

Ingeniería de tráfico para IoT en redes SDN: implementación con controlador RYU y Mininet

Tyrone Miguel Novillo Bravo

Facultad de Ingeniería

Departamento de Telecomunicaciones

Universidad de Cuenca

tyrone.novillo@ucuenca.edu.ec

Pablo Andrés Bermeo García

Facultad de Ingeniería

Departamento de Telecomunicaciones

Universidad de Cuenca

pablo.bermeo@ucuenca.edu.ec

Resumen

This work presents the design and emulation of a Software-Defined Networking (SDN) infrastructure in Mininet, orchestrated by a custom Ryu controller, with integrated traffic engineering for Internet of Things (IoT) applications. The proposed topology comprises six Open vSwitch instances, interconnected to support redundant paths and managed centrally via OpenFlow 1.3. Two IoT services are deployed: an environmental temperature manager using MQTT over TCP, and a proximity radar using UDP sockets. Traffic Engineering (TE) is achieved through OpenFlow group tables—SELECT for weighted load balancing and FAST-FAILOVER for sub-second rerouting—ensuring prioritized delivery of IoT flows over competing IPTV and bulk UDP traffic. A proxy-ARP module centralizes ARP handling, while dynamic monitoring of flow statistics enables automatic adjustment of group weights when link utilization exceeds predefined thresholds. Performance evaluation under both normal and congested conditions demonstrates that the TE mechanisms maintain low jitter (near zero) and stable inter-arrival intervals for critical IoT traffic, recovering nominal behavior within milliseconds upon failover events. These results confirm the viability of SDN-based traffic engineering to guarantee reliable, low-latency communication for heterogeneous IoT services in campus and smart-city scenarios.

Index Terms

Software-Defined Networking, SDN, OpenFlow, Mininet, RYU, MQTT, UDP

I. INTRODUCCIÓN

Las redes definidas por software (SDN, por sus siglas en inglés) representan un cambio fundamental en el diseño, la operación y la gestión de redes. A diferencia de las redes tradicionales, donde los dispositivos de red como switches y routers toman decisiones de forma autónoma, SDN propone una arquitectura centralizada en la que la lógica de control se separa del plano de datos. Esta separación permite que las decisiones sobre cómo manejar el tráfico se tomen desde un controlador centralizado que tiene una visión global de la red, facilitando una gestión más dinámica, flexible y programable.

Una de las principales ventajas de las redes SDN es su capacidad de adaptación a necesidades cambiantes en tiempo real. Dado que las reglas de reenvío se definen por software desde el controlador, es posible modificar el comportamiento de la red sin tener que reconfigurar manualmente cada dispositivo. Esto permite una respuesta más rápida ante fallos, cambios en la demanda del tráfico o necesidades específicas de aplicaciones. Además, al poder programar la red desde un punto central, se reduce el riesgo de configuraciones inconsistentes entre dispositivos.

Otra ventaja clave es la visibilidad global que el controlador tiene sobre toda la infraestructura de red. Esta perspectiva unificada facilita la recolección de estadísticas de tráfico, el análisis de rendimiento, y la detección de anomalías o intrusiones. Además, permite implementar políticas de calidad de servicio (QoS), seguridad o segmentación de red de manera más eficaz y coherente que en arquitecturas tradicionales, donde cada equipo actúa de manera más aislada.

SDN también mejora la innovación y la evolución tecnológica de las redes. Al abstraer el plano de control en software, los operadores pueden experimentar con nuevas políticas, algoritmos de enrutamiento o mecanismos de seguridad sin necesidad de esperar a que los fabricantes de hardware implementen estas funcionalidades. Esta apertura acelera el desarrollo de soluciones personalizadas y reduce los costos operativos y de infraestructura en el mediano y largo plazo.

En la actualidad, las comunicaciones IoT (Internet de las Cosas) se han vuelto esenciales debido al crecimiento exponencial de dispositivos conectados que recopilan, transmiten y procesan datos en tiempo real. Desde sensores industriales hasta

electrodomésticos inteligentes y dispositivos médicos, el ecosistema IoT exige redes capaces de manejar grandes volúmenes de tráfico, con requisitos estrictos de latencia, confiabilidad y escalabilidad. De cara al futuro, se espera que el número de dispositivos IoT supere ampliamente la cantidad de usuarios humanos conectados, lo que plantea desafíos técnicos cada vez mayores. Por ello, la integración de arquitecturas flexibles y adaptativas como SDN se vuelve crucial para garantizar que estas redes puedan responder de forma eficiente a la complejidad y dinamismo del entorno IoT, habilitando servicios más inteligentes, automatizados y seguros en áreas clave como las ciudades inteligentes, la salud, la agricultura y la industria 4.0.

La integración de redes SDN con comunicaciones IoT ofrece una ventaja significativa en términos de gestión dinámica y eficiente del tráfico. Gracias a su arquitectura centralizada y programable, SDN permite establecer políticas de enrutamiento diferenciadas según el tipo de tráfico, priorizando por ejemplo el envío de datos críticos de sensores médicos o de procesos industriales en tiempo real, frente a tráfico menos sensible como el de cámaras o actualizaciones. Esta capacidad de control fino facilita una mejor utilización del ancho de banda y una reducción en la latencia, asegurando que los datos más urgentes lleguen de forma oportuna, incluso en redes altamente congestionadas.

Esta sinergia resulta especialmente valiosa en entornos como fábricas inteligentes, hospitales conectados, o infraestructuras de transporte, donde el correcto funcionamiento depende de la transmisión oportuna de datos IoT. En estos casos, SDN puede implementar rutas específicas, priorizar flujos críticos y adaptar el comportamiento de la red en función de eventos o estados detectados, todo de forma automática. Además, en contextos como ciudades inteligentes o redes eléctricas inteligentes, donde conviven múltiples tipos de tráfico IoT, SDN se convierte en una herramienta clave para orquestar comunicaciones seguras, eficientes y confiables.

En este contexto, se ha planteado la simulación de una red SDN que ofrezca prioridad al tráfico IoT. Específicamente, la simulación de la red SDN se realizará sobre Mininet y el tráfico IoT proviene de un sensor de temperatura que publica valores de temperatura en un servidor Mosquitto: se emplea el protocolo MQTT. El siguiente informe explica la implementación y configuración de la red, y se estructura en capítulos de la siguiente manera: el capítulo II presenta la implementación del *publisher* MQTT y su transmisión de la temperatura...

II. MARCO TEÓRICO

II-A. Protocolo MQTT

El protocolo MQTT (Message Queuing Telemetry Transport) es un estándar de mensajería ligero, ampliamente utilizado en el entorno del Internet de las Cosas (IoT). Fue desarrollado por IBM en 1999 con el objetivo de facilitar la comunicación en dispositivos con recursos limitados y conexiones poco confiables. Basado en TCP/IP, MQTT sigue un modelo de comunicación publicador/suscriptor.

Su funcionamiento gira en torno a un broker (intermediario), que recibe los mensajes de los publicadores y los reenvía a los suscriptores que estén interesados en el mismo tema (o *topic*). Esto permite una arquitectura desacoplada y eficiente, ideal para redes con bajo consumo de energía y dispositivos con capacidad de procesamiento limitada.

MQTT es conocido por su simplicidad, eficiencia energética, y rapidez en la transferencia de datos pequeños. Además, ofrece distintos niveles de Calidad de Servicio (QoS) que definen cómo se asegura la entrega de los mensajes:

- QoS 0: el mensaje se envía una sola vez sin confirmar su recepción.
- QoS 1: el mensaje se reenvía hasta recibir una confirmación.
- QoS 2: se garantiza que el mensaje se entregue una única vez, utilizando un mecanismo de confirmación más complejo.

La arquitectura de MQTT distingue entre clientes (publicadores o suscriptores) y el broker central, el cual gestiona las conexiones, filtra los mensajes y garantiza la entrega según las reglas configuradas. Existen brokers públicos (para pruebas) y privados (para producción), y pueden funcionar sobre TCP o TLS para garantizar la seguridad.

Si bien MQTT ha ganado popularidad en entornos industriales, domésticos y médicos, también presenta desafíos, como:

- Dependencia del protocolo TCP.
- Problemas de escalabilidad debido al uso de brokers centralizados.
- Riesgo de fallos únicos en el broker.
- Complejidad de implementación en comparación con HTTP.
- Requisitos de seguridad más estrictos para entornos sensibles.

En su versión más reciente (MQTT v5), se han incorporado mecanismos de control de flujo, como cuotas de envío y atributos de recepción máxima, mejorando el desempeño en escenarios exigentes y de múltiples proveedores [4].

II-B. El protocolo Openflow

El protocolo OpenFlow define el intercambio de mensajes entre el controlador OpenFlow y los dispositivos que implementan este protocolo. Normalmente, la comunicación entre ambos se establece mediante un canal seguro utilizando SSL o TLS.

Este protocolo permite al controlador agregar, modificar o eliminar entradas en las tablas de flujo de los dispositivos. Los mensajes definidos por OpenFlow se agrupan en tres tipos principales:

- *Mensajes de controlador a dispositivo*: son iniciados por el controlador y, en algunos casos, requieren una respuesta por parte del dispositivo.
- *Mensajes asíncronos*: son enviados por el dispositivo al controlador sin que haya una solicitud previa, generalmente para informar sobre eventos relevantes.
- *Mensajes simétricos*: pueden ser enviados por el controlador o por el dispositivo sin que exista una solicitud explícita. Aunque son simples, resultan útiles. Por ejemplo, los mensajes *hello* se intercambian cuando se establece la conexión entre el controlador y el switch. Los mensajes *echo request* y *echo reply* permiten verificar el estado del canal de comunicación y medir la latencia o el ancho de banda. También existen los mensajes *experimenter*, utilizados para probar o desarrollar nuevas funcionalidades en versiones futuras del protocolo.

Además, OpenFlow proporciona al controlador SDN tres tipos fundamentales de información para la gestión de la red:

- *Mensajes basados en eventos*: se generan cuando ocurren cambios en conexiones o puertos, y se envían al controlador.
- *Estadísticas de flujo*: son recolectadas por los switches en función del tráfico que manejan. Esta información permite al controlador supervisar el comportamiento de la red, reconfigurar rutas y ajustar parámetros para satisfacer requerimientos de calidad de servicio.
- *Paquetes encapsulados*: se envían al controlador cuando un switch necesita una decisión sobre cómo manejar un nuevo flujo, o porque así lo indica la política definida en la tabla de flujo.

II-C. Tablas de grupo en OpenFlow

En OpenFlow, las tablas de grupo permiten definir conjuntos de acciones complejas que no pueden expresarse fácilmente con reglas individuales de flujo. Estas tablas facilitan la implementación de mecanismos avanzados como balanceo de carga, replicación de paquetes, abstracción de acciones comunes y conmutación por fallos. Cada tabla de grupo contiene uno o más *buckets* (cubetas), y el comportamiento exacto depende del tipo de grupo que se utilice. A continuación, se describen los principales tipos de grupos definidos en OpenFlow:

II-C1. Grupo ALL: El grupo ALL funciona duplicando cada paquete recibido y enviando una copia independiente a cada *bucket*. De esta forma, el grupo ALL ejecuta acciones distintas sobre copias separadas del mismo paquete, según las acciones definidas en cada *bucket*. Cada *bucket* puede contener un conjunto diferente de acciones, lo que permite realizar operaciones diversas con cada copia del paquete. Este tipo de grupo se utiliza principalmente para reenvío multicast o broadcast.

II-C2. Grupo SELECT: El grupo SELECT está diseñado específicamente para el balanceo de carga. Cada *bucket* dentro del grupo tiene un peso asignado, y cada paquete que entra al grupo se redirige a uno solo de los *buckets*. El método para seleccionar el *bucket* no está estandarizado y depende del switch; sin embargo, una opción común es el round robin ponderado. Cada *bucket* puede tener cualquier conjunto de acciones compatibles con OpenFlow, al igual que en el grupo ALL, y no es necesario que los *buckets* sean uniformes entre sí.

II-C3. Grupo INDIRECT: Aunque se denomina grupo, el grupo INDIRECT es algo particular porque solo contiene un *bucket*. Ese único *bucket* es el que se encarga de todas las acciones para los paquetes que llegan al grupo. Su propósito es centralizar una secuencia de acciones comunes que son utilizadas por múltiples flujos. Por ejemplo, si los flujos A, B y C coinciden con diferentes encabezados de paquetes pero comparten las mismas acciones, en lugar de repetir esas acciones en cada flujo, todos los flujos pueden apuntar a este grupo INDIRECT. Esto simplifica la configuración de OpenFlow y reduce el uso de memoria.

II-C4. Grupo FAST-FAILOVER: El grupo FAST-FAILOVER está diseñado para detectar y reaccionar ante fallos de puertos. Al igual que los grupos SELECT y ALL, contiene una lista de *buckets*, pero cada *bucket* incluye un puerto o grupo de monitoreo (*watch port* o *watch group*), que determina si el *bucket* está activo o no. Solo se puede utilizar un *bucket* a la vez, y este solo se reemplaza cuando su puerto o grupo de monitoreo cambia de estado (por ejemplo, si el puerto falla).

Cuando un *bucket* falla, no se garantiza un tiempo fijo para el cambio, ya que depende del tiempo que tome detectar y activar el siguiente *bucket* válido. Sin embargo, el uso de FAST-FAILOVER permite que la propia capa de datos (*data plane*) se encargue del monitoreo y recuperación ante fallos, lo cual es más rápido que depender de la capa de control para actualizar los flujos manualmente.

Las ilustraciones de cada grupo se halla en la Figura 1.

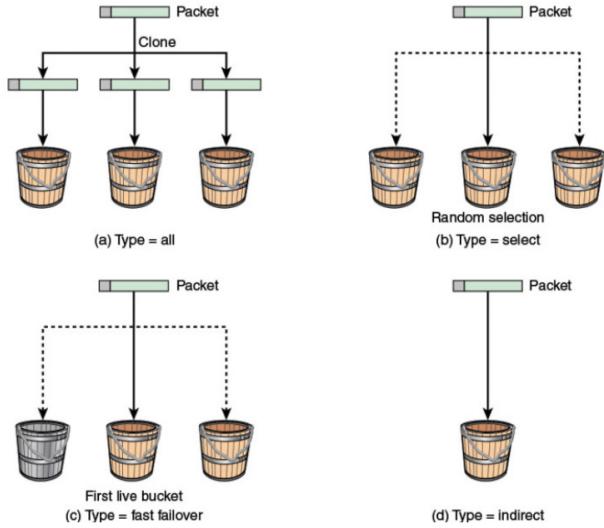


Figura 1: Ilustración de las tablas de grupo. Tomado de [5].

III. METODOLOGÍA

III-A. Descripción general de la solución

La red SDN se implementará en Mininet y los dispositivos insertados en el simulador serán manejados a partir del controlador RYU. La red que se plantea está constituida por seis *openvswitch* que se comunican al controlador RYU. Los *openvswitch* estarán encargados del transporte de paquetes a través de la red en función de las instrucciones generadas por el controlador a través de tablas de flujo y tablas de grupo.

En la Figura 2 se presenta la topología a emplear para el proyecto. Note que se emplean rutas redundantes a fin de generar alternativas para protocolos que requieran una mayor prioridad. Estos protocolos, en un inicio serán MQTT y *sockets UDP*. Tales protocolos proveerán de las aplicaciones IoT que se han diseñado: radar de distancias y gestor de temperatura ambiental. Además la red se inundará a partir de paquetes UDP que provienen del tráfico de IPTV. No obstante también se empleará iperf a fin de generar un tráfico distinto sobre la red.

En el apartado de ingeniería de tráfico se han determinado ciertos criterios a seguir para brindar mayores beneficios a determinados actores en la red. En un inicio el tráfico relacionado a las aplicaciones IoT se transmitirá a por caminos directos o con menor cantidad de saltos. Tales rutas son fácilmente identificables en la topología de la Figura 2. En segundo lugar, el tráfico IPTV tendrá también prioridad en la red a fin de evitar una pérdida masiva de paquetes y una reproducción insatisfactoria. En el caso de que iperf entre en la red e inunde masivamente los enlaces, tal flujo será reenviado a través de rutas largas en comparación a las empleadas en MQTT o UDP en la aplicación de radar. Además se añaden rutas alternas para IoT en caso de la caída de un enlace.

III-B. Entorno de simulación

Se emplea el sistema operativo Kali Linux y dentro del mismo se ha instalado el simulador Mininet. El controlador empleado se denomina Ryu. No se emplearon los módulos que existen en RYU a fin de evitar incompatibilidades o instrucciones no

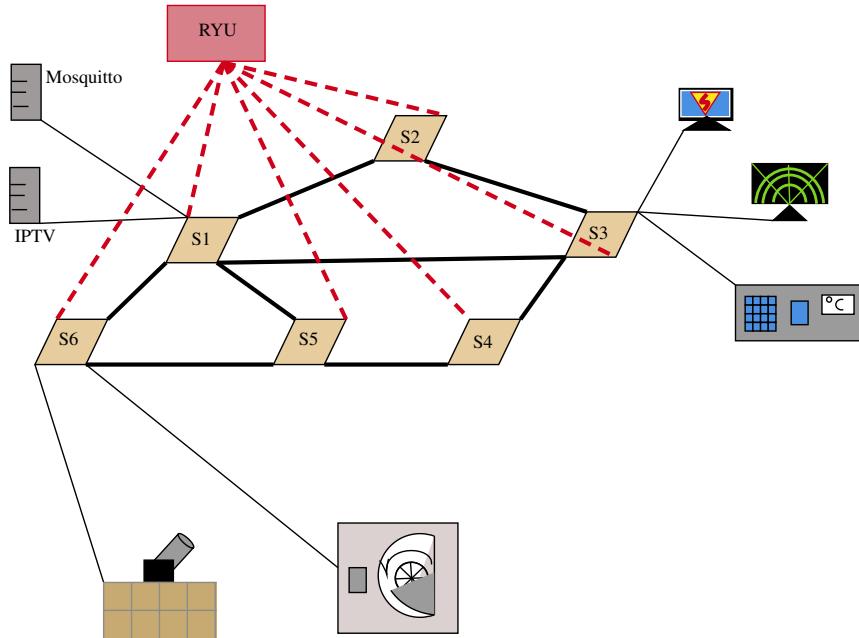


Figura 2: Topología y servicios que se plantea emular sobre Mininet. Note que se han generado caminos redundantes a fin de ofrecer rutas con menos saltos a los protocolos prioritarios.

deseadas en el comportamiento de la red. En cambio se programó un controlador desde la base y se tomaron ciertas instrucciones de los módulos en RYU. El controlador RYU se programa a partir de Python.

En la topología simulada dentro de la Mininet se han ubicado seis *openvswitch* que son controlados mediante RYU. Algunos de los openvswitch tienen conectados un número determinado de *hosts*. Los hosts también son emulados a partir de Mininet. En total, se emularon seis hosts. La distribución y topología de los componentes de la red se presenta en la Figura 3. Note que se ha definido al host 5 con ip 192.168.10.169 como el servidor Mosquitto que será empleado por una de las aplicaciones IoT. En cambio los siguientes Hosts se empleará como clientes de iptv o generadores de tráfico a partir de iperf.

Las direcciones empleadas para los hosts corresponde a la nomenclatura general 192.168.10.X.

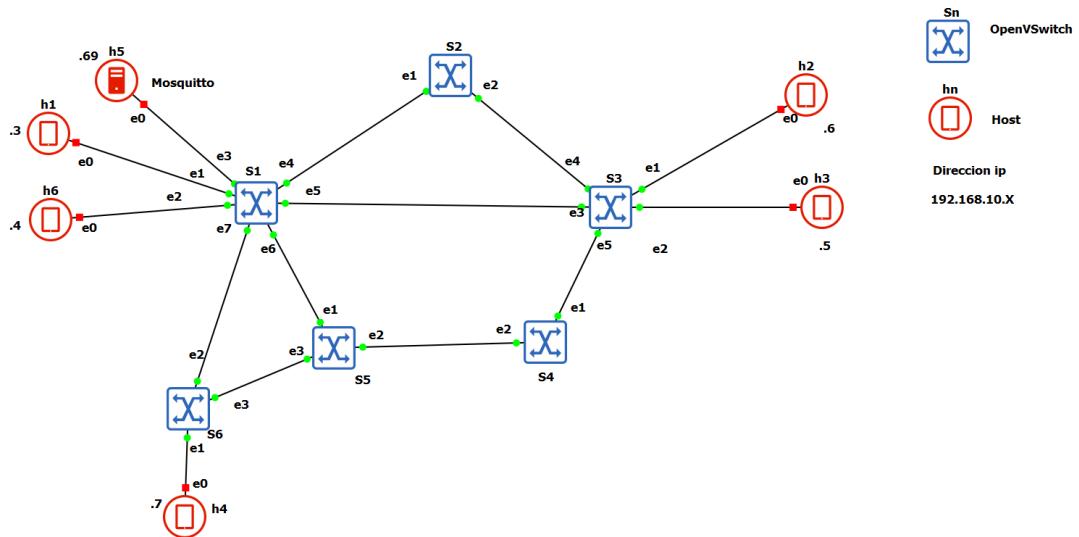


Figura 3: Topología simulada en Mininet.

III-C. Funcionalidades del controlador RYU

El controlador RYU gestiona el encaminamiento y la resiliencia de múltiples flujos de tráfico definidos en la red. Se configura un enrutamiento estático para asegurar que los dispositivos finales y los servicios centrales (como el servidor Mosquitto y el visualizador de radar) mantengan conectividad predecible, al mismo tiempo que se implementan mecanismos de recuperación rápida y balanceo de carga en escenarios de saturación.

1. Se establecerá un enrutamiento estático MQTT entre el ESP32 que publica temperatura y el servidor Mosquitto a través del camino $s_6-s_5-s_1$. Para este flujo se implementan tablas de grupo con capacidad de *fast rerouting*, de modo que ante la caída de un enlace el tráfico se desvíe instantáneamente entre s_6 y s_1 sin pérdida perceptible de mensajes.
2. El controlador configura igualmente un enrutamiento estático MQTT entre la Raspberry Pi y el servidor Mosquitto por el camino $s_1-s_2-s_3$, garantizando un trayecto dedicado para las suscripciones y publicaciones de la Raspberry Pi.
3. Para el monitoreo de radar se definirá un canal de datos UDP en el puerto 2000 que discurre entre el dispositivo Radar y el visualizador de Radar usando el trayecto $s_6-s_1-s_3$. Este flujo UDP también se gestiona mediante reglas estáticas en la tabla de reenvío para asegurar baja latencia.
4. El tráfico IPTV (UDP en puerto 5004) entre los equipos en VLAN 10 y VLAN 30 se enruta por $s_1-s_2-s_3$. En este caso, las entradas en la tabla de grupo se definen con modo *ALL*, de manera que cada paquete multicast se replica automáticamente en todas las salidas correspondientes a ambas VLANs.
5. Para el resto del tráfico entre VLAN 10 y VLAN 30 se disponen dos caminos alternativos: $s_1-s_2-s_3$ y $s_1-s_5-s_4-s_3$. En este escenario se usa un grupo de tipo *SELECT* que dirige el 90 % del tráfico por $s_1-s_5-s_4-s_3$ y el 10 % por $s_1-s_2-s_3$, optimizando la distribución de la carga en condiciones normales.

Cuando el tráfico normal entre VLAN 10 y VLAN 30 alcanza un umbral de saturación, el controlador adapta dinámicamente el balanceo de carga: el grupo de tipo *SELECT* se reconfigura para enviar 50 % de los paquetes por el enlace $s_1-s_2-s_3$ y el 50 % por el enlace $s_1-s_5-s_4-s_3$, de modo que se aprovechen ambos caminos al máximo. Además, se define un desvío específico para el tráfico MQTT de la Raspberry Pi, que en condiciones de alta carga se redirige por el camino directo s_1-s_3 , evitando interferencias con el resto del tráfico entre VLANs. El tráfico IPTV se mantendrá igual bajo esta condición.

III-D. Aplicación IoT: Gestor de temperatura ambiental

Esta aplicación se define como un transmisor que permite la comunicación de un nivel de referencia de temperatura hacia un receptor. El receptor se encuentra conectado a un sensor de temperatura, un motor que simula el ventilador y un servomotor que simula una ventanilla dentro del ambiente. La comunicación en esta aplicación no es unidireccional: el receptor envía su valor de temperatura a un servidor Mosquitto a través de MQTT. Luego el transmisor se suscribe a ese servidor Mosquitto y extrae el valor de temperatura. Note, en tanto, que se genera un flujo bidireccional de información.

Además considere a partir de tal descripción que se empleará MQTT y el broker Mosquitto.

En cuanto a hardware, el transmisor y receptor difieren. En primer lugar, el transmisor está constituido por los siguientes materiales:

- Rasberry PI 400
- Teclado matricial 4x4
- Lector RFID-RC522
- Pantalla Oled 128x64 I2c Display Lcd

En este sentido el planteamiento general en el transmisor es como sigue: Un usuario debe aplastar el ítem D del teclado para habilitar el modo de cambio de referencia; una vez que se haya en ese modo se solicitará una tarjeta de acceso que deberá acercarse al RFID-RC522; una vez se ha acercado el identificador una pantalla solicitará al usuario el ingreso de una temperatura de referencia; dado el ingreso de la referencia, el valor se envía a Mosquitto con el tópico *temp/ref*.

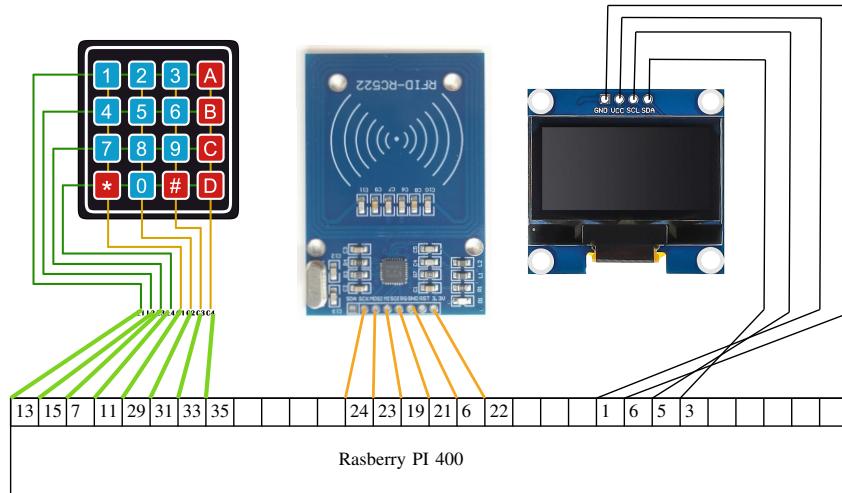


Figura 4: Esquema de conexiones para el transmisor de referencias del gestor de temperatura.

El hardware del receptor está constituido por

- Servomotor TZT MG90S de 180 grados
- Sensor de temperatura 18B20D
- Motor RF 3- 12V
- ESP32
- Fuente de 5V

En este contexto el planteamiento general del receptor es el siguiente: el motor gire continuamente para simular el empleo de un ventilador en el ambiente; al mismo tiempo se envía el valor de temperatura al servidor Mosquitto a través de la ESP32; si el sistema recibe una referencia que está por debajo de la temperatura actual, el servomotor gira 180 grados a fin de simular la apertura de una ventanilla para ventilación.

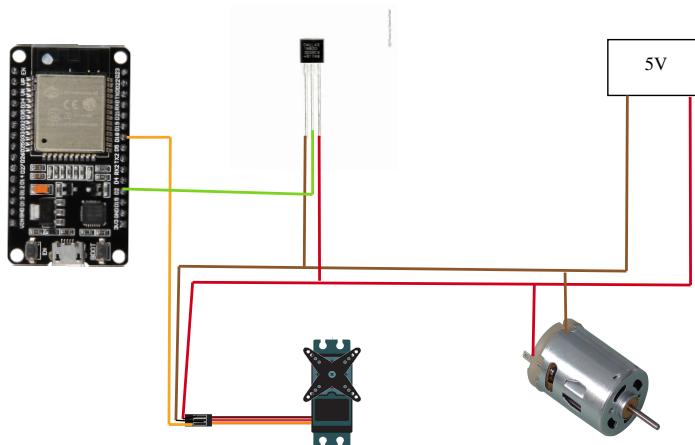


Figura 5: Esquema de conexiones que se empleará para el receptor en la ESP32 en la aplicación de gestión de temperatura.

III-E. Aplicación IoT: Radar con interfaz gráfica

La aplicación de IoT relacionada al radar par del siguiente planteamiento: Un sensor de distancia giratorio tiene un rango de visión de 0 a 180 grados. A medida que el sensor gira envía ángulos y distancia a un receptor en otro lugar de la red. El envío de los valores de ángulo distancia se realizará a través de *sockets UDP*. El receptor genera una gráfica a partir de los datos enviados por el sensor de cercanía.

En la aplicación de radar se plantea la emulación de un detector de distancia que permita realizar un desplazamiento circular de 180 grados. Este desplazamiento se realiza mientras que un detector de cercanía mide las distancias en cada grado de

rotación. De esta manera es posible realizar un *paneo* exacto de los objetos que se hallan alrededor.

En este sentido, el transmisor de distancia se compone de los siguientes materiales:

- Servomotor Digital Servo Hiwonder 20 Kg
- Sensor de proximidad HCSR04
- Resistencias de $1K\Omega$ y $2K\Omega$.
- Fuente de 5V

Las conexiones del transmisor se presentan en la Figura 6.

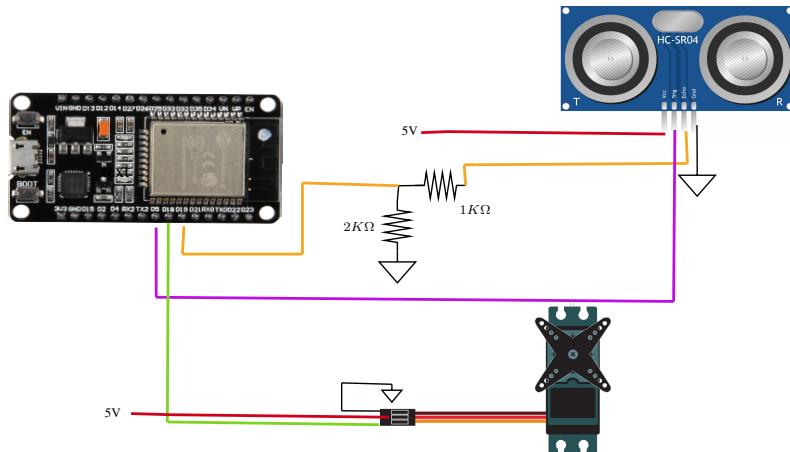


Figura 6: Esquema de conexiones que se usará para el transmisor de distancias y ángulos del Radar con interfaz gráfica.

En cuanto al receptor, únicamente se resalta que se constituye por una codificación en java en el programa Processing [2]. El programa Processing se encarga de recibir la distancia y el ángulo enviado por el receptor y genera una gráfica que se actualiza a cada momento.

IV. DESARROLLO E IMPLEMENTACIÓN

IV-A. Entorno de simulación

IV-A1. Conexión entre red de Mininet con dispositivos reales: Para montar el entorno de simulación se transformó la interfaz Wi-Fi de la máquina Kali Linux en un punto de acceso (AP) al que podían conectarse los hosts de Mininet. Para ello se utilizaron los servicios hostapd y dnsmasq, y se crearon puentes y enlaces virtuales (veth) que interconectan el AP con el resto de la topología. A continuación se detallan los pasos y configuraciones empleadas:

Se instaló y habilitó dnsmasq como servidor DHCP y DNS ligero. En el fichero de configuración se definió la interfaz de puente br0 y el rango de direcciones:

```
interface=br0
bind-interfaces
dhcp-range=192.168.10.10,192.168.10.200,24h
```

Listing 1: Configuración de dnsmasq

Seguidamente, se configuró hostapd para gestionar el punto de acceso inalámbrico sobre wlan0, asignándolo también al puente br0 y estableciendo los parámetros de seguridad WPA2:

```
interface=wlan0
bridge=br0
driver=n180211
ssid=R
hw_mode=g
channel=6
```

```

auth_algs=1
wmm_enabled=0
macaddr_acl=0
ignore_broadcast_ssid=0
wpa=2
wpa_passphrase=claveesp32
wpa_key_mgmt=WPA-PSK
rsn_pairwise=CCMP

```

Listing 2: Configuración de hostapd

A continuación se creó el puente br0 y se levantó la interfaz:

```

sudo ip link add name br0 type bridge
sudo ip link set dev br0 up

```

Listing 3: Creación y activación del puente

A continuación se prepararon los túneles veth que permiten unir la interfaz inalámbrica con el entorno virtual de Open vSwitch:

```

sudo ip link add veth-wlan0 type veth peer name veth-ovs
sudo ip link set veth-wlan0 up
sudo ip link set veth-ovs up
sudo ip link set dev veth-wlan0 master br0
sudo ip link set dev wlan0 master br0

```

Listing 4: Creación de pares veth y asignación a br0

Por último, se configuró la dirección IP del puente y se limpió cualquier asignación previa en wlan0:

```

sudo ip link set dev wlan0 up
sudo ip addr add 192.168.10.1/24 dev br0
sudo ip addr flush dev wlan0

```

Listing 5: Asignación de IP al puente y limpieza de wlan0

Con esto, la tarjeta Wi-Fi opera como un punto de acceso en la red 192.168.10.0/24, distribuyendo direcciones DHCP y conectándose al resto de la simulación de Mininet mediante el puente br0 y el enlace virtual veth-ovs.

Además de la interfaz inalámbrica virtual, se habilitó un segundo punto de acceso mediante un router físico marca TP-Link configurado en modo AP. Este dispositivo se conecta por cable Ethernet al puerto eth0 de la máquina Kali, de modo que todos los paquetes entrantes y salientes del router se integran en la topología de Mininet a través de un puente Ethernet virtual (br1). Para lograr esta interconexión se crearon pares de interfaces veth que vinculan eth0 con Open vSwitch, y se configuró el puente br1 con la dirección IP correspondiente.

```

sudo ip link add veth-eth0 type veth peer name veth-ovs1
sudo ip link set veth-eth0 up
sudo ip link set veth-ovs1 up
sudo ip link set dev veth-eth0 master br1
sudo ip link set dev eth0 master br1

```

Listing 6: Creación de pares veth para eth0

A continuación se creó y levantó el puente br1:

```

sudo ip link add name br1 type bridge
sudo ip link set dev br1 up

```

Listing 7: Creación y activación del puente br1

Finalmente, se habilitó la interfaz física y se asignó la IP al puente, asegurando que el router real opere en la subred 192.168.10.0/24 y dejando limpia la configuración previa de eth0:

```

sudo ip link set dev eth0 up
sudo ip addr add 192.168.10.100/24 dev br1
sudo ip addr flush dev eth0

```

Listing 8: Asignación de IP al puente y limpieza de eth0

Con esta configuración, tanto el AP virtual sobre Wi-Fi como el router físico aportan conectividad a la misma red puenteada, permitiendo que Mininet despliegue y pruebe escenarios que incluyen tráfico inalámbrico y cableado real de forma integrada.

En la interconexión con Mininet se emplean específicamente los enlaces virtuales `veth-ovs` y `veth-ovs1`, que vinculan tanto el punto de acceso inalámbrico como el router físico con la topología virtual. Estos dispositivos de par `veth` permiten replicar el comportamiento de enlaces reales dentro de Mininet, garantizando así que el tráfico procedente de `br0` (Wi-Fi) y `br1` (Ethernet) pueda circular correctamente por la red simulada. A continuación, estos mismos identificadores serán utilizados para definir los enlaces correspondientes en el script de creación de la topología de Mininet.

IV-A2. Topología de la red: La topología simulada en Mininet reproduce la infraestructura descrita en la Figura 3, integrando los enlaces virtuales `veth-ovs` y `veth-ovs1` para conectar los puntos de acceso (Wi-Fi y router físico) a los switches `s6` y `s3`, respectivamente. A continuación se desglosa el script que genera esta topología:

En primer lugar, el script invoca comandos del sistema para volver a crear los pares `veth` asociados a la interfaz `eth0` y asignarlos al puente `br1`, asegurando la persistencia de la conexión del router real:

```
os.system("sudo ip link add veth-eth0 type veth peer name veth-ovs1")
os.system("sudo ip link set veth-eth0 up")
os.system("sudo ip link set veth-ovs1 up")
os.system("sudo ip link set dev veth-eth0 master br1")
os.system("sudo ip link set dev eth0 master br1")
```

Listing 9: Creación y asignación de `veth` para el router

A continuación se instancia la red de Mininet sin topología predefinida, especificando el controlador remoto en la dirección `127.0.0.1:6633`:

```
net = Mininet(topo=None, build=False, ipBase='10.0.0.0/8')
c0 = net.addController('c0',
    controller=RemoteController,
    ip='127.0.0.1',
    port=6633
)
```

Listing 10: Inicialización de Mininet y controlador remoto

El script crea seis switches Open vSwitch en el kernel y seis hosts con direcciones estáticas en la subred `192.168.10.0/24`:

```
s1 = net.addSwitch('s1', cls=OVSKernelSwitch)
h1 = net.addHost('h1', ip='192.168.10.3/24')
h5 = net.addHost('h5', ip='192.168.10.169/24')
```

Listing 11: Definición de switches y hosts

Se establecen enlaces host-switch para cada máquina virtual:

```
net.addLink(h1, s1)
net.addLink(h6, s1)

net.addLink(h4, s6)
```

Listing 12: Conexión de hosts a switches

Y enlaces entre switches para formar la malla de la red:

```
net.addLink(s1, s2)
net.addLink(s1, s3)

net.addLink(s1, s6)
```

Listing 13: Interconexión de switches

Los enlaces virtuales definidos previamente se incorporan a la topología mediante la clase `Intf`, asignando `veth-ovs` a `s6` y `veth-ovs1` a `s3`:

```
Intf('veth-ovs', node=s6)
Intf('veth-ovs1', node=s3)
```

Listing 14: Asignación de interfaces `veth` a switches

Finalmente, la red se construye, se inicia el controlador en todos los switches, y se abre la línea de comandos de Mininet para pruebas interactivas. El script completo se incluye en la sección ??.

IV-B. Controlador

En este informe se emplea un controlador Ryu personalizado que implementa la gestión de flujos para pruebas de iperf UDP en el puerto 5004, IPTV, ARP proxy y MQTT entre los hosts definidos. El controlador extiende la clase RyuApp y utiliza OpenFlow 1.3; configura los flujos iniciales en el evento EventOFPSwitchFeatures y procesa los paquetes entrantes en EventOFPPacketIn.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import (
    CONFIG_DISPATCHER, MAIN_DISPATCHER, DEAD_DISPATCHER,
    set_ev_cls
)
from ryu.ofproto import ofproto_v1_3
from ryu.lib import hub

# Umbral de congestión en bps
UMBRAL_BPS = 1000

class Iperf5004WithARP(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    POLL_INTERVAL = 1 # segundos

    def __init__(self, *args, **kwargs):
        super(Iperf5004WithARP, self).__init__(*args, **kwargs)
        # Estructuras de datos para aprendizaje L2 y ARP
        self.mac_to_port = {}
        self.arp_table = {}
        self.datapaths = {}
        self.prev_flow_bytes = {}
        self.high_congestion = False
        # Lanzar hilo de monitoreo de estadísticas
        self.monitor_thread = hub.spawn(self._monitor)
```

Listing 15: Inicialización del controlador Ryu personalizado

Para centralizar el manejo de ARP, el controlador implementa un “proxy ARP”: todos los paquetes ARP (EtherType 0x0806) se envían al controlador en lugar de ser procesados por los switches. De esta forma, la instancia Ryu construye y despacha las respuestas ARP, evitando el aprendizaje y flooding ARP en el plano de datos. El fragmento siguiente muestra la lógica dentro de _packet_in_handler:

```
#
# Manejador de paquetes entrantes: L2 learning y ARP proxy
#
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    msg = ev.msg
    dp = msg.datapath
    ofp = dp.ofproto
    parser = dp.ofproto_parser
    in_port = msg.match['in_port']

    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocol(ether.ethernet)

    # Solo procesar ARP en este bloque
    if eth.ethertype == ether_types.ETH_TYPE_ARP:
        arp_pkt = pkt.get_protocol(arp.arp)
        src_ip = arp_pkt.src_ip
        dst_ip = arp_pkt.dst_ip
        # Actualizar tabla ARP con MAC y puerto de llegada
        self.arp_table[src_ip] = (arp_pkt.src_mac, in_port)

        if arp_pkt.opcode == arp.ARP_REQUEST and dst_ip in self.arp_table:
            dst_mac, _ = self.arp_table[dst_ip]
            # Construir ARP reply
            arp_reply = packet.Packet()
            arp_reply.add_protocol(ether.ethernet(
                ethertype=eth.ethertype,
                dst=eth.src, src=dst_mac))
```

```

arp_reply.add_protocol(arp.arp(
    opcode=arp.ARP_REPLY,
    src_mac=dst_mac, src_ip=dst_ip,
    dst_mac=arp_pkt.src_mac, dst_ip=src_ip))
arp_reply.serialize()
actions = [parser.OFPActionOutput(in_port)]
out = parser.OFPPacketOut(
    datapath=dp, buffer_id=oif.OFP_NO_BUFFER,
    in_port=oif.OFPP_CONTROLLER,
    actions=actions, data=arp_reply.data)
dp.send_msg(out)
else:
    # Flood si no hay entrada en la tabla ARP
    actions = [parser.OFPActionOutput(ofp.OFPP_FLOOD) ]
    out = parser.OFPPacketOut(
        datapath=dp, buffer_id=msg.buffer_id,
        in_port=in_port, actions=actions,
        data=msg.data)
    dp.send_msg(out)
return

# Procesamiento de otros tipos de paquetes...

```

Listing 16: Implementación del proxy ARP en _packet_in_handler

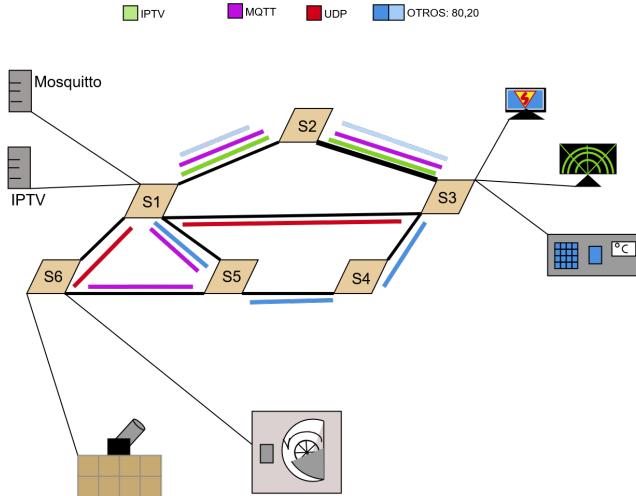


Figura 7: Rutas a generar a través del controlador para el contexto de ausencia de congestión.

IV-B1. Ruta IPTV: La ruta IPTV transporta tráfico UDP en el puerto 5004 entre hosts en VLAN 10 y VLAN 30, utilizando el etiquetado y desencapsulado de VLAN ID para garantizar el aislamiento y la correcta entrega del tráfico multicast. A continuación se detallan las reglas de flujo en los switches S1, S2 y S3 que conforman el camino s1-s2-s3.

Switch S1 (DPID 1)

En S1 se configuran dos grupos de flujos: uno para el tráfico saliente (hosts h1, h6, h5) que etiqueta con VLAN ID 10 y lo envía hacia S2, y otro para el retorno que recibe tráfico con VLAN ID 30 desde S2, elimina la etiqueta y lo entrega a los hosts.

```

if dpid == 1:
    # --- IPTV saliente: tag VLAN 10 y salida por puerto 4 (hacia S2) ---
    for in_p in [1, 2, 3]: # h1=s1-eth1, h6=s1-eth2, h5=s1-eth3
        match_iptv_out = parser.OFPMatch(
            in_port=in_p,
            eth_type=ether_types.ETH_TYPE_IP,
            ip_proto=17,           # UDP
            udp_dst=5004           # puerto IPTV
        )

```

IPTV MQTT UDP OTROS: 50,50

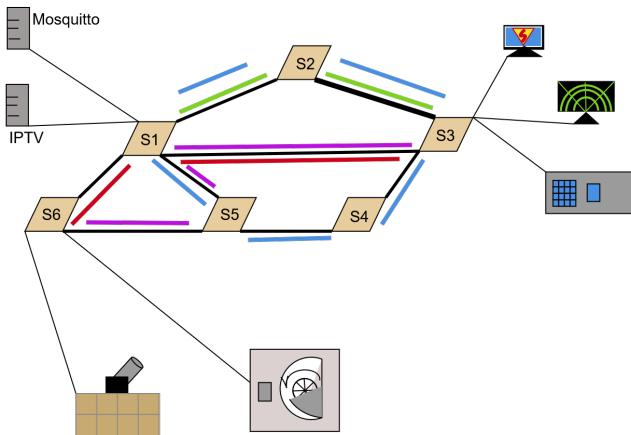


Figura 8: Caminos definidos a través de la red para el contexto de existencia de congestión.

```

actions_iptv_out = [
    parser.OFPActionPushVlan(ether_types.ETH_TYPE_8021Q),
    parser.OFPActionSetField(vlan_vid=(0x1000 | 10)),
    parser.OFPActionOutput(4) # s1-eth4 enlace a S2
]
dp.send_msg(parser.OFPFlowMod(
    datapath=dp,
    priority=30,
    match=match_iptv_out,
    instructions=[parser.OFPIInstructionActions(
        ofp.OFPIT_APPLY_ACTIONS, actions_iptv_out
    )])
)

# --- IPTV retorno: pop VLAN 30, ingreso por puerto 4, salida por 1,2,3 ---
match_iptv_ret = parser.OFPMatch(
    in_port=4, # desde S2
    eth_type=ether_types.ETH_TYPE_IP, # trama 802.1Q
    vlan_vid=(0x1000 | 30), # VLAN ID 30
    ip_proto=17, # UDP
    udp_src=5004 # puerto IPTV de retorno
)
actions_iptv_ret = [
    parser.OFPActionPopVlan(), # quitar etiqueta
    parser.OFPActionOutput(1), # h1 (s1-eth1)
    parser.OFPActionOutput(2), # h6 (s1-eth2)
    parser.OFPActionOutput(3) # h5 (s1-eth3)
]
dp.send_msg(parser.OFPFlowMod(
    datapath=dp,
    priority=30,
    match=match_iptv_ret,
    instructions=[parser.OFPIInstructionActions(
        ofp.OFPIT_APPLY_ACTIONS, actions_iptv_ret
    )])
)

```

Listing 17: Flujos IPTV en S1 (DPID 1)

Switch S2 (DPID 2)

En S2 se hace tránsito de VLAN 10 desde S1 hacia S3 y de VLAN 30 en sentido inverso, manteniendo el ID de VLAN en el camino.

```

if dpid == 2:
    # --- IPTV VLAN 10: ingreso por s2-eth1 egress por s2-eth2 (hacia S3) ---
    match_iptv_s2 = parser.OFPMatch(

```

```

        in_port=1,                                # desde S1
        eth_type=ether_types.ETH_TYPE_IP,
        vlan_vid=(0x1000 | 10),                  # VLAN 10
        ip_proto=17,
        udp_dst=5004
    )
    actions_iptv_s2 = [
        parser.OFPActionOutput(2)                # hacia s2-eth2      S3
    ]
dp.send_msg(parser.OFPPFlowMod(
    datapath=dp,
    priority=30,
    match=match_iptv_s2,
    instructions=[parser.OFFInstructionActions(
        ofp.OFPIT_APPLY_ACTIONS, actions_iptv_s2
    )]
))

# --- IPTV retorno: VLAN 30 ingreso por s2-eth2      egress por s2-eth1 (hacia S1) ---
match_iptv_ret_s2 = parser.OFPMatch(
    in_port=2,                                # desde S3
    eth_type=ether_types.ETH_TYPE_IP,
    vlan_vid=(0x1000 | 30),                  # VLAN 30
    ip_proto=17,
    udp_src=5004
)
actions_iptv_ret_s2 = [
    parser.OFPActionOutput(1)                  # hacia s2-eth1      S1
]
dp.send_msg(parser.OFPPFlowMod(
    datapath=dp,
    priority=30,
    match=match_iptv_ret_s2,
    instructions=[parser.OFFInstructionActions(
        ofp.OFPIT_APPLY_ACTIONS, actions_iptv_ret_s2
    )]
))

```

Listing 18: Flujos IPTV en S2 (DPID 2)

Switch S3 (DPID 3)

En S3 se desencapsula la VLAN 10 para entregar el tráfico a los hosts y al AP, y se etiqueta con VLAN 30 el tráfico de retorno para enviarlo de vuelta a S2.

```

if dpid == 3:
    # --- IPTV VLAN 10 entrante por s3-eth4: pop VLAN y salida a puertos 1,2,6 ---
    match_iptv_in = parser.OFPMatch(
        in_port=4,                                # desde S2
        eth_type=ether_types.ETH_TYPE_IP,
        vlan_vid=(0x1000 | 10),                  # VLAN 10
        ip_proto=17,
        udp_dst=5004
    )
    actions_iptv_in = [
        parser.OFPActionPopVlan(),                 # quitar etiqueta
        parser.OFPActionOutput(1),                  # h2 (s3-eth1)
        parser.OFPActionOutput(2),                  # h3 (s3-eth2)
        parser.OFPActionOutput(6)                  # AP-s3 (s3-eth6)
    ]
    dp.send_msg(parser.OFPPFlowMod(
        datapath=dp,
        priority=30,
        match=match_iptv_in,
        instructions=[parser.OFFInstructionActions(
            ofp.OFPIT_APPLY_ACTIONS, actions_iptv_in
        )]
))

# --- IPTV retorno: tag VLAN 30 en puertos 1,2,6 y salida por s3-eth4 ---
for in_p in [1, 2, 6]:
    match_iptv_bi = parser.OFPMatch(
        in_port=in_p,
        eth_type=ether_types.ETH_TYPE_IP,
        ip_proto=17,                            # UDP

```

```

        udp_src=5004          # tráfico IPTV de retorno
    )
    actions_iptv_bi = [
        parser.OFPActionPushVlan(ether_types.ETH_TYPE_8021Q),
        parser.OFPActionSetField(vlan_vid=(0x1000 | 30)),
        parser.OFPActionOutput(4)           # hacia S2
    ]
    dp.send_msg(parser.OFPPFlowMod(
        datapath=dp,
        priority=30,
        match=match_iptv_bi,
        instructions=[parser.OFFInstructionActions(
            ofp.OFPIT_APPLY_ACTIONS, actions_iptv_bi
        )]
    ))
)

```

Listing 19: Flujos IPTV en S3 (DPID 3)

IV-B2. Comunicación MQTT entre el servidor Mosquitto y la Raspberry: Para habilitar el intercambio de mensajes MQTT entre el servidor Mosquitto (dirección IP 192.168.10.105) y la Raspberry Pi (192.168.10.169), el controlador Ryu instala reglas en los switches que filtran tráfico TCP en el puerto 1883 y lo enrutan de forma bidireccional a lo largo del camino s1-s2-s3.

Switch S1 (DPID 1)

En S1 se definen dos flujos de alta prioridad (100) para el tráfico MQTT:

```

if dpid == 1:
    # --- Forward MQTT: Mosquitto (192.168.10.105)      Raspberry (192.168.10.169) ---
    m_rasp_fwd = parser.OFPMatch(
        in_port=4, eth_type=0x0800, ip_proto=6,
        ipv4_src="192.168.10.105", ipv4_dst="192.168.10.169",
        tcp_dst=1883
    )
    a_rasp_fwd = [parser.OFPActionOutput(3)]
    dp.send_msg(parser.OFPPFlowMod(
        datapath=dp, priority=100, match=m_rasp_fwd,
        instructions=[parser.OFFInstructionActions(
            ofp.OFPIT_APPLY_ACTIONS, a_rasp_fwd
        )]
    ))
    # --- Reverse MQTT: Raspberry      Mosquitto ---
    m_rasp_rev = parser.OFPMatch(
        in_port=3, eth_type=0x0800, ip_proto=6,
        ipv4_src="192.168.10.169", ipv4_dst="192.168.10.105",
        tcp_src=1883
    )
    a_rasp_rev = [parser.OFPActionOutput(4)]
    dp.send_msg(parser.OFPPFlowMod(
        datapath=dp, priority=100, match=m_rasp_rev,
        instructions=[parser.OFFInstructionActions(
            ofp.OFPIT_APPLY_ACTIONS, a_rasp_rev
        )]
    ))
)

```

Listing 20: Flujos MQTT en S1 (DPID 1)

Switch S2 (DPID 2)

En S2 se continúa la ruta, recibiendo el tráfico forward en el puerto 2 y enviándolo al puerto 1, y viceversa para el reverse:

```

if dpid == 2:
    # --- Forward MQTT: ingreso por s2-eth2      salida por s2-eth1 ---
    m2_rasp_fwd = parser.OFPMatch(
        in_port=2, eth_type=0x0800, ip_proto=6,
        ipv4_src="192.168.10.105", ipv4_dst="192.168.10.169",
        tcp_dst=1883
    )
    a2_rasp_fwd = [parser.OFPActionOutput(1)]
    dp.send_msg(parser.OFPPFlowMod(
        datapath=dp, priority=100, match=m2_rasp_fwd,
        instructions=[parser.OFFInstructionActions(
            ofp.OFPIT_APPLY_ACTIONS, a2_rasp_fwd
        )]
    ))
)

```

```

        ))
        # --- Reverse MQTT: ingreso por s2-eth1      salida por s2-eth2 ---
        m2_rasp_rev = parser.OFPMatch(
            in_port=1, eth_type=0x0800, ip_proto=6,
            ipv4_src="192.168.10.169", ipv4_dst="192.168.10.105",
            tcp_src=1883
        )
        a2_rasp_rev = [parser.OFPActionOutput(2)]
        dp.send_msg(parser.OFPPFlowMod(
            datapath=dp, priority=100, match=m2_rasp_rev,
            instructions=[parser.OFPInstructionActions(
                ofp.OFPIT_APPLY_ACTIONS, a2_rasp_rev
            )]
        ))
    )

```

Listing 21: Flujos MQTT en S2 (DPID 2)

Switch S3 (DPID 3)

Finalmente, en S3 se continúa la entrega del tráfico forward desde el AP y hosts conexos (puerto 6) hacia el enlace a S2 (puerto 4), y se enruta el reverse de vuelta al AP por el puerto 6:

```

if dpid == 3:
    # --- Forward MQTT: ingreso por s3-eth6      salida por s3-eth4 ---
    m3_rasp_fwd = parser.OFPMatch(
        in_port=6, eth_type=0x0800, ip_proto=6,
        ipv4_src="192.168.10.105", ipv4_dst="192.168.10.169",
        tcp_dst=1883
    )
    a3_rasp_fwd = [parser.OFPActionOutput(4)]
    dp.send_msg(parser.OFPPFlowMod(
        datapath=dp, priority=100, match=m3_rasp_fwd,
        instructions=[parser.OFPInstructionActions(
            ofp.OFPIT_APPLY_ACTIONS, a3_rasp_fwd
        )]
    ))

    # --- Reverse MQTT: ingreso por s3-eth4      salida por s3-eth6 ---
    m3_rasp_rev = parser.OFPMatch(
        in_port=4, eth_type=0x0800, ip_proto=6,
        ipv4_src="192.168.10.169", ipv4_dst="192.168.10.105",
        tcp_src=1883
    )
    a3_rasp_rev = [parser.OFPActionOutput(6)]
    dp.send_msg(parser.OFPPFlowMod(
        datapath=dp, priority=100, match=m3_rasp_rev,
        instructions=[parser.OFPInstructionActions(
            ofp.OFPIT_APPLY_ACTIONS, a3_rasp_rev
        )]
    ))

```

Listing 22: Flujos MQTT en S3 (DPID 3)

Estas reglas aseguran un camino dedicado y simétrico para la sesión MQTT, manteniendo el flujo dentro de la topología definida.

IV-B3. Comunicación MQTT con Fast Failover: En la ruta entre el ESP32 (IP 192.168.10.138) y el servidor Mosquitto (IP 192.168.10.169) se implementa un mecanismo de *fast failover* mediante grupos de tipo FF en los switches S1 y S6. Cada grupo monitoriza un puerto primario (`watch_port`) y redirige automáticamente el tráfico por un puerto secundario en caso de falla, garantizando alta disponibilidad del servicio MQTT en el puerto TCP 1883.

Switch S1 (DPID 1)

Se crean dos grupos FF:

- **group_id=1** para ESP32→Mosquitto, con buckets que prueban primero el enlace en el puerto 6 y como respaldo el puerto 7, ambos enviando a la salida 3.
- **group_id=2** para Mosquitto→ESP32, igual configuración pero salidas 6 y 7.

Los flujos MQTT utilizan estos grupos para la acción de salida. Además, se define un flujo de respaldo directo que recibe paquetes por el puerto 7 y los reenvía por el puerto 3 con la misma prioridad.

```
# 1) Grupo FF ESP32      Mosquitto (group_id=1)
buckets_fwd = [

```

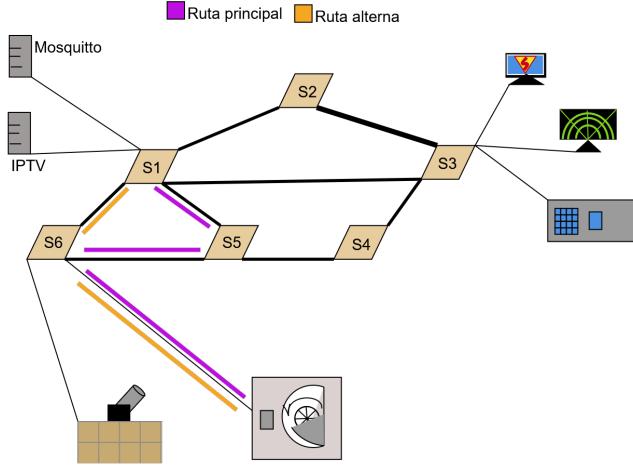


Figura 9: Caminos definidos para generar el re enruteamiento rápido.

```

parser.OFPBucket(watch_port=6, actions=[parser.OFPActionOutput(3)]),
parser.OFPBucket(watch_port=7, actions=[parser.OFPActionOutput(3)])
]
req_fwd = parser.OFPGroupMod(
    datapath=dp, command=oif.OFPGC_ADD, type_=oif.OFPGT_FF,
    group_id=1, buckets=buckets_fwd
)
dp.send_msg(req_fwd)

# 2) Grupo FF Mosquitto      ESP32 (group_id=2)
buckets_rev = [
    parser.OFPBucket(watch_port=6, actions=[parser.OFPActionOutput(6)]),
    parser.OFPBucket(watch_port=7, actions=[parser.OFPActionOutput(7)])
]
req_rev = parser.OFPGroupMod(
    datapath=dp, command=oif.OFPGC_ADD, type_=oif.OFPGT_FF,
    group_id=2, buckets=buckets_rev
)
dp.send_msg(req_rev)

# 3a) Flujo ESP32      Mosquitto usando group_id=1
m_mqtt_fwd = parser.OFPMatch(
    in_port=6, eth_type=ether_types.ETH_TYPE_IP, ip_proto=6,
    ipv4_src="192.168.10.138", ipv4_dst="192.168.10.169",
    tcp_dst=1883
)
dp.send_msg(parser.OFFlowMod(
    datapath=dp, priority=100, match=m_mqtt_fwd,
    instructions=[parser.OFPIInstructionActions(
        ofp.OFPIT_APPLY_ACTIONS, [parser.OFPActionGroup(group_id=1)]
    )]
))
# 3b) Flujo Mosquitto      ESP32 usando group_id=2
m_mqtt_rev = parser.OFPMatch(
    in_port=3, eth_type=ether_types.ETH_TYPE_IP, ip_proto=6,
    ipv4_src="192.168.10.169", ipv4_dst="192.168.10.138",
    tcp_src=1883
)
dp.send_msg(parser.OFFlowMod(
    datapath=dp, priority=100, match=m_mqtt_rev,
    instructions=[parser.OFPIInstructionActions(
        ofp.OFPIT_APPLY_ACTIONS, [parser.OFPActionGroup(group_id=2)]
    )]
))

```

```

# 4) Flujo de respaldo directo (ingreso por 7      salida 3)
match_bkp_s1 = parser.OFPMatch(
    in_port=7, eth_type=ether_types.ETH_TYPE_IP, ip_proto=6,
    ipv4_src="192.168.10.138", ipv4_dst="192.168.10.169",
    tcp_dst=1883
)
actions_bkp_s1 = [parser.OFPActionOutput(3)]
dp.send_msg(parser.OFPPFlowMod(
    datapath=dp, priority=100, match=match_bkp_s1,
    instructions=[parser.OFPIInstructionActions(
        ofp.OFPIT_APPLY_ACTIONS, actions_bkp_s1
    )]
))

```

Listing 23: Grupos FF y flujos MQTT en S1

Switch S5 (DPID 5)

En S5 el tráfico MQTT simplemente se reenvía sin grupos de failover:

```

# Forward MQTT: ingreso por s5-eth3      salida por s5-eth1
m1 = parser.OFPMatch(
    in_port=3, eth_type=0x0800, ip_proto=6,
    ipv4_src="192.168.10.138", ipv4_dst="192.168.10.169",
    tcp_dst=1883
)
a1 = [parser.OFPActionOutput(1)]
dp.send_msg(parser.OFPPFlowMod(
    datapath=dp, priority=100, match=m1,
    instructions=[parser.OFPIInstructionActions(
        ofp.OFPIT_APPLY_ACTIONS, a1
    )]
))

# Reverse MQTT: ingreso por s5-eth1      salida por s5-eth3
m2 = parser.OFPMatch(
    in_port=1, eth_type=0x0800, ip_proto=6,
    ipv4_src="192.168.10.169", ipv4_dst="192.168.10.138",
    tcp_src=1883
)
a2 = [parser.OFPActionOutput(3)]
dp.send_msg(parser.OFPPFlowMod(
    datapath=dp, priority=100, match=m2,
    instructions=[parser.OFPIInstructionActions(
        ofp.OFPIT_APPLY_ACTIONS, a2
    )]
))

```

Listing 24: Flujos MQTT en S5

Switch S6 (DPID 6)

En S6 se repite la creación de grupos FF para manejar la redundancia en ambos sentidos, similar a S1, y se añaden los flujos MQTT que utilizan estos grupos. Además se define un flujo de respaldo directo en caso de caída del enlace primario.

```

# 1) Grupo FF ESP32 Mosquitto (in_port=4)
buckets_fwd_s6 = [
    parser.OFPBucket(watch_port=2, actions=[parser.OFPActionOutput(2)]),
    parser.OFPBucket(watch_port=3, actions=[parser.OFPActionOutput(3)])
]
req_fwd_s6 = parser.OFPGGroupMod(
    datapath=dp, command=oifp.OFPGC_ADD, type_=oifp.OFPGT_FF,
    group_id=1, buckets=buckets_fwd_s6
)
dp.send_msg(req_fwd_s6)

# 2) Grupo FF MosquittoESP32 (in_port=2)
buckets_rev_s6 = [
    parser.OFPBucket(watch_port=2, actions=[parser.OFPActionOutput(4)]),
    parser.OFPBucket(watch_port=3, actions=[parser.OFPActionOutput(4)])
]
req_rev_s6 = parser.OFPGGroupMod(
    datapath=dp, command=oifp.OFPGC_ADD, type_=oifp.OFPGT_FF,
    group_id=2, buckets=buckets_rev_s6
)

```

```

dp.send_msg(req_rev_s6)

# 3a) ESP 3.2 Mosquitto usando group_id=1
m_f = parser.OFPMatch(
    in_port=4, eth_type=ether_types.ETH_TYPE_IP, ip_proto=6,
    ipv4_src="192.168.10.138", ipv4_dst="192.168.10.169",
    tcp_dst=1883
)
dp.send_msg(parser.OFPPFlowMod(
    datapath=dp, priority=100, match=m_f,
    instructions=[parser.OFPIInstructionActions(
        ofp.OFPIT_APPLY_ACTIONS, [parser.OFPActionGroup(group_id=1)]
    )]
))
))

# 3b) Mosquitto ESP32 usando group_id=2
m_r = parser.OFPMatch(
    in_port=2, eth_type=ether_types.ETH_TYPE_IP, ip_proto=6,
    ipv4_src="192.168.10.169", ipv4_dst="192.168.10.138",
    tcp_src=1883
)
dp.send_msg(parser.OFPPFlowMod(
    datapath=dp, priority=100, match=m_r,
    instructions=[parser.OFPIInstructionActions(
        ofp.OFPIT_APPLY_ACTIONS, [parser.OFPActionGroup(group_id=2)]
    )]
))
))

# 4) Flujo de respaldo directo (ingreso por 3      salida 4)
match_bkp_s6 = parser.OFPMatch(
    in_port=3, eth_type=ether_types.ETH_TYPE_IP, ip_proto=6,
    ipv4_src="192.168.10.169", ipv4_dst="192.168.10.138",
    tcp_src=1883
)
actions_bkp_s6 = [parser.OFPActionOutput(4)]
dp.send_msg(parser.OFPPFlowMod(
    datapath=dp, priority=100, match=match_bkp_s6,
    instructions=[parser.OFPIInstructionActions(
        ofp.OFPIT_APPLY_ACTIONS, actions_bkp_s6
    )]
))
)

```

Listing 25: Grupos FF y flujos MQTT en S6

IV-B4. *Tráfico normal entre VLAN 10 y VLAN 30:* Para el tráfico estándar entre VLAN 10 y VLAN 30, el controlador configura en el switch S1 un grupo de tipo SELECT que reparte el tráfico con peso 80/20, etiquetando con VLAN ID 10 en sentido descendente y desencapsulando VLAN ID 30 en el retorno. A continuación se detallan las reglas en cada switch.

IV-B4a. Switch S1 (DPID 1): Se crea un grupo SELECT (group_id=10) para balancear el tráfico VLAN10→VLAN30:

```

# 1) Grupo SELECT para balanceo 80/20 del tráfico VLAN10→VLAN30
buckets = [
    parser.OFPBucket(
        weight=80,
        actions=[
            parser.OFPActionPushVlan(ether_types.ETH_TYPE_8021Q),
            parser.OFPActionSetField(vlan_vid=(0x1000 | 10)),
            parser.OFPActionOutput(6) # a s1-eth6      S5
        ]
    ),
    parser.OFPBucket(
        weight=20,
        actions=[
            parser.OFPActionPushVlan(ether_types.ETH_TYPE_8021Q),
            parser.OFPActionSetField(vlan_vid=(0x1000 | 10)),
            parser.OFPActionOutput(4) # a s1-eth4      S2
        ]
    )
]
dp.send_msg(parser.OFPGGroupMod(
    datapath=dp,
    command=oifp.OFGPC_ADD,
    type_=oifp.OFGPT_SELECT,
    group_id=10,

```

```

        buckets=buckets
    )))
# 2) Flujos que usan el grupo SELECT (hosts en puertos 1,2,3)
for in_p in [1,2,3]:
    match = parser.OFPMatch(
        in_port=in_p,
        eth_type=ether_types.ETH_TYPE_IP
    )
    dp.send_msg(parser.OFFFlowMod(
        datapath=dp,
        priority=10,
        match=match,
        instructions=[parser.OFPIInstructionActions(
            ofp.OFPIT_APPLY_ACTIONS,
            [parser.OFPActionGroup(group_id=10)]
        )]
    )))
# 3) Retorno VLAN30: ingreso desde S2 (puerto 4), pop VLAN y salida a hosts
match_return_s2 = parser.OFPMatch(
    in_port=4,
    eth_type=ether_types.ETH_TYPE_IP,
    vlan_vid=(0x1000 | 30)
)
actions_return = [
    parser.OFPActionPopVlan(),
    parser.OFPActionOutput(1),
    parser.OFPActionOutput(2),
    parser.OFPActionOutput(3)
]
dp.send_msg(parser.OFFFlowMod(
    datapath=dp,
    priority=10,
    match=match_return_s2,
    instructions=[parser.OFPIInstructionActions(
        ofp.OFPIT_APPLY_ACTIONS, actions_return
    )]
))
# 4) Retorno VLAN30: ingreso desde S5 (puerto 6), mismo comportamiento
match_return_s5 = parser.OFPMatch(
    in_port=6,
    eth_type=ether_types.ETH_TYPE_IP,
    vlan_vid=(0x1000 | 30)
)
dp.send_msg(parser.OFFFlowMod(
    datapath=dp,
    priority=10,
    match=match_return_s5,
    instructions=[parser.OFPIInstructionActions(
        ofp.OFPIT_APPLY_ACTIONS, actions_return
    )]
))

```

Listing 26: Grupo SELECT y flujos en S1

IV-B4b. Switch S2 (DPID 2): En S2 se enruta el tráfico etiquetado de VLAN 10 de S1→S3 y VLAN 30 de S3→S1:

```

# VLAN 10: ingreso por s2-eth1      salida por s2-eth2 (hacia S3)
m_vlan10 = parser.OFPMatch(
    in_port=1,
    eth_type=ether_types.ETH_TYPE_IP,
    vlan_vid=(0x1000 | 10)
)
dp.send_msg(parser.OFFFlowMod(
    datapath=dp, priority=20, match=m_vlan10,
    instructions=[parser.OFPIInstructionActions(
        ofp.OFPIT_APPLY_ACTIONS, [parser.OFPActionOutput(2)]
    )]
))
# VLAN 30: ingreso por s2-eth2      salida por s2-eth1 (hacia S1)
m_vlan30 = parser.OFPMatch(

```

```

        in_port=2,
        eth_type=ether_types.ETH_TYPE_IP,
        vlan_vid=(0x1000 | 30)
    )
    dp.send_msg(parser.OFPPFlowMod(
        datapath=dp, priority=20, match=m_vlan30,
        instructions=[parser.OFPInstructionActions(
            ofp.OFPIT_APPLY_ACTIONS, [parser.OFPActionOutput(1)]
        )]
    ))
)

```

Listing 27: Flujos VLAN en S2

IV-B4c. Switch S3 (DPID 3): Para el retorno de VLAN 30 se emplea un grupo SELECT group_id=30 con pesos 80/20 entre los enlaces hacia S4 (puerto 5) y la ruta alternativa (puerto 4). Además, se configuran flujos que usan dicho grupo, y flujos de entrega de VLAN 10 a los hosts y AP.

```

# 1) Grupo SELECT para retorno VLAN30 con peso 80/20
buckets_vlan30_return = [
    parser.OFPBucket(
        weight=80,
        actions=[
            parser.OFPActionPushVlan(ether_types.ETH_TYPE_8021Q),
            parser.OFPActionSetField(vlan_vid=(0x1000 | 30)),
            parser.OFPActionOutput(5) # a s3-eth5      S4
        ]
    ),
    parser.OFPBucket(
        weight=20,
        actions=[
            parser.OFPActionPushVlan(ether_types.ETH_TYPE_8021Q),
            parser.OFPActionSetField(vlan_vid=(0x1000 | 30)),
            parser.OFPActionOutput(4) # ruta alternativa a S2
        ]
    )
]
dp.send_msg(parser.OFPGroupMod(
    datapath=dp,
    command=ofp.OFGC_ADD,
    type_=ofp.OFGT_SELECT,
    group_id=30,
    buckets=buckets_vlan30_return
))

# 2) Flujos de retorno usando group_id=30 (puertos 1,2,6)
for in_p in [1,2,6]:
    match_ret = parser.OFPMatch(
        in_port=in_p,
        eth_type=ether_types.ETH_TYPE_IP
    )
    dp.send_msg(parser.OFPPFlowMod(
        datapath=dp, priority=10, match=match_ret,
        instructions=[parser.OFPInstructionActions(
            ofp.OFPIT_APPLY_ACTIONS,
            [parser.OFPActionGroup(group_id=30)]
        )]
    ))
)

# 3) Entrega VLAN10: pop VLAN y salida a h2,h3,AP (puertos 1,2,6)
for in_p in [5,4]:
    m_h6_all = parser.OFPMatch(
        in_port=in_p,
        eth_type=ether_types.ETH_TYPE_IP,
        vlan_vid=(0x1000 | 10)
    )
    actions_h6_all = [
        parser.OFPActionPopVlan(),
        parser.OFPActionOutput(1),
        parser.OFPActionOutput(2),
        parser.OFPActionOutput(6)
    ]
    dp.send_msg(parser.OFPPFlowMod(
        datapath=dp, priority=10, match=m_h6_all,
        instructions=[parser.OFPInstructionActions(

```

```

        ofp.OFPIT_APPLY_ACTIONS, actions_h6_all
    )
)

```

Listing 28: Grupo SELECT y flujos en S3

IV-B4d. Switch S4 (DPID 4): Ruteo de VLAN 10 de S5→S3 y retorno de VLAN 30:

```

# VLAN10: ingreso por s4-eth2      salida por s4-eth1 (hacia S3)
m_h6_h2 = parser.OFPMatch(
    in_port=2,
    eth_type=ether_types.ETH_TYPE_IP,
    vlan_vid=(0x1000 | 10)
)
dp.send_msg(parser.OFPFlowMod(
    datapath=dp, priority=10, match=m_h6_h2,
    instructions=[parser.OFPIInstructionActions(
        ofp.OFPIT_APPLY_ACTIONS, [parser.OFPActionOutput(1)]
    )]
))

# Retorno VLAN30: ingreso por s4-eth1      salida por s4-eth2 (hacia S5)
m_ret = parser.OFPMatch(
    in_port=1,
    eth_type=ether_types.ETH_TYPE_IP,
    vlan_vid=(0x1000 | 30)
)
dp.send_msg(parser.OFPFlowMod(
    datapath=dp, priority=10, match=m_ret,
    instructions=[parser.OFPIInstructionActions(
        ofp.OFPIT_APPLY_ACTIONS, [parser.OFPActionOutput(2)]
    )]
))

```

Listing 29: Flujos en S4

IV-B4e. Switch S5 (DPID 5): Ruteo de VLAN 10 de S1→S4 y retorno de VLAN 30 de S4→S1:

```

# VLAN10: ingreso por s5-eth1      salida por s5-eth2 (hacia S4)
m_h6_h2 = parser.OFPMatch(
    in_port=1,
    eth_type=ether_types.ETH_TYPE_IP,
    vlan_vid=(0x1000 | 10)
)
dp.send_msg(parser.OFPFlowMod(
    datapath=dp, priority=10, match=m_h6_h2,
    instructions=[parser.OFPIInstructionActions(
        ofp.OFPIT_APPLY_ACTIONS, [parser.OFPActionOutput(2)]
    )]
))

# Retorno VLAN30: ingreso por s5-eth2      salida por s5-eth1 (hacia S1)
m_ret = parser.OFPMatch(
    in_port=2,
    eth_type=ether_types.ETH_TYPE_IP,
    vlan_vid=(0x1000 | 30)
)
dp.send_msg(parser.OFPFlowMod(
    datapath=dp, priority=10, match=m_ret,
    instructions=[parser.OFPIInstructionActions(
        ofp.OFPIT_APPLY_ACTIONS, [parser.OFPActionOutput(1)]
    )]
))

```

Listing 30: Flujos en S5

IV-B5. Balanceo de carga dinámico entre VLAN 10 y VLAN 30: El controlador incorpora un mecanismo automático de balanceo de carga que ajusta en tiempo real la proporción de tráfico entre los enlaces s1-s5 y s1-s2 (grupo 10) y entre s3-s4 y s3-s2 (grupo 30) cuando detecta congestión excesiva. Para ello, se definen las siguientes rutinas:

```

def _monitor(self):
    """Cada POLL_INTERVAL sonda en S1 y S3 todos los flujos IP."""
    while True:
        for dp in self.datapaths.values():

```

```

if dp.id in [1, 3]:
    parser = dp.ofproto_parser
    # Solicitar stats de todos los flujos IP en tabla 0
    req = parser.OFFFlowStatsRequest(
        dp,
        table_id=0,
        match=parser.OFPMatch(eth_type=ether_types.ETH_TYPE_IP)
    )
    dp.send_msg(req)
hub.sleep(self.POLL_INTERVAL)

```

Listing 31: Hilo de monitoreo de estadísticas en S1 y S3

El método `_monitor` envía periódicamente una petición de estadísticas de flujo (`OFPFlowStatsRequest`) a los switches S1 y S3. A continuación, la respuesta se maneja en el evento `EventOFPFlowStatsReply`, donde se acumulan los bytes transmitidos en los puertos de interés y se calcula el bitrate agregado:

```

@set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
def _flow_stats_reply(self, ev):
    dp = ev.msg.datapath
    dpid = dp.id
    if dpid not in [1, 3]:
        return

    total_bits = 0
    for stat in ev.msg.body:
        in_p = stat.match.get('in_port')
        # Filtrar flujos de interes seg n switch
        if (dpid == 1 and in_p in [1, 2, 3]) or \
            (dpid == 3 and in_p in [1, 2, 6]):
            key = (dpid, in_p)
            prev = self.prev_flow_bytes.get(key, 0)
            delta = stat.byte_count - prev
            self.prev_flow_bytes[key] = stat.byte_count
            total_bits += delta * 8

    bps = total_bits / self.POLL_INTERVAL
    self.logger.info("Bitrate VLAN1030: %.2f bps", bps)

    if bps > UMBRAL_BPS and not self.high_congestion:
        self.high_congestion = True
        self.logger.warning("Umbral superado: cambio a 50/50")
        self._set_groups_50_50()
    elif bps <= UMBRAL_BPS and self.high_congestion:
        self.high_congestion = False
        self.logger.info("Trafico normalizado: vuelvo a 80/20")
        self._set_groups_original()

```

Listing 32: Procesamiento de estadísticas de flujo y detección de umbral

Si el bitrate agregado supera el umbral `UMBRAL_BPS`, el controlador invoca `_set_groups_50_50()`, que modifica los grupos SELECT en S1 y S3 para repartir el tráfico en proporción 50/50. Asimismo, ajusta dinámicamente la ruta del tráfico MQTT de la Raspberry para aliviar carga sobreenrutada:

```

def _set_groups_50_50(self):
    # S1: grupo_id=10 pasa a 50/50
    dp1 = self.datapaths.get(1)
    if dp1:
        parser = dp1.ofproto_parser
        buckets = [
            parser.OFPBucket(weight=50,
                actions=[
                    parser.OFPActionPushVlan(...),
                    parser.OFPActionSetField(vlan_vid=(0x1000|10)),
                    parser.OFPActionOutput(6)
                ]),
            parser.OFPBucket(weight=50,
                actions=[
                    parser.OFPActionPushVlan(...),
                    parser.OFPActionSetField(vlan_vid=(0x1000|10)),
                    parser.OFPActionOutput(4)
                ])
        ]

```

```

dp1.send_msg(parser.OFPGGroupMod(
    datapath=dp1, command=dp1.ofproto.OFPGC_MODIFY,
    type_=dp1.ofproto.OFPGT_SELECT, group_id=10,
    buckets=buckets))

# S3: grupo_id=30 pasa a 50/50
dp3 = self.datapaths.get(3)
if dp3:
    parser = dp3.ofproto_parser
    buckets = [
        parser.OFPBucket(weight=50,
            actions=[/* tag VLAN30 + output 5 */]),
        parser.OFPBucket(weight=50,
            actions=[/* tag VLAN30 + output 4 */])
    ]
    dp3.send_msg(parser.OFPGGroupMod(
        datapath=dp3, command=dp3.ofproto.OFPGC_MODIFY,
        type_=dp3.ofproto.OFPGT_SELECT, group_id=30,
        buckets=buckets))

# Reroute MQTT-Raspberry: cambiar puertos en S1 y S3
# en S1: retorno MosquitoRaspberry sale por puerto 5
dp1.send_msg(parser.OFPFlowMod(...))
# en S3: ida RaspberryMosquito sale por puerto 3
dp3.send_msg(parser.OFPFlowMod(...))

```

Listing 33: Modificación de grupos a 50/50 y reroute MQTT

Cuando la congestión remite por debajo del umbral, se llama a `_set_groups_original()`, que restaura las proporciones iniciales (80/20 en S1 y 60/40 en S3) y devuelve las rutas MQTT a su estado primario:

```

def _set_groups_original(self):
    # Restaurar S1: group_id=10 a 80/20
    # Restaurar S3: group_id=30 a 60/40
    # Restaurar flujos MQTT-Raspberry en S1 y S3
    dp1.send_msg(parser.OFPGGroupMod(...))
    dp3.send_msg(parser.OFPGGroupMod(...))
    dp1.send_msg(parser.OFPFlowMod(...))
    dp3.send_msg(parser.OFPFlowMod(...))

```

Listing 34: Restauración de grupos y rutas originales

Este esquema permite que la red se adapte de forma proactiva a variaciones de carga, equilibrando el tráfico de las VLANs cuando sea necesario y garantizando la eficiencia y resiliencia del servicio.

IV-C. Aplicación IoT: Gestión de temperatura

En primer lugar se procedió a la implementación del hardware. Esto último acorde al esquemático presentado en la Figura 4. Las conexiones se realizaron con *jumpers* macho hembras únicamente. El receptor armado se presenta en la Figura 10.

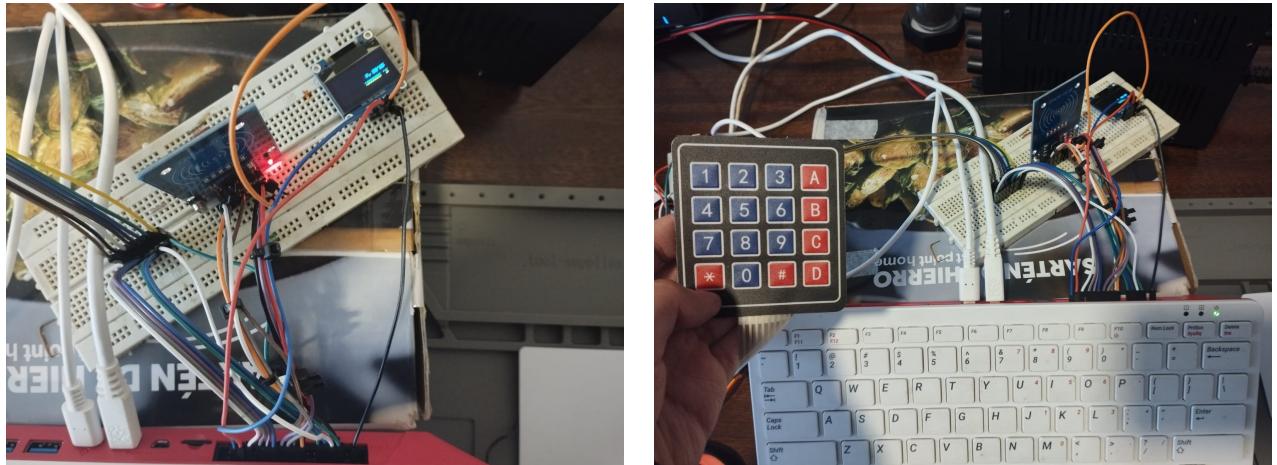


Figura 10: Armado físico del Transmisor de referencias/ Receptor de temperatura del Sistema de gestión de temperatura.

Luego se procedió a codificar el sistema a partir del lenguaje Python en la Raspberry Pi. La codificación parte de ciertos bloques de código que se refieren a la programación de la pantalla, el RFID o el teclado 4x4. Ciertas partes del código en cambio refieren a instrucciones que manejan el flujo general del programa. En este sentido nos centraremos mayormente en el flujo principal del programa pues demuestra el funcionamiento principal y de interés para la aplicación. Cabe recalcar que se hará lo mismo para las siguientes codificaciones IoT.

Aclarado lo anterior, el sistema manejado por la Raspberry PI, el autenticador y transmisor de referencias de temperatura está implementado en base al flujo que se presenta en la Figura 11. Note que el sistema no es persistente y en caso de no generar una conexión exitosa con Mosquitto procede a cerrarse. Luego se genera un modo Visualización que permite, como su nombre lo dice, visualizar la temperatura que proviene del receptor (sensor). Esto último lo consigue a través de la suscripción a Mosquitto al tópico publicado por el receptor "temp/sensor".

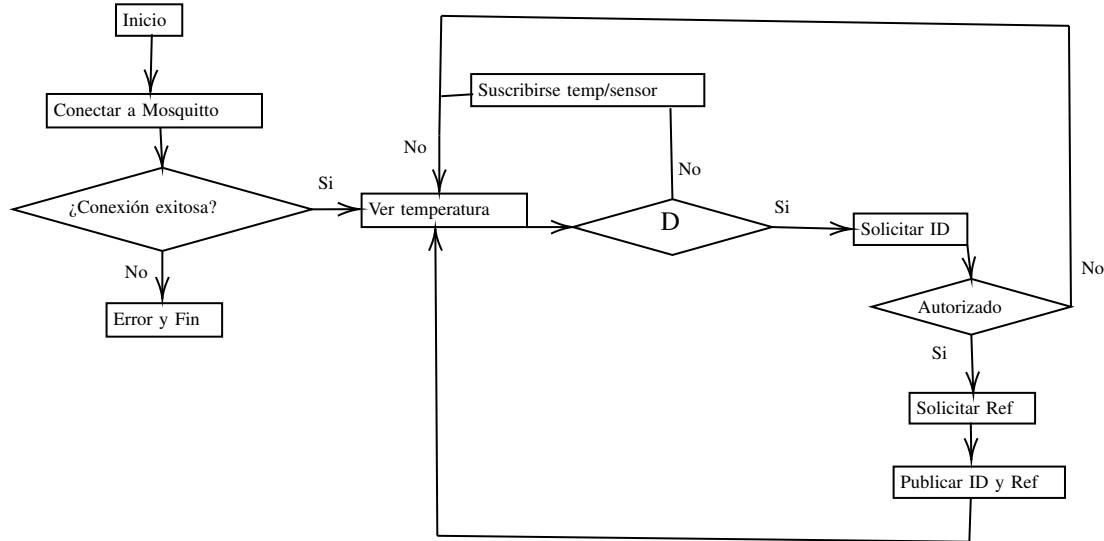


Figura 11: Diagrama de flujo del transmisor de referencia y autenticador del sistema de gestión de temperatura.

A continuación se presentará la implementación de receptor y publicador de temperatura. En este contexto se ha implementado el hardware correspondiente a la Figura 10.

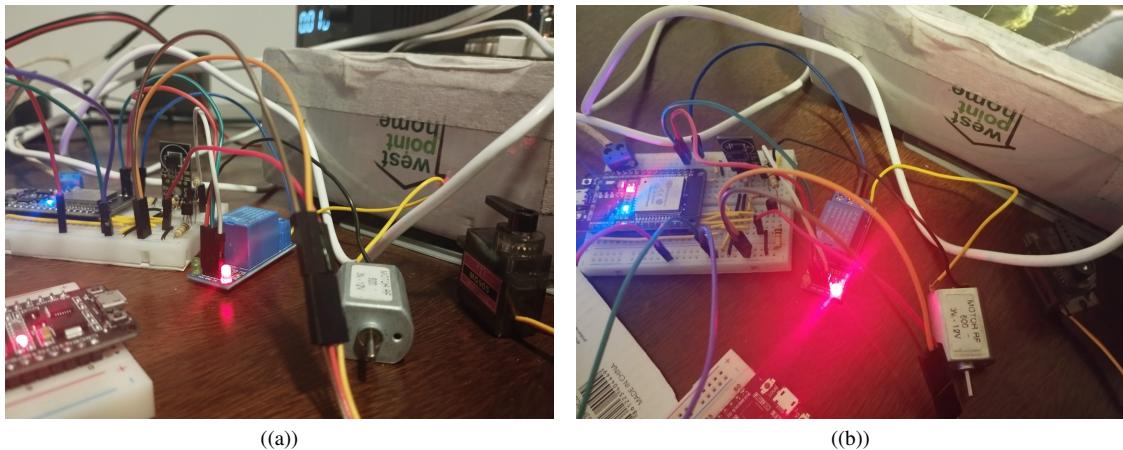


Figura 12: Armado físico del Receptor de referencias/ Transmisor de temperatura del Sistema de gestión de temperatura.

Al igual que antes se procederá a describir el flujo principal de la programación de la ESP32 que hace de receptor. El diagrama de flujo se ubica en la Figura 13. El flujo muestra que principalmente el receptor procede siempre a medir la temperatura para luego publicarla en el tópico "temp/sensor". No obstante se ha empleado un tipo de función *callback* en la

programación del IDE Arduino para generar una variación del valor de ref únicamente cuando exista un cambio en el tópico en la referencia enviada por el sistema implementado en la Raspberry PI.

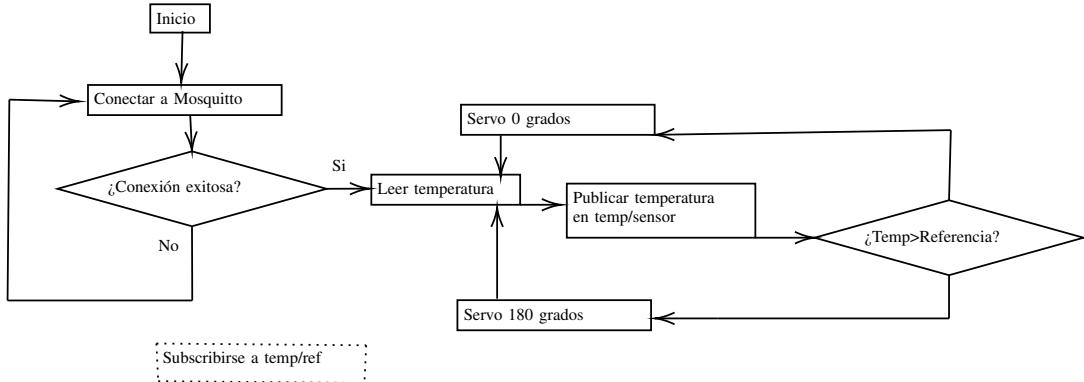


Figura 13: Se presenta el flujo principal del receptor/publicador de temperaturas y aquel que concede la apertura de la ventanilla en el sistema de gestión de temperatura.

IV-D. Aplicación IoT: Radar con interfaz gráfica

La sección de mayor interés del radar procede de su transmisor de ángulos y distancias, pues el programa que grafica el panel de radios constituye únicamente un programa instaurado en Java en el apartado *Processing* [1]. No obstante, se realizará una pequeña descripción de sus utilidades y sus líneas de mayor importancia.

Acorde a estos lineamientos, en primer lugar se procedió a armar el radar giratorio a partir de la aplicación del esquema de la Figura 6. El armado físico se presenta en la Figura 14.

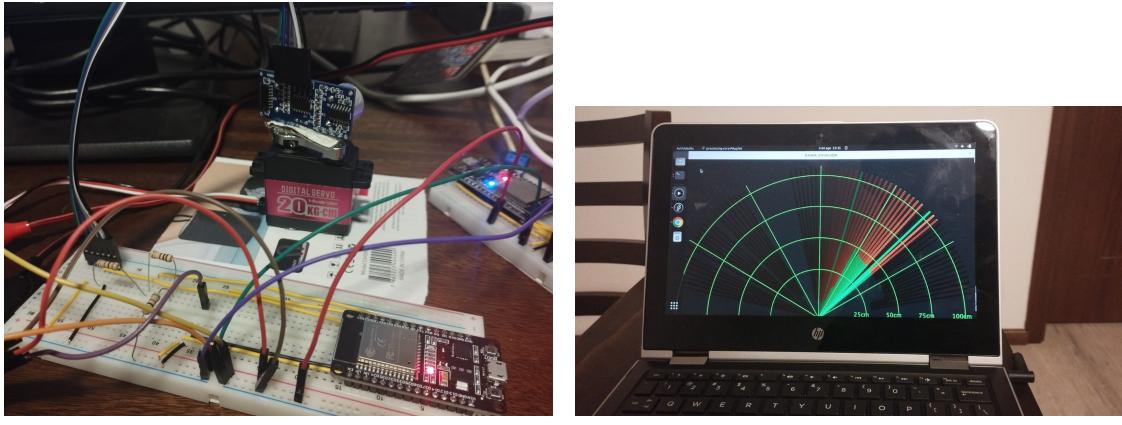


Figura 14: A la izquierda se presenta el radar giratorio implementado en armado físico. A su derecha se halla la interfaz gráfica producida por el receptor.

Dado el armado físico se procede a programar el transmisor a partir del diagrama de flujo presentado en la Figura 15. El esquema es relativamente simple. El servo se dedica a girar dos grados en cada iteración. En cada uno de sus *pasos* toma una medida de la distancia a la que se encuentra un objeto. En este sentido genera un mapeo de grados de giro con distancia calculada. El cálculo de la distancia se realiza en base a las fórmulas típicas relacionadas al viaje de las ondas sonoras en el vacío.

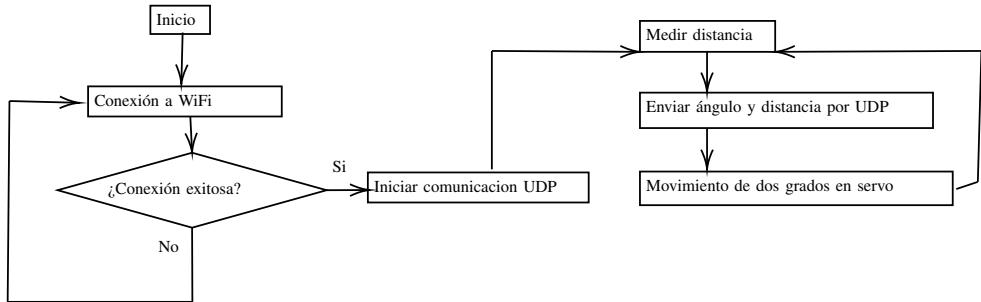


Figura 15: Se presenta el diagrama de flujo del transmisor de radar.

Para presentar

V. RESULTADOS Y ANÁLISIS

V-A. Tablas de flujo creadas

A continuación de listan las tablas de flujo y grupo que fueron creadas con el controlador cuando la red se encuentra en el estado normal (sin congestión). Estas tablas corresponden a los puertos de cada uno de los switches de la topología mencionada previamente. Como se puede observar, los switches de borde cargan con muchas más tablas de flujo y grupo por lo que estos requieren mucho más procesamiento y capacidad. Sin embargo, esta carga solo en los extremos está compensada con la baja capacidad y costo de procesamiento de los switches de core que lo único que hacen es escuchar puertos y sacar paquetes por otros puertos.

Prioridad	Protocolo	Puerto de entrada	IP origen	IP destino	Puerto transporte	Acciones
100	TCP	s1-eth6	192.168.10.138	192.168.10.169	dst 1883	group:1
100	TCP	s1-eth7	192.168.10.138	192.168.10.169	dst 1883	output:s1-eth3
200	UDP	s1-eth7	192.168.10.150	192.168.10.108	dst 2000	output:s1-eth5
100	TCP	s1-eth5	192.168.10.105	192.168.10.169	dst 1883	output:s1-eth3
100	TCP	s1-eth4	192.168.10.105	192.168.10.169	dst 1883	output:s1-eth3
100	TCP	s1-eth3	192.168.10.169	192.168.10.138	src 1883	group:2
200	UDP	s1-eth5	192.168.10.108	192.168.10.150	src 2000	output:s1-eth7
100	TCP	s1-eth3	192.168.10.169	192.168.10.105	src 1883	output:s1-eth4
100	ARP	—	—	—	—	CONTROLLER:65535
30	UDP	s1-eth1	—	—	dst 5004	mod_vlan_vid:10, output:s1-eth4
30	UDP	s1-eth2	—	—	dst 5004	mod_vlan_vid:10, output:s1-eth4
30	UDP	s1-eth3	—	—	dst 5004	mod_vlan_vid:10, output:s1-eth4
30	UDP	s1-eth4	—	—	src 5004	group:20
10	IP	s1-eth1	—	—	—	group:10
10	IP	s1-eth2	—	—	—	group:10
10	IP	s1-eth3	—	—	—	group:10
10	IP	s1-eth4	—	—	—	strip_vlan, output:s1-eth1/output:s1-eth2/output:s1-eth3
10	IP	s1-eth6	—	—	—	strip_vlan, output:s1-eth1/output:s1-eth2/output:s1-eth3
0	ANY	—	—	—	—	drop

Cuadro I: Flujos configurados en el switch S1

Prioridad	Protocolo	Puerto de entrada	IP origen	IP destino	Puerto transporte	Acciones
100	ARP	—	—	—	—	CONTROLLER:65535
100	TCP	s2-eth2	192.168.10.105	192.168.10.169	dst 1883	output:s2-eth1
100	TCP	s2-eth1	192.168.10.169	192.168.10.105	src 1883	output:s2-eth2
30	UDP	s2-eth1	—	—	dst 5004	output:s2-eth2
30	UDP	s2-eth2	—	—	src 5004	output:s2-eth1
20	IP	s2-eth1	—	—	—	output:s2-eth2
20	IP	s2-eth2	—	—	—	output:s2-eth1
0	ANY	—	—	—	—	drop

Cuadro II: Flujos configurados en el switch S2

Prioridad	Protocolo	Puerto de entrada	IP origen	IP destino	Puerto transporte	Acciones
100	ARP	—	—	—	—	CONTROLLER:65535
10	IP	s4-eth2	—	—	—	output:s4-eth1
10	IP	s4-eth1	—	—	—	output:s4-eth2
0	ANY	—	—	—	—	drop

Cuadro IV: Flujos configurados en el switch S4

Prioridad	Protocolo	Puerto de entrada	IP origen	IP destino	Puerto transporte	Acciones
100	ARP	—	—	—	—	CONTROLLER:65535
100	TCP	s5-eth3	192.168.10.138	192.168.10.169	dst 1883	output:s5-eth1
100	TCP	s5-eth1	192.168.10.169	192.168.10.138	src 1883	output:s5-eth3
10	IP	s5-eth1	—	—	—	output:s5-eth2
10	IP	s5-eth2	—	—	—	output:s5-eth1
0	ANY	—	—	—	—	drop

Cuadro V: Flujos configurados en el switch S5

Prioridad	Protocolo	Puerto de entrada	IP origen	IP destino	Puerto transporte	Acciones
100	TCP	veth-ovs	192.168.10.138	192.168.10.169	dst 1883	group:1
200	UDP	veth-ovs	192.168.10.150	192.168.10.108	dst 2000	output:s6-eth3
100	TCP	s6-eth2	192.168.10.169	192.168.10.138	src 1883	group:2
100	TCP	s6-eth3	192.168.10.169	192.168.10.138	src 1883	output:veth-ovs
200	UDP	s6-eth3	192.168.10.108	192.168.10.150	src 2000	output:veth-ovs
100	ARP	—	—	—	—	CONTROLLER:65535
0	ANY	—	—	—	—	drop

Cuadro VI: Flujos configurados en el switch S6

V-B. Tablas de grupo creadas

Group ID	Tipo	Buckets
1	ff	watch_port:"s1-eth6"→ output:"s1-eth3" watch_port:"s1-eth7"→ output:"s1-eth3"
10	select	weight:80 → push_vlan:0x8100, set_field:4106->vlan_vid, output:"s1-eth6" weight:20 → push_vlan:0x8100, set_field:4106->vlan_vid, output:"s1-eth4"
20	all	pop_vlan, output:"s1-eth1" pop_vlan, output:"s1-eth2" pop_vlan, output:"s1-eth3"
2	ff	watch_port:"s1-eth6"→ output:"s1-eth6" watch_port:"s1-eth7"→ output:"s1-eth7"

Cuadro VII: Grupos configurados en el switch S1

Group ID	Tipo	Buckets
No hay grupos definidos en S2		

Cuadro VIII: Grupos en el switch S2

Group ID	Tipo	Buckets
21	all	pop_vlan, output:"s3-eth1" pop_vlan, output:"s3-eth2" pop_vlan, output:"veth-ovs1"
30	select	weight:80 → push_vlan:0x8100, set_field:4126->vlan_vid, output:"s3-eth5" weight:20 → push_vlan:0x8100, set_field:4126->vlan_vid, output:"s3-eth4"

Cuadro IX: Grupos configurados en el switch S3

Group ID	Tipo	Buckets
No hay grupos definidos en S4		

Cuadro X: Grupos en el switch S4

Group ID	Tipo	Buckets
No hay grupos definidos en S5		

Cuadro XI: Grupos en el switch S5

Group ID	Tipo	Buckets
1	ff	watch_port:"s6-eth2" → output:"s6-eth2" watch_port:"s6-eth3" → output:"s6-eth3"
2	ff	watch_port:"s6-eth2" → output:"veth-ovs" watch_port:"s6-eth3" → output:"veth-ovs"

Cuadro XII: Grupos configurados en el switch S6

V-C. *Mediciones de tráfico de paquetes*

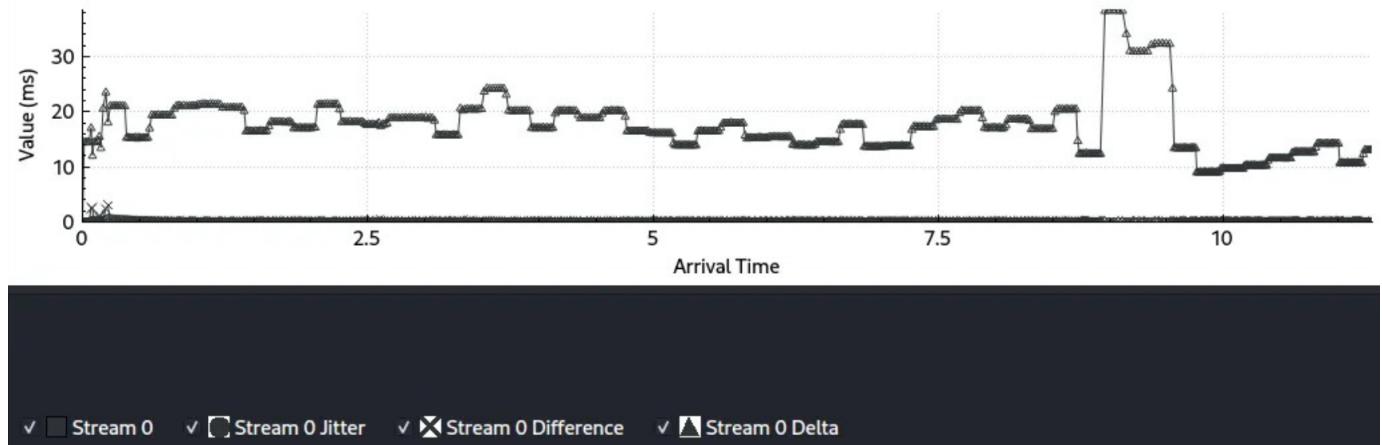


Figura 16: Delta de tiempo de llegada entre paquetes consecutivos en un flujo RTP, extraído de Wireshark.

La Figura 16 muestra el *Delta* de un flujo RTP analizado con Wireshark; el *Delta* se define como la diferencia de tiempo de llegada entre un paquete y el anterior. En este caso el *Delta* permanece prácticamente constante (alrededor de 15–20 ms), tal como corresponde a un escenario sin saturación de paquetes en el tráfico normal entre VLAN 10 y VLAN 30. Además, la línea de estado inferior indica que el *Jitter* es cero, lo que confirma la ausencia de variabilidad en el retardo entre paquetes. Este comportamiento estable constituye el valor de referencia antes de inducir condiciones de congestión en la red.

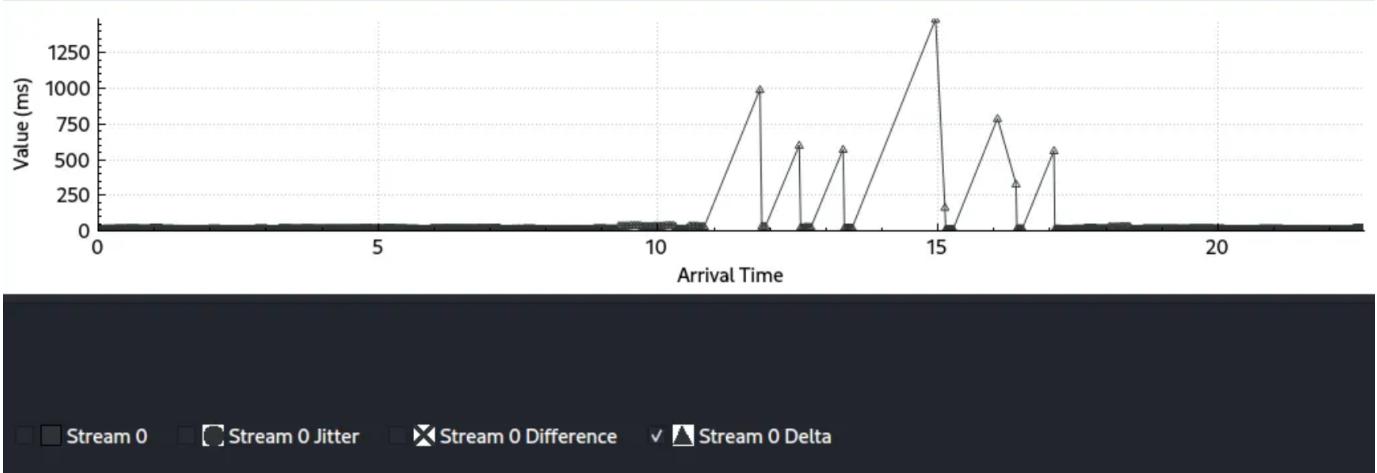


Figura 17: Jitter en la llegada de paquetes RTP durante balanceo de carga.

Durante la fase de balanceo de carga, el tráfico normal entre VLAN 10 y VLAN 30 se reparte en dos rutas ($s_1-s_2-s_3$ compartida con IPTV y $s_1-s_5-s_4-s_3$ para la otra mitad), lo que ocasiona que el flujo RTP experimente variaciones en el retardo, es decir, *jitter*. En la Figura 17 se observa cómo el *jitter* aumenta significativamente al inducir sobrecarga de la red en el segundo 10, y disminuye al cesar la carga en el segundo 17. Este comportamiento confirma que el balanceo está funcionando correctamente: parte del tráfico normal utiliza la ruta alternativa, aliviando la ruta compartida con IPTV y, por ende, modulando el jitter generado por la mezcla de flujos.

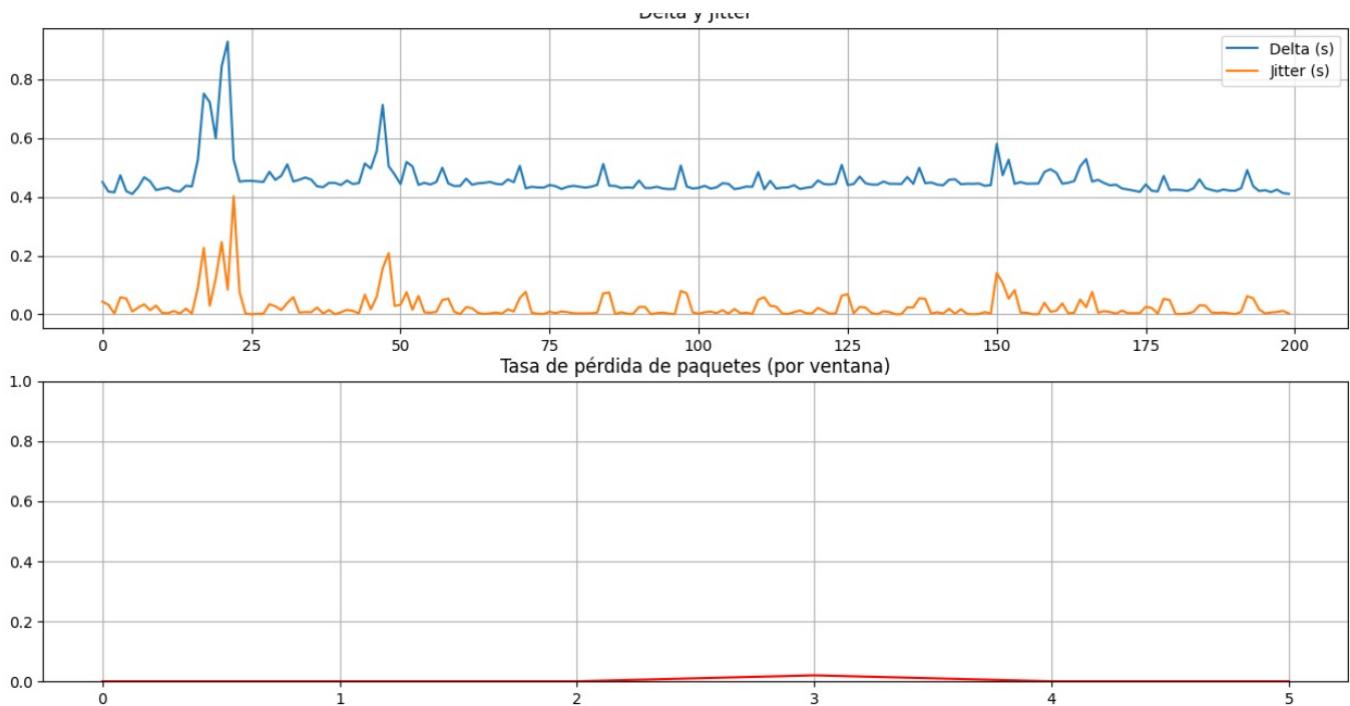


Figura 18: Delta y jitter para el tráfico UDP/2000 (Radar) sin congestión.

El tráfico UDP en el puerto 2000 corresponde al enlace Radar–visualizador, configurado con máxima prioridad y rutas exclusivas que garantizan aislamiento frente al resto de flujos. En la Figura 18 se muestran el *Delta* (línea azul) y el *Jitter* (línea naranja) calculados por ventanas de análisis. El *Delta*, entendido como la diferencia de tiempo entre la llegada de un paquete y el anterior, permanece prácticamente constante a lo largo de todo el muestreo, lo que confirma la estabilidad del enlace dedicado. Asimismo, el *Jitter* se mantiene muy cercano a cero, evidenciando que no existen variaciones apreciables en el retardo cuando no hay congestión de red.

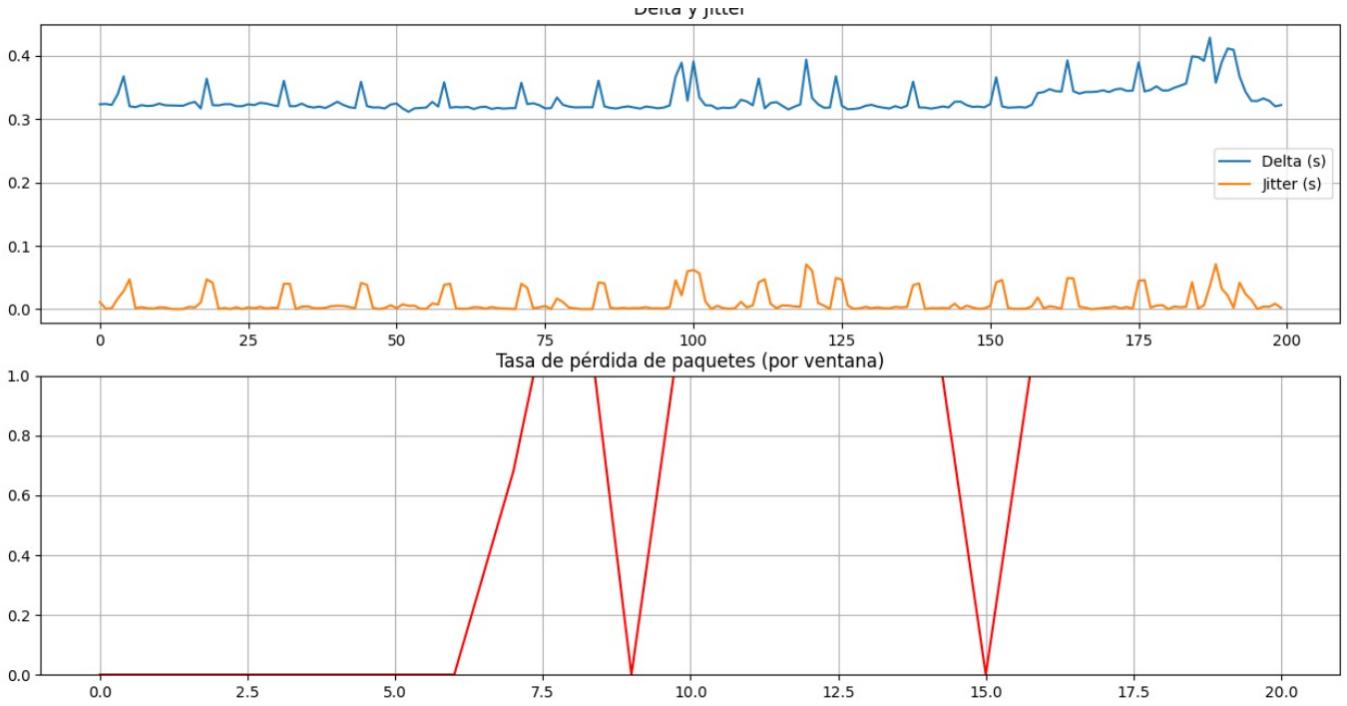


Figura 19: Delta, jitter y tasa de pérdida de paquetes del tráfico UDP/2000 (Radar) durante balanceo de carga.

En la Figura 19 se observa el comportamiento del enlace Radar–visualizador (UDP/2000) cuando deja de ser exclusivo y comparte el tramo s_1-s_3 con el tráfico MQTT. A pesar de la coexistencia de flujos, el *Delta* (diferencia de tiempo entre paquetes consecutivos) se mantiene esencialmente constante alrededor de 0,3–0,4 s, y el *Jitter* permanece cercano a cero, lo que indica que las variaciones de retardo siguen siendo mínimas. En la parte inferior de la figura se muestra la tasa de pérdida de paquetes por ventana, que aunque no es nula —debido a la competencia puntual en el enlace— resulta despreciable en la práctica, dado que el radar envía datos al visualizador con un intervalo aproximado de 3 μ s entre paquetes.

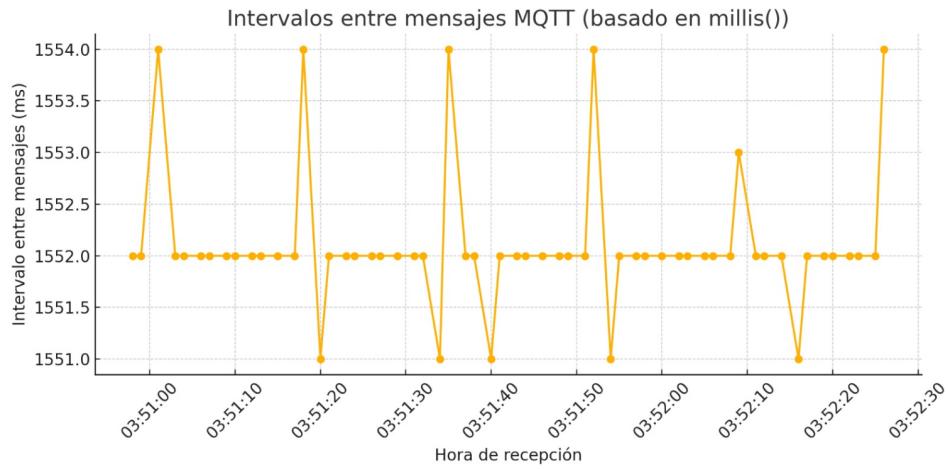


Figura 20: Intervalos de tiempo entre mensajes MQTT medidos con `millis()` en una red sin congestión.

La Figura 20 muestra el *Delta* entre mensajes MQTT en condiciones de red sin congestión. Cada punto representa el intervalo en milisegundos medido con la función `millis()` de la Raspberry Pi. Se aprecia que el valor oscila apenas unos milisegundos alrededor de 1552 ms, lo cual indica una variación mínima y, por tanto, un comportamiento prácticamente constante del flujo MQTT cuando no existe saturación ni interferencia con otros servicios de la red.

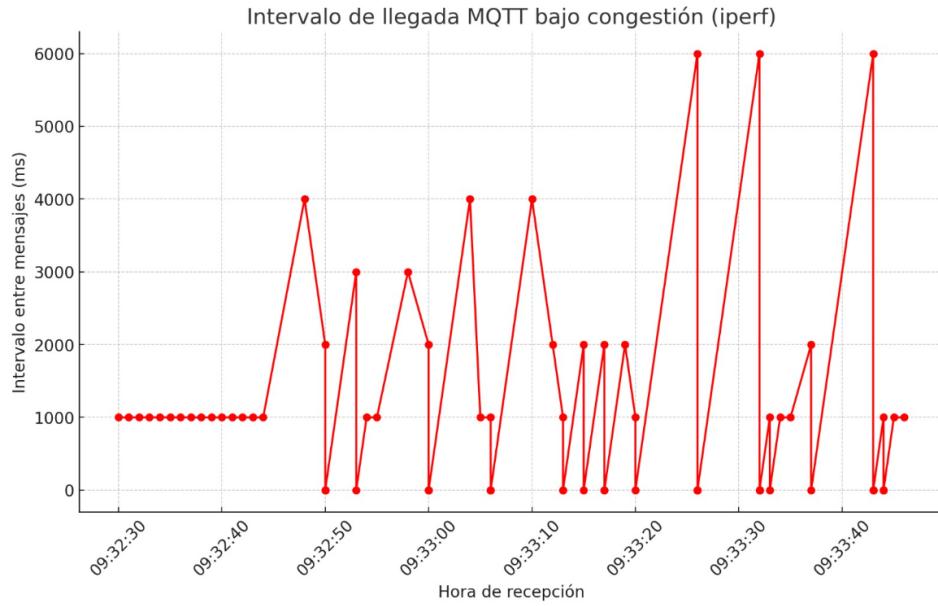


Figura 21: Intervalos de llegada de mensajes MQTT en red congestionada sin ingeniería de tráfico.

La Figura 21 ilustra el comportamiento del flujo MQTT cuando la red se somete a congestión y no se aplican mecanismos de ingeniería de tráfico (TE). En estas condiciones, el *Delta* entre mensajes presenta variaciones muy pronunciadas —con picos que alcanzan varios segundos—, lo que se traduce en un *Jitter* elevado e impredecible. Este aumento de variabilidad puede provocar pérdidas de sincronización en aplicaciones sensibles al retardo, retrasos en la transmisión de datos críticos y empeoramiento de la calidad de servicio.

En particular:

- Los intervalos irregulares indican que los paquetes sufren colas y retrasos variables en los enlaces compartidos ($s_1-s_2-s_3$), comprometiendo la periodicidad deseada de aproximadamente 1,5 s.
- Los saltos bruscos (por encima de 4–6 s) pueden causar desconexiones temporales o expiración de temporizadores en el cliente MQTT, afectando la fiabilidad.
- La falta de rutas alternativas o balanceo mantiene todo el tráfico MQTT sobre la misma ruta congestionada, sin escape frente a cuellos de botella.

Estos resultados confirman la necesidad de emplear TE—por ejemplo, el balanceo de carga basado en grupos SELECT y FAST_FAILOVER descritos anteriormente— para garantizar que el flujo MQTT mantenga un *Delta* estable y un *Jitter* controlado, incluso bajo alta demanda de tráfico normal y servicios de mayor prioridad (IPTV, radar, etc.).

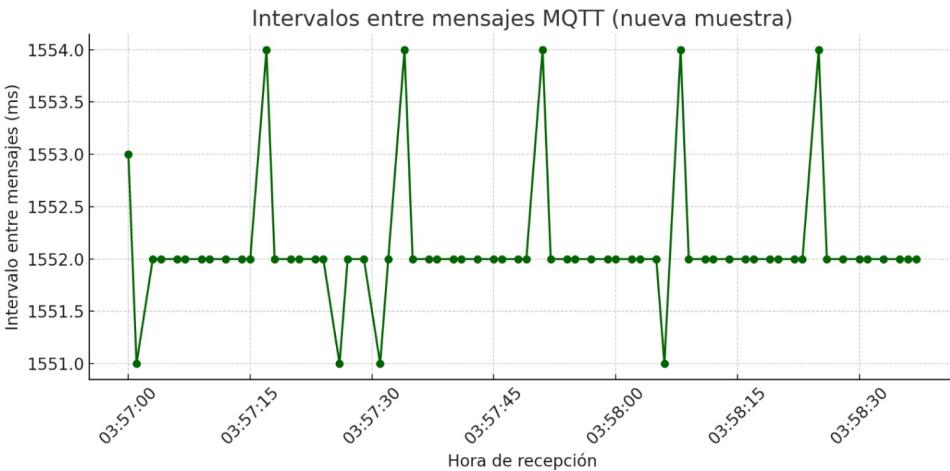


Figura 22: Intervalos entre mensajes MQTT bajo congestión con Ingeniería de Tráfico aplicada.

Al aplicar los mecanismos de Ingeniería de Tráfico (TE) —balanceo de carga con grupos SELECT y rutas alternativas— incluso cuando la red está sometida a congestión, el *Delta* entre mensajes MQTT recupera un comportamiento prácticamente idéntico al observado en condiciones de no congestión (ver Figura 20). En la Figura 22 se aprecia que:

- El valor medio del *Delta* se mantiene alrededor de 1552 ms, con oscilaciones muy pequeñas, lo que indica que la periodicidad de publicación no se ve afectada.
- Las puntas ocasionales corresponden a los momentos de cambio de ruta por el grupo de FAST_FAILOVER o reedición de grupos SELECT, pero retornan inmediatamente al intervalo de referencia.
- El *Jitter* es prácticamente cero (no representado en este gráfico), confirmando que la variabilidad del retardo queda controlada gracias al reparto de carga y la redundancia de enlaces.

Estos resultados demuestran que la TE implementada garantiza la estabilidad del flujo MQTT bajo alta demanda de otros servicios, aislando eficazmente el tráfico crítico de control y evitando la degradación del servicio.

VI. CONCLUSIONES Y RECOMENDACIONES

Se recomienda el empleo de esquemas de amplificación y atenuación de voltaje en la ESP32. Se debe considerar que la ESP32 emplea únicamente 3.3V tanto para sus salidas digitales como para sus entradas. Los dispositivos empleados en este proyecto requieren de 5V para activarse o entregan 5V de sus pines. Considérese el caso particular del pin ECHO del HCSR04. Para lidiar con este problema se recomienda, en caso de enviar salidas digitales al ESP32, emplear un divisor de tensión a partir de una resistencia de 1 KΩ y 2 KΩ. En cuanto a las salidas digitales de la ESP32 se recomienda el empleo de transistores para elevar los 3.3V a 5.

En otro apartado, el entorno de Mininet requiere de *Access Points* para realizar las conexiones. En este sentido se recomienda el uso de una interfaz virtual o el empleo de un enrutador. Ambas opciones permiten realizar un *punto virtual* que luego se emplea como AP hacia la red emulada. Este consejo es de vital importancia debido a que involucra la conexión exitosa de los ESP32 a la red.

El uso de los servomotores o motores DC a partir de la ESP32 es un tanto complicado pues se requieren elementos externos. Los elementos externos se refieren a fuentes de voltaje externas. Las fuentes de voltaje externas permiten el suministro suficiente de corriente hacia los actuadores como los servomotores o motores DC. En este mismo contexto, al usar un servomotor es necesario asegurarse de la forma de actuar que se tenga. En este proyecto se trabajó con servomotores que emplean instrucciones de giro a partir de ángulos. No obstante el proyecto tuvo etapas de estancamiento debido al empleo de servomotores que solo aceptaban velocidad y dirección de giro.

A la hora de implementar tablas de grupo sobre la red SDN se debe considerar que aquello implementado en un opeenswitch afectará también a aquellos que se comunican con aquel switch. Es decir la modificación de las tablas de flujo afecta a todos aquellos vecinos del enrutador que intercambian tráfico con el mismo. En tanto se recomienda mantener un enfoque bidireccional a la hora de implementar cambios en tablas de flujo o el empleo de tablas de flujo.

REFERENCIAS

- [1] Dejan Nedelkovski. *Arduino Radar Project*. HowToMechatronics website. URL: <https://howtomechatronics.com/projects/arduino-radar-project/>. 2015. URL: <https://howtomechatronics.com/projects/arduino-radar-project/> (visitado 07-08-2025).
- [2] Processing Foundation. *Processing: Download*. <https://processing.org/download/>. Versión estable actual (a mayo de 2025: 4.4.4). 2025.
- [3] Ryu Project. *Ryu SDN Framework Documentation*. <https://ryu.readthedocs.io/en/latest/>. Accedido: 2025-08-07. 2022.
- [4] Nael M. Radwan y Jim Alves-Foss. “MQTT in Focus: Understanding the Protocol and Its Recent Advancements”. En: *International Journal of Computer Science and Security (IJCSS)* 18.1 (2024), págs. 1-14. ISSN: 1985-1553. URL: <https://www.researchgate.net/publication/383282006>.
- [5] William Stallings. *Foundations of Modern Networking: SDN, NFV, QoE, IoT, and Cloud*. Boston, MA: Addison-Wesley Professional, 2016. ISBN: 9780134175393.

ANEXOS

El siguiente [enlace](#) corresponde al repositorio que presenta código, archivos, documentación y evidencia del funcionamiento de la red SDN.



Figura 23: Repositorio de prácticas