Despliegue automatizado de servicios L2VPN y L3VPN en entorno MikroTik

Jonathan Sánchez Santiago, 1493132
Escuela de Ingeniería, Universidad Autónoma de Barcelona, Barcelona, España
Ingeniería Informática Mención Tecnologías de la Información, 2022/23

***Resumen*– En este Trabajo de Fin de Grado se propone una solución para implementar la tecnología SD-WAN utilizando dispositivos de bajo costo e imágenes virtualizables de MikroTik. El uso de SD-WAN separa el plano de control del plano de datos en una arquitectura de red permitiendo ser más precisos y flexibles con las configuraciones de dispositivos y aportando capacidades de automatización y monitorización. La solución presentada se basa en una arquitectura cliente-servidor, donde los dispositivos de red virtualizan aplicaciones en contenedores Docker para mantener, generar y aplicar automáticamente las configuraciones de red en tiempo real. Esta operativa se mantiene mediante la utilización de las tecnologías de comunicación HTTP y MQTT que facilitan el intercambio de información entre los dispositivos, y un servidor que mantiene toda la información actualizada de la topología del usuario. La solución, implementada principalmente en Python, permite el escalado de forma significante tanto en número de dispositivos como en posibilidades de arquitectura abriendo la posibilidad de aplicar cualquier topología L2VPN y L3VPN, y siendo fácilmente ampliable para incorporar nuevas tecnologías y topologías de red.**

***Abstract*– This paper proposes a solution to implement SD-WAN technology using low-cost devices and virtualisable images from MikroTik. The use of SD-WAN separates the control plane from the data plane in a network architecture allowing to be more precise and flexible with device configurations and providing automation and monitoring capabilities. The solution presented is based on a client-server architecture, where network devices virtualise applications in Docker containers to automatically maintain, generate and apply network configurations in real time. This operation is maintained through the use of HTTP and MQTT communication technologies that facilitate the exchange of information between devices, and a server that maintains all updated user topology information. The solution, implemented mainly in Python, allows for significant scalability both in terms of number of devices and architectural possibilities, opening up the possibility of applying any L2VPN and L3VPN topology, and being easily extendable to incorporate new technologies and network topologies.**

***Index Terms*—MikroTik, Docker, SD-WAN, Python, Flask, AWS, VPN, OSPF, RouterOS, MQTT, HTTP, API**

I. Introducción

El paradigma de las redes de área amplia (WAN, *Wide Area Networks*) está experimentando cambios significativos debido a las demandas crecientes de reducción de costos, mejora de la calidad de servicio (QoS, *Quality of Service*) y optimización de recursos. Estos nuevos requisitos surgen de la evolución de las tecnologías de acceso y transporte, lo que significa que el enfoque tradicional *best-effort* y las tareas manuales necesarias para crear y mantener una red convencional se vuelven insuficientes para actividades emergentes en Internet, como servicios en la nube, transmisión de video o incluso operaciones médicas a larga distancia. Estas actividades requieren garantizar la calidad de servicio a un costo razonable y plantean desafíos complejos.

Para intentar dar solución a dichos retos aparece el concepto de redes de área amplia definidas por software (SD-WAN). Estas nuevas soluciones buscan centralizar la administración de la red a través de aplicaciones que facilitan el despliegue, mantenimiento y optimización de enlaces. Se enfocan en reducir costos y garantizar parámetros de calidad de servicio (QoS), como baja latencia y alta disponibilidad. Además, eliminan las tareas manuales, los posibles errores humanos y los fallos causados por la caída de dispositivos.Todo esto se puede llevar a cabo gracias a la abstracción de la arquitectura y una división en planos de control y transporte [1]. Estas capas separan los ámbitos de control y datos permitiendo a los operadores gestionar la red de forma centralizada y flexible sin afectar a la infraestructura de transmisión. Además, permiten que las aplicaciones establezcan requisitos que después se traducen automáticamente a configuraciones de red funcionales e instantáneamente aplicables [2].

A pesar de la existencia de la tecnología en la última década y el aumento de la demanda por cada vez más aplicaciones, su uso no es tan común debido al coste asociado a cambiar toda una infraestructura y el desarrollo, todavía por completar, en algunas áreas como la seguridad. Sin embargo, en este Trabajo de Fin de Grado se investigará y propondrá una solución de bajo coste y fácil de desplegar que permitirá desarrollar y desplegar cualquier topología y servicio de red disponible en RouterOS, utilizando tanto equipos físicos como virtualizados de MikroTik [3]. El motivo principal para elegir esta marca reside en la buena relación entra características técnica y coste que ofrecen sus dispositivos. Además desde la versión 7.5 es posible integrar código de terceros en ellos a través de la virtualización de contenedores permitiéndonos crear una arquitectura barata de SD-WAN. El proyecto se centrará en implementar y demostrar el concepto mediante el despliegue de una topología L2VPN y L3VPN usando VXLAN (*Virtual Extensible Local Area Network*) para transportar el servicio de nivel 2 junto con túneles VPN (*Virtual Private Network*) basados en Wireguard que permiten una conexión segura a través de la red pública de Internet y OSPF (*Open Shortest Path First*) como método de enrutamiento dinámico. Se demostrará utilizando esta configuración debido al alto interés por operadoras y centros de datos por ofrecer servicios de conectividad en capa 2 en lugares donde sólo había comunicación en capa 3.

II. Objetivos

El objetivo principal de este Trabajo de Fin de Grado es diseñar y desarrollar una solución para la implementación de una red definida por software (SD-WAN, *Software Defined-Wide Area Network*) para el despliegue automático para un entorno MikroTik. Para la validación del funcionamiento, se presentará la configuración automatizada de una topología en la cual dos sedes de una empresa conectadas a través de Internet pueden comunicarse como si estuvieran conectados en una LAN de capa 2 (*Local Area Network*). Para lograr esta funcionalidad, se plantea un escenario que utiliza la red pública de Internet. En este escenario, se utilizará WireGuard como VPN para garantizar la seguridad de las comunicaciones entre los distintos puntos de la red y proteger la confidencialidad de la información transmitida. Además, se implementará OSPF para distribuir y establecer de forma automática las rutas de los túneles entre los diferentes puntos de la red, lo que brindará resistencia ante cambios y fallos en los enlaces. De esta manera, se creará una red basada en capa 3 con túneles VPN, ofreciendo una solución de L3VPN.

Posteriormente, se añadirá VXLAN a esta infraestructura para que los dispositivos puedan tener conectividad de capa 2, estableciendo un *overlay* sobre la red IP (*Internet Protocol*) y logrando una L2VPN. De esta manera, se establecerá una solución completa que combina distintas tecnologías para habilitar la comunicación transparente y eficiente entre dispositivos en diferentes ubicaciones. Además, los dispositivos estarán permanentemente conectados a una controla central que permitirá realizar cambios en la topología de forma automática siendo este el plano de control SD-WAN.

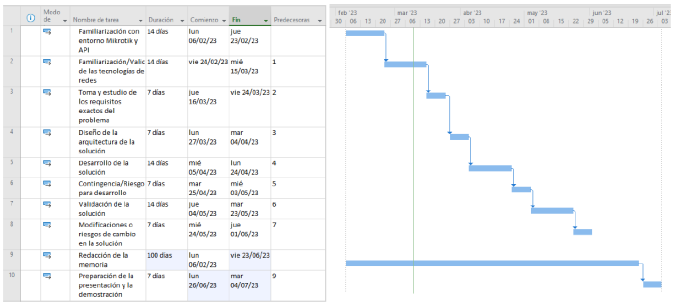
III. Metodología y planificación

Para garantizar un desarrollo adecuado de este Trabajo de Fin de Grado, se ha adoptado una metodología que se basa en la consecución de objetivos a corto y medio plazo, con un intervalo de aproximadamente 2 o 3 semanas entre cada uno. Esto ha permitido mantener un trabajo constante y continuo a lo largo del tiempo. Además, se ha llevado a cabo una supervisión y contraste semanal con el tutor, para asegurar la coherencia y el progreso del trabajo.

La metodología utilizada en este proyecto se enmarca dentro del enfoque Agile [4], que se caracteriza por ser incremental y orientado a la entrega continua. Con este enfoque, se trabajará de manera iterativa, realizando revisiones después de completar cada tarea o *sprint*.

En cuanto a la implementación concreta de esta metodología en el Trabajo de Fin de Grado, se seguirán los siguientes pasos: en primer lugar, se definirán objetivos específicos junto con sus plazos estimados. Estos objetivos han sido incluidos en la herramienta Microsoft Project, lo que nos permitirá establecer una línea base de trabajo y generar un diagrama de Gantt. A través de este diagrama, se podrá visualizar el avance del proyecto y gestionar adecuadamente los tiempos.

Adicionalmente, se traducirán las tareas identificadas a un tablero en la plataforma Trello [5]. En este tablero, se ha llevado un seguimiento exhaustivo de las tareas, junto con su estado actual. Esto ha proporcionado una visión clara y detallada del progreso de cada tarea y permitirá una gestión eficiente del trabajo.



	Modo de	Nombre de tarea	Duración	Comienzo	Fin	Predecesoras
1		Familiarización con entorno Mikrotik y API	14 días	lun 06/02/23	jue 23/02/23	
2		Familiarización/Valic	14 días	vie 24/02/23	mié 15/03/23	1
3		Toma y estudio de los requisitos exactos del problema	7 días	jue 16/03/23	vie 24/03/23	2
4		Diseño de la arquitectura de la solución	7 días	lun 27/03/23	mar 04/04/23	3
5		Desarrollo de la solución	14 días	mié 05/04/23	lun 24/04/23	4
6		Contingencia/Riesgo para desarrollo	7 días	mar 25/04/23	mié 01/05/23	5
7		Validación de la solución	14 días	jue 04/05/23	mar 23/05/23	6
8		Modificaciones o riesgos de cambio en la solución	7 días	mié 24/05/23	jue 01/06/23	7
9		Redacción de la memoria	100 días	lun 06/02/23	vie 23/06/23	
10		Preparación de la presentación y la demostración	7 días	lun 26/06/23	mar 04/07/23	9

Figura 1: Diagrama de Gantt del proyecto y tareas.

A continuación, se presentan los hitos propuestos para el desarrollo, de los cuales los primeros 6 se siguen en un orden secuencial, mientras que los 2 siguientes se ejecutan de forma paralela a los anteriores. La metodología y planificación detallada se puede observar en la Figura 1, que muestra el diagrama de Gantt y la especificación de los objetivos.

1. Familiarización con el Entorno de MikroTik (RouterOS) y su API (Application Programming Interface).
2. Familiarización/validación de las tecnologías de redes.
3. Toma y estudio de los requisitos exactos del problema.
4. Diseño de la arquitectura de la solución.
5. Desarrollo de la solución.
6. Validación de la solución.
7. Redacción de la memoria.
8. Preparación de la presentación y la demostración.

IV. Estado del arte

Tras realizar un estudio de mercado, se ha identificado que existen grandes empresas que ofrecen servicios de SD-WAN que cubren las funcionalidades de generación de configuración de red por parámetros de capa 2, 3 y 4 a través de una consola centralizada [6] [7]. A continuación, se presentan brevemente las principales características de algunos de los productos destacados en el mercado y cómo destacan frente a sus competidores:

- **Cisco**: Cisco ofrece dos productos diferentes para SD-WAN. Cisco Meraki está diseñado para empresas pequeñas y medianas, con una implementación y uso sencillos, aunque con algunas funcionalidades limitadas. Cisco Viptela está dirigido a grandes empresas con topologías complejas y escenarios especiales, ofreciendo una solución totalmente adaptable y *customizable* a casos de uso específicos.
- **Fortinet Secure**: Este producto se destaca por integrar características de seguridad líderes en el mercado a nivel de hardware, proporcionando una de las soluciones de SD-WAN más seguras seguras sin comprometer el rendimiento.
- **Palo Alto**: Esta solución se presenta como la nueva generación de SD-WAN, debido a que permite un control directo sobre las aplicaciones en ejecución en la red y abre la posibilidad a configuraciones personalizadas para

ellas. También se destaca por incluir una fuerte seguridad en su enfoque de SD-WAN.

- **Juniper**: Juniper se enfoca en la experiencia del usuario y la seguridad, y se diferencia al incorporar inteligencia artificial en su solución. Esto le permite proporcionar información avanzada y solucionar automáticamente problemas que afecten a la experiencia del usuario, reduciendo la necesidad de intervención manual.
- **VMware**: VMware ofrece la solución VeloCloud, centrada en optimizar enlaces y mejorar el rendimiento de la red. Su enfoque *zero-touch*, es decir, de conectar y encender, permite un despliegue sencillo y automatiza totalmente las configuraciones y optimizaciones para la estructura.
- **Aryaka**: Aryaka se enfoca en soluciones basadas en la nube y destacan en tener acuerdos previos en regiones asiáticas para facilitar el despliegue en esas áreas, lo que permite una integración global rápida y sencilla. El despliegue de este producto es considerado de los más ágiles permitiendo pasar del WAN tradicional al SD-WAN en dos semanas.

Si bien todos estos productos ofrecen soluciones completas y adaptables a diferentes escenarios, es importante tener en cuenta que son soluciones propietarias que implican altos costos de adquisición, mantenimiento, gestión, actualización y que no permiten interoperabilidad. Como alternativa, en este proyecto se propone una solución *open source* adaptada al entorno MikroTik que se ha desarrollado con Docker y Python. Esta solución ofrece soporte para la tecnología SD-WAN y permite configuraciones L3VPN/L2VPN. Además, se puede escalar y ampliar fácilmente para agregar más tecnologías y adaptarse a escenarios más complejos.

V. Tecnologías vinculadas sobre redes

Antes de continuar con la presentación de la solución y la demostración de funcionamiento, es necesario presentar las tecnologías principales de redes que se usarán en este Trabajo de Fin de Grado y el papel que cumplirán dentro del mismo.

V-A. WireGuard

WireGuard [8] es un protocolo de encapsulación que se considera una evolución y una alternativa a soluciones ampliamente utilizadas como IPSec y OpenVPN. Destaca por su implementación sencilla y por ofrecer mejores resultados en términos de seguridad y rendimiento utilizando las últimas tecnologías de encriptación. Su despliegue implica la creación de una interfaz virtual, la asignación de una clave privada y la configuración de los dispositivos (llamados *peers*) que pueden utilizar el túnel VPN mediante la asignación de su dirección IP origen y clave pública. En este estudio, se utiliza WireGuard para establecer conexiones seguras y encriptadas entre las diferentes ubicaciones de los dispositivos, aprovechando Internet como red subyacente.

V-B. OSPF

OSPF [9] es un protocolo de enrutamiento dinámico de tipo enlace-estado que se basa en el algoritmo de Dijkstra para determinar la mejor ruta (SPF, *Shortest Path First*) entre dos puntos de una red. Todos los routers participan en el protocolo, creando tablas individuales con las rutas conocidas, sus costos asociados y los estados de los enlaces. Estas enlaces y sus respectivos estados se comparten entre los routers participantes, actualizando las entradas individualmente cuando se encuentran rutas mejores o nuevas, lo que permite una interconexión total de la red. Su capacidad para calcular rutas eficientes, su escalabilidad y su capacidad de adaptarse a cambios en la topología de la red lo convierten en una opción confiable para garantizar la conectividad y la eficiencia en entornos de red complejos. Se utiliza en el proyecto como protocolo de enrutamiento dinámico.

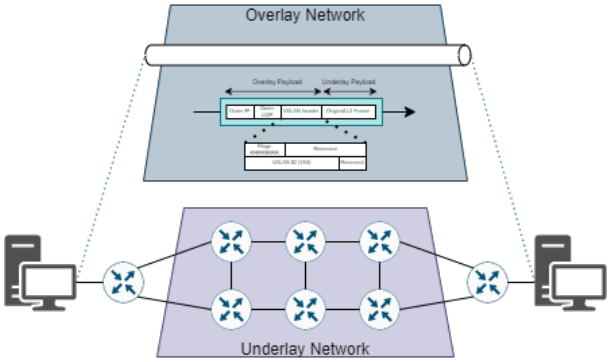


Figura 2: Concepto de red overlay y underlay en VXLAN.

V-C. VXLAN

VXLAN [10] surge de la necesidad de expandir topologías de red basadas en capa 2 sobre múltiples centros de datos sin las limitaciones impuestas por STP (*Spanning Tree Protocol*) y VLAN (*Virtual Local Area Network*) en términos de enlaces y segregación de dispositivos superando los 12 bits de VLAN que permiten un máximo de 4096 VLANs diferentes [11]. VXLAN permite la creación de una red *overlay* que admite transportar mensajes de capa 2 sobre una infraestructura de red interconectada en capa 3 mediante el uso de túneles lógicos [10]. Esto se conoce como el concepto de *overlay* y *underlay* en redes. VXLAN utiliza segmentos identificados con 24 bits para agrupar dispositivos que pueden comunicarse entre sí, lo que permite la coexistencia de hasta 16 millones de VXLANs. Esta capacidad ayuda a superar las limitaciones de VLAN y direcciones MAC, evitando repeticiones. VXLAN opera mediante el encapsulamiento de paquetes de capa 2 dentro de datagramas IP. Estos paquetes encapsulados son enrutados a través de la red de capa 3 como si estuvieran dentro de un túnel. Al llegar al destino, es decir, al final del túnel, se realiza el proceso de desencapsulación para mantener el nivel 2 del paquete original. De esta manera, VXLAN permite que los paquetes de capa 2 sean transportados de forma eficiente sobre una infraestructura de red de capa 3, este concepto de túnel lógico *overlay* se puede observar en la Figura 2. En este proyecto, se utiliza VXLAN para permitir la comunicación de dispositivos en capa 2 transportando tramas

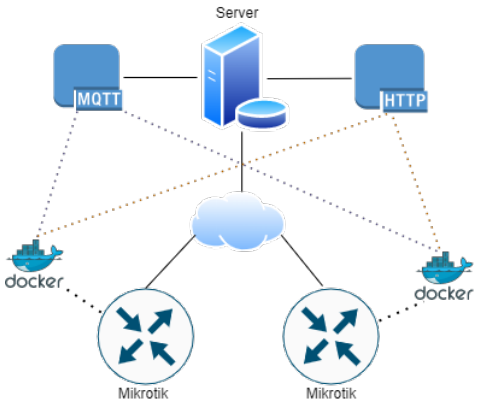


Figura 3: Arquitectura general.

Ethernet encapsuladas en paquetes IP que se transmiten a través de una red en capa 3.

VI. Arquitectura de la solución

La arquitectura mínima de la solución para automatizar la creación de servicios L2VPN se presenta en la Figura 3. Como se puede observar, la propuesta basa su funcionamiento en una estructura cliente-servidor, cuya estructura y objetivos generales se detallan a continuación.

VI-A. Cliente

En la topología, cada router MikroTik actua como un cliente. Para su correcto funcionamiento, es necesario que estos dispositivos utilicen una arquitectura basada en ARM y, que estén actualizados a una versión igual o superior de la 7.6 de RouterOS. Esta configuración permitirá la virtualización de máquinas a través de contenedores con la capacidad de iniciar automáticamente al arrancar.

Los routers MikroTik tienen dos roles principales en esta propuesta. En primer lugar, actúan como enrutadores de red, soportando la topología del cliente y enrutando paquetes en su operación habitual. En segundo lugar, ejecutan una imagen de Linux Alpine, la cual incluye Python y ejecuta el código responsable del plano de control SD-WAN. El verdadero cliente del servidor, por lo tanto, es el contenedor Docker que se ejecuta como contenedor dentro del dispositivo MikroTik (tanto físico como virtual) [12].

VI-B. Servidor

En esta solución, el servidor cumple la función de mantener un punto centralizado de configuración de la red. La idea es similar a las propuestas de grandes marcas en sus productos SD-WAN, donde se utiliza una consola central en la nube para gestionar la red y realizar cambios que se aplican a los dispositivos finales de manera remota.

Para llevar a cabo esta función, el servidor asume responsabilidad en las siguientes tareas:

1. Mantener los archivos que definen las topologías de red aplicadas, es decir, mantener un registro actualizado de la configuración de red en todo momento.

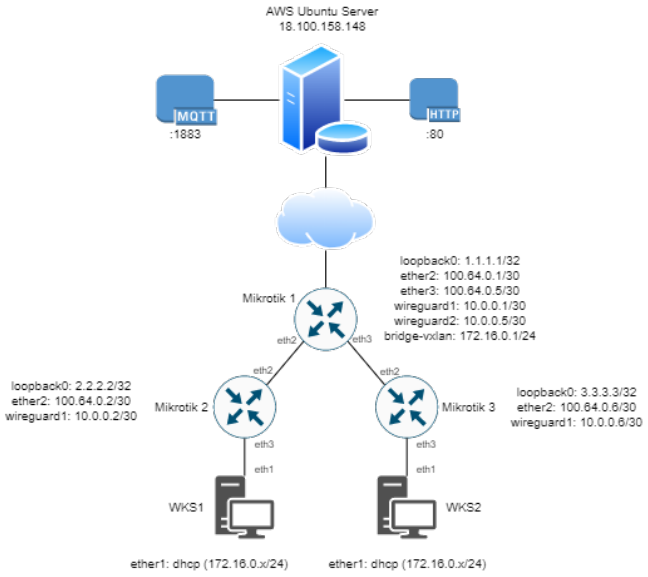


Figura 4: Topología L2VPN demostración.

2. Proporcionar un acceso a estas configuraciones por parte de los dispositivos finales a través de un canal de comunicación seguro y auténticado.
3. Permitir la modificación de las configuraciones.
4. Comunicarse con los clientes para notificar actualizaciones en la configuración a la que pertenecen.

Para cumplir con estas funciones de manera óptima y eficiente, se propone utilizar la combinación de dos tecnologías. Por un lado, un servicio HTTP basado en una API para todo lo relacionado con las configuraciones y la autenticación de usuarios. Por otro lado, un servicio de mensajería ligera MQTT (*Message Queuing Telemetry Transport*) para mantener a los dispositivos actualizados en tiempo real [13]. Aunque la Figura 3 muestra una arquitectura que pasa por Internet, esto en ningún momento en necesario y el despliegue se puede realizar acorde a las necesidades siguiendo esta metodología cliente-servidor.

VII. Desarrollo de los componentes, implementación y funcionamiento

En esta sección se entrará en detalle en cada uno de los actores de la solución, su implementación, estructura y código para entender el funcionamiento global. Previamente, resulta relevante cerciorarse y mantener presente que específicamente se tratará la implementación para una red L2VPN, cuya topología se puede observar en la Figura 4 y que tiene como objetivo la comunicación de los dispositivos finales como si estuvieran en la misma LAN. Se escoge montar esta topología L2VPN porque para realizarla es imprescindible aplicar anteriormente una L3VPN con los túneles Wireguard, por lo que con una sóla demostración se implementa la solución total y más completa.

VII-A. Servidor

Haciendo referencia a la Sección VI-B, el servidor desempeña un papel principal en la propuesta siendo el punto central

de control y notificación. Es el encargado de mantener las configuraciones de los clientes actualizadas en tiempo real.

Para realizar esta tarea, el servidor dispone y mantiene ficheros en formato JSON [14], que contienen la información necesaria para la operativa. En concreto guarda y utiliza los siguientes archivos:

1. `user_db.json`: Este documento contiene la información de los usuarios para realizar una autenticación básica. El sistema considera usuario al cliente de negocio, es decir, a la persona que emplea o posee la solución. Este usuario tendrá a su vez un número `N` de clientes que corresponderán a los routers y utilizarán la autenticación del usuario. Específicamente se guarda el nombre de usuario y un *hash* de la contraseña creado a partir de una sal aleatoria en un esquema `clave:valor`, el cual se demuestra a continuación.

```
{"user":"$2b$12$PZdycgz8KoFgx7.W9txu1.u3G93V98u52.
    NEOQGaVgGy1kfuAmS3a"}
```

2. `templates.json`: Este archivo contiene traducciones de acciones posibles en el entorno RouterOS en forma de cadenas de texto, que pueden utilizarse como plantillas de comandos de configuración. Su objetivo es brindar a los clientes una plantilla de la acción deseada, donde todas las cadenas que contengan el símbolo *'$'* deben reemplazarse por valores específicos del enrutador antes de aplicarla a la configuración. En este archivo, cada acción y su traducción se guardan en un esquema `clave:valor`. Un ejemplo de algunas acciones:

```
{"ipAddresses" : "\n/ip address add address=$address
    interface=$ifname",
"dhcp_client": "\n/ip dhcp-client add interface=
    $ifname",
"interface_bridge" : "\n/interface bridge add name=
    $bridgename"}
```

3. `assets.json`: Este archivo contiene información clave sobre la topología de cada cliente, incluyendo los parámetros específicos necesarios para generar su configuración. Los clientes utilizan esta información para determinar las acciones a realizar, solicitar traducciones al servidor extraídas del fichero anterior, reemplazar los valores en las plantillas y generar la configuración correspondiente. El formato de este archivo puede ser complejo y variable según la topología deseada, pero el concepto general se mantiene y se presenta a continuación:

```
{"user": {
    "router1" : {
        "technology" : "configValues",
        "technology2" : "configValues2"
    },
    "router2" : {
        "technology" : "configValues"
    }
}}
```

La generación de configuraciones se delega a los clientes para evitar sobrecargar el servidor en términos de procesamiento y memoria, así como para evitar problemas de sincronización donde el cliente pueda solicitar una configuración que el servidor aún no ha calculado.

Para continuar con la explicación sobre cómo realiza toda la operativa, es necesario presentar los servicios MQTT y HTTP que ejecuta el servidor. Estos se levantan utilizando imágenes Docker con el siguiente fichero `docker-compose`:

```
version : '3.8'
services:
  mosquitto:
    image: eclipse-mosquitto
    container_name: mosquitto
    volumes:
      - /opt/mosquitto:/mosquitto
      - /opt/mosquitto/data:/mosquitto/data
      - /opt/mosquitto/log:/mosquitto/log
      - /opt/mosquitto/config/mosquitto.conf:/mosquitto/
     config/mosquitto.conf
    ports:
      - 1883:1883
      - 9001:9001
  compose_flask:
    image: saek13/configserverv2
    container_name: flask-server
    volumes:
      - /opt/compose_flask:/app
    ports:
      - 80:5000
```

En él se puede observar que se utiliza la imagen base de Mosquitto para levantar un *broker* MQTT en el puerto 1883 y una imagen personalizada de Flask que contiene el servicio HTTP utilizando Python en el puerto 80 [15]. Ambas imágenes están disponibles públicamente en el repositorio general de DockerHub.

VII-A1. MQTT

MQTT es un protocolo de mensajería ligero que se basa en el modelo de publicación/suscripción. En este modelo, los dispositivos se conectan a un intermediario llamado *broker*, a través del cual pueden publicar mensajes en temas específicos. Otros dispositivos interesados pueden suscribirse a estos temas para recibir los mensajes.

En el escenario de esta solución, el servidor aloja un servicio de *broker* Mosquitto básico. Tanto los clientes como el servidor pueden conectarse a este *broker* para intercambiar mensajes. Específicamente, se crea un tema o hilo para cada usuario, al cual todos los enrutadores clientes se suscriben. En este tema, solo el servicio HTTP tiene la capacidad de publicar dos tipos de mensajes. El primero es una notificación de actualización identificada por la cadena `update`, y el segundo es un *rollback* marcado con la cadena `base`. Posteriormente se entrará más en detalle sobre el uso de estas cadenas.

VII-A2. HTTP

HTTP es un protocolo de comunicación para transferir datos entre un cliente y un servidor basado en peticiones y respuestas entre ambos. Una REST API (*Representational State Transfer API*) es una interfaz de programación que define y estructura un conjunto de *endpoints* o rutas que permiten a los clientes enviar y recibir solicitudes para realizar acciones HTTP con el servidor web.

En la solución propuesta, se crea una aplicación web en formato REST API utilizando Python y la librería Flask que permitirá a los clientes interaccionar con el servidor para recibir la información de los ficheros JSON. Esta aplicación web construye la imagen utilizada en el `docker-compose` utilizando el siguiente Dockerfile, donde básicamente se instalan las librerías necesarias para su funcionamiento y se ejecuta el servicio a nivel de anfitrión con la directiva '–host=0.0.0.0':

```
# syntax=docker/dockerfile:1
FROM python:3.8-slim-buster
```

```
WORKDIR /python-docker
COPY requirements.txt requirements.txt
RUN pip3 install -r requirements.txt
COPY . .
CMD [ "python3", "-m" , "flask", "run", "--host=0.0.0.0"]
```

La aplicación, por lo tanto, debe cumplir con todas las acciones definidas en la Sección VI-B. Ahora se analizará caso por caso cómo se realiza y el código completo se encontrará en el Apéndice A. No obstante, el primer paso resulta en la importación de las librerías necesarias para crear la REST API, realizar operaciones con ficheros JSON y utilizar el servicio MQTT, además de crear la propia aplicación web.

```
from flask import Flask, request
from flask_restful import Resource, Api
from flask_httpauth import HTTPBasicAuth
import json
import paho.mqtt.client as mqtt
import base64
import bcrypt
app = Flask(__name__)
api = Api(app)
```

La aplicación al arrancar lee los archivos que contienen toda la información de la solución y los guarda en variables. Además, define un método `getClient` que permite sacar el usuario directamente desde la cabecera de autorización.

```
with open("user_db.json") as f:
    USER_DATA = json.load(f)
with open("assets.json") as f:
    CONFIG = json.load(f)
with open("templates.json") as f:
    TEMPLATES = json.load(f)
def getClient(message):
    base64_bytes = message.split(" ")[1].encode('ascii')
    message_bytes = base64.b64decode(base64_bytes)
    message = message_bytes.decode('ascii')
    return message.split(":")[0]
```

Se establece una función de autenticación básica en el servidor en la que se comprueba si el usuario y contraseña son válidos según los definidos en el fichero JSON. Para utilizar dicha función posteriormente será necesario únicamente añadir la línea `@auth.login_required`.

```
@auth.verify_password
def verify(username, password):
    if not (username and password and USER_DATA.get(
     username)):
        return False
    return bcrypt.checkpw(password.encode('utf-8'),
     USER_DATA.get(username).encode('utf-8'))
```

La información del fichero `assets.json` es accesible principalmente por dos *endpoints* protegidos con autenticación. El primero permitirá recibir por completo la topología del usuario que autentica. El segundo utilizará un identificador concreto de dispositivo para devolver solo la información específica de ese cliente para el usuario que autentica.

```
@app.route('/api/v1/config', methods=['POST'])
@auth.login_required
def downloadConfig():
    if request.form.get('identifier'):
        id = request.form.get('identifier')
    else:
        return "Error: Identifier not especified."
    authClient = getClient(request.headers['Authorization
     '])
    if authClient in CONFIG:
        if id in CONFIG[authClient]:
            return CONFIG[authClient].get(id)
        else:
            return "Error: Identifier does not exist in
     your namespace"
    else:
        return "Error: You're not allowed to access"
@app.route('/api/v1/wholeConfig', methods=['GET'])
@auth.login_required
def wholeConfig():
    authClient = getClient(request.headers['Authorization
     '])
    if authClient in CONFIG:
        return CONFIG.get(authClient)
    else:
        return "Error: You're not allowed to access"
```

De forma equivalente se define un *endpoint* que permite recibir una plantilla de las definidas en el fichero JSON utilizando el identificador de la acción.

```
@app.route('/api/v1/templates', methods=['POST'])
@auth.login_required
def getTemplate():
    if request.form.get('template'):
        template = request.form.get('template')
    else:
        return "Error: Template not especified."

    if template in TEMPLATES:
        return TEMPLATES[template]
    else:
        return "Error: Template does not exist"
```

Por último, se añaden dos *endpoints* que serán los que interactúen con el *broker* MQTT. El primero simplemente publicará en el hilo pertinente la cadena `base` para hacer un *rollback* a todos los dispositivos de un usuario. El segundo permitirá mediante la recepción de una cadena JSON, la modificación de los valores clave de la topología de un usuario. Una vez actualizados, se publicará el mensaje `update` en el hilo correspondiente para que todos los dispositivos soliciten, generen y apliquen su nueva configuración.

```
@app.route('/api/v1/modifyConfig', methods=['POST', 'GET'])
@auth.login_required
def modifyConfig():
    if request.method == "GET":
      return wholeConfig()
    else:
      request.get_data()
      if not request.data:
        return wholeConfig()
      else:
            authClient = getClient(request.headers['
     Authorization'])
            if request.headers['Content-Type'] == '
     application/json':
                json_data = json.loads(request.get_data().
     decode('utf-8'))
                CONFIG[authClient] = json_data
                with open("assets.json", "w+") as f:
                    json.dump(CONFIG, f)
                client.publish(f"JSNetworkManagement/{
     authClient}", "update", qos = 1)
                return "OK"
@app.route('/api/v1/backToBase', methods=['GET'])
@auth.login_required
def backToBase():
    authClient = getClient(request.headers['Authorization
     '])
    client.publish(f"JSNetworkManagement/{authClient}", "
     base", qos = 1)
    return "Published"
```

De esta forma, el servicio web proporciona todos los requisitos funcionales anteriormente descritos y mantiene una consola central que permite, en tiempo real, mantener a todos los dispositivos siempre actualizados con la última versión de la información.

VII-B. Enrutador MikroTik

Los otros componentes protagonistas de la propuesta son los propios enrutadores MikroTik que forman la topología

del usuario. Como se ha visto anteriormente, estos son los encargados del plano de datos, es decir, de realizar la propia operativa de procesamiento y enrutamiento de paquetes en la red del usuario y, además, de virtualizar un contenedor que será encargado del plano de control SD-WAN.

Para que la solución funcione correctamente, es necesario que estos equipos tengan al menos RouterOS v7.6. Esta versión es crucial, ya que a partir de la versión 7.5 se introdujo la capacidad de virtualización y en la 7.6 se habilitó el arranque desde el inicio, lo que garantiza que el enrutador mantenga el contenedor siempre encendido para que el servicio de SD-WAN esté operativo.

El uso del sistema operativo RouterOS tiene algunas limitaciones en cuanto a funcionalidades como por ejemplo el impedimento de crear ficheros con una extensión diferente a `txt` o que el orden en el que se exportan los comandos no es necesariamente el orden en el que se deben importar para el correcto funcionamiento. Sin embargo, RouterOS también presenta dos ventajas significativas que son las que permiten aplicar configuraciones en tiempo real y mantener activo el plano de control SD-WAN de forma permanente. La primera de ellas radica en que el sistema operativo en sí mismo ofrece una API pública que permite la conexión externa al enrutador mediante cualquier lenguaje de programación [16]. La segunda ventaja, y aún más importante, es que se pueden utilizar archivos de configuración al reiniciar, lo que implica la ejecución de todos los comandos del fichero en forma de script durante el arranque y la eliminación de cualquier configuración anterior.

En este caso, al igual que con el servidor, se desarrolla una aplicación utilizando Python para llevar a cabo todas estas acciones. Utilizando dicha aplicación se construye una imagen de Docker mediante un archivo Dockerfile, la cual se expone públicamente en un repositorio de DockerHub. Posteriormente, esta imagen se importará en el enrutador y se ejecutará para obtener la configuración de la topología correspondiente y unirse al plano de control SD-WAN.

Es importante destacar que la importación del contenedor a los enrutadores a través de DockerHub no es la única opción disponible. En escenarios con altas restricciones, puede ser necesario explorar otras alternativas. Por ejemplo, se puede considerar la opción de grabar la imagen del contenedor en una memoria externa y luego importarla en los enrutadores o incluso también es posible ejecutar el contenedor directamente desde la memoria externa. Estas alternativas ofrecen flexibilidad y permiten adaptarse a diferentes entornos y requisitos específicos.

Entrando más en detalle en la implementación, se procede a analizar el archivo Dockerfile utilizado para construir la imagen. En este archivo se observa cómo se importa una imagen base de Python en Linux Alpine, se instalan las bibliotecas necesarias y se ejecuta la aplicación. Sin embargo, las líneas 4-8 adquieren especial relevancia, ya que definen las variables de entorno de la imagen. Es en este punto en el que se especifica la dirección IP del servidor, así como las credenciales de usuario y contraseña tanto para el servidor como para el enrutador MikroTik.

Estas variables, cuyos valores se definen para cada enrutador y se asignan durante la importación del contenedor, serán utilizadas por la aplicación para establecer comunicación y autenticarse con las diferentes plataformas. De esta forma, se podrá adaptar la imagen del contenedor a cada enrutador específico y garantizar una interacción adecuada con los sistemas de la solución.

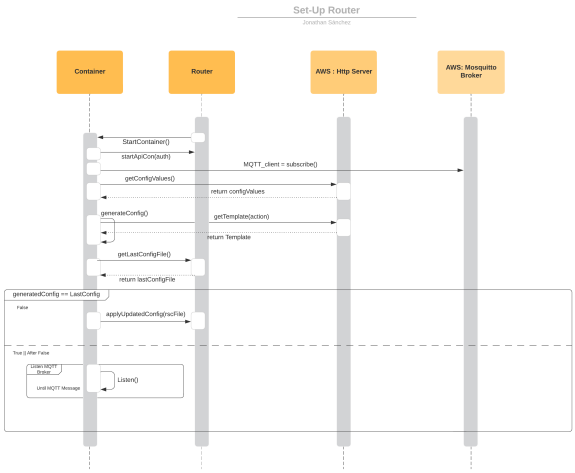


Figura 5: Diagrama de secuencia funcionamiento general.

```
1  FROM python:3.11-rc-alpine
2  RUN apk update
3  RUN apk add git
4  ENV ENDPOINT http://18.100.158.148
5  ENV ROUTER_USER admin
6  ENV ROUTER_PWD ""
7  ENV CLIENT_USER ""
8  ENV CLIENT_PWD ""
9  WORKDIR /app
10 COPY requirements.txt requirements.txt
11 RUN pip3 install --upgrade pip
12 RUN pip3 install -r requirements.txt
13 RUN pip3 install git+https://github.com/sukawatd/RouterOS-
       api.git
14 COPY . .
15 CMD ["python", "app.py"]
```

Antes de continuar con la implementación de la aplicación, es necesario comprender cuál será el funcionamiento de la misma y cómo se relaciona con los demás actores de la solución. Para ello, se presenta el diagrama de secuencia en la Figura 5.

En el proceso descrito, el enrutador MikroTik desempeña un papel fundamental al iniciar la cadena mediante la importación y arranque del contenedor. Una vez que la aplicación comienza a ejecutarse, lo primero que hace es establecer una conexión con el enrutador anfitrión utilizando la API para ejecutar comandos directamente en el dispositivo desde el contenedor. Además, se conecta al *broker* MQTT y se suscribe al canal correspondiente.

A partir de ahí, la aplicación solicita al servidor HTTP los parámetros clave más recientes del enrutador para generar el archivo de configuración. Durante la creación de este archivo, se realizan múltiples llamadas al servidor, donde se traducen las acciones específicas en comandos MikroTik mediante la sustitución de plantillas de comandos con los valores recibidos previamente. Luego, se lee un archivo llamado `newConfig.rsc` desde la memoria del enrutador

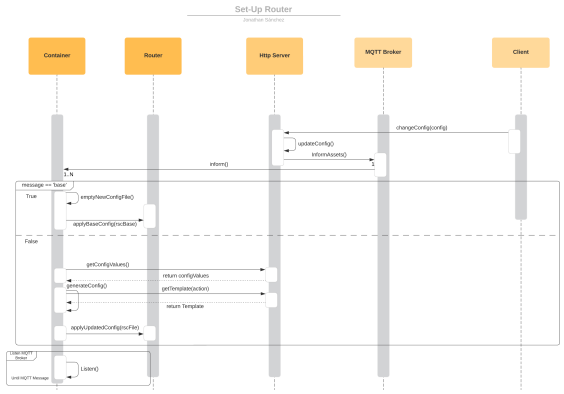


Figura 6: Diagrama de secuencia mensaje MQTT.

MikroTik, donde se guarda la última configuración aplicada al dispositivo.

A continuación, se comprueba si el contenido de ese archivo y la configuración generada son iguales. Si son diferentes, significa que el enrutador no tiene aplicada la última versión de la topología. En tal caso, el contenedor reiniciará el enrutador utilizando el nuevo archivo de configuración y actualizará el contenido de `newConfig.rsc` con esta nueva configuración.

Después de reiniciar y completar nuevamente el proceso, el enrutador tendrá obligatoriamente la última versión de la configuración. Por lo tanto, ambos flujos convergerán y el siguiente paso del contenedor será permanecer en un estado de escucha de mensajes MQTT para actualizarse o realizar un *rollback* según sea necesario. Es importante destacar que el contenedor es capaz de recibir y procesar los mensajes MQTT en cualquier momento durante la ejecución.

El proceso de recepción de un mensaje MQTT se muestra en el diagrama de secuencia de la Figura 6. Como se mencionó anteriormente en la operativa del servidor, los mensajes MQTT se envían únicamente cuando un cliente inicia el proceso de modificar una configuración o realizar un *rollback* a la topología base. En ese momento, el servidor guarda la nueva configuración y notifica a los suscriptores publicando un mensaje en el broker MQTT. Los contenedores reciben el mensaje y realizan un filtrado basado en la cadena recibida, ya sea `base` o `update`.

En el caso de recibir la cadena `base`, se elimina el contenido del archivo con la configuración actual del enrutador y se reinicia utilizando un archivo que contiene una copia de seguridad de la topología base. Por otro lado, si se recibe la cadena `update`, se genera la nueva configuración y se aplica en el enrutador siguiendo el proceso visto anteriormente. Finalmente, en ambos casos, una vez que el enrutador se reinicia, el contenedor vuelve al estado de escucha de mensajes MQTT para estar atento a futuras actualizaciones o rollback.

Cabe destacar que el fichero que funciona como copia de seguridad y que se muestra completo en el Apéndice C, cumple un doble propósito:

1. Mantener la configuración mínima esencial de la topología y del dispositivo en un estado funcional. Esto implica proporcionar una conectividad básica, establecer el identificador del dispositivo, crear el archivo `newConfig.rsc` si no existe, aplicar la configuración necesaria para habilitar la virtualización, definir las variables de entorno del contenedor y finalmente importar y arrancar el contenedor.
2. Servir como plantilla base sobre la que ir añadiendo comandos cuando se genera una nueva configuración.

Continuando con el análisis, el siguiente paso consiste en examinar la implementación de la aplicación para llevar a cabo esta operativa. El código completo y ordenado se encuentra en el Apéndice B. En primer lugar, la aplicación importa las librerías necesarias para su funcionamiento y luego espera 20 segundos para permitir que el anfitrión se inicie completamente. A continuación, lee los valores de las variables de entorno que se pasaron al arrancar el contenedor.

```
import routeros_api
import json as js
import requests
import time
import paho.mqtt.client as mqtt
from string import Template
import os
time.sleep(20)
ENDPOINT = os.environ['ENDPOINT']
ROUTER_USER = os.environ['ROUTER_USER']
ROUTER_PWD = os.environ['ROUTER_PWD']
CLIENT_USER = os.environ['CLIENT_USER']
CLIENT_PWD = os.environ['CLIENT_PWD']
```

Seguidamente, el contenedor establece una conexión mediante la API con el enrutador y procede a leer el archivo base y obtener el identificador del dispositivo. En este punto, la aplicación también establece una conexión con el *broker* MQTT y configura la suscripción al tema correspondiente, definiendo las acciones a realizar con los mensajes recibidos. Una vez establecidas estas configuraciones, la aplicación comienza a escuchar los mensajes de forma paralela a la ejecución principal.

```
def on_message(client, userdata, message):
  print(f"Received message: {message.payload.decode('utf
     -8')}")
  print(f"QoS : {message.qos}")
  msg = message.payload.decode('utf-8')
  if "update" in msg:
    myConfig = js.loads(requests.post(f"{ENDPOINT}/api/v1/
     config", data = { "identifier" : f"{systemIdentity}"},
     auth = (f"{CLIENT_USER}", f"{CLIENT_PWD}")).text)
    newRSC = setUp(myConfig)
    newConfigFileContent = filePointer.get(name="newConfig.
     rsc")[0]['contents']
    if newRSC != newConfigFileContent:
      applyConfig(newRSC)
  elif "base" in msg:
    aux_file = filePointer.get(name="newConfig.rsc")[0]['id
     ']
    filePointer.set(id=aux_file, contents="")
    api.get_binary_resource('/system').call('reset-
     configuration', { 'run-after-reset': "base.rsc", "skip-
     backup":"yes" })
def on_connect(client, userdata, flags, rc):
  client.subscribe(f"JSNetworkManagement/{CLIENT_USER}", 1)
mqttBroker = ENDPOINT.split("//")[1]
client = mqtt.Client(clean_session=True)
client.on_connect = on_connect
client.on_message = on_message
client.connect(mqttBroker, keepalive=0)
client.loop_start()
```

Como se muestra en la función `on_message`, el contenedor realiza un filtrado de la cadena recibida y establece dos flujos de trabajo distintos. En el caso de recibir `update`, se inicia el proceso realizando una solicitud al servidor HTTP

para obtener los valores clave de configuración. A continuación, se llama a la función `setUp`, que generará la nueva configuración completa. Después, se lee el contenido del archivo `newConfig.rsc` y se compara con la configuración generada recientemente. Si son idénticos, se ignora el mensaje de actualización. En caso de que sean diferentes, se aplica la nueva configuración, lo que implica cambiar el contenido del archivo y reiniciar el enrutador con esa configuración actualizada.

Por otro lado, si se recibe la cadena `base`, el proceso es más sencillo. Simplemente se borra el contenido del archivo `newConfig.rsc` y se reinicia el enrutador utilizando la configuración base.

A continuación se presentan las funciones `applyConfig` y `setUp`. La función `setUp`, que es la operativa principal de la solución, se encuentra completamente detallada en el Apéndice B, aquí se añade sólo un fragmento. Esta rutina recibe los parámetros clave del servidor y se encarga de agregar a la plantilla base todos los comandos necesarios para implementar los valores recibidos. El diccionario de parámetros está estructurado de tal manera que las primeras claves corresponden a las tecnologías a aplicar. La función convierte cada tecnología recibida en las acciones necesarias para configurarla. Luego, utiliza estas acciones para solicitar al servidor las plantillas de comandos y reemplazarlas con los valores recibidos, generando así el comando final que se agrega al archivo de configuración. Finalmente, se devuelve el archivo completo generado. Además, se incluye la función `getTemplate`, utilizada por `setUp` para solicitar al servidor las plantillas de comandos, proporcionando una acción como parámetro.

```
def getTemplate(template):
  return Template((requests.post(f"{ENDPOINT}/api/v1/
     templates", data = { "template" : f"{template}"}, auth
     = (f"{CLIENT_USER}", f"{CLIENT_PWD}")).text).replace
     ("'", '"'))
def applyConfig(config):
  idConfig = filePointer.get(name="newConfig.rsc")[0]['id']
  filePointer.set(id=idConfig, contents=config)
  api.get_binary_resource('/system').call('reset-
     configuration', { 'run-after-reset': "newConfig.rsc", "
     skip-backup":"yes" })
def setUp(myConfig):
  newConfig = base
  if "ipAddresses" in myConfig:
    template = getTemplate("ipAddresses")
    for parameters in myConfig["ipAddresses"]:
      substitute = template.substitute(address=parameters["
     inetAddress"], ifname=parameters["inetInterface"])
      newConfig += substitute
   return newConfig
```

Retomando el flujo principal de ejecución, después de configurar la suscripción MQTT y la operativa al recibir un mensaje, la aplicación realiza una primera iteración de generación del archivo de configuración para verificar si lo que se encuentra actualmente aplicado es la versión más actualizada. El proceso es idéntico al que se realiza al recibir una cadena `setUp`, pero se agrega una condición adicional para el archivo `newConfig.rsc`. Esta condición verifica si el archivo está vacío, lo cual indicaría que se ha realizado un *rollback*. Si la condición se cumple, no se realiza ninguna acción, manteniendo la configuración base y preservando que se aplique la última configuración del servidor hasta que se reciba una actualización correspondiente (`update`). Por último, para evitar que el contenedor termine debido a la finalización del hilo principal de la aplicación, se implementa un bucle infinito en el cual la aplicación duerme durante 5 segundos, evitando un uso excesivo de ciclos de la CPU.

```
myConfig = js.loads(requests.post(f"{ENDPOINT}/api/v1/
     config", data = { "identifier" : f"{systemIdentity}"},
     auth = (f"{CLIENT_USER}", f"{CLIENT_PWD}")).text)
newRSC = setUp(myConfig)
newConfigFileContent = filePointer.get(name="newConfig.rsc
     ")[0]['contents']
if newConfigFileContent != "":
  if newRSC != newConfigFileContent:
    applyConfig(newRSC)
while True:
  time.sleep(5)
```

VIII. Demostración

Hasta ahora, se ha explicado el funcionamiento detallado de la solución y se han proporcionado los códigos fuente de las aplicaciones para aplicar configuraciones L3VPN/L2VPN en los apéndices. Sin embargo, para determinar de manera concluyente si el trabajo ha sido exitoso o no, es imprescindible llevar a cabo una demostración y validación del funcionamiento.

Para llevar a cabo la demostración, se implementará la topología descrita en la Figura 4. El objetivo es establecer una conexión en capa 2 utilizando VXLAN entre los enrutadores y los equipos finales. Además, se configurarán túneles WireGuard entre los enrutadores MikroTik2 y MikroTik3 con el MikroTik1 para garantizar la confidencialidad y seguridad de las conexiones debido a que se utilizará la red pública de Internet. Para gestionar el enrutamiento, se utilizará el protocolo OSPF, el cual se encargará de distribuir las rutas dinámicamente entre los enrutadores.

La demostración se realiza utilizando las imágenes virtuales del sistema operativo RouterOS de MikroTik, que se pueden ejecutar en máquinas grandes y en diferentes entornos de virtualización. Estas imágenes virtuales se encuentran disponibles de manera pública y se denominan CHR (*Cloud Hosted Router*). En concreto, se crearán las cinco máquinas descritas anteriormente utilizando VirtualBox y se configurarán los ajustes de red según la topología, simulando el entorno como si estuviéramos trabajando con equipos físicos. Además, a cada máquina se le añadirá en memoria el archivo base correspondiente, que les permitirá arrancar el contenedor y conectarse al plano de control de la red.

Una vez aquí, será necesario crear el archivo de valores claves y tecnologías que posteriormente almacenaremos en el servidor. Esta simple tarea consiste en visualizar cómo se accede a las tecnologías y a los valores en la función `setUp` para añadirlas de igual forma a un archivo en formato JSON. Por ejemplo, para configurar la tecnología de cliente de DHCP (*Dynamic Host Configuration Protocol*) en el MikroTik4 se utiliza el siguiente código:

```
if "dhcp_client" in myConfig:
    ##Create new dhcp clients
    template = getTemplate("dhcp_client")
    for parameters in myConfig["dhcp_client"]:
      substitute = template.substitute(ifname=parameters["
     ifName"])
      newConfig += substitute
```

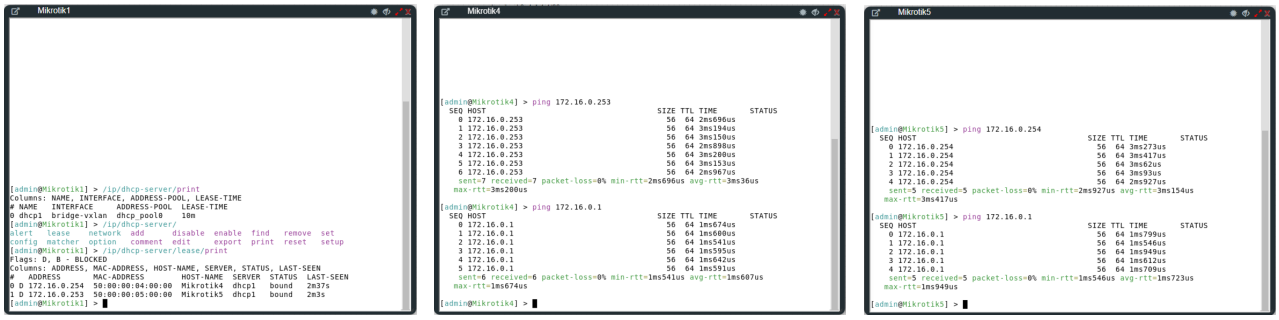


Figura 7: Verificaciones de funcionamiento. Izquierda tabla DHCP. Derecha ping entre dispositivos con TTL 64.

Esto implica que el fichero JSON que se debe generar para aplicar dicha tecnología debe ser:

```
{"user" : {
    "MikroTik4" : {
        "dchp_client" : [
            {"ifName" : "ether1"}
        ]
    }
}}
```

Una vez realizada la traducción completa y subido el fichero al servidor, se procede a arrancar los servicios con el `docker-compose` y a encender las máquinas. A continuación, los dispositivos automáticamente proceden a generar su configuración y se reinician para aplicarla. Una vez reiniciados, se valida el funcionamiento mediante la observación de la tabla de direcciones IP entregadas por el servidor DHCP del MikroTik1. En ella se observan las dos entradas correspondientes a los dispositivos finales. También, se demuestra cómo pueden hacerse ping entre ellos sin disminuir el TTL (*Time To Live*), es decir, imitando el comportamiento de entrega directa en un segmento Ethernet a través de la relación MAC-IP con ARP. Estas verificaciones se pueden observar en la Figura 7. Además, se comprueba el funcionamiento esperado de las cadenas MQTT `base` y `update` mediante la observación de las configuraciones aplicadas después de reiniciarse.

IX. Conclusiones y líneas de trabajo futuro

Como conclusión final, se puede afirmar que se ha logrado el objetivo del Trabajo de Fin de Grado consistente en desarrollar e implementar una solución SD-WAN de bajo coste y con código abierto capaz de aplicar configuraciones de red L2VPN y L3VPN de manera automática y en tiempo real en dispositivos MikroTik, validando su correcto funcionamiento con la topología de red de la Figura 4.

Las principales líneas de trabajo futuras residen en la incorporación de casos de uso a la solución, de modo que sea capaz de soportar un mayor rango de topologías, protocolos y tecnologías de red. Esto permitiría aún más, adaptar la solución al entorno específico de cada usuario ampliando la utilización de la misma. De hecho, otra vía interesante y con mucho futuro consistiría en crear una abstracción de la arquitectura de la solución para poder aplicarla en otras familias de dispositivos.

Otras vertientes para trabajo futuro pueden consistir en añadir certificados a la solución para transportar todo el tráfico encriptado por SSL (*Secure Socket Layer*), implementar autenticación por JWT (*JSON Web Token*) o OAuth2 (*Open Authorization 2*) a la API de Flask e incluso establecer el uso de canales MQTT protegidos por autenticación y SSL para evitar ataques de *spoofing*.

X. Agradecimientos

Me gustaría agradecer a mi tutor, Pere Tuset-Peiró, por su dedicación, mentoría y apoyo durante todo el proyecto.

También me gustaría agradecer a mi familia: Paula, Lucía y Pedro, a mi pareja, Minerva, y a mis amigos porque sin ellos, su cariño y su apoyo, nada de esto habría sido posible ni hubiera valido la pena.

Referencias

[1] Z. Yang, Y. Cui, B. Li, Y. Liu, and Y. Xu, "Software-defined wide area network (sd-wan): Architecture, advances and opportunities," *2019 28th International Conference on Computer Communication and Networks (ICCCN)*, 07 2019.

[2] R. K. Rangan, "Trends in sd-wan and sdn," *CSI Transactions on ICT*, 04 2020.

[3] "Routeros - routeros - mikrotik documentation," help.mikrotik.com. [Online]. Available: https://help.mikrotik.com/docs/

[4] S. Alsaqqa, S. Sawalha, and H. Abdel-Nabi, "Agile software development: Methodologies and trends," *International Journal of Interactive Mobile Technologies*, vol. 14, p. 246–270, 07 2020.

[5] A. Kaur, "App review: Trello," *Journal of Hospital Librarianship*, vol. 18, pp. 95–101, 01 2018.

[6] C. Mulqueen, "8 leading sd-wan providers: A comprehensive analysis," STL Partners. [Online]. Available: https://stlpartners.com/articles/telco-cloud/sd-wan-providers/

[7] "The best managed sd wan providers (with comparison app)," www.netify.com. [Online]. Available: https://www.netify.com/best-managed-sd-wan-providers

[8] J. A. Donenfeld, "Wireguard: Next generation kernel network tunnel," *Proceedings 2017 Network and Distributed System Security Symposium*, 2017. [Online]. Available: https://pdfs.semanticscholar.org/de53/d55470addc0e0d52fd82c2f4bd8e4ca879c2.pdf

[9] J. T. Moy and A. And, *OSPF : anatomy of an Internet routing protocol*. Addison-Wesley, 2002.

[10] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright, "Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks," IETF, 08 2014. [Online]. Available: https://www.rfc-editor.org/info/rfc7348

[11] "Introduction to virtual extensible lan (vxlan)," NetworkLessons.com, 02 2020. [Online]. Available: https://networklessons.com/cisco/ccnp-encor-350-401/introduction-to-virtual-extensible-lan-vxlan

[12] A. Mouat, *Using Docker*. O'reilly, 2016.

[13] R. A Light, "Mosquitto: server and client implementation of the mqtt protocol," *The Journal of Open Source Software*, vol. 2, p. 265, 05 2017.

[14] M. Pandey and R. Pandey, "Json and its use in semantic web," *International Journal of Computer Applications*, vol. 164, pp. 10–16, 04 2017.

[15] "Api — flask documentation (2.3.x)," flask.palletsprojects.com. [Online]. Available: https://flask.palletsprojects.com/en/2.3.x/api/

[16] J. Gocławski, "Routeros-api," GitHub, 03 2023. [Online]. Available: https://github.com/socialwifi/RouterOS-api

APÉNDICE A
CÓDIGO COMPLETO DEL SERVIDOR

```
from flask import Flask, request
from flask_restful import Resource, Api
from flask_httpauth import HTTPBasicAuth
import json
import paho.mqtt.client as mqtt
import base64
import bcrypt

app = Flask(__name__)
api = Api(app)
auth = HTTPBasicAuth()
mqttBroker = "18.100.158.148"
client = mqtt.Client("Server Publisher")
client.connect(mqttBroker, keepalive=0)

with open("user_db.json") as f:
    USER_DATA = json.load(f)

with open("assets.json") as f:
    CONFIG = json.load(f)

with open("templates.json") as f:
    TEMPLATES = json.load(f)

def getClient(message):
    base64_bytes = message.split(" ")[1].encode('ascii')
    message_bytes = base64.b64decode(base64_bytes)
    message = message_bytes.decode('ascii')
    return message.split(":")[0]

@auth.verify_password
def verify(username, password):
    if not (username and password and USER_DATA.get(
     username)):
        return False
    return bcrypt.checkpw(password.encode('utf-8'),
     USER_DATA.get(username).encode('utf-8'))

@app.route('/api/v1/config', methods=['POST'])
@auth.login_required
def downloadConfig():
    if request.form.get('identifier'):
        id = request.form.get('identifier')
    else:
        return "Error: Identifier not especified."

    authClient = getClient(request.headers['Authorization
     '])
    if authClient in CONFIG:
        if id in CONFIG[authClient]:
            return CONFIG[authClient].get(id)
        else:
            return "Error: Identifier does not exist in
     your namespace"
    else:
        return "Error: You're not allowed to access"

@app.route('/api/v1/wholeConfig', methods=['GET'])
@auth.login_required
def wholeConfig():
    authClient = getClient(request.headers['Authorization
     '])
    if authClient in CONFIG:
        return CONFIG.get(authClient)
    else:
        return "Error: You're not allowed to access"

@app.route('/api/v1/publishMQTT', methods=['GET'])
@auth.login_required
def publish():
    client.publish("JSNetworkManagement/server", "I have
     just published a debug message", qos = 1)
    return "Published"

@app.route('/api/v1/modifyConfig', methods=['POST', 'GET'])
@auth.login_required
def modifyConfig():
    if request.method == "GET":
      return wholeConfig()
    else:
      request.get_data()
      if not request.data:
        return wholeConfig()
      else:
            authClient = getClient(request.headers['
    Authorization'])
            if request.headers['Content-Type'] == '
    application/json':
                json_data = json.loads(request.get_data().
    decode('utf-8'))
                CONFIG[authClient] = json_data
                with open("assets.json", "w+") as f:
                    json.dump(CONFIG, f)
                client.publish(f"JSNetworkManagement/{
    authClient}", "update", qos = 1)
                return "OK"

@app.route('/api/v1/backToBase', methods=['GET'])
@auth.login_required
def backToBase():
    authClient = getClient(request.headers['Authorization
     '])
    client.publish(f"JSNetworkManagement/{authClient}", "
     base", qos = 1)
    return "Published"

@app.route('/api/v1/templates', methods=['POST'])
@auth.login_required
def getTemplate():
    if request.form.get('template'):
        template = request.form.get('template')
    else:
        return "Error: Template not especified."

    if template in TEMPLATES:
        return TEMPLATES[template]
    else:
        return "Error: Template does not exist"

@app.route('/api/v1/addUser', methods=['POST'])
@auth.login_required
def addUser():
    if getClient(request.headers['Authorization']) == '
     admin' and (request.form.get('user') and request.form.
     get('password')):
        newUsr = request.form.get('user')
        newPwd = bcrypt.hashpw(request.form.get('password')
    .encode('utf-8'), bcrypt.gensalt()).decode('utf-8')
        if newUsr not in USER_DATA:
            USER_DATA[newUsr] = newPwd
            with open("user_db.json", "w+") as f:
                json.dump(USER_DATA, f)
            return "User created"
        else:
            return "Action can't be done"
    else:
        return "Action can't be done"

@app.route('/', methods=['GET'])
def default():
    return "Hello"

if __name__ == '__main__':
    app.run(debug=True)
```

APÉNDICE B
CÓDIGO COMPLETO DEL CLIENTE

```
import routeros_api
import json as js
import requests
import time
import paho.mqtt.client as mqtt
from string import Template
import os

##### Here we connect to the RouterOs API and load the
     configuration files present in the router.
time.sleep(20)

ENDPOINT = os.environ['ENDPOINT']
ROUTER_USER = os.environ['ROUTER_USER']
ROUTER_PWD = os.environ['ROUTER_PWD']
CLIENT_USER = os.environ['CLIENT_USER']
CLIENT_PWD = os.environ['CLIENT_PWD']

```

```
con = routeros_api.RouterOsApiPool('172.17.0.1', username=f'{ROUTER_USER}', password=f'{ROUTER_PWD}', plaintext_login=True)
api = con.get_api() ##Connection to the API

filePointer = api.get_resource('/file')
base = filePointer.get(name="base.rsc")[0]['contents']

##### Here we get the router identifier needed for next steps
systemIdentity = api.get_resource('/system/identity').get()[0]['name']

def getTemplate(template):
  return Template((requests.post(f"{ENDPOINT}/api/v1/templates", data = { "template" : f"{template}"}, auth = (f"{CLIENT_USER}", f"{CLIENT_PWD}")).text).replace("'", '"'))


def setUp(myConfig):

  newConfig = base
  if "ipAddresses" in myConfig:
    ## Add an ip to an interface
    template = getTemplate("ipAddresses")
    for parameters in myConfig["ipAddresses"]:
      substitute = template.substitute(address=parameters["inetAddress"], ifname=parameters["inetInterface"])
      newConfig += substitute

  if "bridges" in myConfig:
    ## Add a new bridge interface
    template = getTemplate("interface_bridge")
    port_template = getTemplate("interface_bridge_port")
    for parameters in myConfig["bridges"]:
      substitute = template.substitute(bridgename=parameters["bridgeName"])
      newConfig += substitute

      ## Add ports to the new bridge
      for port_parameters in parameters["ports"]:
        substitute = port_template.substitute(bridgename=port_parameters["bridgeName"], ifname=port_parameters["ifName"])
        newConfig += substitute

  if "wireguard" in myConfig:
    for parameters in myConfig["wireguard"]:
      ## Add address to a physical interface which will be the underlay for Wireguard
      template = getTemplate("ipAddresses")
      substitute = template.substitute(address=parameters["inetAddress"], ifname=parameters["inetInterface"])
      newConfig += substitute

      ## Create Wireguard interface
      template = getTemplate("interface_wireguard")
      substitute = template.substitute(wgport = parameters["wgOwnPort"], wgifname= parameters["wgInterfaceName"], privatekey= parameters["wgOwnPrivateKey"])
      newConfig += substitute

      ## Create Peers
      template = getTemplate("wireguard_peers")
      for peer_parameters in parameters['peers']:
        substitute = template.substitute(endpointadd=peer_parameters["endpointAddress"], endpointport=peer_parameters["endpointPort"], publicKey=peer_parameters["endpointPublicKey"], wgifname=parameters["wgInterfaceName"])
        newConfig += substitute

      ## Add address to Wireguard interface
      template = getTemplate("ipAddresses")
      substitute = template.substitute(address=parameters["wireguardAddress"], ifname=parameters["wgInterfaceName"])
      newConfig += substitute

  if "ospf" in myConfig:
    ##Base OSPF Settings
    template = getTemplate("ospf_base")
    substitute = template.substitute(bridgename = myConfig["ospf"]["bridgeName"], routingid = myConfig["ospf"]["routingId"], routingname = myConfig["ospf"]["routingName"], ipaddress = myConfig["ospf"]["ipAddress"])
    newConfig += substitute
    ## Instance OSPF Setting
    template = getTemplate("ospf_instance")
    for parameters in myConfig["ospf"]["instances"]:
      substitute = template.substitute(ospfinstancename = parameters["ospfInstanceName"], ospfrouterid = parameters["ospfRouterId"])
      newConfig += substitute
    ## Area OSPF Setting
    template = getTemplate("ospf_area")
    for parameters in myConfig["ospf"]["areas"]:
      substitute = template.substitute(ospfinstancename = parameters["ospfInstanceName"], ospfareaname = parameters["ospfAreaName"])
      newConfig += substitute
    ## Interface-template OSPF Setting
    template = getTemplate("ospf_interface_template")
    for parameters in myConfig["ospf"]["interfaceTemplates"]:
      substitute = template.substitute(ospfareaname = parameters["ospfAreaName"], ospfnetwork = parameters["ospfNetwork"])
      if "passive" in parameters:
        substitute += " passive"
      newConfig += substitute

  if "vxlan" in myConfig:
    ## Create new vxlan bridge
    template = getTemplate("interface_bridge")
    substitute = template.substitute(bridgename = myConfig["vxlan"]["bridgeName"])
    newConfig += substitute

    ## Create vxlan interfaces
    template = getTemplate("interface_vxlan")
    for parameters in myConfig["vxlan"]["interfaces"]:
      substitute = template.substitute(localaddress=parameters["localAddress"], vxlanname=parameters["name"], vxlanport=parameters["port"], vni=parameters["vni"])
      newConfig += substitute

    ## Associate vxlan interfaces to vxlan bridge
    template = getTemplate("interface_bridge_port")
    for parameters in myConfig["vxlan"]["bridge_ports"]:
      substitute = template.substitute(bridgename=parameters["bridgeName"], ifname=parameters["ifName"])
      newConfig += substitute

    ## Add vteps
    template = getTemplate("interface_vxlan_vteps")
    for parameters in myConfig["vxlan"]["vteps"]:
      substitute = template.substitute(ifname=parameters["ifName"], remoteIP=parameters["remoteIP"])
      newConfig += substitute

  if "dhcp_client" in myConfig:
    ##Create new dhcp clients
    template = getTemplate("dhcp_client")
    for parameters in myConfig["dhcp_client"]:
      substitute = template.substitute(ifname=parameters["ifName"])
      newConfig += substitute

  if "dhcp_server" in myConfig:
    ##Create an ip pool
    template = getTemplate("ip_pool")
    for parameters in myConfig["dhcp_server"]:
      substitute = template.substitute(poolname=parameters["poolName"],  iprange=parameters["ipRange"])
      newConfig += substitute

    ##Create DHCP Server
    template = getTemplate("dhcp_server")
    for parameters in myConfig["dhcp_server"]:
      substitute = template.substitute(poolname=parameters["poolName"], ifname=parameters["ifName"], servername=parameters["serverName"])
      newConfig += substitute

    ##Asign IP address to the main interface
    template = getTemplate("ipAddresses")
    for parameters in myConfig["dhcp_server"]:
      substitute = template.substitute(address=parameters["
```

```
    address"], ifname=parameters["ifName"])
      newConfig += substitute

    ##Create DHCP Server Network
    template = getTemplate("dhcp_server_network")
    for parameters in myConfig["dhcp_server"]:
      substitute = template.substitute(serveraddress=
     parameters["serverAddress"], servergateway=parameters["
     address"])
      newConfig += substitute
  return newConfig

def applyConfig(config):

  idConfig = filePointer.get(name="newConfig.rsc")[0]['id']
  filePointer.set(id=idConfig, contents=config)
  api.get_binary_resource('/system').call('reset-
     configuration', { 'run-after-reset': "newConfig.rsc", "
     skip-backup":"yes" })

def on_message(client, userdata, message):
  print(f"Received message: {message.payload.decode('utf
     -8')}")
  print(f"QoS : {message.qos}")

  msg = message.payload.decode('utf-8')

  if "update" in msg:
    myConfig = js.loads(requests.post(f"{ENDPOINT}/api/v1/
     config", data = { "identifier" : f"{systemIdentity}"},
     auth = (f"{CLIENT_USER}", f"{CLIENT_PWD}")).text)
    newRSC = setUp(myConfig)
    newConfigFileContent = filePointer.get(name="newConfig.
     rsc")[0]['contents']
    if newRSC != newConfigFileContent:
      applyConfig(newRSC)

  elif "base" in msg:
    aux_file = filePointer.get(name="newConfig.rsc")[0]['id
     ']
    filePointer.set(id=aux_file, contents="")
    api.get_binary_resource('/system').call('reset-
     configuration', { 'run-after-reset': "base.rsc", "skip-
     backup":"yes" })

def on_connect(client, userdata, flags, rc):
  client.subscribe(f"JSNetworkManagement/{CLIENT_USER}", 1)

##### Here we connect to our MQTT Broker and specify the
     functionality to realize when a message comes

mqttBroker = ENDPOINT.split("//")[1]
client = mqtt.Client(clean_session=True)
client.on_connect = on_connect
client.on_message = on_message
client.connect(mqttBroker, keepalive=0)
client.loop_start()

myConfig = js.loads(requests.post(f"{ENDPOINT}/api/v1/
     config", data = { "identifier" : f"{systemIdentity}"},
     auth = (f"{CLIENT_USER}", f"{CLIENT_PWD}")).text)
newRSC = setUp(myConfig)
newConfigFileContent = filePointer.get(name="newConfig.rsc
     ")[0]['contents']

if newConfigFileContent != "":
  if newRSC != newConfigFileContent:
    applyConfig(newRSC)

while True:
  time.sleep(5)
```

APÉNDICE C
FICHERO BASE DEL CLIENTE

```
:if ([:len [/ip/dhcp-client find interface="ether1"]] = 0)
     do={/ip/dhcp-client add interface=ether1}
/system identity set name=router-vxlan
:if ([:len [/file find name="newConfig.rsc"]] = 0) do={/
     tool/fetch url="http://18.100.158.148" dst-path="
     newConfig.rsc"}
/interface veth
add address=172.17.0.2/24 gateway=172.17.0.1 name=veth1
/interface bridge
add name=dockers
/ip address
add address=172.17.0.1/24 interface=dockers network
     =172.17.0.0
/interface bridge port
add bridge=dockers interface=veth1
/ip firewall nat
add action=masquerade chain=srcnat src-address
     =172.17.0.0/24
/container config
set registry-url=https://registry-1.docker.io tmpdir=disk1/
     pull
/container envs
add name=pyapp key=ENDPOINT value="http://18.100.158.148"
add name=pyapp key=ROUTER_USER value="admin"
add name=pyapp key=ROUTER_PWD value=""
add name=pyapp key=CLIENT_USER value="admin"
add name=pyapp key=CLIENT_PWD value="superAdmin"
/container
add interface=veth1 logging=yes workdir=/app remote-image=
     saek13/pythonappv4:alpine-amd64 start-on-boot=yes
     envlist=pyapp
:delay 15000ms
start [/container/find interface=veth1]
```