificada. En este caso se ha guardado en *./outputs/demo.pcap*.

Recapitulando, en este escenario hemos emulado 15 segundos con mensajes periódicos cada 4 y 5 segundos empezando en los segundos 1 y 0, respectivamente. Además, tenemos un mensaje único en el segundo 7. De forma sintetizada, deberíamos estar buscando los mensajes descritos en 5.1

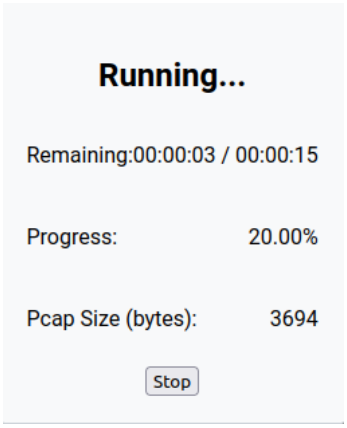

Figura 5.13 Progreso de la emulación.

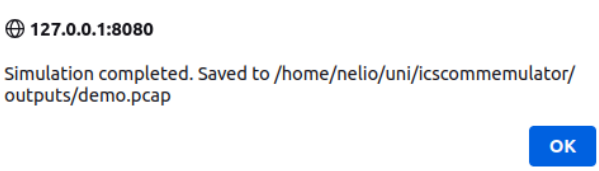

Figura 5.14 Resultado de la emulación.

Tabla 5.1 Mensajes enviados esperados.

Segundo	IP Maestro	IP Esclavo	Solicitud	Respuesta
0	192.168.45.10	192.168.45.100	Holding Registers (0-2)	0, 1, 2
0	192.168.45.10	192.168.45.101	Holding Registers (0)	3
1	192.168.45.10	192.168.45.100	Coils (0-1)	0, 1
1	192.168.45.10	192.168.45.101	Coils (0-2)	1, 0, 1
5	192.168.45.10	192.168.45.100	Holding Registers (0-2)	0, 1, 2
5	192.168.45.10	192.168.45.101	Holding Registers (0)	3
7	192.168.45.11	192.168.45.101	Identidad	Texto en Subsubsección 5.3.5
9	192.168.45.10	192.168.45.100	Coils (0-1)	0, 1
9	192.168.45.10	192.168.45.101	Coils (0-2)	1, 0, 1
10	192.168.45.10	192.168.45.100	Holding Registers (0-2)	0, 1, 2
10	192.168.45.10	192.168.45.101	Holding Registers (0)	3
13	192.168.45.10	192.168.45.100	Coils (0-1)	0, 1
13	192.168.45.10	192.168.45.101	Coils (0-2)	1, 0, 1
15	192.168.45.10	192.168.45.100	Holding Registers (0-2)	0, 1, 2
15	192.168.45.10	192.168.45.101	Holding Registers (0)	3

En la figura 5.15 se muestra el tráfico capturado usando la herramienta Wireshark en el pcap de salida. Podemos apreciar que están todos los mensajes esperados con sus respuestas correctas, así como con sus macs. Por mantener breve el análisis de la demo podemos concluir que la emulación ha sido un éxito rotundo.

modbus

No.	Time	Source	Destination	Protocol	Length	Info
6	0.000150	192.168.45.10	192.168.45.100	Modbus/TCP	78	Query: Trans: 1; Unit: 1, Func: 3: Read Holding Registers
8	0.000662	192.168.45.100	192.168.45.10	Modbus/TCP	81	Response: Trans: 1; Unit: 1, Func: 3: Read Holding Registers
18	0.000990	192.168.45.10	192.168.45.101	Modbus/TCP	78	Query: Trans: 1; Unit: 2, Func: 3: Read Holding Registers
20	0.001541	192.168.45.101	192.168.45.10	Modbus/TCP	77	Response: Trans: 1; Unit: 2, Func: 3: Read Holding Registers
28	1.003150	192.168.45.10	192.168.45.100	Modbus/TCP	78	Query: Trans: 2; Unit: 1, Func: 1: Read Coils
30	1.003809	192.168.45.100	192.168.45.10	Modbus/TCP	76	Response: Trans: 2; Unit: 1, Func: 1: Read Coils
38	1.004291	192.168.45.10	192.168.45.101	Modbus/TCP	78	Query: Trans: 2; Unit: 2, Func: 1: Read Coils
40	1.004938	192.168.45.101	192.168.45.10	Modbus/TCP	76	Response: Trans: 2; Unit: 2, Func: 1: Read Coils
48	5.009939	192.168.45.10	192.168.45.100	Modbus/TCP	78	Query: Trans: 3; Unit: 1, Func: 3: Read Holding Registers
50	5.010620	192.168.45.100	192.168.45.10	Modbus/TCP	81	Response: Trans: 3; Unit: 1, Func: 3: Read Holding Registers
58	5.011083	192.168.45.10	192.168.45.101	Modbus/TCP	78	Query: Trans: 3; Unit: 2, Func: 3: Read Holding Registers
60	5.011778	192.168.45.101	192.168.45.10	Modbus/TCP	77	Response: Trans: 3; Unit: 2, Func: 3: Read Holding Registers
67	5.012184	192.168.45.10	192.168.45.100	Modbus/TCP	78	Query: Trans: 4; Unit: 1, Func: 1: Read Coils
70	5.012680	192.168.45.100	192.168.45.10	Modbus/TCP	76	Response: Trans: 4; Unit: 1, Func: 1: Read Coils
78	5.013073	192.168.45.10	192.168.45.101	Modbus/TCP	78	Query: Trans: 4; Unit: 2, Func: 1: Read Coils
80	5.013498	192.168.45.101	192.168.45.10	Modbus/TCP	76	Response: Trans: 4; Unit: 2, Func: 1: Read Coils
90	7.009623	192.168.45.11	192.168.45.101	Modbus/TCP	77	Query: Trans: 1; Unit: 1, Func: 43/ 1: Read Device Identification
92	7.010351	192.168.45.101	192.168.45.11	Modbus/TCP	170	Response: Trans: 1; Unit: 1, Func: 43/ 1: Read Device Identification
100	9.017763	192.168.45.10	192.168.45.100	Modbus/TCP	78	Query: Trans: 5; Unit: 1, Func: 1: Read Coils
102	9.018462	192.168.45.100	192.168.45.10	Modbus/TCP	76	Response: Trans: 5; Unit: 1, Func: 1: Read Coils
110	9.018914	192.168.45.10	192.168.45.101	Modbus/TCP	78	Query: Trans: 5; Unit: 2, Func: 1: Read Coils
112	9.019633	192.168.45.101	192.168.45.10	Modbus/TCP	76	Response: Trans: 5; Unit: 2, Func: 1: Read Coils
120	10.020614	192.168.45.10	192.168.45.100	Modbus/TCP	78	Query: Trans: 6; Unit: 1, Func: 3: Read Holding Registers
122	10.021079	192.168.45.100	192.168.45.10	Modbus/TCP	81	Response: Trans: 6; Unit: 1, Func: 3: Read Holding Registers
130	10.021461	192.168.45.10	192.168.45.101	Modbus/TCP	78	Query: Trans: 6; Unit: 2, Func: 3: Read Holding Registers
132	10.022012	192.168.45.101	192.168.45.10	Modbus/TCP	77	Response: Trans: 6; Unit: 2, Func: 3: Read Holding Registers
141	13.025577	192.168.45.10	192.168.45.100	Modbus/TCP	78	Query: Trans: 7; Unit: 1, Func: 1: Read Coils
143	13.026171	192.168.45.100	192.168.45.10	Modbus/TCP	76	Response: Trans: 7; Unit: 1, Func: 1: Read Coils
151	13.026587	192.168.45.10	192.168.45.101	Modbus/TCP	78	Query: Trans: 7; Unit: 2, Func: 1: Read Coils
153	13.027276	192.168.45.101	192.168.45.10	Modbus/TCP	76	Response: Trans: 7; Unit: 2, Func: 1: Read Coils
164	15.029649	192.168.45.10	192.168.45.100	Modbus/TCP	78	Query: Trans: 8; Unit: 1, Func: 3: Read Holding Registers
166	15.030339	192.168.45.100	192.168.45.10	Modbus/TCP	81	Response: Trans: 8; Unit: 1, Func: 3: Read Holding Registers
173	15.030694	192.168.45.10	192.168.45.101	Modbus/TCP	78	Query: Trans: 8; Unit: 2, Func: 3: Read Holding Registers
176	15.031390	192.168.45.101	192.168.45.10	Modbus/TCP	77	Response: Trans: 8; Unit: 2, Func: 3: Read Holding Registers

```
▸ Ethernet II, Src: CompalIn_d1:11:1a (e4:a8:df:d1:11:1a), Dst: CompalIn_d1:11:0a (e4:a8:df:d1:11:0a)
▸ Internet Protocol Version 4, Src: 192.168.45.100, Dst: 192.168.45.10
▸ Transmission Control Protocol, Src Port: 502, Dst Port: 35688, Seq: 1, Ack: 13, Len: 15
▸ Modbus/TCP
▾ Modbus
    .000 0011 = Function Code: Read Holding Registers (3)
    [Request Frame: 6]
    [Time from request: 0.000512000 seconds]
    Byte Count: 6
  ▸ Register 0 (UINT16): 0
  ▸ Register 1 (UINT16): 1
  ▸ Register 2 (UINT16): 2
```

Figura 5.15 Tráfico en Wireshark.

6 Conclusiones

En este capítulo se presentan las conclusiones derivadas del desarrollo del proyecto, así como una reflexión final sobre los logros alcanzados, los desafíos superados y las posibles aplicaciones futuras del emulador de tráfico de red desarrollado. A lo largo de este documento, se ha detallado el proceso de diseño, implementación y validación del sistema, destacando su capacidad para generar tráfico sintético y su utilidad en el entrenamiento de algoritmos de inteligencia artificial. Además, se exploran otros usos potenciales, como la emulación de ataques y la enseñanza de protocolos industriales.

6.1 Usos derivados

Como se mencionó en la motivación del proyecto, el fin último de un proyecto de esta índole es el de la generación de tráfico sintético para el entrenamiento de algoritmos de inteligencia artificial. Tras observar con detenimiento el funcionamiento del proyecto junto a una demostración de su ejecución, podemos ir un paso más allá y pensar qué otros usos pueden nacer de esta aplicación.

6.1.1 Emulación de Ataques

El código ha resultado ser un emulador de tráfico de red fiel y versátil. Si analizamos los distintos tipos de ataque, podemos ver que el simulador está totalmente preparado para hacer simulaciones de ataques de capa de aplicación aunque no exista una interfaz específica para ello.

Por ejemplo, un ataque de denegación de servicio (DoS) se puede emular fácilmente. Un ataque de denegación de servicio consiste en saturar a un servidor con muchas peticiones. Para realizar este ataque, lo único que habría que hacer sería mandar un mensaje periódico con un periodo muy pequeño. Con poco trabajo más se podría configurar los esclavos para que se queden sin memoria, permitiendo un escenario de DoS donde se produzca discontinuidad en el servicio.

Otro ejemplo claro sería el de tráfico con valores fuera de rango. Normalmente los sistemas de control industrial están diseñados para trabajar con valores dentro de un rango determinado. Si se mandan valores fuera de rango, el sistema puede comportarse de forma inesperada. Puesto que la única diferencia en este tipo de ataques está en cómo lo procesa, para nuestro emulador de tráfico no hay diferencia alguna entre un valor dentro de un rango o fuera del mismo.

6.1.2 Enseñanza

El emulador de tráfico de red también puede ser usado para enseñar a estudiantes interesados el protocolo industrial Modbus. En la actualidad, la mayoría de las soluciones de emulación no tienen la precisión adquirida por este emulador de tráfico de red. Además, la interfaz gráfica es muy intuitiva.

6.2 Reflexión Final

En el proyecto se establecieron unos objetivos abiertos que luego han derivado en objetivos específicos. En primer lugar, se ha desarrollado un emulador de tráfico, a cuya lógica principal se le ha llamado Core o

Modelo a lo largo de esta memoria. En segundo lugar, ha quedado manifiesto en la demostración que la interfaz es intuitiva y abstrae al usuario de la tecnología subyacente.

Además, gracias a la fase de diseño se han podido identificar necesidades funcionales a tiempo, permitiendo incluirlas con facilidad adaptando la arquitectura. Por otro lado, se ha podido comprobar que la arquitectura propuesta es escalable y flexible, permitiendo así la inclusión de nuevas funcionalidades sin necesidad de rehacer los módulos.

A lo largo del proyecto se han codificado 3037 líneas de código en ficheros .py y .js. Esta suma no incluye las librerías usadas ni archivos de configuración, como los diversos archivos YAML, JSON, CSV o los archivos de configuración de Docker.

Por último, se han creado imágenes de Docker para este escenario, reduciendo su peso en disco y en memoria con respecto a las disponibles públicamente en Docker Hub.

7 Limitaciones y Líneas Futuras

I believe we've reached the end of our journey. All that remains is to collapse the innumerable possibilities before us. Are you ready to learn what comes next?

Solanum, Outer Wilds (Mobius Digital, 2019)

El proyecto comenzó con una serie de objetivos que se han alcanzado con éxito, marcando un sólido punto de partida. Aunque algunas funcionalidades iniciales quedaron fuera del alcance debido al tiempo disponible, esto abre oportunidades de mejora interesantes. A continuación, se presentan algunas líneas futuras que podrían potenciar aún más el proyecto.

7.1 Incrementar cantidad de protocolos

La primera limitación viene dada porque este trabajo solo soporta el protocolo Modbus TCP, lo que restringe su aplicabilidad a otros protocolos industriales como DNP3 o IEC 104. Aunque el código es personalizable, se puede mejorar la automatización de los ataques a Modbus.

La conveniencia de ampliar el número de protocolos ya se ha contemplado en este trabajo durante la fase de diseño, lo que facilitará la futura implementación. La arquitectura se diseñó para que se pudiese implementar cualquier otro protocolo únicamente buscando la librería y haciendo cambios leves a la interfaz gráfica. Las librerías que se proponen de los protocolos mencionados son *pyiec61850* y *dnp3-python*.

7.2 Retroalimentación durante la ejecución

Uno de los principales motivos por los cuales se escogió la representación en grafo fue para poder representar el tráfico generado por la red en tiempo real. Actualmente no está implementado, pero estuvo presente la posibilidad de añadirlo hasta las últimas fases del proyecto.

Este punto fue descartado por falta de tiempo. La librería utilizada para el frontend permite la congelación de los nodos, evitando la interacción con el usuario. En este estado, las peticiones que se realizan al backend para actualizar sobre el estado de la emulación se pueden hacer con mayor frecuencia y que lleven información de los mensajes mandados por cada nodo.

Desde el core, se tendría que añadir un módulo que recogiese los mensajes mandados por cada nodo. Una de las formas más sencillas sería inspeccionando el socket de Docker, aunque no se ha estudiado con la suficiente profundidad como para establecerla como la mejor solución.

7.3 Velocidad de enlace no infinita

Otra limitación es que el enlace de datos no tiene condiciones realistas (p.ej., retardo constante o cero), lo que puede afectar a la representatividad de los escenarios simulados frente a redes reales con latencias y

pérdidas de paquetes. Una posible línea de trabajo es añadir una opción para limitar la velocidad de enlace entre los nodos.

7.4 Dependencia de Docker

Desde el punto de vista de la arquitectura, el sistema depende de Docker para la virtualización de los dispositivos, lo que puede generar sobrecarga en sistemas con recursos limitados. Esto podría llegar a ser una limitación pero hay que tener en cuenta que en el trabajo ya se ha intentado paliar esta limitación reduciendo el tamaño de las imágenes, aunque como futura línea podría profundizarse más en esta línea.

Índice de Figuras

2.1 Campos de la cabecera del protocolo Modbus 3
2.2 Arquitectura de Docker 6
2.3 Aplicación del patrón Modelo-Vista-Controlador 7

4.1 Diagrama de componentes: Backend y Core 16
4.2 Estructura de directorios de /tmp/ICSCommEmulator 21
4.3 Diagrama de Secuencia para Crear un Escenario 29
4.4 Diagrama de Secuencia para Cargar un Escenario 29
4.5 Diagrama de secuencia para la creación de un escenario 33
4.6 Diagrama de secuencia para la carga de un escenario 33
4.7 Diagrama de secuencia para el guardado de un escenario 34
4.8 Diagrama de secuencia para la ejecución de un escenario 34
4.9 Diagrama de secuencia para la generación de archivos necesarios para la emulación 35
4.10 Diagrama de secuencia para la detención de la emulación 35

5.1 Ejecución exitosa de los tests unitarios 38
5.2 Formulario de creación de escenario 39
5.3 Formulario de carga de escenario 39
5.4 Pantalla de red 40
5.5 Configuración de los maestros 40
5.6 Configuración de esclavo1 41
5.7 Configuración de esclavo2 41
5.8 Configuración de mensajes del maestro insistente al esclavo1 42
5.9 Configuración de mensajes del maestro insistente al esclavo2 42
5.10 Configuración de mensajes del maestro relajado al esclavo2 42
5.11 Mensaje de éxito al guardar el escenario 43
5.12 Parámetros de ejecución de la emulación 43
5.13 Progreso de la emulación 44
5.14 Resultado de la emulación 44
5.15 Tráfico en Wireshark 45

Índice de Tablas

2.1	Códigos de función Modbus	4
2.2	Tipos de Registros en Modbus	4
2.3	Comandos comunes para la gestión de imágenes en Docker	5
2.4	Instrucciones comunes de Dockerfile	5
2.5	Equivalencia entre operaciones CRUD y métodos HTTP	8
3.1	Comparativa entre Software	12
4.1	Endpoints de la REST API	14
4.2	Librerías Usadas en el Proyecto	15
4.3	Herramientas Externas Usadas en el Proyecto	15
4.4	Validaciones realizadas para un escenario	26
4.5	Campos de creación de un escenario	28
4.6	Campos de configuración de un nodo	30
4.7	Campos de configuración específicos de un esclavo	30
4.8	Validaciones realizadas para los parámetros de comunicación	31
4.9	Campos requeridos para la creación de un escenario	32
5.1	Mensajes enviados esperados	44

Índice de Códigos

2.1	Ejemplo básico docker-compose.yml	6
4.1	Ejemplo de network	16
4.2	Ejemplo de nodo maestro	17
4.3	Ejemplo de nodo esclavo	17
4.4	Dockerfile del maestro	18
4.5	Dockerfile del esclavo	19
4.6	Ejemplo de master.csv	19
4.7	Ejemplo de slave.yaml	19
4.8	Ejemplo de uso de docker-compose-generator.py	21
4.9	Ejemplo de uso de scenario-config-generator.py	22
4.10	Ejemplo de configuración YAML	26

Bibliografía

[1] I. N. Fovino, A. Carcano, M. Masera, and A. Trom-Betta, "Design and implementation of a secure modbus protocol," *IFIP Advances in Information and Communication Technology*, 2009, cited by: 111; All Open Access, Bronze Open Access. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84891808534&doi=10.1007%2f978-3-642-04798-5_6&partnerID=40&md5=ec190d7dad22b95152062417b0d9a9a7

[2] INCIBE. (2020) Evolucionando a modbus seguro. [Online]. Available: https://www.incibe.es/incibe-cert/blog/evolucionando-modbus-seguro

[3] Modbus Organization, Inc., *MODBUS Application Protocol Specification V1.1b3*, Apr. 2012. [Online]. Available: https://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf

[4] S. Electric. (2020) Protocolo modbus para interruptores masterpact nt/nw. Schneider Electric. [Online]. Available: https://product-help.schneider-electric.com/ED/ES_Power/NT-NW_Modbus_IEC_Guide/EDMS/DOCA0054EN/DOCA0054xx/Master_NS_Modbus_Protocol/Master_NS_Modbus_Protocol-2.htm

[5] Wikipedia. (2009) Model-view-controller. [Online]. Available: https://en.wikipedia.org/wiki/Model-view-controller

[6] S. O. Blog, "Best practices for rest api design," 2020. [Online]. Available: https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/

[7] M. Masse, *REST API Design Rulebook*. Sebastopol, CA: O'Reilly Media, 2011.

[8] I. McGregor, "The relationship between simulation and emulation," in *Proceedings of the Winter Simulation Conference*, vol. 2, 2002, pp. 1683–1688 vol.2.

[9] Wikipedia. (2024) Packet generators. [Online]. Available: https://en.wikipedia.org/wiki/Packet_generator

[10] D. Antonioli and N. O. Tippenhauer, "Minicps: A toolkit for security research on CPS networks," *CoRR*, vol. abs/1507.04860, 2015. [Online]. Available: http://arxiv.org/abs/1507.04860

[11] C. Queiroz, A. Mahmood, and Z. Tari, "Scadasim—a framework for building scada simulations," *IEEE Transactions on Smart Grid*, vol. 2, no. 4, pp. 589–597, 2011.

[12] A. Dehlaghi-Ghadim, A. Balador, M. H. Moghadam, H. Hansson, and M. Conti, "Icssim — a framework for building industrial control systems security testbeds," *Computers in Industry*, vol. 148, p. 103906, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166361523000568

[13] A. Candia and C. Cappo, "Tinyics: An industrial control system simulator based on ns-3," in *2024 43rd International Conference of the Chilean Computer Science Society (SCCC)*, 2024, pp. 1–8.

[14] D. D. Team. (2016) Docker networking design philosophy. [Online]. Available: https://www.docker.com/blog/docker-networking-design-philosophy/

[15] IONOS. (2021) Multicast dns: qué es y cómo funciona. [Online]. Available: https://www.ionos.es/digitalguide/servidores/know-how/multicast-dns/

[16] Wikipedia. (2010) Simple service discovery protocol (ssdp). [Online]. Available: https://es.wikipedia.org/wiki/SSDP

[17] D. D. Team, *Resource constraints for containers*, 2024. [Online]. Available: https://docs.docker.com/engine/containers/resource_constraints/

[18] I. Team. (2024) Rest api standards: Best practices and guidelines. [Online]. Available: https://www.integrate.io/blog/rest-api-standards/

[19] K. Mathioudakis, N. Frangiadakis, A. Merentitis, and V. Gazis, "Towards generic scada simulators : A survey of existing multi-purpose co-simulation platforms , best practices and use-cases," 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:12441860

Siglas

ACR Azure Container Registry. 5
AJAX Asynchronous JavaScript And XML. 8
API Application Programming Interface. 7, 8, 13, 14, 23

CoAP Constrained Application Protocol. 7
CRUD Create, Read, Update, Delete. 8

ECR Amazon Elastic Container Registry. 5

FC Function Code. 3, 4

HMI Human-Machine Interface. 12
HTML Hypertext Markup Language. 23, 24, 26
HTTP Hypertext Transfer Protocol. 7

ICS Industrial Control System. 1, 13
IP Internet Protocol. 3, 12, 16, 17, 20, 24, 26, 28, 30, 40

JSON JavaScript Object Notation. 7, 8, 14, 24–28
JSP JavaServer Pages. 23
JWT JSON Web Token. 9

MAC Media Access Control. 12, 16, 17, 30, 40
mDNS Multicast DNS. 23
MVC Modelo-Vista-Controlador. 7–9, 13

PLC Programmable Logic Controller. 12

RAE Real Academia Española. 11
REST Representational State Transfer. 7, 8, 14, 23
RTU Remote Terminal Unit. 3, 12

SSDP Simple Service Discovery Protocol. 23

TCP Transmission Control Protocol. 1, 3, 49

UDP User Datagram Protocol. 3, 23
UPnP Universal Plug and Play. 23
URI Uniform Resource Identifier. 7, 8

XML Extensible Markup Language. 7, 8

YAML YAML Ain’t Markup Language. 5, 14, 19, 20, 26, 28