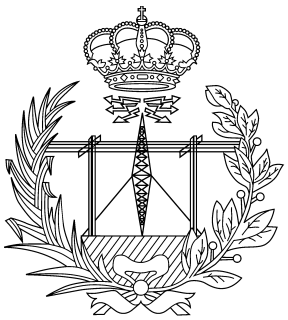
ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA



Trabajo Fin de Máster

DISEÑO E IMPLEMENTACIÓN DE ALGORITMOS ADAPTATIVOS DE GESTIÓN DE TRÁFICO EN REDES SDN

(Adaptive scheduling implementation and validation in SDN networks)

Para acceder al Título de

Máster Universitario en Ingeniería de Telecomunicación

Autor: Jorge Tielve Viejo

Junio- 2025

UNIVERSIDAD DE CANTABRIA

E.T.S. DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

CALIFICACIÓN DEL TRABAJO FIN DE MÁSTER

Realizado por: Jorge Tielve Viejo
Director del TFG: Luis Francisco Diez Fernández
Título: "Diseño e implementación de algoritmos adaptativos de gestión de tráfico en redes SDN"
Title: "Adaptive scheduling implementation and validation in SDN networks"

Presentado a examen el día: 16 de Junio de 2025

para acceder al Título de

GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Composición del tribunal:
Presidente (Apellidos, Nombre): Muñoz Gutiérrez, Luis
Secretario (Apellidos, Nombre): García Gutiérrez. Alberto Eloy
Vocal (Apellidos, Nombre): Miguel Diaz, Jose Ángel

Este Tribunal ha resulto otorgar la calificación de: .....

Fdo.: El Presidente

Fdo.: El Secretario

Fdo.: El Vocal

Fdo.: El Director del TFM
(solo si es distinto del Secretario)

V^o B^o del Subdirector

Trabajo Fin de Máster N^o
(a asignar por Secretaría)

Agradecimientos

Quiero expresar mi más sincero agradecimiento a mi familia y amigos, por su constante apoyo y ánimo durante todo el proceso de este proyecto. Agradezco especialmente a mi director de proyecto, Luis Francisco Díez, por su paciencia, por acompañarme y por brindarme su ayuda en todas las etapas del desarrollo. También quiero dar las gracias a mi compañera de trabajo, María, por escuchar mis ideas y aportaciones incluso en los momentos más inesperados durante nuestras jornadas laborales.

Resumen

Con la creciente demanda de servicios que requieren un mayor ancho de banda y alta disponibilidad de los recursos, aumenta la necesidad de una gestión más eficiente por parte de los administradores de red y proveedores de servicio que deberán enfrentar nuevos escenarios aprovechando la infraestructura de redes IP tradicionales. La RAN ha evolucionado hacia arquitecturas SDN, separando el plano de datos y el plano de control, y centralizando este último mediante el uso de controladores inteligentes. Los centros de datos y las infraestructuras *cloud*, permiten el despliegue de controladores SDN con capacidad para gestionar cientos de dispositivos que operan en la integración del *fronthaul* y del *backhaul* dentro de la red 5G. Existen varios modelos de controladores programables y con capacidad para desarrollar e implementar algoritmos de *scheduling* con el objetivo de optimizar el tráfico de red. En este proyecto se propone el uso de Ryu, un *framework* de *python*, con las ventajas en programabilidad que esto ofrece, para la implementación de un algoritmo planificador y demostrar su versatilidad y las ventajas que ofrece el uso de estas herramientas en un entorno SDN que gestiona diferentes flujos de tráfico de red a ráfagas en las redes 5G.

Palabras clave: SDN; RAN; Ryu; *python*; *fronthaul*; *backhaul*; 5G; controlador de red; *scheduler*...

Abstract

With the growing demand for services requiring higher bandwidth and greater resource availability, there is an increasing need for more efficient network management by administrators and service providers. These stakeholders must address new challenges while leveraging traditional IP network infrastructures. The Radio Access Network (RAN) has evolved towards Software-Defined Networking (SDN) architectures, separating the data and control planes and centralizing the latter through the use of intelligent controllers. Data centers and cloud infrastructures enable the deployment of SDN controllers capable of managing hundreds of devices operating within the integration of fronthaul and backhaul networks in 5G environments. Several models of programmable controllers are available, allowing the development and deployment of scheduling algorithms aimed at optimizing network traffic. This project proposes the use of Ryu, a Python-based framework, leveraging its programmability to implement a scheduling algorithm and demonstrate both its versatility and the advantages of using such tools in an SDN-managed environment.

Keywords: SDN; RAN; Ryu; *python*; *fronthaul*; *backhaul*; 5G; controlador de red; *scheduler...*

Índice general

Índice de figuras

Índice de tablas

Índice de acrónimos

3GPP	3rd Generation Partnership Project.
ACL	Access Control List.
API	Application Programming Interface.
AS	Autonomous System.
ASIC	Application-Specific Integrated Circuit.
BBU	Baseband Unit.
BGP	Border Gateway Protocol.
BH	Backhaul.
CPRI	Common Public Radio Interface.
C-RAN	Cloud-RAN.
CU	Centralized Unit.
DDoS	Distributed Denial-of-Service.
DL	Downlink.
DoS	Denial of Service.
DPDK	Data Plane Development Kit.
D-RoF	Digital Radio over Fiber.
DRR	Deficit Round Robin.
DSCP	Differentiated Services Code Point.
DU	Distributed Unit.
eCPRI	enhanced Common Public Radio Interface.
EMC	Exact Match Cache.
FCFS	First-Come, First-Serve.
FH	Fronthaul.
FIFO	First-in, First-out.

FPGA	Field-Programmable Gate Array.
FQ	Fair Queuing.
GNS3	Graphical Network Simulator-3.
GPU	Graphics Processing Unit.
HARQ	Hybrid Automatic Repeat reQuest.
IETF	Internet Engineering Task Force.
ISP	Internet Service Provider.
LTE	Long Term Evolution.
MFC	MegaFlow Cache.
MH	Midhaul.
MIMO	Multiple-input Multiple-output.
MPLS	Multi-Protocol Label Switching.
NFV	Network Functions Virtualization.
NOS	Network Operating System.
NR	New Radio.
OFT	OpenFlow Table.
ONF	Open Networking Foundation.
O-RAN	Open Radio Access Network.
OVS	Open vSwitch.
PMD	Poll Mode Driver.
PS	Head of Line Processor Sharing.
QoE	Quality of Experience.
QoS	Quality of Service.
RAM	Random Access Memory.
RAN	Radio Access Network.
RFC	Request for Comments.
RIC	RAN Intelligent Controllers.
RoF	Radio over Fiber.
RRH	Remote Radio Head.
RSS	Receive Side Scaling.
RU	Radio Unit.

SDN	Software Defined Networks.
SJF	Shortest Job First.
TC	Traffic Control.
TCAM	Ternary Content Addressable Memory.
TCP	Transmission Control Protocol.
TSS	Tuple Space Search.
UDN	Ultra-Dense Network.
UDP	User Datagram Protocol.
UL	Uplink.
VLAN	Virtual Local Area Network.
WDM	Wavelength Division Multiplexing.
WFQ	Weighted Fair Queuing.

Capítulo 1

Introducción

En los últimos años, las redes celulares han visto un aumento significativo en el tráfico de datos, impulsado principalmente por la creciente adopción de dispositivos inteligentes como *smartphones* y *tablets*. Se espera que esta tendencia continúe, y la expansión de las comunicaciones máquina a máquina y los servicios en la nube en tiempo real está generando nuevos desafíos. Para hacer frente a estas demandas, las redes de próxima generación deberán ofrecer un rendimiento mayor, con hasta 1000 veces más capacidad, 100 veces más velocidad de datos y una latencia de tan solo 1 ms en comparación con los sistemas 4G Long Term Evolution (LTE). Las tecnologías clave que permitirán alcanzar estos niveles incluyen Multiple-input Multiple-output (MIMO) masivo, Ultra-Dense Network (UDN) y el uso de espectro de alta frecuencia [1].

El aumento de prestaciones conllevará una mayor complejidad de las redes celulares con nuevos sistemas de comunicaciones inalámbricos de nueva generación sustentados en tecnologías heterogéneas y que trabajan en diversas bandas de frecuencia. La llegada del 5G supuso el replanteamiento de la arquitectura de las redes para dar soporte al acceso vía radiofrecuencia a múltiples servicios y la necesidad de gestionar el consiguiente incremento de ancho de banda demandado. Esto implica un incremento en las inversiones de capital y operativas para los operadores de redes, quienes deberán actualizar y mantener continuamente su infraestructura para adaptarse a las nuevas tendencias del mercado, avances tecnológicos y las crecientes demandas de los clientes [2].

Uno de los segmentos de red que se verá más afectado es el de acceso, o Radio Access Network (RAN). Este segmento ha estado tradicionalmente compuesto por unidades monolíticas, soluciones integrales que implementan todas las capas de la pila de protocolos celulares. Estas soluciones, proporcionadas por un número reducido de proveedores, son percibidas por los operadores como cajas negras. La dependencia de estas cajas negras ha generado varias limitaciones: la RAN tiene una reconfigurabilidad limitada, lo que impide ajustar de forma óptima el soporte a diferentes implementaciones y perfiles de tráfico; la coordinación entre los nodos de la red es escasa, dificultando la optimización conjunta y el control de los componentes de la RAN; y los operadores quedan sujetos al bloqueo de proveedores (*vendor lock-in*), con pocas opciones para integrar equipos de RAN de diferentes fabricantes. En este contexto, gestionar de manera óptima los recursos radioeléctricos y utilizar el espectro de forma eficiente mediante la adaptación en tiempo real se convierte en un reto considerable [2].

La gestión y optimización de estas redes requiere soluciones como O-RAN. La Alianza O-RAN, creada

en 2018, tiene como objetivo la desagregación de la RAN en tecnologías 4G LTE y 5G New Radio (NR), tal como han sido definidos por 3rd Generation Partnership Project (3GPP). En concreto, la Alianza O-RAN adopta y amplía la división NR 7.2 de 3GPP para estaciones base, desglosando sus funciones en tres unidades funcionales: Centralized Unit (CU), Distributed Unit (DU) y Radio Unit (RU). Además, la arquitectura O-RAN define la interconexión entre estas unidades y controladores de red mediante interfaces abiertas, permitiendo la transmisión de telemetría desde la RAN y la implementación de acciones y políticas de control hacia ella. La arquitectura de O-RAN incluye dos RAN Intelligent Controllers (RIC) que gestionan y controlan la red en diferentes escalas de tiempo: casi en tiempo real (entre 10 ms y 1 s) y en no tiempo real (más de 1 s). Finalmente, la Alianza O-RAN está desarrollando una plataforma de virtualización para la RAN y ampliando la definición de interfaces 3GPP y enhanced Common Public Radio Interface (eCPRI) para conectar los nodos de la RAN [2].

Para el despliegue de las entidades definidas en la arquitectura O-RAN se hace uso de técnicas de virtualización, dando lugar al concepto de vRAN o Cloud-RAN (C-RAN), donde las redes ya no dependen de un hardware propietario particular, lo que hace la gestión y optimización mucho más eficiente y sencilla. *Cloud* RAN es una evolución consecuente de las vRAN por todas las ventajas que aporta la infraestructura física y virtual de la nube. Cualquier proveedor *cloud* te ofrece infraestructura física de servidores donde se puede desplegar cualquier tipo de infraestructura virtual dependiendo de los requisitos de carga y latencia, además de una arquitectura donde las funciones de red se pueden ejecutar como microservicios en contenedores o kubernetes, lo que lo convierte en una estructura simplificada, automatizada y escalable.

Como se ha mencionado anteriormente, la arquitectura O-RAN requiere del rediseño y reestructuración de las redes que comunican los elementos virtualizados: red de Fronthaul (FH) entre la RU y DU; red de Midhaul (MH) entre la DU y CU; y red de Backhaul (BH) desde la CU al núcleo de la red del operador. El *fronthaul* es un componente esencial para las redes 5G, pero presenta unos requisitos muy altos. Requiere un gran ancho de banda, baja latencia y una estricta sincronización en la red de transporte, y tradicionalmente solo ha admitido una topología punto a punto, lo que restringe su uso en muchos escenarios de 5G. La arquitectura O-RAN impone un *fronthaul* capaz de soportar topologías más flexibles y permitir la incorporación de diferentes esquemas de división funcional en la red de acceso, lo que ayudará a satisfacer los exigentes requisitos de ancho de banda y latencia en redes ultra densas, arquitecturas de desacoplamiento de control/datos y comunicaciones sensibles al retardo. Asimismo, se podría habilitar de manera más eficiente la cooperación masiva y las redes centradas en dispositivos mediante un *fronthaul* que permita enlaces punto a multipunto. Se identifican tres requisitos clave para el diseño del *fronthaul*: la capacidad de manejar varios tipos de tráfico, el soporte de topologías lógicas diversas y la garantía de latencias diferenciadas [1].

La tecnología de transmisión física predominante para el *fronthaul* (FH) clásico es Digital Radio over Fiber (D-RoF). Aunque existen otras tecnologías competidoras, como Radio over Fiber (RoF), el mayor rango de transporte en arquitecturas C-RAN otorga a RoF ventajas significativas, principalmente por su baja degradación de la señal. Se han desarrollado diversas especificaciones para garantizar la interoperabilidad entre productos FH de diferentes fabricantes. Un ejemplo de esto es la especificación de la Common Public Radio Interface (CPRI), que abarca las capas 1 y 2 del FH y define aspectos como la topología física, las tasas de datos de línea y el formato de enmarcado, entre otros. [1].

Aunque el FH clásico ha sido ampliamente adoptado en estaciones base distribuidas y C-RAN, enfrentará

serios desafíos frente a las redes 5G. Estos desafíos incluyen:

1. **Requisitos masivos de ancho de banda de FH:** Los requisitos de ancho de banda en un enlace FH clásico son proporcionales al producto del ancho de banda de radio, el número de antenas y la resolución de cuantificación. Por ejemplo, una eNodeB típica de 4G LTE con 20 MHz y 8 antenas necesita aproximadamente 10 Gb/s de ancho de banda en enlace ascendente (Uplink (UL)) o descendente (Downlink (DL)). En las redes 5G, se espera que la densidad de antenas y el ancho de banda sean aún mayores, lo que aumentará significativamente la demanda de ancho de banda en FH. A pesar de que las técnicas de compresión más avanzadas solo alcanzan una tasa de compresión de hasta tres veces, la opción más directa para el transporte es el uso de fibra oscura, donde un solo núcleo de fibra puede transportar un enlace FH. Sin embargo, esto consume enormes recursos de fibra, lo que lo hace viable solo para operadores con abundante capacidad de fibra o en escenarios donde el despliegue de fibra es económico. Otra opción es multiplexar enlaces FH en un único núcleo de fibra mediante Wavelength Division Multiplexing (WDM), aunque los módulos WDM son considerablemente más costosos que los módulos estándar. [1].

2. **Restricciones estrictas de latencia:** El procesamiento de señales inalámbricas requiere cumplir con restricciones de latencia muy estrictas. Por ejemplo, en LTE, Hybrid Automatic Repeat reQuest (HARQ) tiene un límite de 3 ms para la decodificación de cada subtrama de radio. Como estas subtramas deben transportarse desde las Remote Radio Head (RRH) a las unidades de banda base (Baseband Unit (BBU)) antes de ser procesadas, la latencia del transporte FH debe ajustarse a este límite. Excluyendo el tiempo necesario para el procesamiento de la señal (alrededor de 2.5 ms), solo quedan entre 300 y 500 μs para el transporte FH, lo que descarta tecnologías de conmutación que introduzcan latencias excesivas. Además, en ciertos escenarios de 5G con comunicaciones de submilisegundos, las restricciones de latencia serán aún más estrictas, lo que podría prohibir el uso de transporte a larga distancia [1].

3. **Requisitos estrictos de sincronización:** La sincronización es fundamental en los sistemas de comunicación por radio. Muchas RRH no pueden generar un reloj preciso por sí mismas, ya sea porque los receptores GPS son demasiado costosos o porque la señal satelital está bloqueada en entornos interiores. Por ello, FH debe proporcionar la información de sincronización desde las BBU hacia las RRH. En los enlaces CPRI, la sincronización se transporta mediante los bordes de pulso de la señal, pero la infraestructura de red subyacente puede introducir fluctuaciones que degraden el rendimiento de la comunicación. La necesidad de una sincronización precisa será aún más crítica en las redes 5G, donde la cooperación masiva entre nodos de acceso es clave. Si los nodos cooperantes presentan un desplazamiento de frecuencia, las señales transmitidas se superpondrán en el equipo del usuario, lo que afectará negativamente la formación de haces y degradará el rendimiento [1].

La integración de todos los elementos dentro de la arquitectura 5G se puede ver en la Figura 1.1, dónde se incluyen los principales elementos de la red de acceso. Se muestran además las pilas de los protocolos que intervienen en las diferentes redes de interconexión áreas. Para dar respuesta a estos requisitos se está desplegando una red de muy alta capacidad, lo que conlleva un alto coste. A esto se une que las entidades que constituyen la estación base se encuentran virtualizadas y que cada vez se centralizan más, aumentando la distancia física que cubre la red FH. Con todo ello, el tráfico de FH (muestras IQ en banda

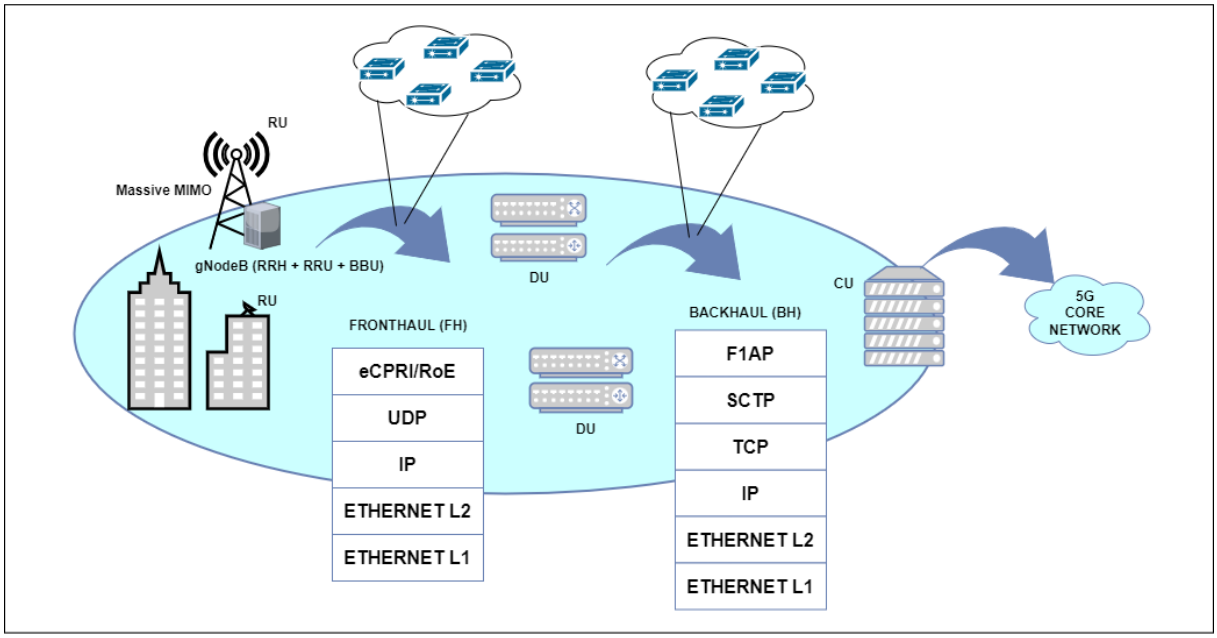


Figura 1.1: Escenario O-RAN 5G

base) coexiste con tráficos de otros segmentos de red (MH o BH), lo que hace necesario que los nodos de reenvío (switches o routers) gestionen el tráfico adecuadamente [3]. En la actualidad, esta gestión se realiza de forma estática mediante el marcado de tráfico con Differentiated Services Code Point (DSCP) y las colas de prioridad absoluta [4]. Sin embargo, este enfoque puede perjudicar de forma relevante al resto de flujos de tráfico.

1.1. Planteamiento y Objetivos

Basado en la introducción, este proyecto tiene como objetivo general la implementación y evaluación de técnicas adaptativas de gestión de QoS para tráficos heterogéneos. Estas técnicas se desarrollarán haciendo uso de técnicas SDN, que otorgan una gran adaptabilidad y facilidad de desarrollo. Para ello se empleará OVS como *switch* virtual (o *vSwitch*) que soporta una larga lista de protocolos, entre ellos *Openflow* que será el de mayor interés a lo largo del proyecto.

El *switch* virtual se configurará mediante un controlador Openflow. El controlador elegido para el desarrollo del proyecto es Ryu, un proyecto *open-source*, que define un framework para *component-based software defined networking*. Ryu permite crear una aplicación que gestione los dispositivos de la red de la manera que se desee, por lo que con un script de *python* se podrá implementar un controlador capaz de configurar el *vSwitch* usando el protocolo OVSDB y de manejar eventos Openflow.

El objetivo del proyecto es establecer reglas de QoS dinámicamente sobre un interfaz del *vSwitch* para impedir que el ancho de banda de algún servicio se haga con toda la capacidad disponible en un enlace compartido para el tráfico agregado de varios servicios. A continuación se muestra un ejemplo visual sencillo que ejemplifica el funcionamiento de la aplicación de QoS sobre OVS, y que se desarrollará con más detalle en el Capítulo 4.

El escenario de la Figura 1.2, se tiene un entorno virtual con un enlace agregado limitado a 50 *Mbps*. Este enlace compartido da servicio a dos tasas de tráfico de red, uno mantiene un servicio con tráfico

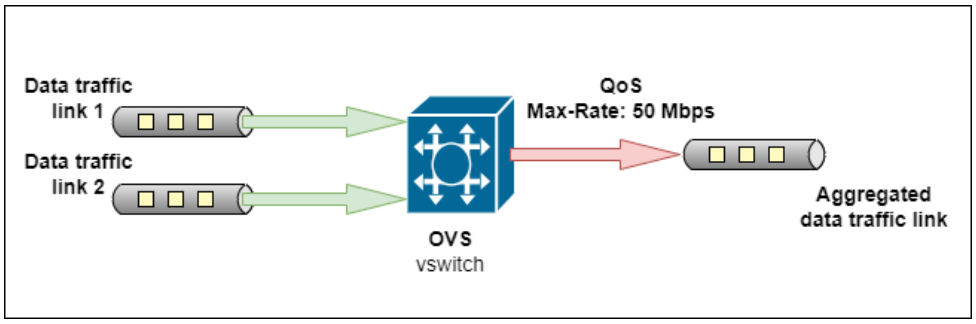
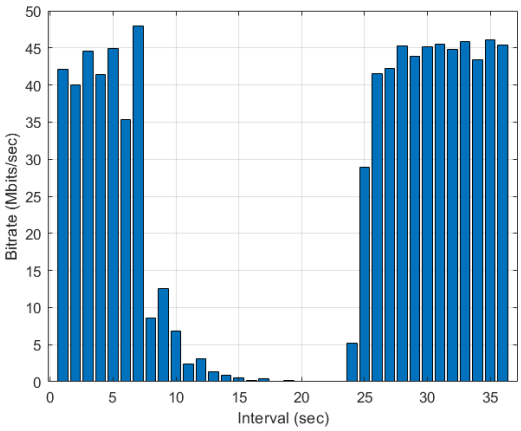
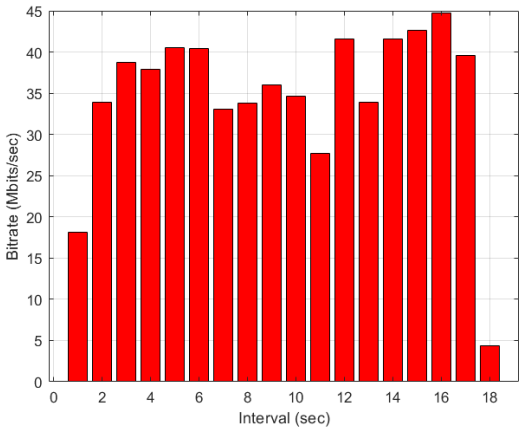


Figura 1.2: Escenario introducción



(a) Tráfico TCP con origen en *Data Traffic Link 1*



(b) Tráfico UDP con origen en *Data Traffic Link 2*

Figura 1.3: Tasa de servicio sin QoS sobre *Aggregated Data Traffic Link*

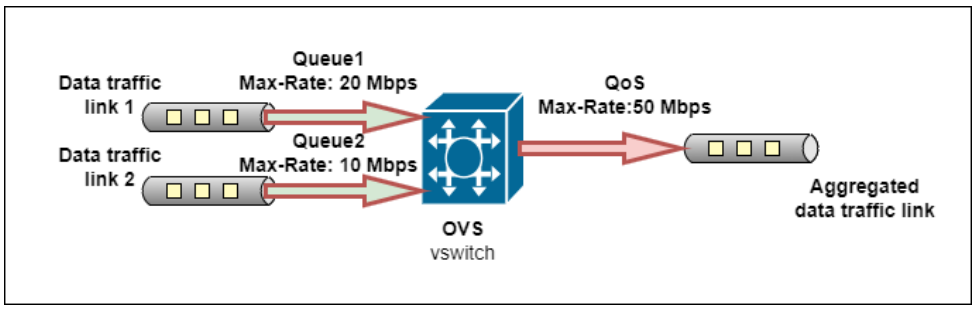
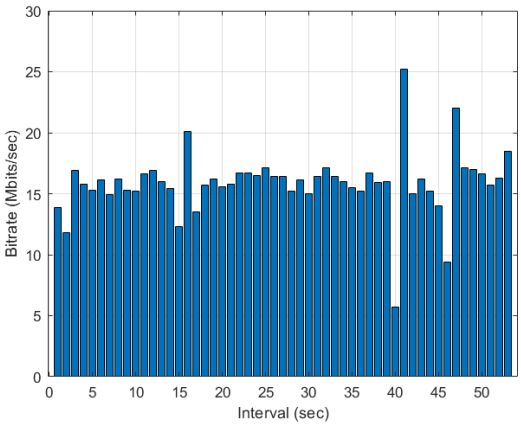


Figura 1.4: Escenario introducción. Creación de colas

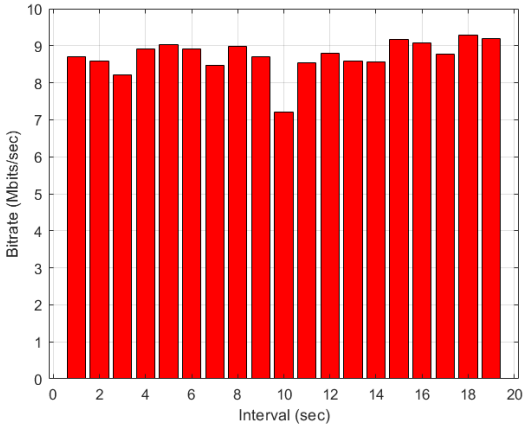
Transmission Control Protocol (TCP) y el otro con tráfico User Datagram Protocol (UDP). Si ambos intentan transmitir a la tasa limitada, se presenta el problema.

Se hace un análisis del comportamiento simultáneo del tráfico en la Figura 1.3, y como se puede comprobar, el servicio UDP consume todo el ancho de banda agregado disponible. Para evitar esta situación, se configura el *switch* aplicando reglas de QoS sobre el interfaz que absorbe el tráfico agregado. Se configura un QoS con una tasa máxima de 50 *Mbps* y se crean dos colas asociadas que limitarán a 20 *Mbps* para el tráfico TCP y a 10 *Mbps* para el tráfico UDP tal y como se muestra en el escenario de la Figura 1.4.

Ambos servicios tratarán de transmitir a 50 Mbits/sec de la misma manera que en el experimento anterior; sin embargo, las colas asociadas limitan ambos flujos, evitando que uno de ellos consuma todo el ancho de banda agregado y permitiendo mantener a los usuarios un ancho de banda mínimo que pueda mantener



(a) Tráfico TCP con origen en *Data Traffic Link 1*



(b) Tráfico UDP con origen en *Data Traffic Link 2*

Figura 1.5: Tasa de servicio con QoS sobre *Aggregated Data Traffic Link*

el servicio. El efecto de las colas de QoS se puede observar en la Figura 1.5.

A la vista de los resultados, se puede comprobar que el uso de estas herramientas permite gestionar el ancho de banda y distribuir la capacidad agregada entre distintos servicios de forma eficaz. Esta primera demostración del funcionamiento y los beneficios de aplicar reglas de QoS sirve como base para el desarrollo principal del proyecto, que consiste en implementar reglas de QoS dinámicas. Estas reglas se adaptarán en tiempo real al tráfico de red, asignando el ancho de banda disponible según las necesidades de cada servicio y tomando decisiones basadas en la ocupación del *buffer* de los interfaces.

Los objetivos concretos que se plantean en este proyecto son los siguientes:

1. Introducir la problemática acerca de la necesidad de soluciones SDN para la gestión de la red ante la creciente demanda y como afecta a la reestructuración de la misma. Dar a conocer las iniciativas *open-source* que ofrecen soluciones en este ámbito.

2. Crear un escenario próximo a un entorno virtualizado como podría ser una solución dentro de la RAN situando el controlador SDN como el encargado de la gestión de los flujos de red.

3. Implementación, en un controlador SDN, de algoritmos para gestionar la QoS dinámicamente en función de tres métricas: la ocupación de los *buffers* interfaces, el retardo acumulado de los paquetes en los mismos y el retardo máximo.

4. Evaluar y comparar el rendimiento de las técnicas desarrolladas.

1.2. Estructura del documento

En este breve apartado, se pretende explicar escuetamente el documento y cómo se van a desarrollar los temas propuestos en los objetivos. En primer lugar, en el Capítulo 2 se aportará información acerca del estado del arte de las herramientas y el origen de las tecnologías sobre las que se apoya el proyecto, así como la importancia de las mismas para cumplir con los objetivos. Toda la información está referenciada a los artículos indicados en la bibliografía, que se han usado para documentar correctamente los fundamentos del proyecto.

En el Capítulo 3, se expone la implementación del controlador, cuáles son sus limitaciones y la justificación de los parámetros que se emplean para su evaluación. Posteriormente, en el Capítulo 4, se evalúa y compara el funcionamiento de los algoritmos implementados. Por último, en el Capítulo 5 se desarrollan las conclusiones del proyecto.

Capítulo 2

Estado del Arte

En este capítulo se presentan los conceptos técnicos más relevantes que se utilizarán a lo largo del trabajo. Concretamente, en la Sección 2.1 se describirá el paradigma SDN de forma general. Posteriormente, en las Secciones 2.2 y 2.3 se describirán los dos protocolos SDN que se utilizarán en el trabajo.

2.1. Redes definidas por software (SDN)

Las tecnologías de red se han consolidado como un elemento esencial en prácticamente todas las actividades humanas. El número de dispositivos conectados, así como la cantidad de tráfico presente en Internet, está creciendo exponencialmente. No obstante, el desarrollo de nuevos protocolos y servicios, tales como movilidad, VoIP o servicios con requisitos de QoS (calidad de servicio), se ha visto limitado por la fuerte dependencia entre el *hardware* especializado para el reenvío de paquetes y los sistemas operativos que gestionan cientos de protocolos estáticos (algunos de ellos propietarios), lo cual restringe las opciones de configuración disponibles para los administradores de red [5].

A pesar de su adopción generalizada, las redes IP tradicionales son complejas y difíciles de gestionar. Esto se debe a varios factores, como la necesidad de aplicar y gestionar políticas de red en una amplia gama de dispositivos, que incluyen *routers*, *switches*, *firewalls* y otros componentes de infraestructura. Estas políticas pueden abarcar desde reglas de seguridad y acceso, hasta configuraciones de QoS y gestión del tráfico, lo que dificulta su implementación de manera coherente en todos los dispositivos. Además, se requiere una alta conectividad y disponibilidad, junto con tolerancia a fallos, un manejo eficiente del tráfico y una configuración adecuada del encaminamiento. La superposición de múltiples mecanismos en diferentes capas de la red no solo complica aún más su administración, sino que también expone la red a mayores vulnerabilidades de seguridad. Los protocolos de transporte y el control distribuido que se utilizan en las redes actuales también introducen desafíos operativos significativos en estas infraestructuras tradicionales [6].

Las redes definidas por *software*, o SDN, constituyen una innovación desde el punto de vista de arquitectura que propone la separación del plano de control y del plano de datos, permitiendo así su evolución y desarrollo de manera independiente. A principios de la década de 2000, el uso de *Ethernet* como medio de comunicación generalizado en redes, junto con iniciativas para integrar capacidades en tiempo real en estándares ampliamente adoptados, preparó el terreno para las innovaciones que vemos hoy. Las SDN buscan establecer estándares universales aplicables a diversos sectores, como el control de maquinaria, la

transmisión de audio y vídeo, o las redes automotrices. Sin embargo, cuando estos conceptos surgieron en el entorno de los centros de datos, no lograron mostrar todo su potencial [7].

Ethane fue una de las primeras iniciativas dentro del campo de las SDN. Este modelo, propuesto para redes empresariales, utilizaba una arquitectura de control centralizada. Sin embargo, su implementación requería el despliegue de conmutadores personalizados (OpenWrt, NetFPGA, Linux) para soportar el protocolo *Ethane* [5]. El proceso de implementación de nuevas soluciones, desde su diseño, simulación, pruebas, publicación en un estándar, hasta su instalación en equipos de red, puede llevar varios años. Por ello, resulta evidente que las redes de nueva generación deben superar la rigidez de las arquitecturas tradicionales [5].

En la actualidad, una de las soluciones SDN más relevantes es OpenFlow, un protocolo y arquitectura abierta, que fue desarrollada inicialmente como una forma de permitir a los investigadores realizar experimentos en redes heterogéneas sin afectar el tráfico de los usuarios reales. OpenFlow tiene como objetivo proporcionar mayores opciones de configuración en el plano de datos de los conmutadores. La especificación de OpenFlow define las reglas de comunicación entre el plano de datos y un plano de control centralizado, y permite controlar toda la red a través de interfaces de aplicación, o Application Programming Interface (API), desarrolladas por los usuarios. La iniciativa Open Networking Foundation (ONF) agrupa a cerca de 90 empresas y se dedica a la publicación, promoción y adopción de la especificación OpenFlow [5], entre otras soluciones SDN.

Como se ha comentado, las soluciones SDN ofrecen una mayor adaptabilidad para cubrir los requisitos de los nuevos servicios de la era digital y de la demanda de grandes anchos de banda. Para ello implementan un control de la red programable e independiente de la infraestructura subyacente, de manera que los distintos servicios de red sean abstractos para él. La arquitectura SDN se basa en cinco conceptos:

1. **Separación de los planos de control y datos**: este es un principio fundamental que permite la evolución y desarrollo de los mismos de manera independiente. Además, el plano de control centralizado, proporciona una visión global de la red.

2. **Programabilidad de la red**: los responsables de TI pueden ajustar dinámicamente las asignaciones de red para satisfacer los requisitos de aplicaciones y usuarios finales específicos.

3. **Flexibilidad y agilidad de abstracción de flujo**: relacionado con el anterior punto, la programabilidad proporciona una flexibilidad que permite garantizar que las demandas de los usuarios sean atendidas de manera eficiente y precisa.

4. **Neutralidad del proveedor**: con el desarrollo y estandarización de protocolos y *hardware* no propietario que los soporta permite la independencia del fabricante y de Internet Service Provider (ISP).

5. **Control centralizado y capacidad de administración**:se definen canales de comunicación entre el plano de control y de datos que permite a los administradores configurar y gestionar los recursos de la red mediante APIs accesibles, ya sean propietarias o de código abierto.

El estrecho acoplamiento entre los dispositivos de red ha ralentizado el avance de las tecnologías de red. A medida que aumentan los requisitos de latencia y ancho de banda, el proceso de transmisión o reenvío

de paquetes ha pasado a depender cada vez más del *hardware* especializado. Esto ha llevado a que la gestión de la red se delegue progresivamente a herramientas de software, ya que los servidores ofrecen mayor capacidad de procesamiento y memoria que los dispositivos de red individuales. Esta necesidad de optimizar la gestión ha impulsado el desarrollo del protocolo OpenFlow, diseñado para facilitar el control y la administración de redes. OpenFlow se ha convertido en el estándar más adoptado en la comunidad investigadora y ha servido como base para numerosas iniciativas enfocadas en mejorar la gestión de las redes actuales [6].

Dentro del concepto SDN, el protocolo OpenFlow elimina las limitaciones impuestas por los protocolos estáticos, abre la posibilidad de una innovación rápida y ha impulsado a la comunidad investigadora a desarrollar nuevos paradigmas en las tecnologías de red. Algunos ejemplos de esta evolución incluyen: virtualización, lenguajes de programación de redes de alto nivel, y optimización de aplicaciones para Quality of Experience (QoE). Sin embargo, nuevos desafíos y preguntas sin resolver han surgido a medida que se intenta implementar estas ideas en redes de producción [5]:

1. **Modelado y Rendimiento**: La implementación de SDN/OpenFlow requiere la estimación del número de controladores necesarios para una topología determinada y la localización óptima de estos controladores. Este es un problema complejo de resolver. SDN establece la separación de los planos de control y de datos, pero no necesariamente prescribe la existencia de un único controlador. El plano de control SDN puede estar compuesto por un único controlador, un conjunto de controladores en una topología jerárquica o incluso por una topología de controladores dinámica. El número y la ubicación de los controladores dependen de los límites de reacción deseados, la elección de métricas y la topología de la red en sí [5].

   Otro factor importante es la selección de un Network Operating System (NOS) apropiado y la evaluación de su rendimiento. En otras palabras, se debe considerar la rapidez con la que el controlador responde a las solicitudes del plano de datos y cuántas solicitudes de datos puede manejar por segundo. Además, es necesario establecer un equilibrio entre el alto rendimiento y la productividad de los desarrolladores (un lenguaje amigable para el desarrollador). El rendimiento del controlador depende directamente del lenguaje de programación (C++, Java, *python*) y del entorno de desarrollo utilizado [5].

2. **Resiliencia y Recuperación**: Uno de los problemas relacionados con el control centralizado propuesto por OpenFlow es la vulnerabilidad de la red. Un fallo en el controlador puede comprometer negativamente la resiliencia de toda la red. Además, OpenFlow permite la configuración de uno o más controladores de respaldo, pero no proporciona un mecanismo de coordinación entre ellos[5].

   El componente *CPRecovery* es un mecanismo primario-secundario que permite la replicación entre el controlador SDN primario y el secundario en dos fases: la fase de replicación y la fase de recuperación. La fase de replicación actúa durante el funcionamiento normal del controlador y envía actualizaciones periódicas al controlador de respaldo. La fase de recuperación actúa durante un estado de fallo del controlador primario y arranca el controlador de respaldo como controlador principal. El estado de fallo puede ser provocado por la interrupción abrupta de un proceso del NOS, un error inesperado de la aplicación (API) o un ataque Distributed Denial-of-Service (DDoS). Los resultados experimentales muestran un tiempo de recuperación (desde el estado de fallo hasta la conexión de respaldo) de 800 ms [5].

3. **Seguridad**: La arquitectura actual de Internet presenta grandes problemas para soportar, verificar y hacer cumplir las propiedades de seguridad. Los sistemas de seguridad tradicionales se basan en proteger los *hosts* (por ejemplo, con antivirus) o en instalar dispositivos de red especializados (middleboxes) para detectar anomalías en la red. Sin embargo, estas soluciones se vuelven ineficaces cuando el *hosts* está comprometido (vulnerabilidades de día cero) o requieren actualizaciones constantes y esfuerzo por parte del administrador de red para, por ejemplo, seleccionar el tráfico que debe ser filtrado. En este punto, SDN puede contribuir a mejorar y desarrollar nuevos modelos de seguridad [5].

   Pedigree es un sistema de seguridad basado en SDN para redes empresariales. Este modelo utiliza OpenFlow y una función simple del núcleo del sistema operativo en los *hosts* para ayudar al controlador de red a determinar el origen del tráfico de red. Una vez validada la conexión, el controlador habilita y establece la ruta de red en los conmutadores. Este sistema está diseñado con dos componentes: el etiquetador (*tagger*) y el árbitro (*arbiter*). El etiquetador monitoriza todos los eventos en el sistema operativo y crea una etiqueta para cada proceso. Cuando un proceso requiere enviar datos a través de la red, se envía un flujo de etiquetas (*tagstream*) al controlador. El controlador cuenta con una función de árbitro que verifica las etiquetas, las valida y ordena la instalación de las tablas de flujo correspondientes en los conmutadores. Si el árbitro detecta un intento de conexión no autorizado, el controlador ordena el bloqueo del flujo de este enlace. Aunque estos procesos generan una carga adicional en el *hosts* y en la red, la sobrecarga es comparable a la ejecución de un software antivirus [5].

4. **Convergencia y Gestión**: La arquitectura SDN-OpenFlow se desarrolló originalmente para redes de campus empresariales. Sin embargo, al intentar extender esta arquitectura a redes más grandes, se deben abordar ciertos problemas, como el enrutamiento entre dominios (*Interdomain Routing*) [5].

   Hoy en día, el protocolo más implementado para la comunicación entre Autonomous System (AS) es Border Gateway Protocol (BGP). BGP es un paradigma de reenvío basado en el destino. En otras palabras, la información de enrutamiento entre Sistemas Autónomos (AS) depende de la dirección de destino en el encabezado del paquete IP. Un AS intercambia información y controla el flujo de tráfico solo con sus AS vecinos directos. Esto dificulta el control de los flujos de tráfico a lo largo de toda una ruta o en una parte remota de la red. La tarea consiste en integrar la arquitectura SDN para interconectar diferentes AS [5].

   Otro desafío a considerar es la integración con redes heredadas, es decir, redes sin capacidad para OpenFlow (como la infraestructura actual de Internet). Aunque es posible actualizar o incluso reemplazar el equipamiento para soportar OpenFlow, este cambio implica una costosa reingeniería de la red. Por esta razón, es necesaria la coordinación entre el nuevo equipamiento SDN y la infraestructura heredada [5].

2.1.1. Arquitectura SDN

La arquitectura de las SDN requiere de un controlador conectado a todos los dispositivos de reenvío de paquetes (capa 2 y capa 3) en la medida en que estén abiertos y tengan interfaces programables. La API necesaria para la comunicación entre el controlador y el resto de dispositivos gestionados por el mismo se puede sostener sobre diferentes protocolos, entre los que se encuentra OpenFlow.

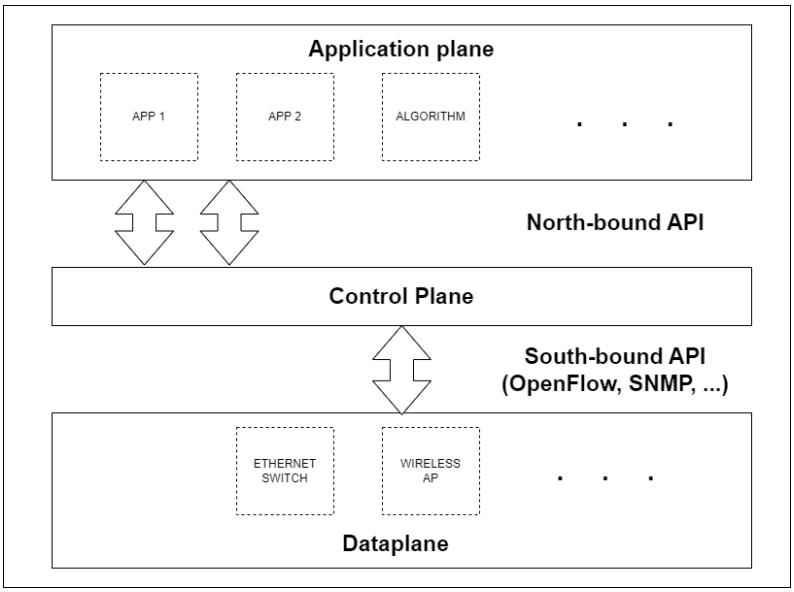


Figura 2.1: Arquitectura SDN [7]

Una SDN basada en OpenFlow generalmente tiene tres capas: aplicación, control e infraestructura. Los conmutadores y enrutadores son ejemplos de *hardware* de red de la capa de infraestructura que reenvían paquetes en respuesta a las instrucciones de la capa de control. Un elemento de la capa de control es el controlador SDN, que supervisa los dispositivos de red e indica las reglas de la red. Las aplicaciones de red poseen una visión abstracta de la red e indican al controlador los requisitos extremo a extremo en función de los servicios soportados. En la Figura 2.1 se muestra la arquitectura de SDN [8].

Las redesSDN se presentan, por lo tanto, como una opción prometedora para el desarrollo de las redes del futuro, ya que al cambiar la arquitectura y la gestión de las redes, se abren nuevas posibilidades en diversos ámbitos a nivel de aplicaciones.

- **Capa de Infraestructura**: Los elementos físicos de la red, como conmutadores, enrutadores y puntos de acceso, están incluidos en la capa de infraestructura, que es la capa más baja de una red. Estos componentes reenvían paquetes de acuerdo con sus configuraciones y tablas de enrutamiento. En una SDN basada en OpenFlow, estos dispositivos están acoplados con un conmutador o agente OpenFlow, que se comunica con el controlador SDN para recibir instrucciones de enrutamiento de paquetes [8].

- **Capa de Control**: La capa de control, que es la segunda capa, juega un papel crucial en el control de la red y la aplicación de políticas de red. La capa incluye el controlador SDN, una entidad central basada en software que se comunica con los dispositivos de red a través del protocolo OpenFlow. Su principal responsabilidad es recopilar y analizar la información del estado de la red, de modo que se puedan tomar decisiones sobre la mejor manera de configurar los dispositivos de red para cumplir con las políticas de red deseadas. La capa de control, que actúa como el centro de gestión principal de la red, hace posible una administración más efectiva y eficiente de los recursos de la red [8].

- **Capa de Aplicación**: La capa de aplicación es la capa superior. Incluye una serie de aplicaciones que están diseñadas para interactuar con los dispositivos de red y ejecutarse en el controlador SDN con el fin de proporcionar servicios de red a los usuarios finales [8].

La visión para las redes del futuro se centra en crear una infraestructura que pueda configurarse de manera dinámica y permita la interoperabilidad entre diferentes aplicaciones, sobre una infraestructura de comunicación compartida. Se espera que las redes industriales enfrenten una creciente heterogeneidad debido a la incorporación de tecnologías de comunicación cada vez más complejas. Además, deberán gestionar un número creciente de dispositivos impulsado por la mayor conectividad entre productos, máquinas y personas, y soportar comunicaciones *ad hoc* que resulten de conexiones temporales entre estos actores [7].

2.1.2. *Schedulers*: algoritmos de encolamiento

En la actualidad existen numerosos algoritmos de planificación (*schedulers*) que abordan problemas de gestión de colas en redes de comunicaciones. Estos algoritmos, cada uno con sus propias características, ventajas y limitaciones, están diseñados para optimizar el flujo de paquetes y reducir el tiempo de espera en redes, mejorando así la calidad del servicio (QoS). Los algoritmos de planificación son aplicables no solo en redes, sino también en escenarios prácticos de la vida real. Uno de los objetivos de este proyecto, es proponer un algoritmo de planificación (desarrollado a lo largo de la Sección 3.1), con la intención de corroborar las ventajas del uso de *schedulers* adaptativos, soluciones que han demostrado eficacia tanto en entornos técnicos como empresariales [9].

Existen diversas variaciones de algoritmos de planificación, cada una ofreciendo un equilibrio diferente entre eficiencia y complejidad. Uno de los algoritmos más comunes es First-Come, First-Serve (FCFS), que garantiza que todos los procesos en la cola se atiendan eventualmente y es fácil de implementar. Sin embargo, una de sus limitaciones es el alto tiempo promedio de espera en comparación con otros algoritmos. Por otro lado, el Shortest Job First (SJF) minimiza el tiempo de espera, pero corre el riesgo de generar inanición, ya que los procesos con tiempos de ejecución más largos pueden quedar pendientes indefinidamente. Además, el algoritmo SJF solo es viable cuando se pueden estimar con precisión los tiempos de ejecución, lo cual es poco común en entornos reales [9].

En los últimos años, se han propuesto políticas de encolamiento para asegurar la equidad en la gestión de solicitudes en un punto de servicio. El algoritmo Fair Queuing (FQ), desarrollado por Demers, Keshav y Shenkar, busca una distribución equitativa del ancho de banda, logrando una equidad que se aproxima a la alcanzada en un modelo de flujo continuo. En redes de datos, el algoritmo Head of Line Processor Sharing (PS) es considerado el más equitativo. La diferencia de rendimiento entre FQ y PS, para cualquier cola o patrón de llegada, nunca excede el tamaño máximo de un paquete, lo que convierte a esta diferencia en una métrica de equidad. No obstante, FQ presenta un alto costo de procesamiento de paquetes con una complejidad dependiente del número de flujos activos [10].

El algoritmo Weighted Fair Queuing (WFQ) es fundamental para garantizar el ancho de banda y el aislamiento de flujos en redes de alta velocidad. Sin embargo, implementar el algoritmo WFQ en el *hardware* de *switches* comerciales resulta complicado debido a su alta complejidad. Como alternativa, se han desarrollado versiones aproximadas de WFQ que operan con colas First-in, First-out (FIFO), más económicas y ampliamente disponibles. Sin embargo, estos planificadores aproximados de WFQ no logran asignar el ancho de banda de manera equitativa en flujos TCP, debido a la naturaleza variable del tráfico TCP. Además, las soluciones aproximadas de WFQ tienden a degradar la equidad de la planificación debido a las pérdidas de paquetes excesivas [11].

Los algoritmos FQ y WFQ han sido ampliamente estudiados y adoptados, tanto por la comunidad investigadora como por soluciones comerciales, por su alta equidad y limitación de retardo. Sin embargo, ambos requieren una cola de prioridad para determinar el siguiente paquete a ser procesado, lo cual mantiene su complejidad. Para reducir el alto costo de procesamiento de FQ, Shreedhar y Varghese propusieron el algoritmo Deficit Round Robin (DRR), que disminuye el procesamiento de paquetes. Sin embargo, la equidad de DRR, medida por la diferencia de rendimiento entre el modelo ideal PS y DRR, es significativamente menor en comparación con FQ [10].

La selección de rutas en redes de alto tráfico se basa en restricciones o políticas de QoS, como el retardo y el ancho de banda. La tecnología Multi-Protocol Label Switching (MPLS), que utiliza el enrutamiento basado en etiquetas, ha sido un catalizador en el campo de la ingeniería de tráfico. Permite una mayor flexibilidad en el control de la admisión y en la planificación de flujos, además de ofrecer configuraciones avanzadas para cumplir con políticas específicas de QoS.

Dependiendo de los criterios de las políticas de red (como el retardo, la confiabilidad y la capacidad), se pueden seleccionar diversas alternativas de rutas con diferentes niveles de complejidad, dependiendo de si el objetivo es cumplir con un umbral específico o maximizar el criterio.

A continuación se enumeran de forma resumida las técnicas más relevantes a la hora de configurar colas de QoS en dispositivos de red, entre los que se encuentran el planificador, descarte de paquetes y dimensionado de la memoria:

- **Planificación (*Scheduling*)**

  - **FCFS (First-Come-First-Served)**: Procesa las solicitudes estrictamente en el orden en que llegan, sin prioridad ni diferenciación entre ellas. Es simple pero puede provocar tiempos de espera largos si las primeras solicitudes tardan mucho.

  - **SJF (*Shortest Job First*)**: Atiende primero las solicitudes que requieren menos tiempo de procesamiento. Minimiza el tiempo promedio de espera, pero puede provocar inanición de trabajos largos si siempre llegan trabajos cortos.

  - **Round-Robin**: Asigna un intervalo de tiempo fijo (cuanto) a cada flujo o proceso en una cola circular. Después de usar su tiempo, el flujo pasa al final de la cola. Garantiza equidad temporal, útil en sistemas interactivos.

  - **DRR (Deficit Round Robin)**: Mejora Round-Robin para flujos con diferentes tamaños de paquetes. Cada flujo tiene un crédito (déficit), y puede enviar paquetes si su tamaño es menor o igual al crédito. Si no, espera al próximo turno acumulando su crédito.

  - **FQ (Fair Queueing)**: Simula que todos los flujos comparten un enlace de forma equitativa, dividiendo la capacidad del enlace entre ellos. Se ordenan los paquetes según un tiempo virtual de finalización para garantizar equidad en la transmisión.

  - **WFQ (Weighted Fair Queueing)**: Es una extensión de FQ que asigna un peso a cada flujo. Los flujos con mayor peso obtienen más ancho de banda. Así, se puede priorizar ciertos servicios mientras se mantiene una distribución justa de recursos.

- **Eliminación de Paquetes (Drop)**
  - **Drop-Tail**: Descarta el último paquete recibido cuando se llena la cola.
  - **Active Queue Management (AQM)**: Implementa la gestión dinámica de colas descartando paquetes antes de que el router se sature, como en el caso de **Random Early Detection (RED)**.
- **Dimensionado de la Capacidad del Buffer**
  - La capacidad del buffer se determina como el producto entre la capacidad del enlace y el RTT ($C \cdot RTT$).
  - Con N flujos TCP activos, la capacidad del buffer se estima como $\frac{C \cdot RTT}{\sqrt{N}}$.

2.2. Protocolo Openflow

Ante el incremento exponencial del volumen del tráfico en la red surge la necesidad de plantear estrategias más eficientes de gestión. Los proveedores comienzan a desarrollar *traffic engineering* ya que los algoritmos de encaminamiento y procesamiento tradicionales no son los más adecuados.

El protocolo OpenFlow surgió del programa *Clean Slate* de la universidad de *Stanford*, un proyecto que buscaba rediseñar Internet desde cero. El objetivo de este programa era replantear la infraestructura de la red, ya que se estaba quedando obsoleta y su limitada escalabilidad impedía su evolución. Dentro de este proyecto, nació la idea de utilizar un controlador centralizado que permitiera a los administradores de red definir políticas de control de seguridad basadas en los flujos de red y aplicarlas en múltiples dispositivos. Esto permitía implementar un control unificado sobre toda la comunicación en la red.

Este proyecto reveló que al separar los módulos de control de enrutamiento y el reenvío de paquetes en las redes tradicionales, sería posible utilizar un controlador centralizado para gestionar y configurar múltiples dispositivos de red a través de interfaces estándar. Esto abre nuevas posibilidades en el diseño, gestión y aprovechamiento de los recursos de la red, facilitando la innovación y el desarrollo tecnológico. Como resultado, se propuso el concepto de OpenFlow, y en 2008 se publicó el artículo titulado *OpenFlow: Enabling Innovation in Campus Networks* [12], en el que se detallaban por primera vez los principios y aplicaciones de OpenFlow.

Basado en OpenFlow, el equipo propuso en 2009 el concepto de Redes Definidas por Software (SDN), lo que despertó un gran interés en la industria. En 2011, empresas como Google, Facebook y Microsoft, entre otras, fundaron conjuntamente la Open Networking Foundation (ONF), una organización centrada en promover y facilitar la adopción de SDN. La ONF definió OpenFlow como la primera interfaz estándar de comunicación entre las capas de control y reenvío en la arquitectura SDN, y se encargó de su estandarización.

La arquitectura de OpenFlow se compone de tres elementos principales: un conmutador OpenFlow, un controlador externo y el protocolo OpenFlow, que permite la comunicación entre el conmutador y el controlador a través de un canal seguro. Un conmutador OpenFlow está formado por una o más tablas de flujo y una tabla de grupos, que se utilizan para la búsqueda y el reenvío de paquetes. Cada tabla de flujo contiene campos de coincidencia, contadores e instrucciones. Las cabeceras del paquete que se comparan

con los campos de coincidencia pueden pertenecer indistintamente a la segunda, tercera o cuarta capa de la arquitectura TCP/IP, dependiendo de la versión del protocolo OpenFlow. La versión más reciente, OpenFlow v1.5, soporta hasta 40 campos de coincidencia, incluidos aquellos para IPv6. Sin embargo, la versión más utilizada sigue siendo OpenFlow v1.0, que cuenta con 12 campos de coincidencia [5].

Los puertos OpenFlow son las interfaces de red utilizadas para el envío de paquetes entre el procesamiento de OpenFlow y el resto de la red. Estos puertos permiten que los *switches* OpenFlow se conecten lógicamente entre sí. Un paquete puede ser reenviado desde un *switch* OpenFlow a otro únicamente a través de un puerto de salida en el primer *switch* y un puerto de entrada en el segundo. Cada *switch* OpenFlow tiene disponibles varios puertos OpenFlow para el procesamiento de los paquetes. Sin embargo, no todos los puertos de red físicos del *hardware* del *switch* tienen que estar habilitados para OpenFlow, ya que algunos pueden estar desactivados, y el *switch* OpenFlow puede definir puertos adicionales específicos para OpenFlow. Los paquetes que ingresan a través de un puerto se procesan mediante el *pipeline* de OpenFlow y pueden ser reenviados a un puerto de salida. A continuación se indican los tipos de puertos OpenFlow [13]:

- Puertos físicos: son los puertos definidos por el *switch* y corresponden directamente a interfaces de *hardware*. Por ejemplo, en un *switch* Ethernet, los puertos físicos están vinculados a las interfaces Ethernet del dispositivo [13].
- Puertos lógicos: son abstracciones dentro del *switch* que no están asociadas directamente con una interfaz de *hardware*. Estos puertos se utilizan para funciones como la agregación de enlaces, túneles o interfaces de *loopback* [13].
- Puertos reservados: son puertos especiales utilizados para funciones internas o de control dentro del *switch* (por ejemplo, puerto de inyección de tráfico interno o el puerto de *loopback*) [13].
    - *CONTROLLER* (requerido): representa el canal de control con los controladores OpenFlow. Puede utilizarse como un puerto de entrada o como un puerto de salida. Cuando se usa como puerto de salida, encapsula el paquete en un mensaje *packet-in* y lo envía utilizando el protocolo del *switch* OpenFlow. Cuando se usa como puerto de entrada, identifica un paquete que se origina desde el controlador.
    - *LOCAL* (opcional): representa la pila de red local del *switch* y su pila de gestión. Puede utilizarse como puerto de entrada o como puerto de salida. El puerto *LOCAL* permite que entidades remotas interactúen con el *switch* y sus servicios de red a través de la red OpenFlow, en lugar de hacerlo mediante una red de control separada. Con un conjunto adecuado de entradas de flujo (*flow entries*), puede utilizarse para implementar una conexión con el controlador dentro de la propia red de datos (*in-band controller connection*), aunque esto está fuera del alcance de esta especificación.
    - *NORMAL* (opcional): representa el reenvío mediante el canal tradicional no basado en OpenFlow del *switch*. Solo puede usarse como puerto de salida y procesa el paquete utilizando el canal de procesamiento normal. En general, reenviará o encaminará el paquete; sin embargo, el resultado real depende de la implementación. Si el *switch* no puede reenviar paquetes desde el canal OpenFlow al canal tradicional, debe indicar que no admite esta acción.

- *FLOOD* (opcional): representa el envío masivo (*flooding*) mediante el canal tradicional no basado en OpenFlow del *switch*. Solo puede usarse como puerto de salida, y el resultado real depende de la implementación.

El comportamiento de estos puertos es fundamental para el control y la gestión dinámica del tráfico en redes SDN, donde OpenFlow juega un papel clave en el encaminamiento flexible y centralizado de los paquetes [13].

Los *switches* compatibles con OpenFlow se dividen en dos tipos: OpenFlow-only y OpenFlow-híbrido. Los *switches* OpenFlow-only procesan todos los paquetes exclusivamente a través del canal OpenFlow, sin utilizar otros métodos de procesamiento. En cambio, los *switches* OpenFlow-híbrido admiten tanto las operaciones OpenFlow como las tradicionales de *switching* Ethernet, incluyendo *switching* L2, aislamiento Virtual Local Area Network (VLAN), enrutamiento L3 (IPv4 e IPv6), Access Control List (ACL) y procesamiento de calidad de servicio (QoS). Estos *switches* híbridos necesitan un mecanismo externo a OpenFlow para clasificar el tráfico y dirigirlo hacia el canal OpenFlow o el tradicional. Por ejemplo, un *switch* híbrido puede usar etiquetas VLAN o el puerto de entrada de un paquete para decidir si procesarlo mediante el canal OpenFlow o no. Este mecanismo de clasificación no forma parte de la especificación de OpenFlow. Además, en los *switches* híbridos, los paquetes pueden pasar del canal OpenFlow al canal tradicional mediante los puertos reservados *NORMAL* y *FLOOD* [13].

Un *switch* OpenFlow debe tener al menos una tabla de flujo, pero puede tener más tablas si se requiere. Los *switches* con una sola tabla de flujo son válidos y simplifican el procesamiento de la canalización. Las tablas de flujo están numeradas de manera secuencial, comenzando desde 0, y los paquetes las atraviesan en ese orden. El procesamiento interno del tráfico se divide en dos etapas: entrada y salida (ver Figura 2.3). La primera tabla de salida marca el límite entre ambas etapas. Las tablas con un número menor se utilizan como tablas de entrada, mientras que las tablas con un número mayor se utilizan como tablas de salida. El procesamiento siempre comienza en la primera tabla de entrada (tabla 0). Si un paquete coincide con una entrada de flujo en esta tabla, se ejecutan las acciones definidas para esa entrada, y el paquete puede ser reenviado a otra tabla de entrada o a un puerto de salida. Si el paquete es reenviado a un puerto de salida, puede ser procesado en las tablas de salida, si están configuradas. Si no hay tablas de salida configuradas, el paquete se reenvía directamente fuera del *switch*. En caso de que haya tablas de salida configuradas, el paquete debe coincidir con las entradas de estas tablas y el procesamiento continuará según los resultados de esa coincidencia como sigue en la Figura 2.2 [13].

La canalización de OpenFlow es una abstracción que se asigna al *hardware* físico del *switch*. En algunos casos, el *switch* OpenFlow puede estar virtualizado dentro del *hardware*, como cuando se manejan múltiples instancias de OpenFlow o en el caso de un *switch* híbrido. Incluso si el *switch* no está virtualizado, la estructura del *hardware* puede no coincidir completamente con la canalización OpenFlow. Por ejemplo, una tabla de flujo que solo admite paquetes Ethernet requerirá que los paquetes no Ethernet se mapearan previamente a Ethernet. Otro caso es cuando ciertos *switches* almacenan información de VLAN en metadatos internos, mientras que para OpenFlow la VLAN es tratada como parte del paquete. Además, algunos *switches* OpenFlow definen puertos lógicos que implementan encapsulaciones complejas que alteran los encabezados de los paquetes. Como resultado, un paquete transmitido a través del *hardware* puede representarse de manera distinta en la canalización OpenFlow [13].

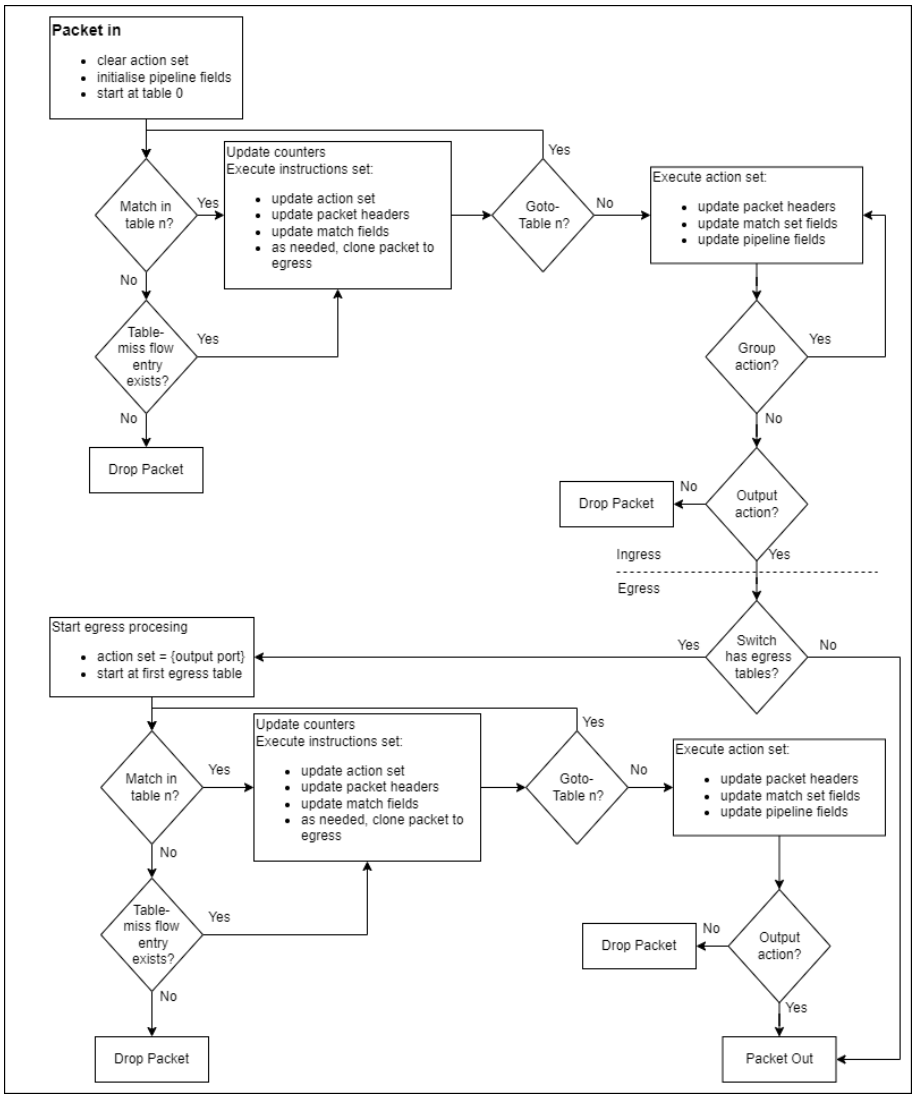


Figura 2.2: Openflow Matching [13]

En algunos casos, una tabla de flujo puede configurarse para el procesamiento de entrada o de salida mediante un cambio en la primera tabla de salida. Al inicio del procesamiento de entrada, el conjunto de acciones está vacío. Las tablas de flujo usadas para el procesamiento de entrada solo pueden dirigir paquetes a otras tablas de entrada mediante la instrucción *Goto-Table* y no pueden redirigirlos a tablas de salida. Por lo general, las tablas de entrada no admiten el campo *Action Set Output*, aunque en algunas excepciones puede ser soportado [13].

En el procesamiento de salida, el conjunto de acciones inicialmente contiene solo la acción de salida hacia el puerto de salida actual. Los valores válidos para esta acción de salida incluyen puertos físicos, lógicos o los puertos reservados *CONTROLLER* y *LOCAL*; los demás puertos reservados deben sustituirse por sus valores reales. Los valores de los campos de la canalización se mantienen, excepto en el campo *Action Set Output*; por ejemplo, el valor inicial del campo Metadata coincide con el que tenía cuando el paquete ingresó en el procesamiento de salida [13].

Las tablas de flujo de salida solo pueden dirigir paquetes a otras tablas de salida mediante la instrucción *Goto-Table*. Estas tablas deben admitir el campo *Action Set Output* para que los flujos puedan basarse en

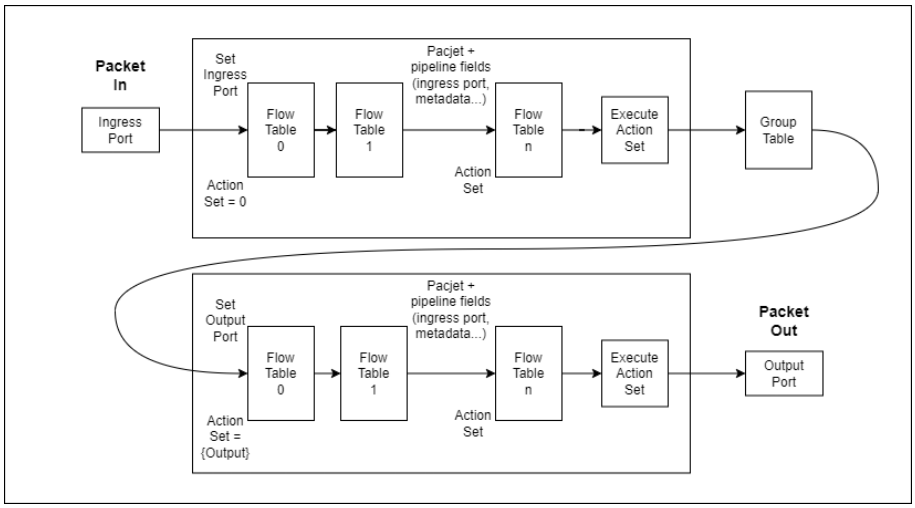


Figura 2.3: Openflow Tables [13]

el contexto del puerto de salida. También deben prohibir el uso de la acción de salida o de grupo en la instrucción *write-action* para evitar que el puerto de salida del paquete sea modificado. Estas restricciones deben especificarse en las características de la tabla de flujo[13].

Opcionalmente, las tablas de salida pueden permitir el uso de la acción de salida o la acción de grupo en la instrucción apply-action. Estas acciones se comportan de la misma forma que en una tabla de entrada: clonan el paquete hacia los puertos especificados, y esos clones inician el procesamiento de salida desde la primera tabla de salida. Esta funcionalidad es útil, por ejemplo, para implementar el mirroring (técnica que permite copiar el tráfico de una o varias interfaces y enviarlo a otra interfaz para su monitorización o análisis) selectivo de salida [13].

Un *switch* OpenFlow debe implementar mecanismos de protección contra bucles infinitos de paquetes cuando utiliza acciones de salida o de grupo en sus tablas de salida. La implementación de este mecanismo depende de cada fabricante y está fuera del alcance de la especificación [13].

El protocolo OpenFlow define tres tipos de mensajes: del controlador al switch, síncronos y simétricos. Los mensajes de controlador/*switch* son iniciados por el controlador y pueden o no requerir una respuesta del *switch*, entre ellos se incluyen:

- **Características**: El controlador puede solicitar la identidad y las capacidades básicas de un *switch* enviando una solicitud de características; el *switch* debe responder con una respuesta de características que especifique la identidad y las capacidades básicas del *switch*. Esto se realiza comúnmente al establecer el canal OpenFlow [13].

- **Configuración**: El controlador puede establecer y consultar parámetros de configuración en el *switch*. El *switch* solo responde a una consulta del controlador [13].

- **Modificar Estado**: El controlador envía mensajes para modificar el estado del *switch*, incluyendo la adición, eliminación y modificación de entradas de flujo o grupo, y la gestión de las propiedades de los puertos del *switch*. Estos mensajes permiten al controlador configurar dinámicamente las tablas OpenFlow, ajustando aspectos como los action buckets de un grupo[13].

- **Leer Estado**: El controlador utiliza estos mensajes para obtener información sobre el estado actual, estadísticas y capacidades del *switch*. La mayoría de las solicitudes de lectura de estado emplean secuencias de mensajes multipartes para transmitir datos de gran tamaño[13].
- ***Packet-out***: Estos mensajes son utilizados por el controlador para enviar paquetes desde un puerto específico en el *switch*, y para reenviar paquetes recibidos a través de mensajes Packet-in. Los mensajes Packet-out deben contener un paquete completo o un ID de buffer que haga referencia a un paquete almacenado en el *switch*. El mensaje también debe contener una lista de acciones que se aplicarán en el orden especificado; una lista vacía de acciones descarta el paquete [13].
- **Barrera**: Los mensajes de solicitud/respuesta de barrera son usados por el controlador para asegurar que las dependencias de los mensajes se han cumplido o para recibir notificaciones de operaciones completadas [13].
- **Solicitud de rol (*Role-Request*)**: Permiten al controlador establecer o consultar su rol en el canal OpenFlow y configurar su ID de controlador. Esto es especialmente útil cuando el *switch* está conectado a múltiples controladores [13].
- **Configuración asíncrona (*Asynchronous-Configuration*)**: Mediante estos mensajes, el controlador puede definir filtros para los mensajes asíncronos que desea recibir o consultar el filtro configurado. Esta función es especialmente útil en escenarios donde un *switch* está conectado a múltiples controladores y se realiza generalmente al establecer el canal OpenFlow [13].

Los mensajes asíncronos son enviados por el *switch* sin que el controlador los solicite. Estos mensajes notifican al controlador sobre la llegada de paquetes o cambios de estado en el *switch*, manteniendo la coherencia en la visión del controlador sobre el estado del *switch*. Los tipos principales de mensajes asíncronos incluyen *flow-removed*, *port-status* y *packet-in*. Estos mensajes pueden ser filtrados según la configuración asíncrona establecida, permitiendo al controlador recibir solo los eventos relevantes [13]. Finalmente, los mensajes simétricos los puede enviar tanto el switch como el controlador e incluyen mensajes de error y los utilizados en el establecimiento de la conexión OpenFlow.

El protocolo OpenFlow asegura una entrega confiable de mensajes, aunque no garantiza automáticamente la confirmación o el procesamiento en orden. La entrega de mensajes está asegurada en el canal OpenFlow, salvo en caso de fallo total del canal. Si esto ocurre, el controlador no puede asumir ningún estado específico del *switch* y podría activarse un modo autónomo de fallo en el *switch* [13].

Los *switches* deben procesar completamente cada mensaje recibido de un controlador, lo que posiblemente genere una respuesta. Si un *switch* no puede procesar completamente un mensaje recibido de un controlador, debe enviar un mensaje de error. Para los mensajes de *packet-out*, procesar completamente el mensaje no garantiza que el paquete incluido realmente salga del *switch*. El paquete incluido puede ser descartado después del procesamiento de OpenFlow debido a congestión en el *switch*, política de QoS o si se envía a un puerto bloqueado o no válido [13].

El *switch* debe enviar al controlador todos los mensajes asíncronos generados por cambios de estado en OpenFlow, como flow-removed, port-status o packet-in, para que el controlador pueda mantener una visión actualizada del estado del *switch*. Sin embargo, algunos de estos mensajes asíncronos pueden filtrarse para reducir la carga de procesamiento, de acuerdo con la configuración asíncrona establecida.

Version	Date	Header Fields
OF 1.0	Dec 2009	12 Fields (Ethernet, TCP/IPv4)
OF 1.1	Feb 2011	15 Fields (MPLS, inter-table metadata)
OF 1.2	Dec 2011	36 Fields (ARP, ICMP, IPv6, etc.)
OF 1.3	June 2012	40 Fields
OF 1.4	Oct 2013	41 Fields

Tabla 2.1: Versiones OpenFlow y campos de las cabeceras

Por ejemplo, los paquetes recibidos en ciertos puertos que deberían reenviarse al controlador pueden ser descartados debido a políticas de QoS o congestión en el *switch*, sin que se genere un mensaje packet-in [13].

Es recomendable implementar mecanismos de control para los paquetes destinados al controlador, con el fin de prevenir posibles ataques Denial of Service (DoS) en la conexión del controlador. Esto se puede lograr mediante el uso de colas en el puerto asignado al controlador o mediante medidores por flujo. Aunque los controladores pueden optar por ignorar ciertos mensajes recibidos, deben responder a ciertos mensajes para evitar que el *switch* finalice la conexión [13].

El protocolo OpenFlow ha evolucionado desde la versión 1.0 hasta la versión 1.5. En sus inicios, solo contaba con 12 campos de coincidencia fijos y una única tabla de flujo, pero ha crecido significativamente desde entonces. A medida que se añadieron más campos de coincidencia, surgió un problema: los conmutadores existentes no podían implementarlos porque su arquitectura es estática y no permite la reconfiguración. Esto significa que una vez que los campos de coincidencia se integran en el *hardware* del conmutador, no pueden modificarse. Esta limitación generó ineficiencias, ya que los nuevos campos de coincidencia no podían implementarse en los conmutadores existentes. Inicialmente, los conmutadores reconfigurables eran demasiado lentos, pero con los avances en la tecnología de sistemas microelectromecánicos, estos conmutadores mejoraron su velocidad. En la Tabla 2.1 se muestran las distintas versiones del protocolo OpenFlow y el número de campos de encabezado que soporta cada una. [14].

2.2.1. Controlador Openflow

Como se ha comentado antes, el switch OpenFlow procesa paquetes del plano de datos en función de la configuración de sus tablas. En el caso de que el paquete no coincida con ningún campo de coincidencia, el conmutador puede configurarse para descartarlo o enviarlo al controlador para su procesado mediante un mensaje *PACKET_IN*.

Ante esta funcionalidad aparecen cuestiones relativas al rendimiento de los controladores: (i) ¿Qué tan rápido puede responder un controlador?, (ii) ¿Cuántos mensajes puede gestionar un controlador por segundo?. Para responder a estas preguntas, es fundamental evaluar el rendimiento de los controladores SDN tanto de manera cuantitativa como cualitativa, ya que esto permite comprender sus ventajas y limitaciones. Dicha evaluación ofrece una guía clara para seleccionar el controlador más adecuado para cada escenario. Se pueden comparar los distintos controladores SDN en función de indicadores clave de rendimiento, como la latencia, el rendimiento y la resiliencia [15].

En la actualidad, existen varios controladores OpenFlow ampliamente utilizados, entre los que destacan OpenDayLight, ONOS, Ryu, Floodlight y POX. Estos cinco controladores son los más relevantes y ofrecen

distintas características. La mayoría de estos controladores han alcanzado un alto grado de madurez, lo que hace relevante una reevaluación de sus rendimientos actuales. El lenguaje de programación que utiliza cada controlador SDN tiene un impacto significativo en su rendimiento. Java, por ejemplo, es una opción popular debido a su compatibilidad multiplataforma y soporte para *multithreading*. *Python*, en cambio, se descartó en algunos casos debido a sus limitaciones en *multithreading*, mientras que C# fue descartado debido a su dependencia de Windows. C y C++ también han perdido relevancia debido a su compleja gestión de memoria. En términos de rendimiento, el controlador Beacon, basado en Java, ha mostrado los mejores resultados [15]. A continuación se resumen las características de algunos de los controladores más utilizados.

OpenDayLight y ONOS

OpenDayLight y ONOS se destacan por ser los controladores más completos en cuanto a funcionalidades. Ambos soportan múltiples plataformas y son altamente modulares gracias a OSGi, lo que permite tiempos de carga rápidos y una alta flexibilidad en el desarrollo. Ambos controladores cuentan con una interfaz gráfica de usuario intuitiva y una comunidad activa que contribuye constantemente con mejoras. Su arquitectura distribuida los hace ideales para implementaciones de SDN a gran escala en escenarios realistas. Además, estos controladores soportan interfaces hacia el plano de datos adaptadas para SDN híbrido, lo cual es crucial para implementaciones que combinan tecnologías tradicionales y definidas por software. Sin embargo, ONOS no cuenta con soporte para la orquestación en la nube (por ejemplo, OpenStack), lo cual puede ser una limitación en la gestión de recursos virtuales [15].

RYU

Ryu ofrece un conjunto de características adecuadas para implementaciones de SDN a pequeña escala. Es moderadamente modular y utiliza *python* como lenguaje de programación, lo que facilita el desarrollo, pero limita su capacidad de escalado. Su arquitectura centralizada y el hecho de estar diseñado exclusivamente para sistemas Linux también restringen su aplicación a redes de menor escala [15].

Floodlight

Floodlight es el controlador con menos características en comparación con los anteriores. Una de sus limitaciones principales es que solo soporta OpenFlow como interfaz hacia el plano de datos. Además, carece de modularidad, no ofrece un esquema de arquitectura distribuida ni soporte para orquestación en la nube, lo que lo hace más adecuado para aplicaciones SDN a pequeña escala [15].

2.3. Open Virtual Switch (OVS)

Open vSwitch (OVS) es la implementación de código abierto más popular de un *switch* basado en OpenFlow y se ha consolidado como el estándar de facto en este ámbito. Su importancia ha crecido rápidamente, convirtiéndose en una pieza fundamental en numerosos proyectos de redes definidas por software (SDN) de código abierto, como OpenStack. No obstante, cuando se abordó la implementación de la configuración de OVS, no existía un estándar formal para esta área. Por ello, los desarrolladores de OVS optaron por crear su propio protocolo, la base de datos de *Open vSwitch* (OVSDB), para gestionar y manipular los datos de configuración del *switch*. Aunque el protocolo fue publicado como un documento

Request for Comments (RFC) informativo por Internet Engineering Task Force (IETF) en 2013, el esquema de la base de datos solo está documentado en un documento interno de OVS [16].

Open vSwitch (OVS) es un *switch* virtual que ha ganado un papel central en todas las principales plataformas de hipervisor. Una amplia variedad de aplicaciones de red, que incluyen redes definidas por software (SDN), Network Functions Virtualization (NFV) y computación en la nube de alto rendimiento, se benefician de la flexibilidad y escalabilidad que ofrece OVS. El rendimiento de estas aplicaciones depende directamente de OVS, que actúa como el núcleo de sus funcionalidades, y la eficiencia de OVS, a su vez, está estrechamente vinculada a su capacidad para gestionar la clasificación de paquetes de manera efectiva [17].

La clasificación de paquetes es crucial para diversas funciones de red, como la calidad de servicio (QoS), la seguridad y otras funcionalidades avanzadas empleadas en *switches*, *routers*, *firewalls*, balanceadores de carga y otros dispositivos de red. Su propósito es identificar los paquetes entrantes que coinciden con un conjunto de reglas definidas, basadas en múltiples campos como las direcciones IP de origen o destino, puertos TCP o UDP, entre otros. Los paquetes que coinciden con la misma regla se agrupan en un flujo y se les aplica la misma acción o tratamiento [17].

La clasificación de paquetes debe enfrentar dos requisitos que a menudo son contradictorios. Por un lado, la clasificación de paquetes necesita procesar los paquetes a la velocidad que permita la red en el plano de datos, minimizando el retardo asociado al procesamiento. Esto requiere integrar procesamiento de paquetes eficientes, altamente optimizados y, a veces, muy complejos en las rutas críticas de procesamiento de paquetes de los dispositivos de red. Por estas razones, los componentes de clasificación de paquetes incluso han sido integrados en componentes de *hardware* específicos, como Application-Specific Integrated Circuit (ASIC), Ternary Content Addressable Memory (TCAM), Field-Programmable Gate Array (FPGA) o Graphics Processing Unit (GPU). Por otro lado, los conjuntos de reglas para la clasificación de paquetes suelen ser dinámicos, actualizándose constantemente. Estas actualizaciones pueden afectar al canal de procesamiento de paquetes, obligando a detener parcial o completamente el procesamiento y a almacenar temporalmente los paquetes entrantes obligando a aumentar la ocupación de los *buffers* [17].

Las aplicaciones que utilizan *Open vSwitch* (OVS) deben manejar ambos requisitos: alto rendimiento y la capacidad de realizar numerosas actualizaciones de reglas por segundo. Dado que OVS generalmente se implementa en servidores comerciales que no cuentan con *hardware* especializado como ASIC, TCAM o FPGA, las plataformas que lo utilizan deben depender de algoritmos de clasificación de paquetes basados en software que permitan tanto una clasificación rápida como actualizaciones ágiles de las reglas. Además, se han identificado ataques recientes contra la infraestructura de OVS, los cuales explotan los algoritmos de clasificación al generar tráfico diseñado para degradar considerablemente el rendimiento de OVS. Por ello, las plataformas OVS deben ser robustas y capaces de resistir estos ataques [17].

Open vSwitch (OVS) emplea un clasificador TSS en las tablas de coincidencia (*match tables*). Este clasificador fue elegido debido a sus propiedades, como la capacidad de mantener una complejidad constante en las actualizaciones. Sin embargo, TSS requiere consultar todas las tablas hash, lo que reduce su rendimiento en las búsquedas [17].

A continuación se introducirán conceptos relacionados con la arquitectura de OVS y la técnica de *tuple space search* que implementa para comprender mejor su funcionamiento:

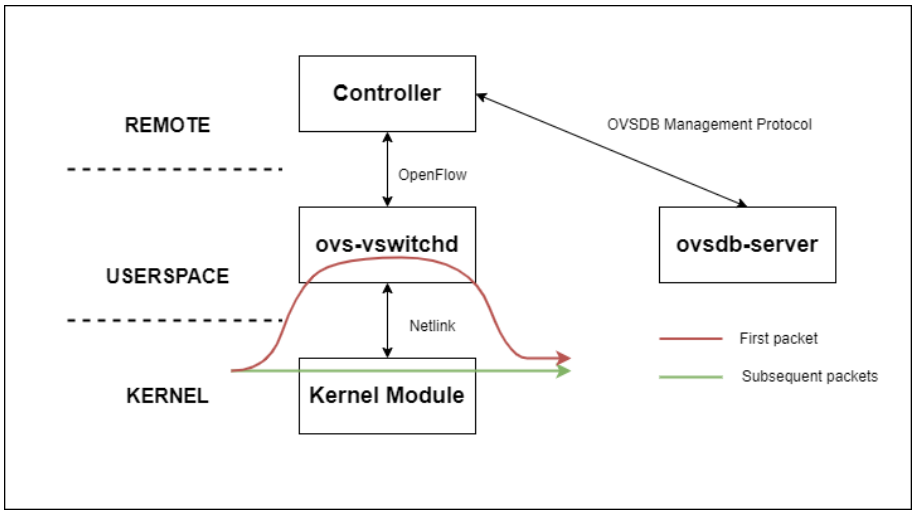


Figura 2.4: Arquitectura OVS [18]

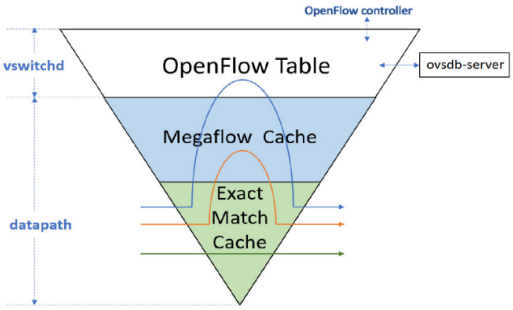


Figura 2.5: Arquitectura OVS. TSS [17]

2.3.1. OVS: Arquitectura

Open vSwitch se compone de dos elementos claves, uno de ellos mencionado anteriormente: la base de datos OVSDB y *Open vSwitch Daemon* (ovs-vswitchd) . Ambos elementos definidos en la arquitectura del *vSwitch* en la Figura 2.4.

OVS ejecuta el servidor OVSDB como un proceso independiente que se comunica con los clientes de OVSDB. Este protocolo utiliza el intercambio de mensajes basados en JSON-RPC, tal como se define en su especificación. Los mensajes permiten gestionar los datos de configuración administrados por el servidor OVSDB [16].

Open vSwitch (OVS) se estructura en dos planos: el plano de datos, que maneja todos los paquetes entrantes al *switch* , y el plano de control, que contiene las reglas que definen el comportamiento aplicado a los paquetes de flujos coincidentes. El plano de control de OVS se ejecuta en el espacio de usuario mediante el proceso vswitchd, que gestiona una OpenFlow Table (OFT) con todas las reglas. Por otro lado, el plano de datos de OVS puede ejecutarse en el núcleo del sistema operativo o en el espacio de usuario utilizando Data Plane Development Kit (DPDK). Este plano de datos está compuesto por dos memorias caché: Exact Match Cache (EMC) y MegaFlow Cache (MFC). Cuando un paquete ingresa al sistema, primero se compara con los flujos en el EMC, que contiene descripciones completas de los flujos observados recientemente, sin reglas comodín. Si un paquete coincide con una regla en el EMC, se

aplican las acciones asociadas a esa regla. Las reglas en el EMC están definidas sin prefijos, por lo que la coincidencia puede realizarse de manera eficiente mediante una tabla hash. Sin embargo, dado que el número de flujos en la red puede ser muy grande, el EMC no puede almacenar todos los flujos. Para ello, se utiliza un segundo nivel de caché, el MFC. Cuando un paquete no encuentra coincidencia en el EMC, se envía al MFC, que sí contiene reglas con comodines. Si el paquete encuentra coincidencia en el MFC, se inserta una nueva descripción de flujo en el EMC con todos los detalles del paquete coincidente y la acción correspondiente. Si no se encuentra coincidencia en el MFC, el paquete se envía a la OFT, que contiene el conjunto completo de reglas. Si la OFT encuentra una coincidencia con una regla comodín, esta se reenvía al plano de datos y se inserta en el MFC, mientras que en el EMC se añade una entrada exacta correspondiente. Cada regla en el MFC tiene un temporizador de envejecimiento que elimina la entrada después de cierto tiempo [17].

Esta arquitectura de tres capas (ver Figura 2.5) proporciona a *Open vSwitch* (OVS) un alto rendimiento. La mayoría de los paquetes se comparan en el EMC o en la caché MFC, sin necesidad de llegar a la OFT y comparar con todo el conjunto de reglas. No obstante, incluso con un alto índice de aciertos, el rendimiento general de la plataforma OVS depende en gran medida de la eficiencia de coincidencia en el MFC. Mientras que la coincidencia en el EMC se realiza mediante una tabla hash simple, la coincidencia en el MFC y en la OFT es más compleja. OVS nativo utiliza el enfoque de Búsqueda de Espacio de Tuplas (TSS) para implementar la clasificación de paquetes y la coincidencia de reglas tanto en el MFC como en la OFT. Este enfoque se selecciona por varias razones: primero, permite actualizaciones eficientes en tiempo constante, incluso con un número arbitrario de campos coincidentes; segundo, maneja de manera eficaz cambios algorítmicos en las coincidencias; y tercero, TSS hace una utilización de memoria relativamente pequeña, que crece de manera lineal con el número de flujos activos. Estas ventajas siguen siendo relevantes en la implementación actual de OVS, por lo que resulta beneficioso continuar utilizando una variante de búsqueda de tuplas [17].

Antes de profundizar en TSS, es importante señalar otro cuello de botella de rendimiento en el algoritmo de clasificación, identificado en la literatura. En una red, el sistema operativo actúa como intermediario entre los programas en el espacio de usuario y los dispositivos de *hardware*, como las tarjetas de red. El acceso a los paquetes a través de las protecciones del sistema operativo y la pila de protocolos resulta costoso, ya que la ruta de datos suele implementarse en el núcleo del sistema operativo o del hipervisor. Este acceso se vuelve aún más complejo en entornos virtualizados, donde varios sistemas comparten la infraestructura *hardware*. El Kit de Desarrollo de Plano de Datos (DPDK) es un proyecto de software de código abierto, gestionado por la Fundación Linux, que aborda este problema. DPDK proporciona un conjunto de librerías de plano de datos y controladores de red que permiten la comunicación directa entre la tarjeta de red y el espacio de usuario, evitando así la necesidad de atravesar la pila de protocolos en el núcleo. DPDK-OVS, una rama de OVS, aprovecha DPDK para permitir que todo el funcionamiento de OVS se realice en el espacio de usuario, facilitando así su despliegue en entornos virtualizados. Con DPDK-OVS, tanto el plano de datos como el plano de control se ejecutan en el espacio de usuario [17].

DPDK proporciona acceso directo a la tarjeta de red a través de un puerto DPDK en el espacio de usuario. DPDK-OVS se ejecuta como un proceso de múltiples hilos que contiene varios hilos Poll Mode Driver (PMD), cada uno asignado a un núcleo separado. Cada hilo PMD contiene un caché EMC y MFC separados y está conectado al puerto DPDK por colas Rx y Tx utilizadas, respectivamente, para recibir y

enviar paquetes. Un mecanismo Receive Side Scaling (RSS) distribuye los paquetes recibidos desde el puerto DPDK sobre las colas Rx salientes utilizando una función hash sobre las tuplas del encabezado del paquete. Este mecanismo permite un balanceo de carga eficiente de los múltiples hilos PMD y asegura que todos los paquetes pertenecientes a un flujo sean procesados en el mismo PMD, y que agregar o eliminar una entrada en el EMC o MFC solo impacte al hilo correspondiente. Además, dividir el MFC entre varios hilos hace que el tamaño del MFC sea más pequeño y simplifica la coincidencia de paquetes. Sin embargo, la caché MFC se actualiza frecuentemente, por lo que su rendimiento de actualización se convierte en un cuello de botella importante para DPDK-OVS [17].

2.3.2. OVS: *Quality of Service*

OVS soporta QoS para el tráfico de entrada y el tráfico de salida del *switch* , descartando aquellos paquetes que excedan la tasa configurada. Sin embargo *Open vSwitch* no implementa QoS sino que emplea los mecanismos Traffic Control (TC) de Linux. TC emplea disciplinas de colas o *qdisc* (*queueing discipline*), de manera que cada vez que el kernel tiene que enviar un paquete a una interfaz, este se encola en la *qdisc* configurada en dicha interfaz. El kernel trata de sacar el mayor número de paquetes de esa *qdisc* para enviarlos al driver de la tarjeta de red.

La qdisc determinará qué paquetes se reenviarán antes que otros. Existen dos tipos:

- ***classless qdisc***:

  - ***ingress***: cada interfaz puede tener una *ingress* qdisc que no se utiliza para enviar paquetes al adaptador de red. En cambio, te permite aplicar filtros TC a los paquetes que llegan por la interfaz, ya sea que tengan un destino en la red local o deban ser reenviados (Figura 2.6). Dado que los filtros TC contienen una implementación completa de Token Bucket Filter y también pueden hacer coincidir con el estimador de flujo del kernel, hay una amplia funcionalidad disponible. Esto te permite controlar el tráfico entrante antes de que entre en la pila de protocolos IP.

  - **fq_codel**: Fair Queuing Controlled Delay es una técnica de gestión de colas que integra Fair Queuing con el esquema de gestión activa de colas CoDel (Controlled Delay). Esta combinación utiliza un modelo estocástico para clasificar los paquetes entrantes en distintos flujos, asegurando una distribución equitativa del ancho de banda para todos los flujos que utilizan la cola. Cada uno de estos flujos es gestionado por la disciplina de encolamiento CoDel, que garantiza un control efectivo del retardo. Internamente utiliza una cola FIFO. FIFO perjudica a TCP debido a que una pérdida de un paquete en TCP activa los mecanismos de control de congestión que provocan una disminución de la tasa de envío para la conexión TCP. Una pérdida de un paquete UDP no tiene ningún impacto en la tasa de envío para esta comunicación.

  - **netem**: proporciona una funcionalidad de emulación de red, que es útil para probar protocolos de red al simular las condiciones de redes de un entorno real. La qdisc netem simula diferentes aspectos y problemas que pueden ocurrir en una red, afectando a los paquetes de datos. Algunos de estos aspectos incluyen: delay, perdidas, duplicación...

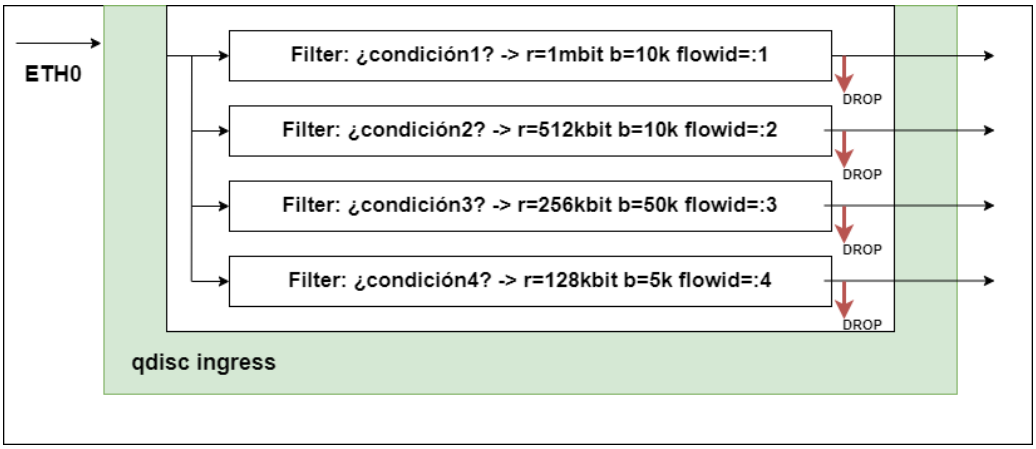


Figura 2.6: *Ingress* qdisc [19]

- **tbf**: Token Bucket Filter, limita la velocidad del tráfico de un interfaz basándose en tres parámetros. Ancho de banda, tamaño del *bucket* (que almacena los *tokens*) y latencia que se corresponde al tiempo de espera de un paquete por un token antes de ser descartado.

- ***classful qdisc***: una clase especifica las diferentes categorías de tráfico. Una clase de tráfico está asociada a una qdisc. Incluye filtros, que identifica el tráfico que se corresponde con cada clase.

  - **HFSC**: Hierarchical Fair Service Curve garantiza una asignación precisa de ancho de banda y delay asignando el exceso de ancho de banda de manera justa. A diferencia de HTB, utiliza el dropping para lograr delays bajos.

  - **HTB**: TBF se utiliza cuando se desea que todo el tráfico que sale por una determinada interfaz tenga unas características determinadas en cuanto a ancho de banda, tamaño de cubeta y latencia. Hierarchy Token Bucket es necesario cuando se quiere clasificar el tráfico que sale por una determinada interfaz en varias clases, cada uno de ellas con unas características diferentes de ancho de banda. Si alguna clase no utiliza todo el ancho de banda que se le ha definido, dependiendo de la configuración, se podría utilizar dicho ancho de banda para algún otra clase que lo necesite.

Para cada flujo de tráfico puede haber reglas a la entrada y a la salida de los interfaces. Para un tráfico de entrada (*ingress* qdisc) solo se puede definir una política con filtros, es decir, un *token bucket filter* (tbf) con una tasa y una longitud de ráfaga (*burst*).

Para el tráfico de salida (*egress traffic*) se puede hacer *shaping* como un tbf con prioridades o un HTB. En los interfaces de salida de tráfico se tiene más control sobre el tbf pudiendo añadir el parámetro de latencia que se aplica antes que la ráfaga, y define el tiempo máximo de espera de un paquete por un token. Este parámetro sirve para los paquetes que entran en la cola y se quedan esperando a que caiga un *token* en el *bucket* cuánto tiempo como máximo están ahí.

En la documentación de *Open vSwitch*, los QoS con y sin *class qdisc* compatibles con OVS y sus parámetros configurables mediante el protocolo OVSDB son los referidos en la Figura 2.8.

De la misma manera, y como se especifica en la documentación de OVS, en la Figura 2.9 se muestran los parámetros configurables de las colas asociadas a un QoS.

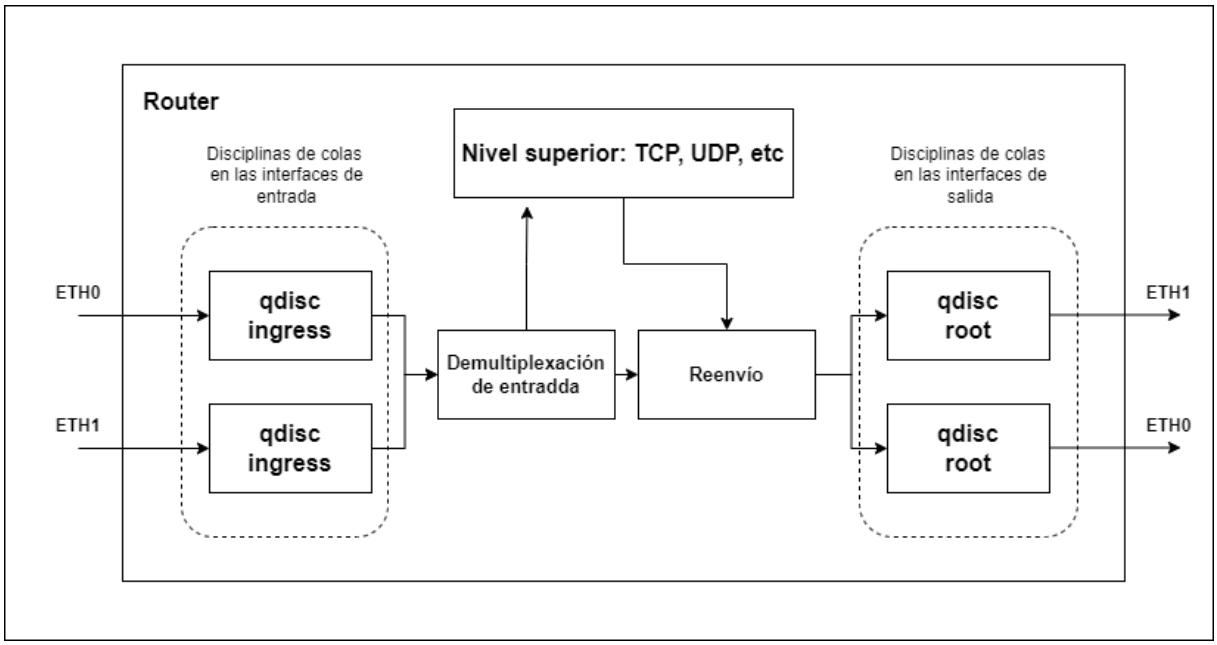


Figura 2.7: Esquema *Ingress*/*Egress* qdisc [19]

```
# QoS TABLE
# Quality of Service (QoS) configuration for each Port that references it.

# Summary:
type                        string
queues                      map of integer-Queue pairs, key in range 0 to 4,294,967,295

# Configuration for linux-htb and linux-hfsc:
other_config : max-rate     optional string, containing an integer

# Configuration for egress-policer QoS:
other_config : cir          optional string, containing an integer
other_config : cbs          optional string, containing an integer
other_config : eir          optional string, containing an integer
other_config : ebs          optional string, containing an integer

# Configuration for linux-sfq:
other_config : perturb      optional string, containing an integer
other_config : quantum      optional string, containing an integer

# Configuration for linux-netem:
other_config : latency      optional string, containing an integer
other_config : limit        optional string, containing an integer
other_config : loss         optional string, containing an integer
other_config : jitter       optional string, containing an integer

# Common Columns:
other_config                map of string-string pairs
external_ids                map of string-string pairs
```

Figura 2.8: QoS OVSDB [20]

```
# Queue TABLE
# A configuration for a port output queue, used in configuring Quality of
# Service (QoS) features. May be referenced by queues column in QoS table.

# Summary:
dscp                        optional integer, in range 0 to 63

# Configuration for linux-htb QoS:
other_config : min-rate     optional string, containing an integer, at least 1
other_config : max-rate     optional string, containing an integer, at least 1
other_config : burst        optional string, containing an integer, at least 1
other_config : priority     optional string, containing an integer,
                            in range 0 to 4,294,967,295

# Configuration for linux-hfsc QoS:
other_config : min-rate     optional string, containing an integer, at least 1
other_config : max-rate     optional string, containing an integer, at least 1

# Common Columns:
other_config                map of string-string pairs
external_ids                map of string-string pairs
```

Figura 2.9: Queue OVSDB [20]

La calidad de servicio (QoS) en OpenFlow es compatible con dos características principales de OVS: las colas y la tabla de medidores. Las colas son mecanismos de salida de paquetes en los puertos del *switch* OpenFlow. La compatibilidad con colas se introdujo en OpenFlow 1.0, con una configuración inicial que permitía solo una tasa mínima garantizada. Posteriormente, en OpenFlow 1.2, se añadió la posibilidad de establecer una tasa máxima, lo que permite limitar el rendimiento de una cola. Aunque las colas están definidas en la especificación de OpenFlow, el protocolo no gestiona su administración. La gestión de colas (creación, eliminación y modificación) se realiza mediante protocolos de configuración de *switches*, como OF-Config o la base de datos *Open vSwitch* (OVSDB). OpenFlow en sí mismo solo puede consultar estadísticas de las colas en el *switch* [21].

La tabla de medidores es una característica introducida en OpenFlow 1.3, que permite monitorizar la tasa de ingreso de un flujo y realizar operaciones basadas en esa tasa. Estas operaciones pueden incluir el descarte de paquetes o la modificación de los bits DSCP de los paquetes. A diferencia de las colas, que son propiedades de los puertos del *switch* , los medidores se adjuntan a las entradas de flujo [21].

En algunos estudios, se utilizan colas en OpenFlow para asegurar un ancho de banda específico. Para cada flujo que requiere QoS, se crea una cola en el *switch* de entrada y en los *switches* intermedios, estableciendo una velocidad mínima y una máxima igual al ancho de banda garantizado para dicho flujo. Sin embargo, este enfoque tiene problemas de escalabilidad, ya que el número de colas aumenta con cada nuevo flujo. Para mejorar la escalabilidad, se propone una estrategia de agregación. En lugar de asignar una cola independiente a cada flujo de QoS, se utiliza una sola cola compartida para reenviar todos los flujos de QoS a través de un puerto del *switch* . La velocidad de esta cola es dinámica y se ajusta en función de la suma de los anchos de banda de todos los flujos de QoS que se transmiten por ese puerto. Sin embargo, el límite máximo de cada flujo de QoS se mantiene en su tasa garantizada. Si un flujo de QoS excede este límite, podría competir por el ancho de banda disponible con otros flujos de QoS. [21].

A continuación se muestra una configuración básica de OVS mediante comandos para ejemplificar la definición de colas. Cabe destacar que la implementación de algoritmos de QoS que se presentarán en la siguiente sección se realiza desde el controlador. Inicialmente se puede consultar el las instancias de colas

```
/ # tc qdisc show
qdisc noqueue 0: dev lo root refcnt 2
qdisc fq_codel 0: dev eth0 root refcnt 2 limit 10240p flows 1024 quantum 1514 target 5ms
    interval 100ms memory_limit 32Mb ecn drop_batch 64
qdisc fq_codel 0: dev eth1 root refcnt 2 limit 10240p flows 1024 quantum 1514 target 5ms
    interval 100ms memory_limit 32Mb ecn drop_batch 64
qdisc fq_codel 0: dev eth2 root refcnt 2 limit 10240p flows 1024 quantum 1514 target 5ms
    interval 100ms memory_limit 32Mb ecn drop_batch 64
qdisc fq_codel 0: dev eth3 root refcnt 2 limit 10240p flows 1024 quantum 1514 target 5ms
    interval 100ms memory_limit 32Mb ecn drop_batch 64
qdisc fq_codel 0: dev eth4 root refcnt 2 limit 10240p flows 1024 quantum 1514 target 5ms
    interval 100ms memory_limit 32Mb ecn drop_batch 64
```

Figura 2.10: Enumeración de colas qdisc instanciadas

```
ovs-vsctl -- \
  add-br br0 -- \
  add-port br0 eth1 -- \
  add-port br0 eth2 -- \
  add-port br0 eth3 -- \
  add-port br0 eth4 -- \
  set port eth3 qos=@newqos -- \
  --id=@newqos create qos type=linux-htb \
      other-config:max-rate=50000000 \
      queues:1=@10queue \
      queues:2=@20queue -- \
  --id=@10queue create queue other-config:max-rate=20000000 -- \
  --id=@20queue create queue other-config:max-rate=10000000
```

Figura 2.11: Definición de cola de tasa máxima en OVS

```
/ # tc qdisc show
qdisc noqueue 0: dev lo root refcnt 2
qdisc fq_codel 0: dev eth0 root refcnt 2 limit 10240p flows 1024 quantum 1514 target 5ms
    interval 100ms memory_limit 32Mb ecn drop_batch 64
qdisc fq_codel 0: dev eth1 root refcnt 2 limit 10240p flows 1024 quantum 1514 target 5ms
    interval 100ms memory_limit 32Mb ecn drop_batch 64
qdisc fq_codel 0: dev eth2 root refcnt 2 limit 10240p flows 1024 quantum 1514 target 5ms
    interval 100ms memory_limit 32Mb ecn drop_batch 64
qdisc htb 1: dev eth3 root refcnt 2 r2q 10 default 0x1 direct_packets_stat 0 direct_qlen
    1000
qdisc fq_codel 0: dev eth4 root refcnt 2 limit 10240p flows 1024 quantum 1514 target 5ms
    interval 100ms memory_limit 32Mb ecn drop_batch 64
```

Figura 2.12: Configuración de esquema de colas

configuradas en el switch tal como se muestra en la Figura 2.10.

Posteriormente, de acuerdo a la documentación de *Open vSwitch*, la configurar QoS sobre un interfaz de un switch OVS se realizaría como se muestra en la Figura 2.11.

Con esta configuración se está aplicando un QoS sobre el interfaz físico eth3 con una tasa máxima para el tráfico de salida de 50 Mbps. Posteriormente, se han definido dos colas asociadas a ese interfaz: una de 10 Mbps y otra de 20 Mbps. La configuración resultante se muestra en la Figura 2.12.

Una vez definidas las colas en OVS, se puede utilizar OpenFlow para asociar flujos de tráfico a esas colas, controlando de esta manera el uso de los recursos. A continuación se muestra la definición de dos entradas OpenFlow en la tabla por defecto (tabla 0) de forma que el tráfico de entrada de los puertos eth1 y eth2 se envía por el puerto eth3 a través de las colas 1 y 2 respectivamente:

```
$ ovs-ofctl add-flow br0 in_port=eth1,actions=set_queue:1,normal
```

```
/ # ovs-appctl qos/show eth3
QoS: eth3 linux-htb
max-rate: 50000000

Default:
    burst: 12512
    min-rate: 12000
    max-rate: 50000000
    tx_packets: 17
    tx_bytes: 2881
    tx_errors: 0

Queue 1:
    burst: 12512
    min-rate: 12000
    max-rate: 20000000
    tx_packets: 4061
    tx_bytes: 5892624
    tx_errors: 510905

Queue 2:
    burst: 12512
    min-rate: 12000
    max-rate: 10000000
    tx_packets: 0
    tx_bytes: 0
    tx_errors: 0

/ # tc -s qdisc show dev eth3
qdisc htb 1: root refcnt 2 r2q 10 default 0x1 direct_packets_stat 0 direct_qlen 1000
 Sent 5895715 bytes 4081 pkt (dropped 510905, overlimits 3950 requeues 0)
 backlog 0b 0p requeues 0
```

Figura 2.13: Validación de comportamiento

```
$ ovs-ofctl add-flow br0 in_port=eth2,actions=set_queue:2,normal
```

Además de la redirección a la cola se define la acción *normal* añadiendo la configuración de *switching* que el *switch* OVS crea por defecto cuando no existe ninguna otra entrada en la *flow-table*.

En la Figura 2.13 se ha usado la configuración de OVS mostrada anteriormente y se han realizado pruebas de capacidad con iperf. Se usan los comandos *ovs-appctl qos/show eth3* y *tc -s qdisc show dev eth3* para obtener las estadísticas de OVS. En este último se pueden ver el tamaño del *buffer* de la cola *direct_qlen 1000*, donde el tamaño del *buffer* es de 1000 paquetes. Además, en el *backlog* se puede ver cómo los paquetes van llenando la cola cuando la tasa excede el ancho de banda asignado a la cola.

Capítulo 3

Scheduler basado en la ocupación de los interfaces y los retardos acumulados

Para el desarrollo de este proyecto se ha elegido *Ryu* como plataforma para el diseño e implementación del controlador SDN. La elección de *Ryu* se basa principalmente en su sencillez, flexibilidad y en el uso de Python como lenguaje de programación, lo cual facilita la creación y personalización de un entorno de desarrollo adecuado para evaluar el funcionamiento y comportamiento real de un controlador SDN. La estructura modular de *Ryu* permite una rápida adaptación, ya que proporciona herramientas fáciles de usar para la gestión de flujos y la integración con el protocolo OpenFlow. Además, la arquitectura de *Ryu* permite una fácil depuración y modificación del código, lo que resulta fundamental en etapas de prueba y ajuste fino del sistema.

Aunque en la Sección 2.2.1 se mencionó que *Ryu* podría no ser la opción más adecuada para entornos de producción a gran escala (donde se gestionan cientos de dispositivos y donde la latencia es un factor crítico para el procesamiento y aplicación eficiente de reglas OpenFlow), su uso es perfectamente válido y efectivo para el propósito de este proyecto. En contextos académicos, de investigación o pruebas controladas, *Ryu* ofrece un entorno sencillo para experimentar con algoritmos y validar propuestas técnicas, como es el caso del algoritmo de planificación (*scheduler*) que se aborda en este trabajo, cuyos fundamentos matemáticos se presentan más adelante en este documento (Sección 3.1).

A lo largo de este capítulo se detallarán los aspectos más relevantes de la configuración y parametrización del controlador, así como las decisiones tomadas durante su implementación. También se realizará un análisis de las características y limitaciones de *Ryu*, poniendo en contexto su aplicabilidad dentro del ámbito del proyecto y su impacto en el desempeño general del sistema SDN que se propone.

Antes de profundizar en estos aspectos técnicos, es importante describir el entorno de pruebas empleado para la validación del proyecto. Las simulaciones y pruebas se llevaron a cabo utilizando la herramienta de emulación Graphical Network Simulator-3 (GNS3), que permite la creación de redes mediante entornos virtuales, facilitando la emulación de escenarios reales de red y la interacción directa con el controlador SDN.

El entorno de emulación se ha ejecutado en una máquina virtual (VM) basada en el sistema operativo Lubuntu 20.04 (una versión ligera de Ubuntu), seleccionada por su bajo consumo de recursos y su estabilidad en entornos virtualizados. Para garantizar un rendimiento adecuado durante las pruebas y simular condiciones similares a las de un entorno cloud, la máquina virtual fue configurada con los siguientes recursos: 10 vCPU asignadas, permitiendo el procesamiento paralelo de múltiples flujos de red y la gestión eficiente de las tareas del controlador y 8 GB de memoria Random Access Memory (RAM), proporcionando la capacidad necesaria para gestionar la carga de trabajo generada durante las simulaciones de tráfico de red.

Para los escenarios creados en GNS3 se diseñaron diferentes condiciones de red para evaluar tanto el funcionamiento del controlador como la eficacia de los algoritmos de *scheduling* desarrollados. Las pruebas incluyeron la gestión de flujos, la aplicación de políticas de calidad de servicio (QoS) y la respuesta del sistema ante variaciones en la carga de tráfico, asegurando así una validación completa del proyecto en un entorno controlado. Además, permitió evaluar el rendimiento del controlador SDN bajo condiciones de red más aleatorias para proporcionar una visión más realista sobre su comportamiento y capacidad de adaptación.

A lo largo de este capítulo se detalla la estructura del controlador y las técnicas empleadas para la implementación de los algoritmos propuestos, proporcionando una visión integral del proceso de desarrollo y validación del sistema.

3.1. Fundamentos teóricos del *scheduler*

En esta sección se presenta el modelo del sistema y el planteamiento general de la política de *scheduling* que se basa en la teoría de control de Lyapunov a través del marco teórico desarrollado por Neely [22].

Tabla 3.1: Definiciones y símbolos

Símbolo	Definición
	Variables aleatorias
$a_m(t)$	Llegadas a la cola del servicio m en el slot t (bytes).
$b_m(t)$	Salidas de la cola del servicio m en el slot t (bytes).
$w_n(t)$	Capacidad de transmisión en el slot t (bytes/slot)
$Q_m(t)$	Valor de la cola del servicio m en el slot t
M	Número de servicios.
$\alpha_{i,j}(t)$	Decisión para transferir tráfico entre las colas i y j (bytes).
$W_{i,j}$	Diferencia entre la métrica de las colas i y j

La Tabla 3.1 proporciona un resumen de los símbolos empleados en el modelo junto con sus respectivas definiciones. Se asume que en cada *switch* hay M servicios que generan tráfico de forma independiente. El tiempo está ranurado, de forma que en cada ranura (o *slot*) cada uno de los *switches* recibe una cantidad de tráfico (medido en bytes) de cada uno de los servicios y toma una decisión.

La decisión, tomada por el controlador, permite modificar la cantidad de tráfico que se drena de cada cola, a fin de asegurar la estabilidad de las mismas. En nuestro caso, las decisiones consisten en modificar la capacidad de transmisión dedicada a cada uno de los servicios en los *switches*.

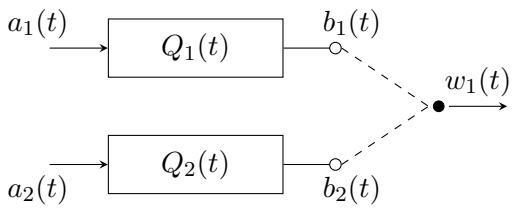


Figura 3.1: Modelo del sistema con 2 servicios y en interfaz de salida

La Figura 3.1 muestra un ejemplo de un *switch* con colas para dos servicios y un interfaz de salida. En cada *slot* t, el tráfico generado por un servicio k se indica como $a_k(t)$. Este tráfico se almacena en la cola $Q_k(t)$, cuya ocupación se actualiza en cada *slot*. Por otro lado, en cada ranura una cantidad de tráfico $b_k(t)$ abandona la cola, como consecuencia de la decisión tomada y, potencialmente, de otras circunstancias sobre las que no se tiene control. De este modo, en cada *slot* t la ocupación de las colas se actualiza como se indica en la Ec. (3.1):

$$Q_k(t+1) = \text{máx}[Q_k(t) - b_k(t), 0] + a_k(t) \tag{3.1}$$

El objetivo principal es asegurar la estabilidad promedio, para lo que se hace uso de la métrica *Mean Rate Stability* que se define como sigue:

$$\lim_{T\to\infty} \frac{1}{T}\,\mathbb{E}\{Q_k(t)\} = 0 \tag{3.2}$$

siendo $Q_k(t)$ el tamaño de la cola en el *slot* t, y $\mathbb{E}$ la esperanza matemática. Existe una métrica de estabilidad más estricta llamada *Strong Stability*, que se define en la Ec. (3.3):

$$\lim_{T\to\infty} \frac{1}{T} \sum_{\tau=1}^{\tau=T} \mathbb{E}\{Q_k(\tau)\} \leq \infty \tag{3.3}$$

El marco teórico adoptado en este trabajo asegura que se alcanza *Mean Rate Stability* y en la práctica permite tener *Strong Stability* en la mayoría de los casos. Teniendo en cuenta lo anterior, en cada *slot* se toma una decisión $\alpha(t)$ dentro de un conjunto de posibilidades $\mathrm{A}(t)$ que modifica la cantidad de tráfico que abandona las colas en el *slot*. De este modo, se puede definir la salida de la cola m, $b_m(t)$, como una función de $\alpha(t)$ como se muestra en la Eq.(3.4), donde $\hat{b}$ indica una función genérica. Además, se añade la variable $\omega(t)$ que modela otros mecanismos aleatorios que pudieran influir en la cantidad de tráfico drenado. Este parámetro podría representar limitaciones en la capacidad de transmisión o escritura en memoria.

$$b_m(t) = \hat{b}(\alpha(t), \omega(t)) \tag{3.4}$$

Por simplicidad se asumirá que la decisión tomada es directamente el tráfico saliente. Sin embargo, para evitar asignar más tráfico que el existente en las colas, la función $\hat{b}$ como se indica como se muestra en la Ec.(3.5). Esta función se puede plantear como una restricción en el espacio de decisiones en cada *slot*, de forma que se asume que $\mathrm{A}(t)$ excluye las decisiones que implican sacar más tráfico de una cola que el contenido en la misma.

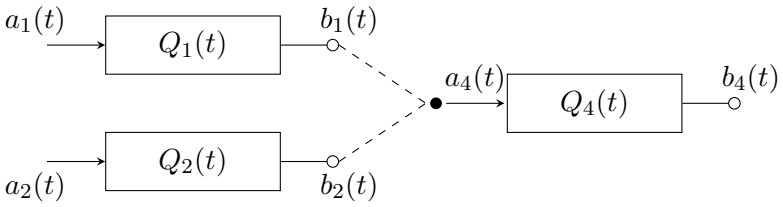


Figura 3.2: Ejemplo modelo del sistema con 2 servicio y 1 interfaz

$$b_m(t) = \hat{b}(\alpha(t), \omega(t)) = \text{máx}[Q_m(t), \alpha(t)] \quad \forall m \in \{1, \cdots, M\} \tag{3.5}$$

En redes con varios saltos o sistemas de colas conectados, la estabilización de colas se puede alcanzar mediante el algoritmo *backpressure*, tal como se indica en [22]. En general consideramos un modelo de sistema con M servicios y N interfaces. A modo de ejemplo, en la Figura 3.2 se ilustra un sistema con 2 servicios y un interfaz.

En cada slot el algoritmo observa las colas y selecciona los valores de $\alpha(t)$ que maximice la Ec. (3.6).

$$\underset{\alpha(t)}{\text{máx}} \quad \sum_{i,j} b_{i,j}(t) \cdot W_{ij}(t) \tag{3.6}$$

donde $W_{ij}(t)$ se calcula como:

$$W_{ij}(t) = Q_i(t) - Q_j(t) \tag{3.7}$$

Aunque la métrica basada en ocupación de los *buffer* es la más natural, el significado de cola dentro del esquema aceptado no está limitado a una cola física. En este sentido, existen trabajos que han utilizado retardos como medidas de la ocupación de las colas [23, 24]. Como se indicará en las siguientes secciones, se han implementado tanto un algoritmo basado en ocupación de colas, como en medidas de retardo (máximo y agregado). En todos los casos, merece la pena destacar que el marco teórico adoptado asegura la estabilidad de las colas incluso si las decisiones son sub-óptimas, siempre y cuando la diferencia entre la decisión tomada y la óptima esté acotada.

3.2. Diseño e implementación del controlador con *Ryu*

Para comprender el funcionamiento del controlador, se parte del escenario de la Figura 3.3. En este escenario disponemos de un *vSwitch* OVS en el que se han configurado 3 interfaces para la gestión del tráfico de diferentes servicios.

El diagrama de la Figura 3.3 representa el esquema que se utilizará para validar la implementación del controlador *Ryu* y también para la medición de los resultados finales, que se desarrollarán a lo largo del Capítulo 4. El diagrama representa un *vSwitch* OVS y sus interfaces con sus respectivos *buffers*, y un controlador que envía configuraciones de control de calidad de servicio sobre el mismo, aplicando políticas de QoS basadas en decisiones con origen en el algoritmo de *scheduling* planteado.

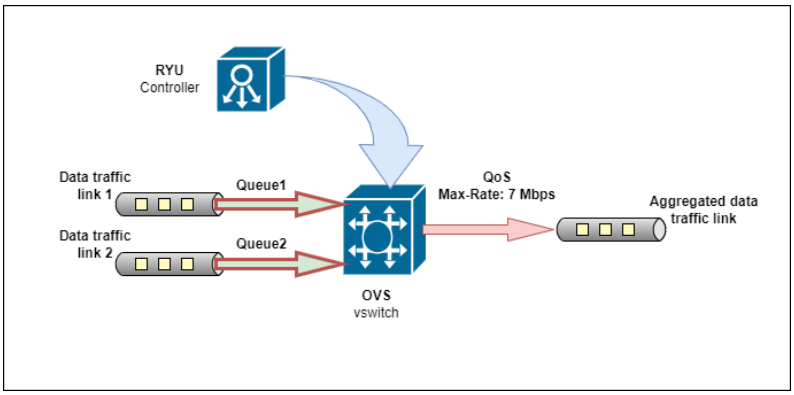


Figura 3.3: Escenario de diseño

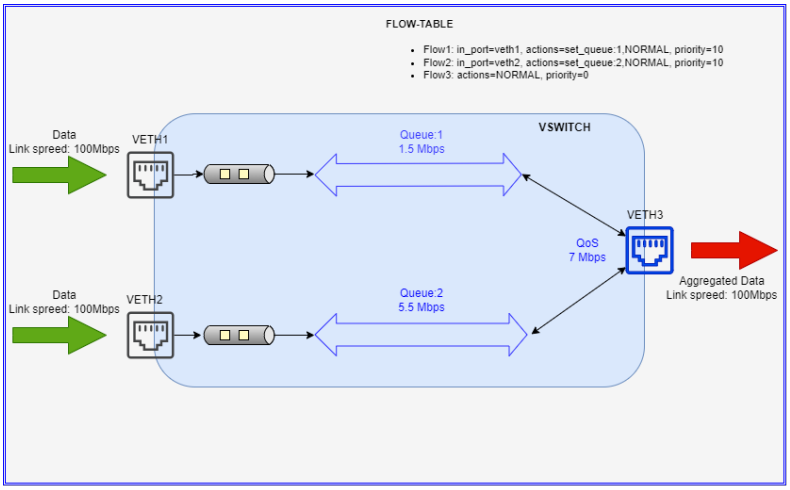


Figura 3.4: Diagrama *vSwitch* OVS asignación estática

Para comprender mejor qué es lo que sucede cuando el controlador interactúa con OVS, se puede ver representado qué es lo que está sucediendo realmente dentro del *vSwitch* en la Figura 3.4. Se dispone de un *vSwitch* OVS con 3 interfaces y una primera configuración QoS estática. Los dos primeros interfaces reciben tráfico a ráfagas y el tercero transmite el tráfico agregado de los dos primeros. En este esquema, no existe aún interacción alguna con un controlador SDN, sino que se aplican directamente configuraciones de QoS y entradas manuales en la tabla de flujos (*flow-table*) de OVS. La configuración estática, crea una cola o *queue* para cada uno de los interfaces del *vSwitch*, limitando la capacidad máxima mediante una *egress queue*. Esta configuración y el estudio de su comportamiento, basados en lo descrito a lo largo del Capítulo 2 (más concretamente en la Sección 2.3.2), se encuentran explicados con más detalle en el Anexo A. Lo que hará el controlador será modificar los parámetros que se requieran usando el protocolo OVSDB. Concretamente estableciendo el *max-rate* de el QoS y las *queues* asociadas e introducir cambios en la *flow-table* mediante OpenFlow.

El controlador tiene dos funciones principales: monitorizar en tiempo real la ocupación y el retardo de los paquetes acumulados en los *buffers* de los interfaces del *vSwitch* y aplicar políticas de calidad de servicio (QoS) dinámicamente, basándose en el algoritmo de *scheduling* (cuyos fundamentos ya se han descrito en la Sección 3.1). El controlador será, por tanto, el encargado de tomar decisiones y aplicar dichas reglas. Para ello, deberá calcular la ocupación de los interfaces y registrar las marcas de tiempo de los paquetes que se van acumulando en los *buffers*, y basándose en el algoritmo, aplicar las políticas de QoS, otorgando

al tráfico de los diferentes servicios una capacidad ajustada a la máxima capacidad disponible en el enlace agregado, tratando de impedir que ningún servicio se quede sin capacidad, evitando saturación de la red y pérdida de paquetes.

El controlador *Ryu* opera de forma asíncrona y utiliza OpenFlow para comunicarse con OVS a través de un mecanismo de suscripción a eventos. Gracias a este enfoque, el controlador puede detectar dinámicamente los *vSwitches* conectados, ya que tanto las direcciones IP de los *vSwitches* como la del controlador son conocidas de antemano. El controlador asigna a cada *vSwitch* un identificador único (*datapath id*), lo que permite gestionar varios dispositivos, permitiendo la escalabilidad en la capacidad de gestión del mismo.

Mediante OpenFlow cada *vSwitch* reporta periódicamente su estado al controlador, proporcionando información sobre la configuración y definición de los interfaces de red, el estado y métricas de ocupación de los *buffers* de los puertos y la gestión y actualización de las tablas de flujo. Con estos datos, el controlador obtiene una visión en tiempo real de la carga de tráfico en cada interfaz, la ocupación de los *buffers* y el retardo acumulado de los paquetes. Basándose en esta información, se implementa el algoritmo de *scheduling* que administra la calidad de servicio (QoS) en la red.

Para simular tráfico de red a lo largo de todo el proyecto se empleará *iperf3*, herramienta que nos permite generar, tanto tráfico TCP como UDP, bajo diferentes condiciones y así poder verificar cómo se comporta el controlador con diferentes cargas y tipos de tráfico. Concretamente, para este proyecto, se generará tráfico UDP con la intención de tener control sobre el tráfico y poder ver con más claridad el comportamiento del controlador, ya que con TCP entraría también en juego el control de congestión. Para poder hacer un estudio del funcionamiento del *scheduler* propuesto para este proyecto se realizan varias pruebas para comprobar su funcionamiento y también su comportamiento en situaciones límite (buscando los límites de operación del controlador). Estas pruebas consisten en generar tráfico UDP mediante *iperf3* y se analiza su comportamiento. Se analizarán tres puntos clave en esta sección:

- En la Sección 3.2.1, se van a tratar las decisiones de implementación, donde se detallan las estrategias adoptadas para integrar el *scheduler* dentro del controlador *Ryu* y su interacción con OVS. En este apartado se abordan aspectos clave como la elección del modelo de comunicación asíncrona mediante OpenFlow, la definición de los criterios de decisión para la integración del algoritmo de *scheduling* (que se pondrán a prueba en el Capítulo 4), y la estructura de las reglas de flujo aplicadas sobre OVS, describiendo la manera en que se asignan dinámicamente, los recursos de ancho de banda en función de la demanda de tráfico.

- Análisis de la frecuencia con la que el controlador toma decisiones y actualiza las estadísticas de ocupación y retardos de los paquetes en los *buffers*. En un sistema SDN, la capacidad de reacción del controlador ante cambios en la red depende de la periodicidad con la que recopila métricas y su capacidad de ajustar la asignación de recursos. Una frecuencia demasiado alta puede generar sobrecarga en el controlador, afectando su capacidad de procesamiento y la estabilidad del sistema. Por otro lado, una frecuencia demasiado baja podría hacer que las decisiones de asignación de ancho de banda no reflejen con precisión las condiciones actuales del tráfico, lo que podría generar congestión o una distribución ineficiente de los recursos. En este apartado se analiza cómo la periodicidad de actualización de estadísticas afecta la estabilidad y eficiencia del sistema, así como el equilibrio entre la latencia de respuesta y el consumo de recursos computacionales (Sección 3.2.2). Además,

se analizan las limitaciones identificadas durante el desarrollo y las posibles mejoras que podrían implementarse para optimizar el rendimiento del sistema.

- Finalmente en la Sección 3.2.3, se hará un estudio de la pérdida de rendimiento en cuanto a *bitrate* se refiere, asociada a la toma de decisiones. Cada vez que el controlador SDN ajusta las reglas de calidad de servicio (QoS), puede producirse una interrupción momentánea en la transmisión del tráfico debido a los tiempos de convergencia en la actualización de las reglas de enrutamiento. Es importante cuantificar en qué medida este proceso afecta al rendimiento de la red y si se generan pérdidas de paquetes significativas o caídas temporales en el *throughput*. En este análisis se busca determinar si la estrategia de asignación de recursos minimiza estas pérdidas y si es necesario optimizar los tiempos de convergencia del sistema.

3.2.1. Implementación de los algoritmos de *scheduling*

A continuación se exponen los diferentes comportamientos del *vSwitch*, gestionado por el controlador mediante el *scheduler* propuesto, ante diferentes criterios de decisión. Cuando el controlador toma una decisión, crea un nuevo QoS y asocia a cada interfaz una cola (*egress queue*) que permite repartir la capacidad agregada en base a dos criterios fundamentales:

1. **Ocupación de los *buffers* de los interfaces**: este es el criterio principal a la hora de implementar el controlador y se ha realizado de dos maneras diferentes. Como se mencionó en la Sección 3.1, el marco teórico asegura estabilidad de las colas (en este caso ocupación de *buffers*) incluso si las decisiones no son óptimas, pero el error cometido está acotado. Las dos opciones que se comentan a continuación son decisiones sub-optimas desde el punto de vista teórico pero que presentan beneficios prácticos. La primera se basa en la diferencia relativa entre la ocupación de los interfaces (pseudocódigo disponible en la Figura 3.5). Este criterio ofrece una respuesta rápida ante diferencias abruptas en la carga, ya que se ajusta más agresivamente cuando una cola tiene mucha más carga que la otra. Este enfoque nos permite crear una lista de posibles capacidades para cubrir por rangos; es decir, se define un conjunto de posibles capacidades para cada uno de los flujos (que en su conjunto suma la capacidad agregada) basándose en esa diferencia de ocupación. La segunda se trata de un balanceo de carga con reparto equitativo y superado un umbral, de tal forma que se reparte la capacidad del enlace agregado de manera proporcional al nivel de ocupación de los interfaces. El balanceo proporcional dinámico es la mejor opción, ya que ajusta el ancho de banda en función del tráfico total y mantiene una distribución proporcional. Se define el pseudocódigo en la Figura 3.6.

2. **El retardo de los paquetes en los *buffers***: en este caso se hace uso de métricas sintéticas de ocupación de las colas basadas en el retardo, de forma que el *scheduler* asegura estabilidad de esta métrica de retardo. Para ello se genera un registro, en tiempos absolutos, del tiempo que llevan los paquetes esperando, lo que nos permite tomar decisiones no solo basándose en la ocupación, sino también en la antigüedad de los paquetes. En la implementación se han calculado dos factores que se pueden incluir como criterio de decisión: retardo acumulado (Figura 3.7) y retardo máximo.

Estos son los dos criterios de decisión a los que se va a recurrir a la hora de aplicar QoS mediante el controlador. Es importante añadir que esta es otra de las claras ventajas que ofrece el uso de controladores

```
INICIO
DEFINIR RANGOS_DIFERENCIA  [20%, 40%, 80%]
DEFINIR MAX_QOS_RATE  7000000

MIENTRAS VERDADERO HACER
    PARA CADA SWITCH EN LA RED HACER
        OBTENER avg_queue_1, avg_queue_2

        SI avg_queue_1 > avg_queue_2 ENTONCES
            diferencia:  1 - (avg_queue_2 / avg_queue_1)
        SINO
            diferencia:  1 - (avg_queue_1 / avg_queue_2)
        FIN SI

        SI diferencia < 20% ENTONCES
            queue_config:  [50%, 50%]
        SINO SI diferencia < 40% ENTONCES
            queue_config:  [60%, 40%]
        SINO SI diferencia < 80% ENTONCES
            queue_config:  [70%, 30%]
        SINO
            queue_config:  [90%, 10%]
        FIN SI

        APLICAR CONFIGURACION DE QoS (queue_config)
        REGISTRAR DECISION EN LOG
    FIN PARA
    ESPERAR INTERVALO_DE_ACTUALIZACION
FIN MIENTRAS
FIN
```

Figura 3.5: Pseudocódigo - diferencia relativa entre la ocupación *buffers*

```
INICIO
DEFINIR UMBRAL
DEFINIR MAX_QOS_RATE 7000000

MIENTRAS VERDADERO HACER
    PARA CADA SWITCH EN LA RED HACER
        OBTENER avg_queue_1, avg_queue_2
        total_carga:  avg_queue_1 + avg_queue_2

        SI total_carga > UMBRAL ENTONCES
            porcentaje_1:  avg_queue_1 / total_carga
            porcentaje_2:  avg_queue_2 / total_carga

            queue_config: [porcentaje_1 * MAX_QOS_RATE, porcentaje_2 * MAX_QOS_RATE]
        SINO
            queue_config: [50%, 50%]
        FIN SI

        APLICAR CONFIGURACION DE QoS (queue_config)
        REGISTRAR DECISION EN LOG
    FIN PARA
    ESPERAR INTERVALO_DE_ACTUALIZACION
FIN MIENTRAS
FIN
```

Figura 3.6: Pseudocódigo - reparto proporcional de la capacidad agregada en función de la ocupación de los *buffers*

```
INICIO
DEFINIR MAX_QOS_RATE 7000000
DEFINIR OFFSET
    MIENTRAS VERDADERO HACER
        PARA CADA SWITCH EN LA RED HACER
        SI SWITCH REGISTRADO EN sw_dp_id ENTONCES
            BLOQUEAR mutex_lock
            OBTENER total_delay_q1, total_delay_q2, max_delay_q1, max_delay_q2
            DESBLOQUEAR mutex_lock

            SI total_delay_q1 = 0 Y total_delay_q2 = 0 ENTONCES
                queue_config: [50 %, 50 %]
            SINO
                total: total_delay_q1 + total_delay_q2
                aporcentaje_1: total_delay_q1 / total
                porcentaje_2: total_delay_q2 / total

                queue_config: [porcentaje_1 * MAX_QOS_RATE, porcentaje_2 * MAX_QOS_RATE]
            FIN SI

            APLICAR CONFIGURACION DE QoS (queue_config)
            REGISTRAR DECISION EN LOG
        FIN PARA
        ESPERAR INTERVALO_DE_ACTUALIZACION
    FIN MIENTRAS
FIN
```

Figura 3.7: Pseudocódigo - reparto proporcional de la capacidad agregada basada en el retardo acumulado

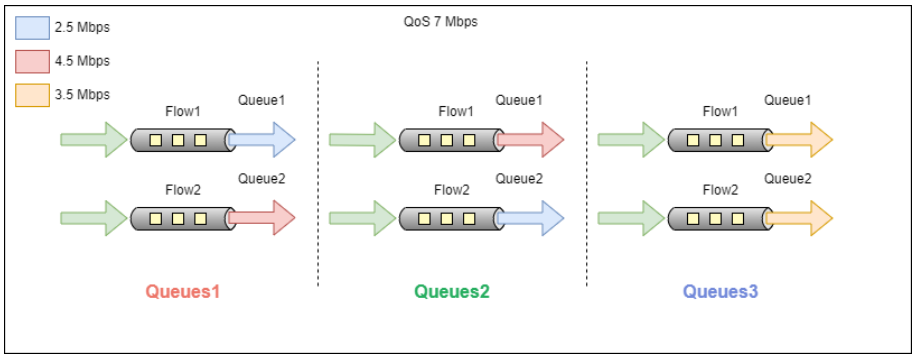


Figura 3.8: Diferencia relativa entre los *buffers* - Asignación de QoS

SDN, que permite una respuesta proactiva adaptándose al tráfico en base a diferentes métricas (ocupación y retardos) con el objetivo de lograr el mejor *throughput* con la menor latencia posible.

A fin de validar que las implementaciones propuestas dan lugar al comportamiento esperado, a continuación se muestran unos resultados iniciales sobre un escenario controlado (Figura 3.3). Aunque más adelante durante el Capítulo 4 se profundiza más en el uso de los diferentes *schedulers* dependiendo del criterio de decisión, en esta primera prueba se pretende mostrar unos primeros resultados para validar el funcionamiento del controlador. Para ello, la implementación propuesta será de un *scheduler* que toma decisiones basándose en la diferencia relativa de la ocupación de los *buffers*.

En la Figura 3.8 se muestran tres posibles elementos de un conjunto de decisiones para los dos flujos de tráfico. Cada flujo tiene asociado su correspondiente cola (*egress queue*) y recibirá una asignación de ancho de banda impuesta por la política de QoS. Cada flujo se corresponde con un interfaz del *vSwitch* y el controlador tomará una decisión cada t_s segundos (tiempo de *slot*) basándose en la ocupación del *buffer* de ambos interfaces. El controlador se encarga de recoger estadísticas de ocupación del *buffer* cada t_{stats} segundos y almacena los últimos n valores de ocupación obtenidos. Finalmente, el valor de ocupación se define con la media aritmética de los n valores, con el objetivo de no tomar decisiones basadas solo en el último registro. Si la diferencia relativa está por debajo del 20 %, el controlador asignará una tasa de 3.5

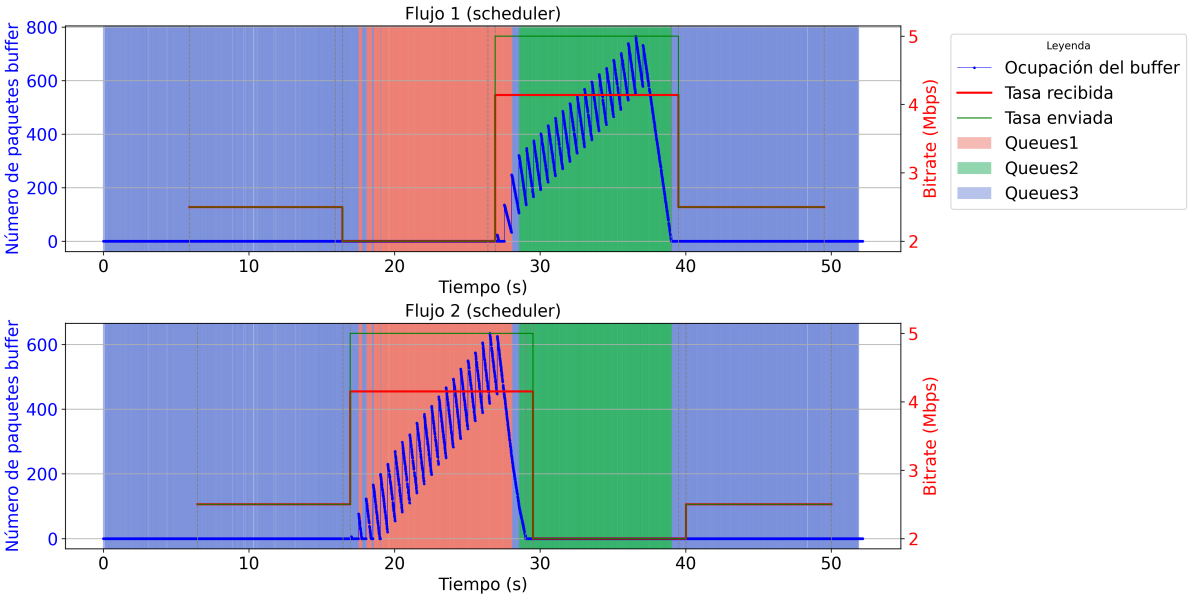


Figura 3.9: diferencia relativa entre los *buffers* - Asignación de QoS evolución temporal

Tabla 3.2: Configuración de los tests *iperf* para estudio del comportamiento de Ryu

Parámetro	**Flujo 1**	**Flujo 2**
Número de tests	5	
Duración por test (s)	5	
Bitrate test 1 (Mbps)	2.5	2.5
Bitrate test 2 (Mbps)	2	5
Bitrate test 3 (Mbps)	5	2
Bitrate test 4 (Mbps)	2.5	2.5

Mbps a cada flujo, definido como el elemento *Queues3* dentro del conjunto de decisiones de la Figura 3.8; es decir, se asume que la diferencia de ocupación no es significativa como para plantear otra solución que no sea un reparto equitativo de la capacidad. Si la diferencia es mayor del 20 %, la asignación será de 4.5 *Mbps* y 2.5 *Mbps* respectivamente a favor del *buffer* más ocupado que se corresponderían con *Queues1* y *Queues2* respectivamente.

Una vez descrito el funcionamiento, y con el objetivo de visualizar el comportamiento del controlador a la hora de asignar la capacidad a las colas basándose en la ocupación de los *buffers* empleando como criterio de decisión la diferencia relativa, se ha generado el tráfico *iperf* según lo especificado en la Tabla 3.2. El tráfico se genera con la intención de forzar al controlador a tomar una decisión predecible dentro del conjunto de decisiones de la Figura 3.8.

Los resultados del experimento se muestran en la Figura 3.9. Aquí lo que se puede observar es la evolución de los flujos en el tiempo y los intervalos sombreados de diferente color para el conjunto de posibles decisiones del controlador que se corresponden con *Queues1*, *Queues2* y *Queues3* de la Figura 3.8. Como era de esperar, durante los primeros segundos, ambos flujos tratan de transmitir por debajo de la capacidad y la ocupación de los *buffer* es cero, lo que se traduce en que el controlador asigna la misma capacidad a ambos flujos. Cuando uno de los flujos aumenta la tasa y el *buffer* comienza a crecer, el controlador favorece al flujo del *buffer* más ocupado, asignándole más capacidad.

En esta primera configuración e implementación del *scheduler* que se ha realizado, se muestra de manera bastante sencilla y visual el funcionamiento del controlador. Aunque se podría haber optado por otra decisión de diseño para este ejemplo de validación, por ejemplo, aumentando el posible conjunto de decisiones para asignar diferentes límites de capacidad a los flujos, o usar como criterio el reparto proporcional de la capacidad agregada, que es una solución más eficiente tanto para cumplir con la tasa que demanda el servicio, el objetivo es introducir y validar el funcionamiento del controlador.

3.2.2. Impacto de la frecuencia en la actualización de estadísticas y toma de decisiones en el rendimiento del controlador

En esta sección se profundiza en el análisis del comportamiento del controlador, enfrentándolo a situaciones que permiten encontrar sus límites. Estos límites pueden venir dados o bien por decisiones tomadas a la hora de implementar el controlador, por límites de las propias herramientas empleadas (OVS y *Ryu* en este caso) o bien por los límites de procesamiento de las propias herramientas de virtualización empleadas para simular los experimentos.

Para buscar estos límites, se configuran los parámetros del controlador que nos permiten localizar los rangos dentro de los cuales su funcionamiento entra dentro de lo esperado y fuera de ellos su rendimiento y/o comportamiento se aleja de lo deseado. No solo se trata de buscar sus fallos y límites, sino también de buscar aquellas configuraciones que nos permitan optimizar el funcionamiento del controlador. Concretamente, este análisis se centra en dos aspectos: respuesta del controlador frente a diferentes patrones de tráficos y frecuencia de actualización datos y toma de decisiones del controlador. Para realizar las diferentes pruebas, se emplea el escenario en GNS3 de la Figura 3.3.

En este apartado se realiza un análisis de rendimiento, ya que tal y como se ha expuesto durante el Capítulo 2, una de las características esenciales a la hora de elegir un controlador, es el rendimiento que este puede proporcionar a la hora de gestionar varios dispositivos, en la que entran en juego factores como latencia, capacidad de procesamiento, etc. Otro de los aspectos que tendremos en cuenta a la hora de implementar el controlador *Ryu* es la frecuencia con la que este es capaz de tomar decisiones.

El controlador *Ryu* consiste en una aplicación multihilo en Python, en la que se ejecutan dos procesos simultáneamente, cada uno con una frecuencia predefinida y que se puede parametrizar. Uno de los hilos se ocupa de hacer la lectura de la ocupación de los *buffers* de los interfaces del *vSwitch* manteniendo en memoria los últimos valores de ocupación registrados y las marcas de tiempo que permiten conocer la antigüedad de los paquetes en los *buffers*. La comunicación OpenFlow entre el controlador y el *vSwitch* OVS sigue el modelo cliente-servidor.

El segundo hilo es el encargado de tomar decisiones basándose en los datos recogidos por el anterior. Se ocupará de establecer comunicación con OVSDB del *vSwitch* para crear reglas de QoS usando el algoritmo concreto que se esté utilizando. La primera comprobación es la búsqueda del límite de frecuencia a la que el controlador es capaz de tomar decisiones. Para poder obtener los mejores resultados, sería lógico suponer que cuanto más rápido tome decisiones, mejor, ya que permite adaptarse más rápidamente al tráfico de red fluctuante.

El controlador se ha diseñado de manera que se definen los tiempos entre las llamadas para los dos hilos de ejecución. En la Figura 3.10 se muestra a nivel de código, la definición de ***send_buffer_stats_request***

```
# Class Definition
class L2Switch(app_manager.RyuApp):
    # We will use OF version 1.3
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    # Initialize the controller
    def __init__(self, *args, **kwargs):
        super(L2Switch, self).__init__(*args, **kwargs)
            # Thread buffer stats request
            threading.Thread(target=self.send_buffer_stats_request, daemon=True).start()
            # Thread queue-load-balance request
            threading.Thread(target=self.send_queue_load_balance_request,daemon=True).start()
    def send_buffer_stats_request(self):
        while True:
            time.sleep(0.0010)
            for datapath_id in self.dprefs.keys():
                datapath = self.dprefs[datapath_id]
                self.send_queue_stats_request(datapath)
                self.send_flow_stats_request(datapath)
    def send_queue_load_balance_request(self):
        while True:
            time.sleep(0.010)
            for datapath_id in self.dprefs.keys():
                self.queue_load_balance(datapath_id)
```

Figura 3.10: Definición clase app *Ryu vSwitch* capa 2 - *threads*

y ***send_queue_load_balance_request***, que representa cómo se realiza la llamada a los *threads* o hilos que realizan las operaciones descritas. Es aquí donde se definen t_s o tiempo de *slot*, y t_{stats}, que se corresponden con el tiempo en el que el controlador obtiene una actualización de los datos de ocupación de los *buffers* interfaces cada *1 ms* y de la toma de decisiones cada *10 ms*.

Antes de mostrar las representaciones de los resultados, es importante describir que tanto en este capítulo como en el Capítulo 4, se mostrarán los resultados en *box plots*, o gráficos de cajas, que permiten analizar la distribución del conjunto de datos medidos y detectar patrones de comportamiento y valores atípicos. Además, este método de representación nos da más información, ya que obtenemos la mediana, rango intercuartil (que contiene el 50 % central de los datos) y los bigotes, que se extienden hasta 1.5 veces el rango intercuartil. Los valores más allá de este rango se consideran atípicos.

En la Figura 3.11 se puede apreciar mediante el uso de un diagrama de cajas cómo se distribuyen los tiempos entre decisiones durante la ejecución del controlador. Sin embargo, aunque el valor objetivo esté definido en 10 ms, la llamada al hilo se produce en media, cada 50 ms en este experimento, con valores atípicos que pueden llegar a estar por encima de los 150 ms.

Para poder entender mejor por qué ocurre, se realiza un estudio introduciendo marcas de tiempo en el código para tratar de localizar dónde se produce esta latencia añadida. Al tratarse de una aplicación diseñada y programada en Python y ser un lenguaje interpretado, el rendimiento se verá afectado tal y como se ha comentado a lo largo del Capítulo 2, concretamente en la Sección 2.2.1.

En la Figura 3.12 se muestra la distribución de la frecuencia de acceso a la base de datos OVSDB. A pesar de estar dentro del mismo entorno, donde la latencia de comunicación entre OVS y el controlador es menor a 1 ms, se puede ver que el acceso a la base de datos OVSDB está en media en torno a los 40 ms. Podemos concluir, por lo tanto, que el acceso a OVSDB es el factor que determinará realmente la velocidad en la toma de las decisiones. Sin embargo, como el acceso a OVSDB es bloqueante, podemos mantener la ejecución del hilo por debajo de los 40 ms, a pesar de que aparentemente no tenga sentido tomar decisiones por debajo de este valor, al forzar al controlador a ejecutar el hilo más rápido, el acceso a OVSDB

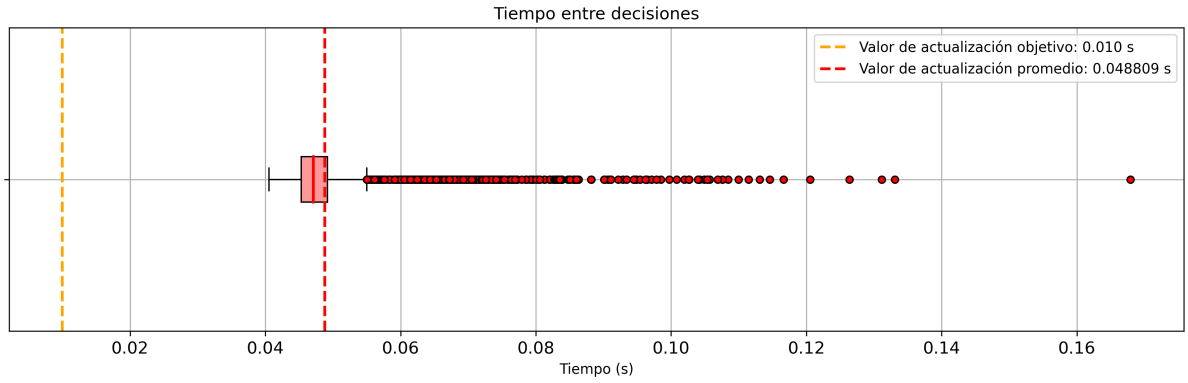


Figura 3.11: Boxplot - Frecuencia de toma de decisiones

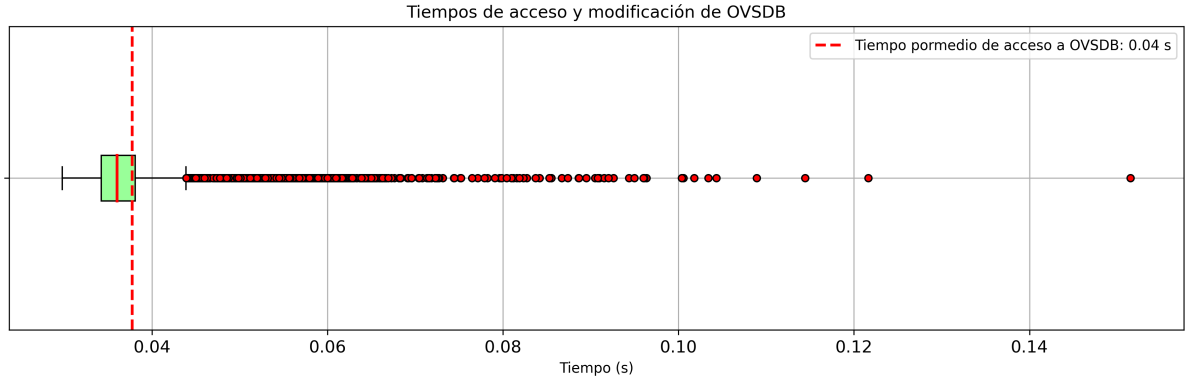


Figura 3.12: Boxplot - Frecuencia de acceso a modificar OVSDB

Estos 10 milisegundos, realmente, son la frecuencia con la que se ejecuta el hilo encargado de tomar las decisiones. Sin embargo, como se puede apreciar en la figura 3.11, en el experimento el controlador no es capaz de tomar decisiones por debajo de los 50 milisegundos.

Esta forma de funcionar es una decisión de implementación de la aplicación y las primeras pruebas consisten en jugar con estos parámetros para ajustar el controlador. Se pretende conseguir la actualización de estadísticas del *vSwitch* OVS con la mayor frecuencia posible. Esto permite al controlador disponer de la información suficiente para conocer el estado de los interfaces del *vSwitch*. Al igual que en el *thread* encargado de tomar las decisiones, se puede definir la frecuencia con la que se desea que se ejecute el *thread*. En este caso, no se realizan modificaciones sobre OVSDB, y por lo tanto no se introducen las latencias antes mencionadas, sino que se realizan las lecturas de los valores disponibles de paquetes transmitidos por las *queues* y por los *flows* de la *flow-table*. Estos valores se emplean para calcular y monitorizar la ocupación del *buffer*. En la Figura 3.13 se puede ver que los tiempos de acceso a la lectura de estadísticas y actualización de los valores del *buffer* se acercan al valor de 1ms, establecido en el tiempo entre ejecuciones de *send_buffer_stats_request*.

3.2.3. Comportamiento del controlador bajo límite de tráfico

En esta sección se abordará una cuestión principalmente, y es describir el comportamiento del controlador cuando se enfrenta a la gestión de tasas muy próximas a los límites de *max-rate* establecidos por la aplicación de *egress policies* en los interfaces de OVS. Tras estudiar esta cuestión, se comentará cómo se

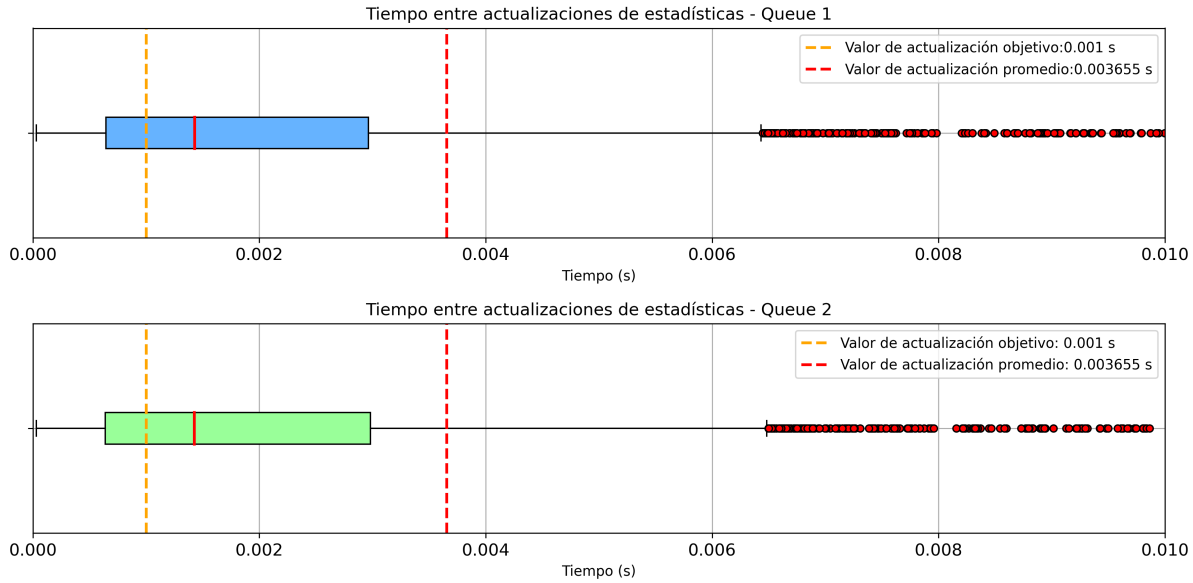


Figura 3.13: Boxplot - Frecuencia de actualización de estadísticas de ocupación

anticipa que será su comportamiento frente a tráfico a ráfagas, que es en definitiva, lo que se busca con su diseño, su capacidad de adaptación a variaciones de tráfico.

Sin la intervención del controlador, se ha configurado en OVS un QoS estático sobre el interfaz para el tráfico agregado de 7 *Mbps* y dos colas limitando el *bitrare* de los otros interfaces a 2 *Mbps* y 5 *Mbps* respectivamente. A continuación, se genera tráfico *iperf* UDP con intervalos cortos de separación. Se genera tráfico de forma incremental para cada uno de los interfaces hasta alcanzar la máxima capacidad agregada limitada por el QoS y se puede apreciar que, cuando nos acercamos al límite, se experimenta una pérdida de capacidad independientemente de la implementación del controlador SDN.

En la Figura 3.14 se pueden observar los resultados que demuestran este hecho. Se han realizado varios experimentos, y se puede apreciar claramente un incremento de la pérdida de la tasa cuando nos acercamos a los límites de ambas colas. Con *iperf3* se genera tráfico controlado, con un tiempo predefinido de 5 segundos de duración por cada test *iperf*. El tráfico se genera de manera simultánea por ambos interfaces, ya que se busca simular situaciones en las que el tráfico está por debajo de la capacidad y otras en las que se acerca al límite.

En otras palabras, se pierde capacidad cuando el tráfico se acerca a la máxima capacidad asignada -sin llegar a superarla donde lógicamente ya se perderían paquetes- ya sea por la cola por defecto generada con la creación de un QoS en OVS o por cualquiera de sus colas asociadas siendo este uno de los límites de las herramientas empleadas para este proyecto. En la figura 3.14, lo que se puede ver en las dos primeras gráficas es la cantidad de tasa perdida para cada uno de los flujos, pudiendo identificar en la leyenda la tasa a la que cada flujo está tratando de transmitir. En la tercera gráfica se ve el tráfico agregado para ambos flujos, y dónde claramente se puede apreciar que al acercarse al límite establecido por el QoS, siempre se transmite por debajo de la capacidad. El tráfico generado para demostrar esta limitación, es constante durante varios test *iperf*; es decir, no es tráfico aleatorio a ráfagas y sin la intervención del controlador. Sin embargo, es importante detectarla, ya que en el próximo capítulo se pondrán a prueba diferentes *schedulers*.

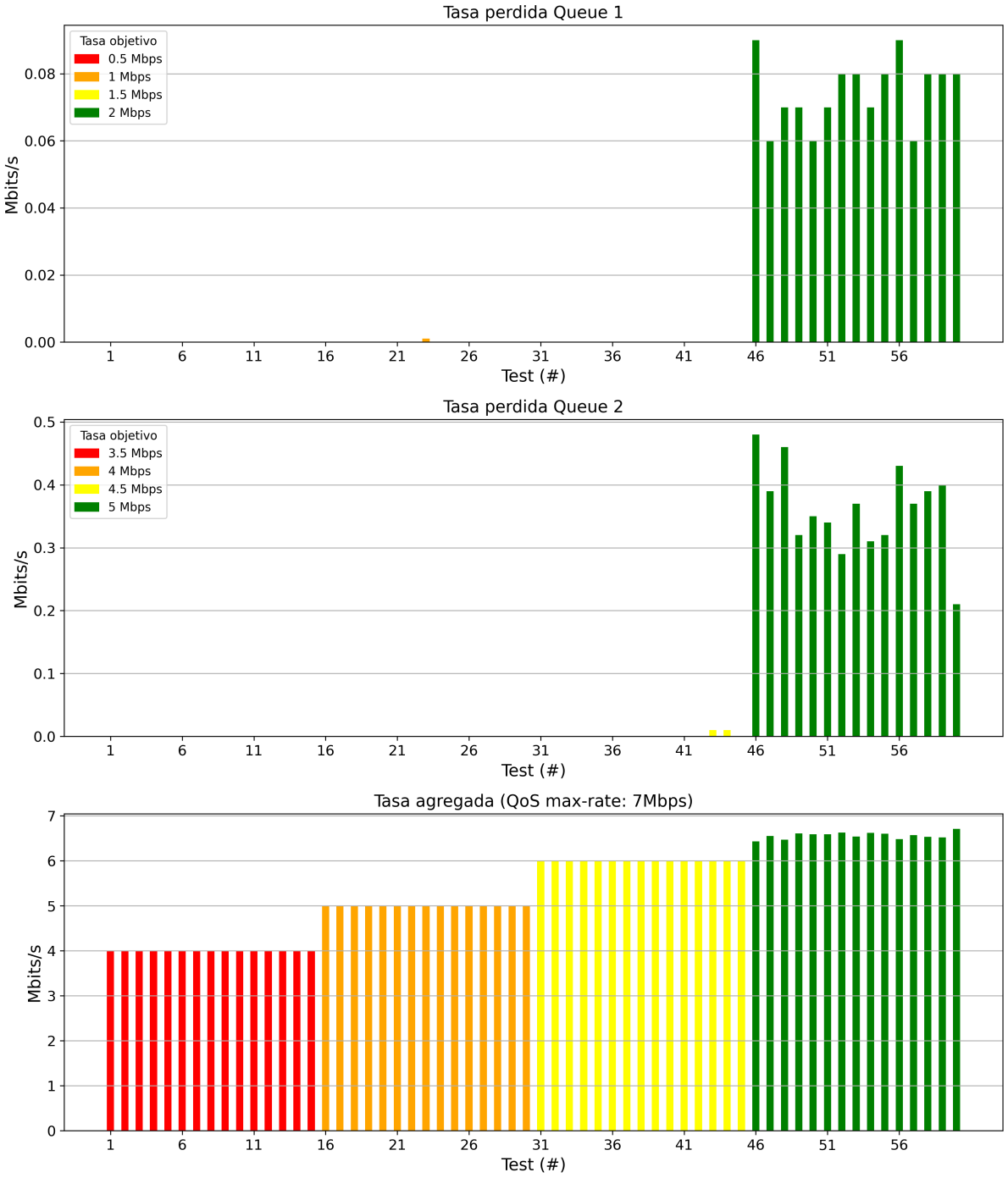


Figura 3.14: Pérdida de capacidad cerca del límite

Finalmente, en la Figura 3.15 se presentan los resultados del mismo experimento, esta vez con la intervención del controlador. En este escenario, el controlador toma decisiones de forma periódica siguiendo la frecuencia y los patrones descritos en la Sección 3.2.2 de este capítulo. El *scheduler* implementado utiliza como criterio principal de decisión un esquema de balanceo proporcional dinámico, basado en la ocupación de los *buffers* de los interfaces. Este criterio, centrado en la ocupación, resulta ser el más relevante para los resultados obtenidos, tal como se argumentó previamente en la Sección 3.2.1.

Uno de los objetivos de esta sección es mostrar que, incluso sin la intervención del controlador, ya se observaba una pérdida de *bitrate* al aproximarse al límite de capacidad. Esta situación se agrava cuando el controlador entra en funcionamiento, lo cual es esperable, ya que está continuamente aplicando nuevos valores de *max-rate* a los diferentes flujos en función de la ocupación de los *buffers*. Como se ha demostrado, cuando se impone un límite de capacidad, rara vez se alcanza por completo dicha tasa. Por esta razón, se ha optado por no asignar la totalidad del ancho de banda agregado disponible a una única cola, incluso cuando esta presente una mayor ocupación que las demás. En su lugar, se garantiza una capacidad mínima para la cola con menor ocupación, lo que permite al controlador mantener un comportamiento más equilibrado y eficiente.

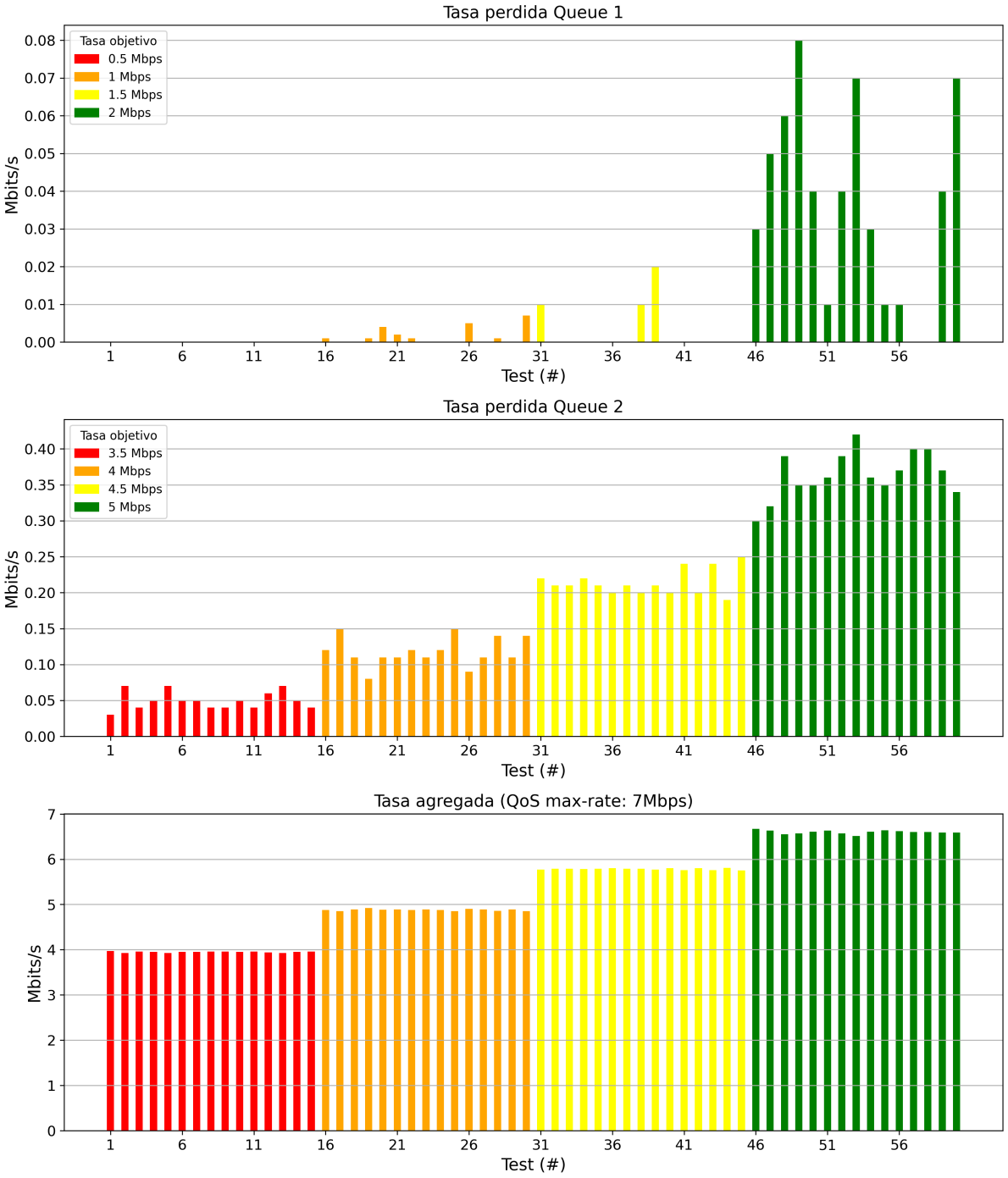


Figura 3.15: Pérdida de capacidad cerca del límite - Controlador Ryu

Capítulo 4

Controlador *Ryu*: Resultados y análisis

Finalmente, en este capítulo se exponen y analizan los resultados obtenidos a partir de varios experimentos diseñados para evaluar el funcionamiento del controlador y la efectividad del algoritmo *scheduler* propuesto en la Sección 3.1, con el objetivo de validar si realmente cumple con los requisitos definidos y demostrar las ventajas de aplicar reglas de QoS dinámicas dentro de una arquitectura SDN mediante la configuración de controladores inteligentes.

Los resultados de los experimentos realizados mostrarán el impacto de las decisiones del controlador en función de los criterios de decisión, basados en las métricas de ocupación y retardos, como se ha detallado en la Sección 3.2.1. En los diferentes casos, se utilizarán las siguientes métricas para optimizar el *scheduler*: ocupación, retardo acumulado y retardo máximo de los paquetes en los *buffers*. Para estructurar el análisis se plantean las pruebas en dos entornos diferenciados:

1. Escenario con flujos de tráfico controlado: se genera tráfico utilizando *iperf* con tasas predefinidas, permitiendo anticipar el comportamiento esperado del sistema. El objetivo es comprobar que el controlador actúa de acuerdo con lo previsto y que el algoritmo *scheduler* responde correctamente bajo condiciones conocidas y reproducibles.

2. Escenario con flujos de tráfico aleatorio: en este apartado será donde realmente se pone a prueba el controlador y las ventajas que aporta su gestión dinámica de QoS. Para ello se genera tráfico aleatorio dentro de unos rangos predefinidos, lo que nos permite acercarnos mas al comportamiento real de un controlador SDN gestionando tráfico de red.

Los experimentos se realizan sobre el escenario (configurado sobre GNS3) de la Figura 4.1, en el que se genera tráfico a ráfagas *iperf* desde *user-1* y *user-2* simultáneamente hacia *user-3*. El *vSwitch* cuenta con 3 interfaces configuradas y su función se limita a reenviar paquetes en función de las instrucciones que le proporciona periódicamente el controlador. Mediante *iperf* se genera tráfico a ráfagas UDP, realizando múltiples tests que simulan el tráfico de diferentes servicios. Esto se consigue variando las tasas de cada test, ya que cada servicio transmitirá una tasa diferente. Cuando se trata de un escenario controlado, las tasas generadas serán fijadas previamente para verificar que el controlador tiene el comportamiento esperado. Por otra parte, en el escenario con flujos de tráfico aleatorio, aunque siga tratándose del mismo esquema que el de la Figura 4.1, se definen tasas aleatoriamente dentro de unos rangos. Estos rangos se

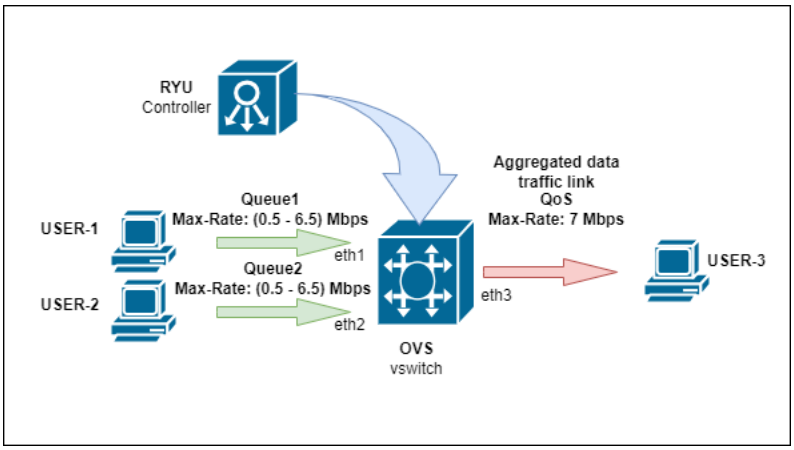


Figura 4.1: Escenario resultados

definen de manera que el tráfico generado por uno de los flujos es en media de 2 *Mbps* (*user-1*) y el otro flujo (*user-2*) genere, en media, tráfico a una tasa de 5 *Mbps*. Además, se realizarán diferentes mediciones manteniendo el valor medio para cada flujo pero variando la dispersión en torno al valor medio de las tasas.

En el momento en el que el controlador reciba la notificación de la presencia de un *vSwitch*, aplicará una política de QoS sobre el interfaz (conocido) por el que se reenviará el tráfico agregado. El QoS crea por defecto una cola con un *max-rate* de 7 *Mbps* y dos colas asociadas cuyo *max-rate* variará en función de las métricas que se quieran establecer relacionadas con la ocupación de los *buffers* y/o del retardo acumulado de los paquetes en sus respectivos *buffers*.

Es importante mencionar que a la hora de asignar la capacidad disponible a las colas asociadas (*queue1* y *queue2*), se establece siempre una capacidad mínima asignable de manera que nunca se le asigna todo el ancho de banda disponible a uno de los flujos en función de los criterios que se establezcan (se puede ver en la Figura 4.1). Teóricamente se pretendía que así fuera, de manera que, por ejemplo, cuando la ocupación de uno de los *buffers* está muy por encima del otro, todo el ancho de banda disponible de la cola *default queue* (*max-rate* asignado al establecer un QoS) se le asigna al flujo correspondiente. Sin embargo, a la hora de realizar la implementación, y como se ha descrito en la Sección 3.2.3, el controlador no permite asignar un cero absoluto como capacidad asignada y responde mejor si asigna, aunque sea proporcionalmente mucho menor, una capacidad mínima que no sea cero.

4.1. Escenario con flujos de tráfico controlado

En esta sección se exponen los primeros resultados del controlador Ryu. Es el escenario con el que se ha trabajado durante el proceso de implementación, ya que se somete a unas condiciones específicas, esperando un comportamiento predecible que permite validar y verificar el correcto funcionamiento del controlador.

En este apartado se muestra el comportamiento del controlador bajo las condiciones conocidas y especificadas en la Tabla 4.1. Para estudiar el comportamiento de las políticas dinámicas, se compara el comportamiento del sistema al aplicar una asignación estática de capacidad, con el obtenido al utilizar el *scheduler* basado en la ocupación del *buffer*. Con estas configuraciones, se obtienen los resultados de la

Tabla 4.1: Resumen de configuración de QoS y pruebas *iperf* para verificación del *scheduler*

Configuración de QoS		
Cola	**QoS Estático (Mbps)**	**QoS con Controlador (Mbps)**
Default queue	7.0	7.0
Queue 1	1.5	[0.5, 6.5]
Queue 2	5.5	[0.5, 6.5]
Configuración de los tests *iperf*		
Parámetro	**Flujo 1**	**Flujo 2**
Número de tests	5	5
Duración por test (s)	5	5
Bitrate test 1 (Mbps)	2.0	5.0
Bitrate test 2 (Mbps)	0.5	6.5
Bitrate test 3 (Mbps)	3.5	3.5
Bitrate test 4 (Mbps)	3.0	4.0
Bitrate test 5 (Mbps)	3.5	4.5

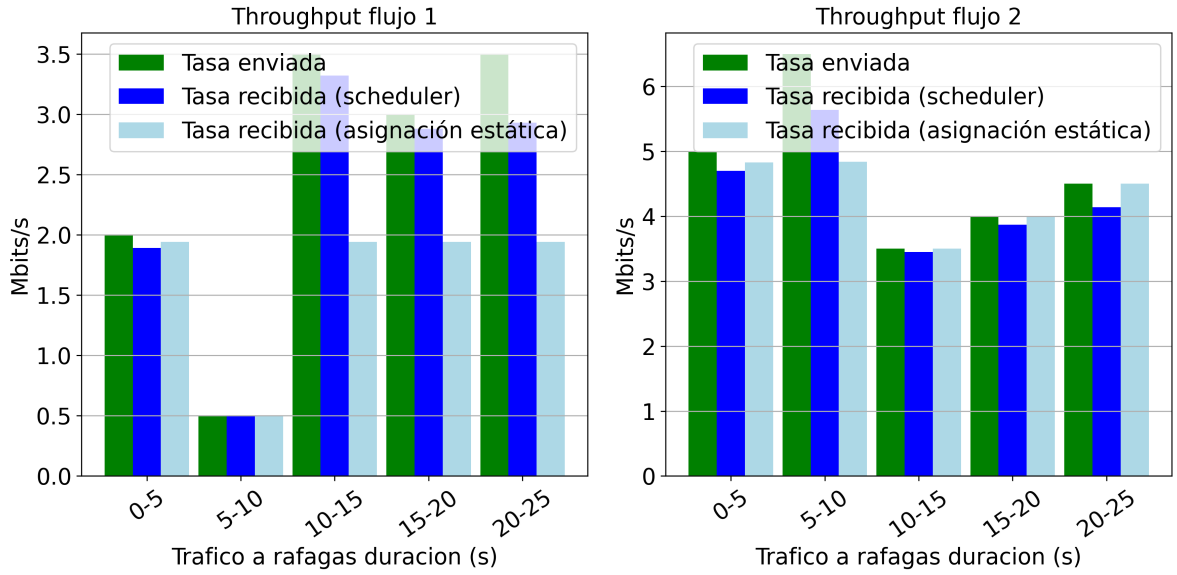


Figura 4.2: Bitrate - QoS *scheduler* vs asignación estática

Figura 4.2.

Esta gráfica muestra la tasa enviada y recibida canto haciendo uso del *scheduler* como con una asignación estática en la que se asignan las capacidades iguales al tráfico generado por cada flujo. Como se puede observar, el flujo 1 nunca podrá enviar a tasas superiores al límite impuesto por su cola, lo que provocará un considerable aumento en los niveles de ocupación de los *buffers* como veremos a continuación, pudiendo provocar congestión e incluso pérdida de paquetes.

Para tener una imagen más clara del comportamiento obtenido, en la Figura 4.3 se muestra la evolución temporal de los valores de ocupación de los *buffers* de ambas interfaces del *vSwitch* a lo largo del tiempo junto con las tasas de tráfico recibido y enviado. En el primer caso, el controlador no juega ningún papel más allá de aplicar simplemente, *egreess policies* directamente sobre el interfaz *eth3* del *vSwitch* y recoger datos en tiempo real de la ocupación de los interfaces de OVS (actualización de datos de ocupación cada 1 ms). Simplemente se realiza una asignación estática de capacidad y se genera el mismo tráfico descritos

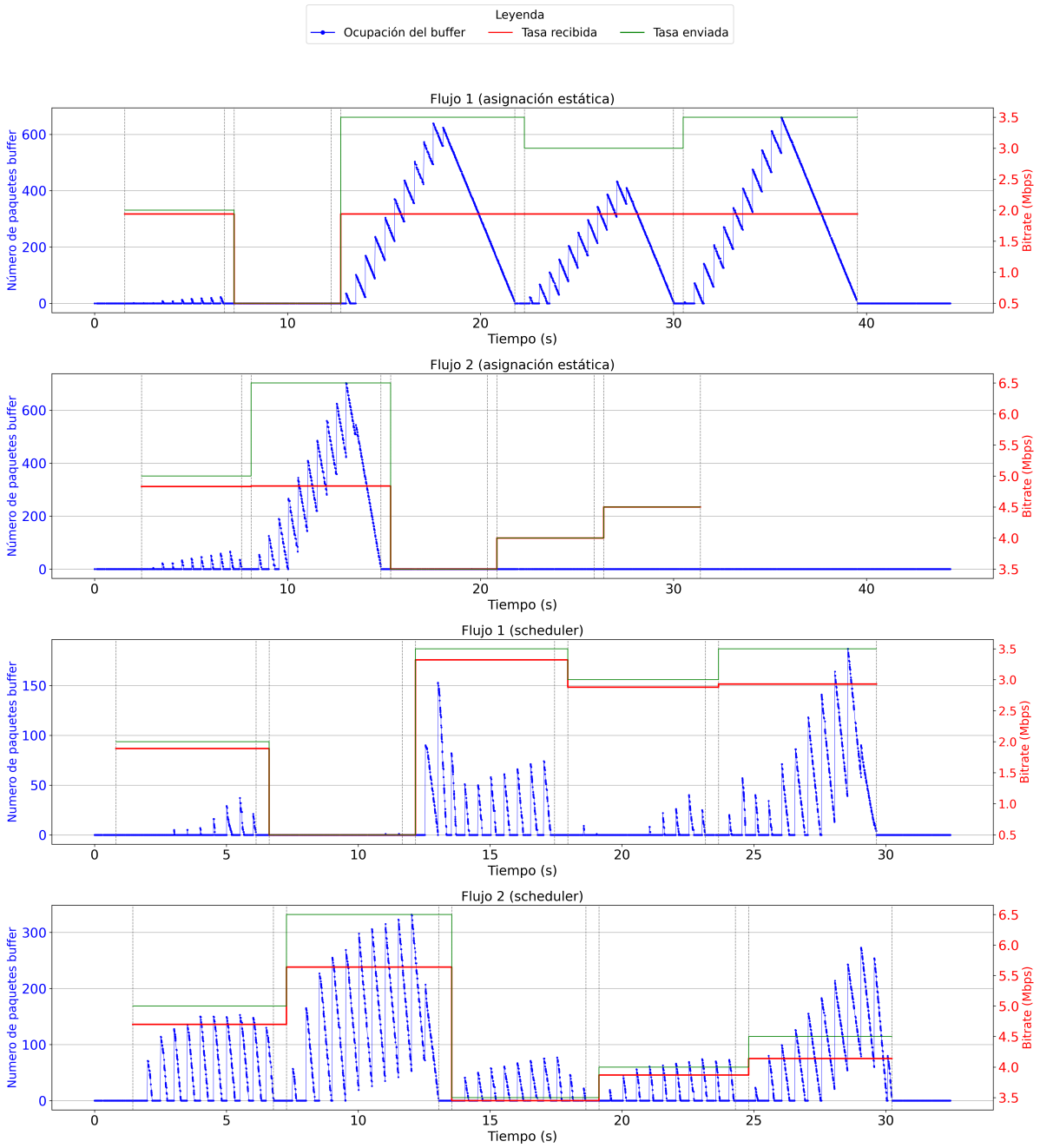


Figura 4.3: Evolución temporal de la ocupación del *buffer* - QoS *scheduler* vs asignación estática

en la Tabla 4.1.

Por otro lado, en el segundo caso de la Figura 4.3, se muestra el comportamiento del sistema al aplicar el planificador basado en la ocupación de los *buffers*. En esta ocasión, el controlador realiza asignaciones dinámicas de capacidad sobre las colas asociadas a los dos flujos según la Tabla 4.1, empleando como criterio de decisión la ocupación de las interfaces, repartiendo la capacidad agregada proporcionalmente a la ocupación siempre que la diferencia de ocupación entre ambos *buffers* sea superior al 20 %. En caso contrario, el reparto de la capacidad será equitativo. Como se puede observar, la gestión dinámica asegura que el nivel de ocupación máximo de los *buffers* es mucho menor, siendo en torno a la mitad para el flujo 2 y a una cuarta parte para el flujo 1.

Tabla 4.2: Parámetros del experimento *iperf* - sin dispersión

Parámetro *Iperf3*	**Valor**
Número de tests por flujo	50
Duración de cada test	5 segundos
Tasa objetivo flujo 1	1.5 – 2.5 Mbps
Tasa objetivo flujo 2	4.5 – 5.5 Mbps

Este primer experimento permite observar una ventaja sobre una asignación estática de QoS. Para realizar una comparación justa, se plantearon las mismas condiciones de tráfico para ambos casos (Tabla 4.1) y se puede apreciar claramente que con la intervención del *scheduler* del controlador, los *buffers* se descargan más rápido y alcanzan niveles máximos de ocupación mucho menores que con una asignación estática, y además logra en términos de *bitrate* resultados más ajustados a la necesidad, especialmente en el flujo 1, que resulta más castigado con la asignación estática al no poder superar los 2 Mbps.

4.2. Escenario con flujos de tráfico aleatorio

Es en esta sección donde se muestran los resultados finales de este proyecto. El objetivo, en definitiva, es mostrar de lo que es capaz el controlador, comparando los diferentes criterios de decisión ya comentados durante la Sección 3.2.1 y simulando unas condiciones de red más próximas a un entorno real. En este sentido, el tráfico *iperf* generado será totalmente aleatorio, pero sujeto a la condición de que en media el tráfico generado se corresponde con las limitaciones de asignación del QoS de 2 Mbps para el flujo 1 y de 5 Mbps para el flujo 2.

En los primeros resultados se mostrarán los niveles de ocupación de los *buffers* para los diferentes *schedulers*, entendiendo que en cada uno de los casos, la métrica empleada para tomar decisiones es una de las mostradas en la Tabla 4.3. En base a cada uno de los criterios, los *schedulers* establecen un reparto proporcional de la capacidad agregada. Para el experimento, se genera tráfico *iperf* UDP con tasas aleatorias para crear las condiciones de red de tráfico a ráfagas bajo los parámetros definidos en la Tabla 4.2.

En una primera configuración se genera en media 2 *Mbps* para el flujo 1 y de 5 *Mbps* para el flujo 2 , pero con valores de tasas generadas con poca dispersión. Bajo estas premisas, se obtienen los resultados de la Figura 4.4. En este *boxplot* la métrica que se compara es la ocupación del *buffer* para los diferentes criterios de decisión de los diferentes *schedulers*. En esta situación, al no existir mucha dispersión en torno al valor medio de la tasa de transmisión, las diferentes tasas de transmisión se aproximan al valor delimitado por el *max-rate* de las colas. Por lo tanto al realizar la comparación entre un QoS estático y el uso de un *scheduling* no se aprecia una mejora sustancial en cuanto a la ocupación del *buffer* se refiere y, como es lógico, la mejor solución es la asignación estática.

Se repite el experimento, pero esta vez se aumenta la dispersión en torno al valor medio a la hora de generar el tráfico respetando la condición de generar en media 2 *Mbps* para el flujo 1 y de 5 *Mbps* para el flujo 2. Los detalles del tráfico *iperf* generado para el resto de los resultados se resumen en la Tabla 4.4. Para el flujo 1 no se reduce la tasa al 0 absoluto, porque no tiene sentido no generar tráfico *iperf* con tasa 0, por lo que la tasa media no es exactamente 2, pero sí un valor muy próximo.

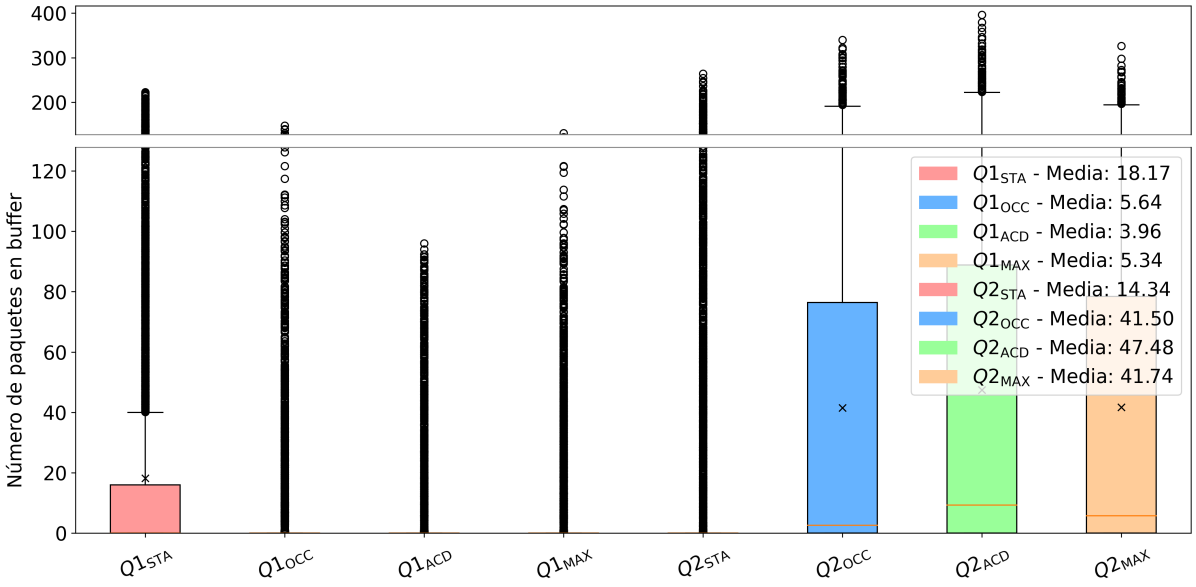


Figura 4.4: Ocupación de los *Buffers* - QoS asignación estática vs QoS *scheduling*

Tabla 4.3: Significado de las etiquetas usadas en los gráficos de comparación de buffers

Etiqueta	**Descripción**
$Q1_{\mathrm{STA}}$	Buffer de la cola 1 bajo asignación estática de capacidad
$Q1_{\mathrm{OCC}}$	Buffer de la cola 1 bajo asignación basada en ocupación
$Q1_{\mathrm{ACD}}$	Buffer de la cola 1 bajo asignación por retardo acumulado
$Q1_{\mathrm{MAX}}$	Buffer de la cola 1 bajo asignación por retardo máximo
$Q2_{\mathrm{STA}}$	Buffer de la cola 2 bajo asignación estática de capacidad
$Q2_{\mathrm{OCC}}$	Buffer de la cola 2 bajo asignación basada en ocupación
$Q2_{\mathrm{ACD}}$	Buffer de la cola 2 bajo asignación por retardo acumulado
$Q2_{\mathrm{MAX}}$	Buffer de la cola 2 bajo asignación por retardo máximo

Tabla 4.4: Parámetros del experimento iperf para la verificación del *scheduler*

Parámetro *Iperf3*	**Valor**
Número de tests por flujo	50
Duración de cada test	5 segundos
Tasa objetivo flujo 1	0.1 – 4 Mbps
Tasa objetivo flujo 2	3 – 7 Mbps

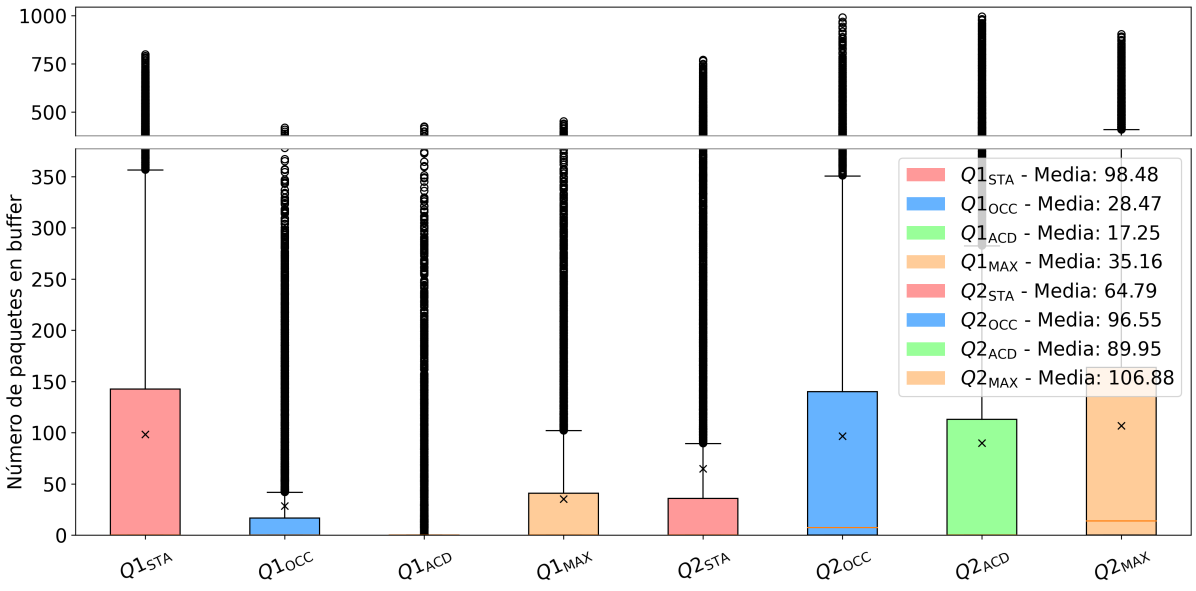


Figura 4.5: Ocupación de los *Buffers* - QoS asignación estática vs QoS *scheduling*

Se obtienen finalmente los resultados de la Figura 4.5 donde se compara para cada *scheduler* la métrica de ocupación de los *buffers*. Se puede apreciar que al aumentar la dispersión, además de cubrir las necesidades de *bitrate* con más eficiencia, es más favorable el uso de cualquier *scheduler* en cuanto a ocupación del *buffer* con una diferencia considerable para el flujo 1 siendo este el más perjudicado en un QoS estático. En estas condiciones de tráfico más próximo a la realidad, la asignación estática es la peor de las soluciones.

Un detalle a destacar en los resultados de la Figura 4.5 es que el *scheduler* ACD sea superior a OCC, ganándole en su propia métrica. Sin embargo, este comportamiento podría ser posible. Para explicar por qué se puede producir esta situación, se muestra un ejemplo en la Figura 4.6 para ver cómo es el comportamiento de ambos *schedulers* y cómo ACD puede mostrar mejores resultados que OCC incluso en la propia métrica de ocupación.

La Figura 4.6 muestra la evolución temporal del estado de los buffers bajo dos políticas de planificación: ACD y OCC. En cada subgráfico se representan las colas Q1 (en azul) y Q2 (en verde) en los instantes: $t = 10s$, $t = 11s$, $t = 12s$ y $t = 13s$. Cada bloque corresponde a un paquete, etiquetado con su identificador (una letra) y el retardo acumulado (en segundos) que lleva en la cola en ese instante. Los paquetes resaltados con borde rojo indican nuevas llegadas a los interfaces. Se puede observar que la política ACD drena la cola Q2 antes de que se acumulen retardos extremos, mientras que OCC, al priorizar únicamente la ocupación, deja crecer el retardo de algunos paquetes, como puede apreciarse en el caso del paquete S en $t = 13s$, que alcanza un retardo superior al que ocurre bajo ACD. Esta visualización permite comparar el impacto de cada *scheduler* en la evolución de los retardos, evidenciando cómo ACD logra un mejor equilibrio entre colas y previene picos extremos de retardo. Además, cabe destacar que OCC reparte la capacidad equitativamente en caso de no detectar diferencias relevantes de ocupación entre los *buffers*, lo que limita su capacidad de anticipación frente a patrones de retardo acumulado. Por eso, en el instante $t = 13s$ OCC solo ha vaciado un paquete de cada cola y se puede ver el resumen del ejemplo en la Tabla 4.5.

El *scheduler* ACD evalúa el retardo acumulado de cada cola, priorizando la gestión de aquellos paquetes

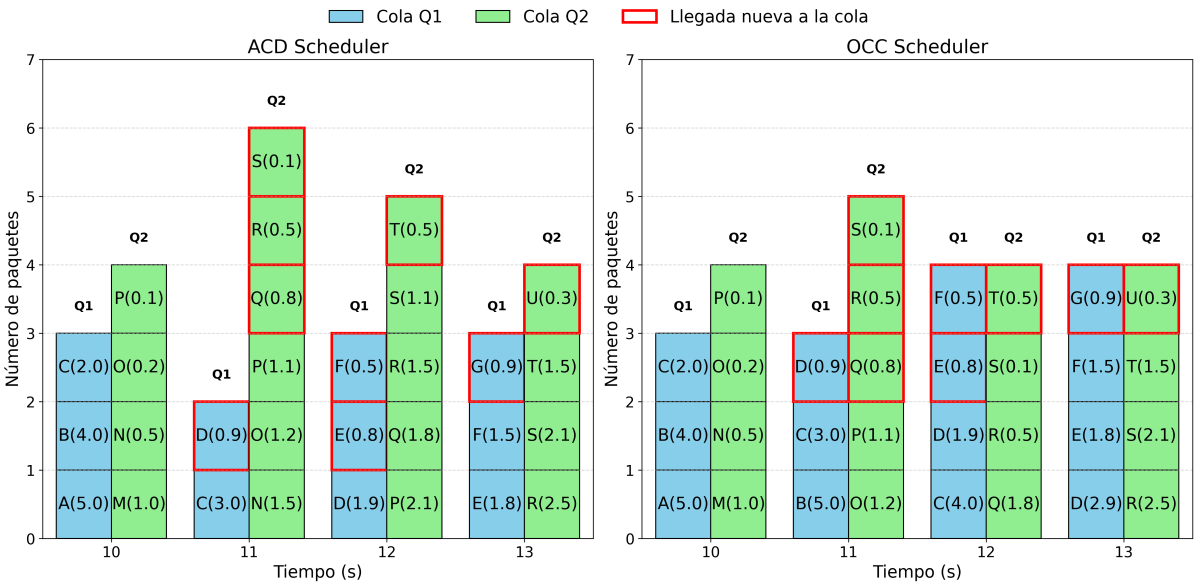


Figura 4.6: Ejemplo comparación de *scheduling*

Tabla 4.5: Resumen de métricas ACD vs OCC

Scheduler	Cola	Ocupación (paquetes)	Retardo acumulado (s)
ACD/OCC t=10	Q1	3	11.0
	Q2	4	1.8
ACD t=11	Q1	2	3.9
	Q2	6	5.2
OCC t=11	Q1	3	8.9
	Q2	5	3.7
ACD t=12	Q1	3	3.2
	Q2	5	7.0
OCC t=12	Q1	4	7.2
	Q2	4	2.9
ACD t=13	Q1	3	4.2
	Q2	4	6.4
OCC t=13	Q1	4	7.1
	Q2	4	6.4

con mayor antigüedad en espera. A diferencia de OCC, que basa sus decisiones únicamente en el número de paquetes en cada cola en un instante dado, ACD incorpora una perspectiva histórica, valorando el envejecimiento de los paquetes. Esta diferencia resulta crucial cuando las colas presentan ocupaciones similares. En ese caso, OCC no prioriza ninguna cola, ya que no percibe asimetría en su métrica principal. Sin embargo, puede darse la situación de que una de las colas se haya llenado bruscamente en los últimos instantes, acumulando paquetes con mayores retardos. ACD detecta este envejecimiento y reacciona antes, adaptándose mejor al tráfico, lo que le permite vaciar las colas más rápidamente. En consecuencia, aunque OCC busca igualar la ocupación entre colas, ACD puede lograr una ocupación media global más baja. Este comportamiento pone de manifiesto que una política basada en retardo acumulado no solo mejora la latencia percibida por los paquetes más antiguos, sino que también puede traducirse en una mejor eficiencia global en la gestión de los *buffers*.

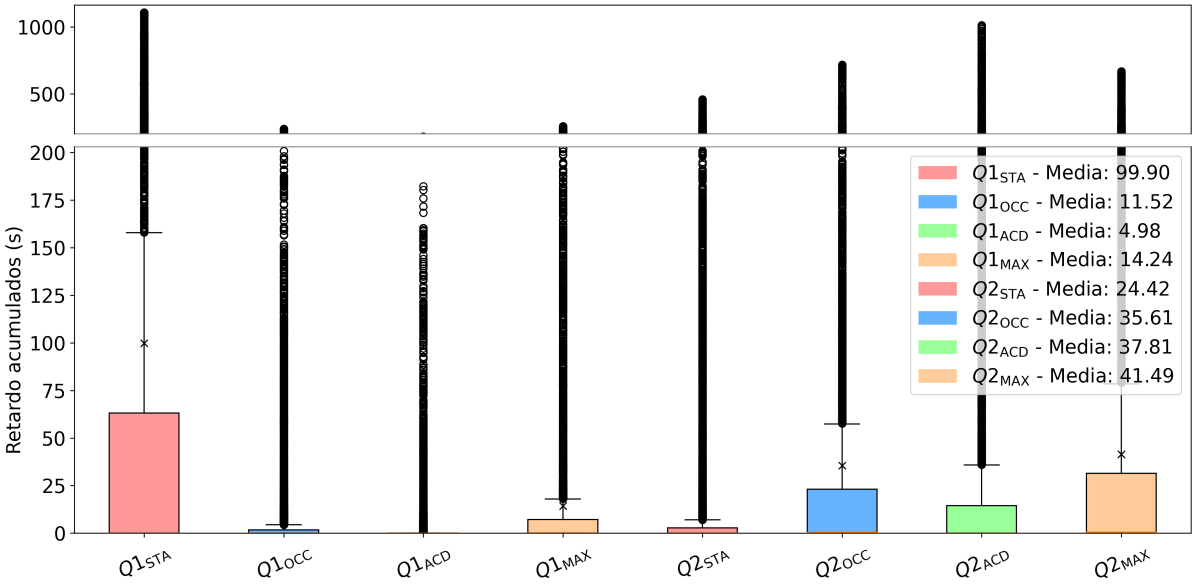


Figura 4.7: Retardo acumulado - QoS asignación estática vs QoS *scheduling*

Continuando con los resultados, en el siguiente experimento se realiza la comparación del retardo acumulado en los *buffers* para cada *scheduler*. En la Figura 4.7 se puede ver la comparativa de esta métrica y comportándose tal y como era de esperar, ACD es el que tiene la mejor respuesta.

Por último, se obtienen los resultados de la comparativa para el retardo máximo en la Figura 4.8. Una vez más, la asignación estática es la peor solución posible a todos los niveles: ocupación, retardo acumulado y retardo máximo. Sin embargo, vuelve a llamar la atención que el *scheduler* que toma decisiones basándose en el retardo máximo no sea el mejor en esta métrica en comparación con los *schedulers* OCC y ACD. En este caso, MAX es un *scheduler greedy* que solo busca minimizar la cola que contenga el paquete más antiguo, sin tener en cuenta ni ocupación ni el tiempo de espera del resto de los paquetes de la cola. Esto puede provocar, en un tráfico a ráfagas y variable, que los retardos máximos aumenten por la toma de decisiones no balanceadas.

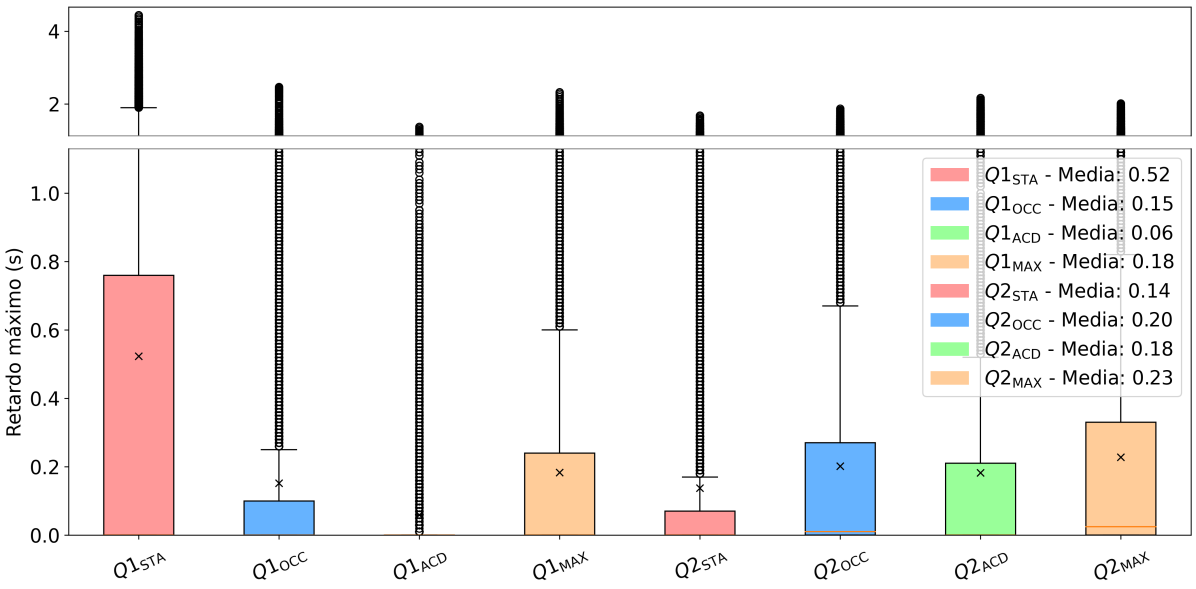


Figura 4.8: Retardo máximo - QoS asignación estática vs QoS *scheduling*

Capítulo 5

Conclusiones

Este proyecto tiene como objetivo proponer soluciones de gestión de tráfico adaptativas para dar respuesta a los entornos y nuevos segmentos de red que aparecen en sistemas de nueva generación. Para ello se han propuesto, implementado y validado soluciones de *scheduling* para redes de transporte bajo el paradigma proporcionado por el concepto SDN.

Este nuevo modelo separa el plano de datos y el plano de control, posibilitando la centralización de la gestión. Esto no solo permite un uso más eficiente de la infraestructura de las redes, sino que también favorece la aparición de herramientas *open source* con independencia del *hardware* e integrables con una amplia gama de diferentes dispositivos gestionados por los controladores inteligentes.

Concretamente, las soluciones desarrolladas en este proyecto se han basado en las herramientas abiertas Ryu y *Open vSwitch*. Ambas se pueden utilizar en entornos reales de producción para administradores y proveedores de servicio, como pueden ser recursos *cloud* dedicados a la gestión de infraestructura. Con estas herramientas se ha implementado un controlador programado en *Python* capaz de aplicar varios criterios de decisión y comunicarse con el plano de datos usando los protocolos OpenFlow y OVSDB.

A lo largo del Capítulo 3 se han mostrado cuáles eran las condiciones de implementación del controlador y cuáles son los límites de su rendimiento. Entre lo más destacado está la velocidad de reacción del controlador para tomar decisiones modificando OVSDB, ya que cuanto mayor pueda ser la capacidad de reacción, más control se puede aplicar sobre la adaptación de los diferentes flujos gestionados por el controlador.

Posteriormente, en el Capítulo 4 se ha evaluado el rendimiento de los diferentes criterios de decisión implementados: la ocupación, el retardo acumulado o el retardo máximo en los *buffers* de las interfaces del OVS.

En primer lugar, se ha podido observar que el uso de políticas dinámicas ofrece un mejor rendimiento que la reserva estática de recursos. Además, la comparativa entre los diferentes *schedulers* muestra que emplear el retardo acumulado proporciona los mejores resultados. En el escenario planteado —con tráfico a ráfagas generado mediante *iperf3* para simular distintos servicios— el uso del retardo acumulado de los paquetes en los *buffers* supone una ventaja clara, ya que considera no solo la ocupación, sino también el envejecimiento de las colas. Esto permite un reparto de capacidad más robusto frente a variaciones bruscas o picos de tráfico que, si solo se tuviera en cuenta la ocupación, podrían conducir a congestión.

Del mismo modo, el uso del retardo máximo no ofrece una respuesta tan eficiente frente al tráfico a ráfagas, ya que, debido a la naturaleza del tráfico de red, resulta más sensible a las fluctuaciones en la asignación de capacidad.

Bibliografía

[1] J. Liu, S. Xu, S. Zhou y Z. Niu. «Redesigning fronthaul for next-generation networks: beyond baseband samples and point-to-point links». En: *IEEE Wireless Communications* 22.5 (2015), págs. 90-97. DOI: 10.1109/MWC.2015.7306542.

[2] M. Polese, L. Bonati, S. D'Oro, S. Basagni y T. Melodia. «Understanding O-RAN: Architecture, Interfaces, Algorithms, Security, and Research Challenges». En: *IEEE Communications Surveys & Tutorials* 25.2 (2023), págs. 1376-1411. DOI: 10.1109/COMST.2023.3239220.

[3] Open RAN Alliance. *O-RAN Working Group 4 (Open Fronthaul Interfaces WG). Control, User and Synchronization Plane Specification (V14.00)*. TS. O-RAN, 2024.

[4] Open RAN Alliance. *O-RAN Working Group 9 (Open X-haul Transport WG). O-RAN Xhaul Packet Switched Architectures and Solutions (V7.00)*. TS. O-RAN, 2024.

[5] A. L. Valdivieso Caraguay, L. I. Barona Lopez y L. J. Garcia Villalba. «Evolution and Challenges of Software Defined Networking». En: *2013 IEEE SDN for Future Networks and Services (SDN4FNS)*. 2013, págs. 1-7. DOI: 10.1109/SDN4FNS.2013.6702542.

[6] Q. Waseem, W. Isni Sofiah Wan Din, A. Aminuddin, M. Hussain Mohammed y R. F. Alfa Aziza. «Software-Defined Networking (SDN): A Review». En: *2022 5th International Conference on Information and Communications Technology (ICOIACT)*. 2022, págs. 30-35. DOI: 10.1109/ICOIACT55506.2022.9972067.

[7] D. Henneke, L. Wisniewski y J. Jasperneite. «Analysis of realizing a future industrial network by means of Software-Defined Networking (SDN)». En: *2016 IEEE World Conference on Factory Communication Systems (WFCS)*. 2016, págs. 1-4. DOI: 10.1109/WFCS.2016.7496525.

[8] S. Imran Hussain, R. Yesvanth y V. Yuvarajapathi. «Implementing OpenFlow, Exploring the Present and Future Software-Defined Networks Ecosystem». En: *2023 International Conference on Sustainable Communication Networks and Application (ICSCNA)*. 2023, págs. 280-285. DOI: 10.1109/ICSCNA58489.2023.10370398.

[9] M.-F. C. Ndupu, A. A. Adeniyi y S. A. Adeshina. «An Optimized Solution to Harmonized First-Come-First-Serve Based Multiple Queues Using Sets of Predefined Burst Time». En: *2023 2nd International Conference on Multidisciplinary Engineering and Applied Science (ICMEAS)*. Vol. 1. 2023, págs. 1-5. DOI: 10.1109/ICMEAS58693.2023.10379263.

[10] A. Sen, L. Mohammed, R. Samprathi y S. Bandyopadhyay. «Fair queuing with round robin: a new packet scheduling algorithm for routers». En: *Proceedings ISCC 2002 Seventh International Symposium on Computers and Communications*. 2002, págs. 1001-1006. DOI: 10.1109/ISCC.2002.1021794.

[11] W. Chen, Y. Tian, X. Yu, B. Zheng y X. Zhang. «Enhancing Fairness for Approximate Weighted Fair Queueing With a Single Queue». En: *IEEE/ACM Transactions on Networking* 32.5 (2024), págs. 3901-3915. DOI: 10.1109/TNET.2024.3399212.

[12] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker y J. Turner. «OpenFlow: enabling innovation in campus networks». En: *SIGCOMM Comput. Commun. Rev.* 38.2 (2008), págs. 69-74. DOI: 10.1145/1355734.1355746. URL: https://doi.org/10.1145/1355734.1355746.

[13] Open Networking Foundation. *OpenFlow Switch Specification, Version 1.5.1 (Protocol version 0x06)*. Inf. téc. ONF TS-025. Open Networking Foundation (ONF), 2015. URL: https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf.

[14] N. Makondo, H. I. Kobo y T. E. Mathonsi. «The Latest Developments in Software Defined Networking: Adoption Rate and Challenges». En: *2023 IEEE AFRICON*. 2023, págs. 1-6. DOI: 10.1109/AFRICON55910.2023.10293567.

[15] L. Mamushiane, A. Lysko y S. Dlamini. «A comparative evaluation of the performance of popular SDN controllers». En: *2018 Wireless Days (WD)*. 2018, págs. 54-59. DOI: 10.1109/WD.2018.8361694.

[16] T. Čejka y R. Krejčí. «Configuration of open vSwitch using OF-CONFIG». En: *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*. 2016, págs. 883-888. DOI: 10.1109/NOMS.2016.7502920.

[17] X. Zhang, K. Salamatian y G. Xie. «Improving Open Virtual Switch Performance Through Tuple Merge Relaxation in Software Defined Networks». En: *IEEE Transactions on Network and Service Management* 19.3 (2022), págs. 2078-2091. DOI: 10.1109/TNSM.2022.3155592.

[18] P. Documentation. *Introduction to Open vSwitch*. Accessed: 2024-02-23. 2024. URL: https://pica8-fs.atlassian.net/wiki/spaces/PicOS442sp/pages/4268173/Introduction+to+Open+vSwitch.

[19] E. Castro. *Control de tráfico y DiffServ en Linux*. Accessed: 2024-02-23. 2024. URL: https://evacastro.gitbooks.io/internet/content/control_de_trafico.html.

[20] *Open vSwitch Database Configuration*. https://www.openvswitch.org/support/dist-docs/ovs-vswitchd.conf.db.5.txt.

[21] H. Krishna, N. L. M. van Adrichem y F. A. Kuipers. «Providing bandwidth guarantees with OpenFlow». En: *2016 Symposium on Communications and Vehicular Technologies (SCVT)*. 2016, págs. 1-6. DOI: 10.1109/SCVT.2016.7797664.

[22] M. J. Neely. *Stochastic Network Optimization with Application to Communication and Queueing Systems*. Synthesis Lectures on Communication Networks. Morgan & Claypool Publishers, 2010. URL: http://dx.doi.org/10.2200/S00271ED1V01Y201006CNT007.

[23] B. Ji, C. Joo y N. B. Shroff. «Delay-Based Back-Pressure Scheduling in Multihop Wireless Networks». En: *IEEE/ACM Transactions on Networking* 21.5 (2013), págs. 1539-1552. DOI: 10.1109/TNET.2012.2227790.

[24] L. Hai, Q. Gao, J. Wang, H. Zhuang y P. Wang. «Delay-Optimal Back-Pressure Routing Algorithm for Multihop Wireless Networks». En: *IEEE Transactions on Vehicular Technology* 67.3 (2018), págs. 2617-2630. DOI: 10.1109/TVT.2017.2770183.

Apéndice A

Configuración de OVS

En este anexo, se documentan las pruebas realizadas para la configuración de *Open vSwitch* (OVS), con el objetivo de comprender su funcionamiento, familiarizarse con el entorno de desarrollo del proyecto y sus herramientas, y evaluar la integración con el controlador Ryu. Las pruebas se llevaron a cabo utilizando el entorno de virtualización GNS3, que permitió replicar un entorno de red controlado y reproducible, facilitando así el análisis y la validación del comportamiento de OVS en diferentes escenarios. Estas pruebas experimentales constituyen la base de todos los resultados descritos a lo largo de la memoria del proyecto.

Durante el proceso, se emplearon comandos y procedimientos recomendados en la documentación oficial de OVS y de Ryu, asegurando la compatibilidad con las mejores prácticas y la correcta utilización de las herramientas involucradas. El principal objetivo de estas pruebas es validar la interoperabilidad entre OVS y Ryu, así como comprobar la correcta aplicación de políticas de control de tráfico, como la gestión de calidad de servicio (QoS).

A.1. Prueba 1: QoS sobre OVS

En esta primera prueba, se lleva a cabo la configuración de una política de calidad de servicio (QoS) estática en una instancia de OVS sobre una topología muy simple (ver Figura A.1), utilizando los comandos descritos en su documentación oficial y sin la intervención de un controlador SDN. La configuración se centra en establecer una cola de salida (*egress queue*) sobre un interfaz específico del *switch* virtual, limitando el ancho de banda asignado a determinados flujos de tráfico.

Open vSwitch no implementa QoS por sí mismo, sino que emplea *Traffic Control* (TC) de *Linux*. Se expondrán los conceptos necesarios en la propia memoria del proyecto (sección 2.3.2).

El objetivo de esta primera prueba es comprobar y analizar el comportamiento de reglas QoS sobre OVS. En este sencillo escenario aplicamos las siguientes configuraciones sobre el *vSwitch* (Figura A.2). Para ello se ha creado un escenario en GNS3 con un único *switch* y dos usuarios: *user-1* y *user-2*.

Sobre el escenario de la Figura A.1 se configura un QoS sobre el interfaz eth2 de OVS, asignando una tasa máxima de 50 Mbps. Esta será la cola *default* del interfaz eth2 y se añadirá una cola *queue* con una tasa máxima de 10 Mbps asociada a *user-1*. Al tratarse de una *egress policy*, implica que el tráfico es gestionado internamente dentro del *switch*, que se encargará de descartar paquetes cuando se excedan

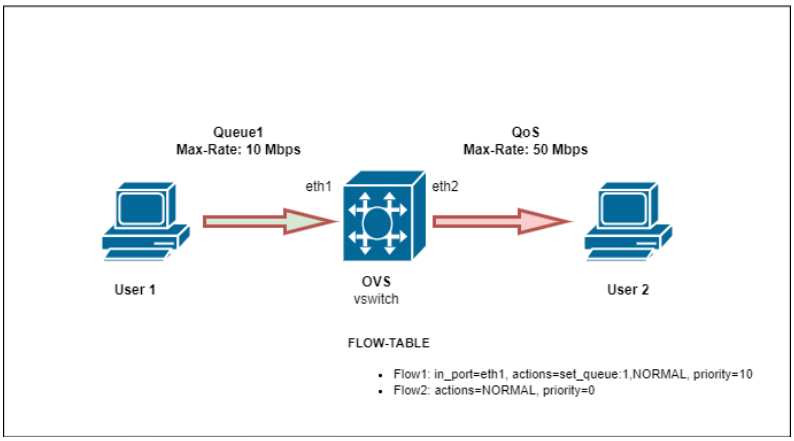


Figura A.1: Prueba 1 - Escenario

```
/ # ovs-vsctl -- \
      add-br br0 -- \
      add-port br0 eth1 -- \
      add-port br0 eth2 -- set interface eth2 ofport_request=2 -- \
      set port eth2 qos=@newqos -- \
      --id=@newqos create qos type=linux-htb \
          other-config:max-rate=50000000 \
          queues:1=@10queue -- \
      --id=@10queue create queue other-config:max-rate=10000000
/ # ovs-vsctl list qos
_uuid               : e69d34c0-3ede-47bf-8656-af79da9557c7
external_ids        : {}
other_config        : {max-rate="50000000"}
queues              : {1=70fa3fb0-596c-4270-a37b-d8e29cdba87c}
type                : linux-htb
/ #
```

Figura A.2: Prueba 1 - Configuración OVS

los límites del *buffer* del interfaz asociado a la cola. Cuando la tasa de transmisión supere el ancho de banda indicado por el *max-rate*, este *buffer* crecerá más rápido y se producirá pérdida de paquetes. En OVS, cuando se crea una QoS *egress policy* sobre un interfaz, se genera una *default queue* para encolar los paquetes que atraviesan el *switch* y son transmitidos por el interfaz. Además, permite crear más *queues* asociadas al QoS, de manera que se pueden crear todas las *queues* que se deseen implementar y aplicar *egress policies* para todos aquellos interfaces cuyo propósito sea el reenvío de paquetes.

Una vez se han aplicado los comandos para crear el *switch* OVS, se verifican las configuraciones aplicadas usando los comandos de la Figura A.2. Se pueden ver las entradas creadas dentro de la *flow-table* y la asignación de los interfaces en la Figura A.3.

A continuación se encienden los usuarios en GNS3 y se realizan mediante *iperf3* las pruebas para comprobar que se está aplicando correctamente el QoS y el tráfico desde *user-1* hacia *user-2* va a través de la cola establecida. En la Figura A.4, se pueden ver las estadísticas de las colas.

El siguiente paso será modificar la tasa máxima de la cola mientras está OVS en funcionamiento y ya hay estadísticas del tráfico a través del *switch*. Esta es una de las primeras pruebas donde se quiere comprobar cómo se comporta el *vSwitch* al recibir comandos que modifican OVSDB cuando ya está gestionando flujos de red. Para ello, se envían los comandos de la Figura A.5. En esta figura podemos ver cómo queda registrada la modificación y estadísticas del tráfico por OVS.

```
/ # ovs-vsctl show
3a06106b-f5ac-4c2d-be38-97192e9333e9
    Bridge "br0"
        Port "eth2"
            Interface "eth2"
        Port "eth1"
            Interface "eth1"
        Port "br0"
            Interface "br0"
                type: internal
    ovs_version: "2.12.3"
/ # ovs-ofctl dump-flows br0
cookie=0x0, duration=55.961s, table=0, n_packets=0, n_bytes=0, priority=10,in_port=eth1
    actions=set_queue:1,NORMAL
cookie=0x0, duration=159.078s, table=0, n_packets=0, n_bytes=0, priority=0 actions=NORMAL
/ #
```

Figura A.3: Prueba 1 - *flow-table* OVS

```
/ # ovs-ofctl dump-flows br0
cookie=0x0, duration=218.533s, table=0, n_packets=71784, n_bytes=106911735,
    priority=10,in_port=eth2 actions=set_queue:1,NORMAL
cookie=0x0, duration=321.649s, table=0, n_packets=31, n_bytes=2483, priority=0
    actions=NORMAL
/ # ovs-appctl qos/show eth2
QoS: eth2 linux-htb
max-rate: 50000000
Default:
    burst: 12512
    min-rate: 12000
    max-rate: 50000000
    tx_packets: 2
    tx_bytes: 140
    tx_errors: 0
Queue 1:
    burst: 12512
    min-rate: 12000
    max-rate: 10000000
    tx_packets: 8618
    tx_bytes: 12795818
    tx_errors: 63166
/ #
```

Figura A.4: Prueba 1 - Estadísticas de las colas *switch* OVS

```
/ # ovs-vsctl set queue 70fa3fb0-596c-4270-a37b-d8e29cdba87c
    other-config=max-rate=30000000
/ # ovs-appctl qos/show eth2
QoS: eth2 linux-htb
max-rate: 50000000
Default:
    burst: 12512
    min-rate: 12000
    max-rate: 50000000
    tx_packets: 0
    tx_bytes: 0
    tx_errors: 0
Queue 1:
    burst: 12512
    min-rate: 12000
    max-rate: 30000000
    tx_packets: 252854
    tx_bytes: 376641012
    tx_errors: 513782
/ #
```

Figura A.5: Prueba 1 - Estadísticas de las colas (modificación de tasa máxima)

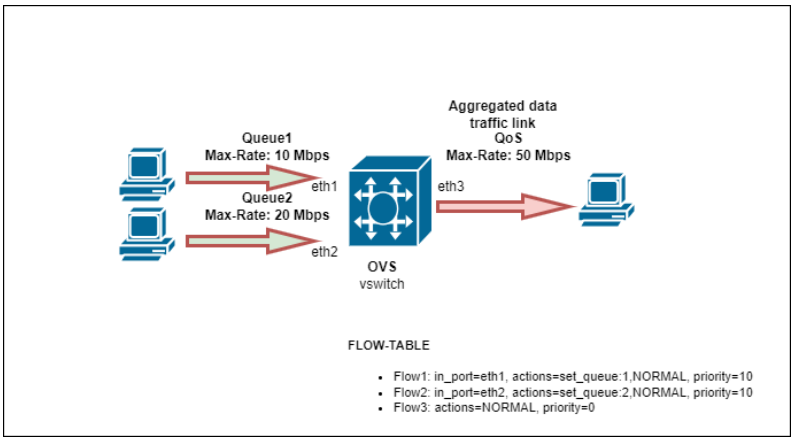


Figura A.6: Prueba 1 - Escenario ampliación

```
Trying ::1...
Connected to localhost.
Escape character is '^]'.
OVS-1 console is now available... Press RETURN to get started.
/ # bin/sh boot.sh > /dev/null 2>&1 &
/ # ovs-vsctl -- \
    add-br br0 -- \
    add-port br0 eth1 -- \
    add-port br0 eth2 -- \
    add-port br0 eth3 -- \
    add-port br0 eth4 -- \
    set port eth3 qos=@newqos -- \
    --id=@newqos create qos type=linux-htb \
        other-config:max-rate=50000000 \
        queues:1=@10queue \
        queues:2=@20queue -- \
    --id=@10queue create queue other-config:max-rate=20000000 -- \
    --id=@20queue create queue other-config:max-rate=10000000
/ # ovs-ofctl add-flow br0 in_port=1,priority=10,actions=set_queue:1,normal
/ # ovs-ofctl add-flow br0 in_port=3,priority=10,actions=set_queue:2,normal
/ #
```

Figura A.7: Prueba 1 - Configuración OVS ampliación usuarios

En este ejemplo en concreto, se accede a los parámetros a través de su *queue id*, y solo para modificar, en este caso, ya que es suficiente para los objetivos del proyecto, el parámetro de tasa máxima: *other config-max rate*.

A.1.1. QoS sobre OVS: ampliación de egress policies

Continuando con esta prueba, se amplía el escenario (Figura A.6). Introducimos un nuevo usuario (*user-2*) y, por lo tanto, entra en juego la gestión de múltiples flujos de tráfico, ya que la idea es controlar múltiples flujos de tráfico a ráfagas. Esto implica la necesidad de ampliar el número de colas asociadas al QoS y asociar cada cola (*queue*) a cada usuario.

En este caso OVS tendrán la misma configuración de la Figura A.7, y se asigna a *user-1* la cola *queue1* con una tasa máxima de 20 Mbps y a *user-2* con una tasa máxima de 10 Mbps.

Se realizan test de ancho de banda con *iperf* verificando que las colas aplicadas están aplicando un QoS a cada usuario consultando las estadísticas de las colas en la Figura A.8.

```
/ # ovs-appctl qos/show eth3
QoS: eth3 linux-htb
max-rate: 50000000
Default:
    burst: 12512
    min-rate: 12000
    max-rate: 50000000
    tx_packets: 109144
    tx_bytes: 163995193
    tx_errors: 192540
Queue 1:
    burst: 12512
    min-rate: 12000
    max-rate: 20000000
    tx_packets: 72444
    tx_bytes: 109639714
    tx_errors: 154
Queue 2:
    burst: 12512
    min-rate: 12000
    max-rate: 10000000
    tx_packets: 9074
    tx_bytes: 13492160
    tx_errors: 240063
/ # ovs-ofctl dump-flows br0
cookie=0x0, duration=399.115s, table=0, n_packets=72600, n_bytes=109873254,
    priority=10,in_port=eth1 actions=set_queue:1,NORMAL
cookie=0x0, duration=398.590s, table=0, n_packets=249139, n_bytes=371180622,
    priority=10,in_port=eth3 actions=set_queue:2,NORMAL
cookie=0x0, duration=966.234s, table=0, n_packets=374495, n_bytes=455849099, priority=0
    actions=NORMAL
/ #
```

Figura A.8: Prueba 2 - Estadísticas de las colas *switch* OVS

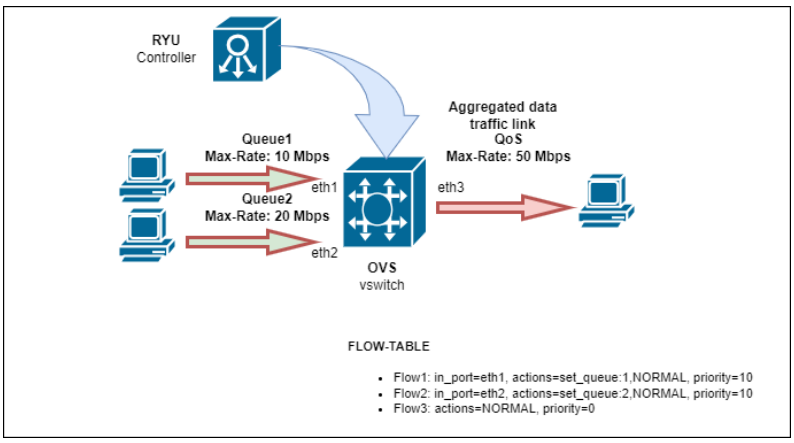


Figura A.9: Prueba 2 - Escenario

A.2. Prueba 2: OVS gestionado por controlador Ryu

Una vez se ha probado la configuración deseada mediante el uso de comandos, se comienza a realizar pruebas introduciendo el controlador. Tanto para la configuración OVSDB de OVS como la lectura de estadísticas con OpenFlow se usará el controlador Ryu. Se comienza con una configuración muy simple de OVSDB empleando la librería *OVSDB Library* de Ryu, que permite al controlador interactuar con el *vSwitch* usando el protocolo OVSDB para enviarle comandos del tipo *ovs-vsctl*. El escenario empleado será el mismo escenario ampliado de la prueba 1, pero incluyendo el controlador Ryu como se puede ver en la Figura A.9.

```
#!/bin/sh
ovsdb-tool create /etc/openvswitch/conf.db /usr/share/openvswitch/vswitch.ovsschema
ovsdb-server --detach --remote=punix:/var/run/openvswitch/db.sock
ovs-vswitchd --detach
ovs-vsctl --no-wait init
/usr/share/openvswitch/scripts/ovs-ctl start
ovsdb-tool create /etc/openvswitch/conf.db /usr/share/openvswitch/vswitch.ovsschema
ovsdb-server --detach --remote=punix:/var/run/openvswitch/db.sock
ovs-vswitchd --detach
ovs-vsctl --no-wait init
/bin/sh
```

Figura A.10: Prueba 2 - *script bash boot.sh* OVS

```
Trying ::1...
Connected to localhost.
Escape character is '^]'.
OVS-1 console is now available... Press RETURN to get started.
/ # bin/sh boot.sh > /dev/null 2>&1 &
/ # ovs-vsctl set-manager ptcp:6640
/ # ovs-vsctl show
d1d848e1-2c7b-4b66-be3c-02825374c0d2
    Manager "ptcp:6640"
    ovs_version: "2.12.3"
/ #
```

Figura A.11: Prueba 2 - Configuración *switch* OVS

En este escenario el *switch* únicamente está configurado para establecer conexión con el controlador mediante el protocolo OVSDB (RFC7047). Se configura OVS para escuchar en el puerto TCP 6640, definido como puerto predeterminado para OVSDB, hasta que el controlador establece conexión. Para ello se utiliza un *script* de *bash* (Figura A.10) para configurar OVS, arrancar la instancia debidamente, configurar el *bridge* con sus puertos e indicarle que recibirá comunicación del controlador en el puerto 6640.

Este *script* (Figura A.10), simplemente se utiliza para arrancar OVSDB y la instancia OVS. Esta información se puede obtener de la documentación oficial de *Open vSwitch* y es independiente del controlador. Una vez OVS está arrancado, se lanzan los comandos de la Figura A.11 para indicarle a la instancia que debe escuchar en el puerto 6640.

A.2.1. Controlador Ryu

El siguiente paso es ejecutar el *script* del controlador Ryu para comprobar que conecta correctamente con el *switch* OVS. En una primera configuración se prueba una parametrización muy simple con un único *bridge*, cuatro puertos asociados y se establece el puerto de escucha OVSDB y el puerto OpenFlow con la dirección IP del controlador. Para ello se crea otro *script* de *bash* (*config.sh* ver en Figura A.12), para configurar OVS donde ya se dispone de los comandos necesarios para crear el *vSwitch*. Este *script* crea la configuración que se puede ver en la Figura A.13.

En la Figura A.13, se obtienen los identificadores para el *bridge* y para el QoS y las colas *queue*. El *script* de Ryu, funciona por eventos, de manera que recibirá información periódicamente de OVS empleando OpenFlow en el puerto TCP 6633.

Se puede ver en la Figura A.13 cómo se le informa a OVS de la IP del controlador Ryu. De la misma

```
#!/bin/sh
ovs-vsctl -- \
    add-br br0 -- \
    add-port br0 eth1 -- set interface eth1 ofport_request=1 --\
    add-port br0 eth2 -- set interface eth2 ofport_request=2 --\
    add-port br0 eth3 -- set interface eth3 ofport_request=3 --\
    add-port br0 eth4 -- set interface eth4 ofport_request=4 --\
    set port eth3 qos=@newqos -- \
    --id=@newqos create qos type=linux-htb \
        other-config:max-rate=50000000 \
        queues:1=@10queue \
        queues:2=@20queue -- \
    --id=@10queue create queue other-config:max-rate=10000000 -- \
    --id=@20queue create queue other-config:max-rate=20000000

ovs-vsctl set-controller br0 "tcp:10.0.2.15:6633"
ovs-vsctl set-manager "ptcp:6640"
```

Figura A.12: Prueba 2 - *script config.sh*

```
/ # ovs-vsctl show
232ea71d-d1b1-46a6-801e-a168ba26fcb1
    Manager "ptcp:6640"
    Bridge "br0"
        Controller "tcp:10.0.2.15:6633"
            is_connected: true
        Port "eth2"
            Interface "eth2"
        Port "eth1"
            Interface "eth1"
        Port "eth3"
            Interface "eth3"
        Port "eth4"
            Interface "eth4"
        Port "br0"
            Interface "br0"
                type: internal
    ovs_version: "2.12.3"
/ # ovs-vsctl list qos
_uuid               : 3cee6bc1-9dbe-46f4-9eaa-94079e336aeb
external_ids        : {}
other_config        : {max-rate="50000000"}
queues              : {0=333521b2-89fd-49c1-a5a0-22a26c936cc9,
    1=e421da06-21f0-4399-9575-b3a3dca6bc77}
type                : linux-htb
/ # ovs-vsctl list queue
_uuid               : e421da06-21f0-4399-9575-b3a3dca6bc77
dscp                : []
external_ids        : {}
other_config        : {max-rate="10000000"}
_uuid               : 333521b2-89fd-49c1-a5a0-22a26c936cc9
dscp                : []
external_ids        : {}
other_config        : {max-rate="20000000"}
/ #
```

Figura A.13: Prueba 2 - Configuración *config.sh* OVS

```
@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
    msg = ev.msg
    dp = ev.msg.datapath
    ofproto = dp.ofproto
    parser = dp.ofproto_parser
    match = parser.OFPMatch()
    actions = [parser.OFPActionOutput(ofproto.OFPP_NORMAL)]
    c.add_flow(datapath=dp, match=match, actions=actions, priority=0)
    match = parser.OFPMatch(in_port=1)
    actions = [parser.OFPActionSetQueue(1), parser.OFPActionOutput(ofproto.OFPP_NORMAL)]
    c.add_flow(datapath=dp, match=match, actions=actions, priority=10)
    match = parser.OFPMatch(in_port=2)
    actions = [parser.OFPActionSetQueue(2), parser.OFPActionOutput(ofproto.OFPP_NORMAL)]
    c.add_flow(datapath=dp, match=match, actions=actions, priority=10)
```

Figura A.14: Prueba 2 - *switch_features_handler*

```
OVSDB_ADDR = "tcp:20.0.0.200:6640"
max_qos_rate = 50000000 # 50 Mbps
queues_config = [
 {'max-rate': str(max_qos_rate)},
 {'max-rate': str(int(max_qos_rate/2))},
 {'max-rate': str(int(max_qos_rate/2))}
]
ovs_bridge = bridge.OVSBridge(self.CONF, datapath_id=dp_id, ovsdb_addr=OVSDB_ADDR)
ovs_bridge.set_qos(port_name='eth3', type='linux-htb', max_rate=str(max_qos_rate),
    queues=queue_config)
```

Figura A.15: Prueba 2 - *set_qos()*

manera, en el controlador han de estar registradas las IP de los *vSwitch* que este vaya a gestionar; es decir, deberán ser conocidas. El controlador cuenta con un manejador *switch_enter_handler*, que registra los *datapath_id*, identificadores únicos de cada *switch* OVS que vaya a gestionar. Estos *datapath_id* serán almacenados en un diccionario para después diferenciar sobre los diferentes *vSwitches* gestionados.

En esta prueba, el controlador ya genera entradas en la *flow-table* del *vSwitch*. Se dispone de la función *switch_features_handler*, que nos permite crear estas entradas y asociar dinámicamente interfaces con colas del QoS. Por ejemplo, se puede ver en la Figura A.14, cómo se asocian al *of_port=1* y al *of_port=2* las colas *queue1* y *queue2* respectivamente .Una vez generadas las entradas en la flow-table, se realizan las mismas pruebas con *iperf* para comprobar que funciona correctamente y se visualizan las estadísticas de las colas generadas. En este caso se procede a manejar directamente los interfaces, pero se podría profundizar mucho más y establecer reglas para tráfico mucho más específicas, ya que dispone de gran variedad de campos en OVS.

Aunque se está realizando una configuración inicial de OVS con el *script config.sh*, Ryu también dispone de la librería *OVSDB library* que permite enviar directamente comandos para modificar OVSDB (*ovs-vsctl* entre otros) convirtiendo al controlador en la parte activa que inicia la comunicación con OVS. En concreto, el método de mayor interés es *set_qos()* dentro de la clase *OVSBridge*, que permite crear un nuevo QoS con sus respectivas *queues*. Se puede ver un ejemplo de uso del método en la Figura A.15.

Con el método *set_qos()* podemos realizar la configuración del *max-rate* para la *default queue* y el resto de colas asociadas al QoS. En el ejemplo de la Figura A.15, se está aplicando un reparto equitativo de la capacidad total agregada definida en la *default queue*. Este es el punto de partida para realizar modificaciones en el QoS y se ha comprobado que cada vez que se ejecuta este método, se crea un nuevo

```
threading.Thread(target=self.send_buffer_stats_request, daemon=True).start()
```

Figura A.16: Prueba 2 - *send_buffer_stats_request*

```
student@student:~/ryu$ ryu-manager prueba.py
loading app prueba.py
loading app ryu.controller.ofp_handler
loading app ryu.topology.Switches
loading app ryu.controller.ofp_handler
instantiating app prueba.py of L2Switch
Valid Connection: True
VSctlCommand(args=['br0'],command='add-br',options=[],result=None)
VSctlCommand(args=['br0', 'eth1'],command='add-port',options=['--may-exist'],result=None)
VSctlCommand(args=['br0', 'eth2'],command='add-port',options=['--may-exist'],result=None)
VSctlCommand(args=['br0', 'eth3'],command='add-port',options=['--may-exist'],result=None)
VSctlCommand(args=['br0', 'eth4'],command='add-port',options=['--may-exist'],result=None)
VSctlCommand(args=['br0', 'tcp:10.0.2.15:6633'],command='set-controller',options=[],result=None)
THREAD ARRANCADO
self.dprefs.items(): dict_keys([])
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ryu.topology.switches of Switches
########################################
OFPSwitchFeatures received:
version=0x4,msg_type=0x6,msg_len=0x20,xid=0x11881eca,
OFPSwitchFeatures(auxiliary_id=0,capabilities=79,datapath_id=46829586864971,n_buffers=0,n_tables=254)
########################################
Received switch enter:
46829586864971
########################################
OFPSwitchFeatures received:
version=0x4,msg_type=0x6,msg_len=0x20,xid=0xedd7e07d,
OFPSwitchFeatures(auxiliary_id=0,capabilities=79,datapath_id=86730037578566,n_buffers=0,n_tables=254)
########################################
Received switch enter:
86730037578566
46829586864971
86730037578566
46829586864971
86730037578566
```

Figura A.17: Ryu *controller* detectando *vswitch* (*threading* testing)

QoS sustituyendo al anterior; es decir, cada vez que se aplica, crea nuevos identificadores para estos elementos.

A.2.2. Controlador Ryu: estadísticas OVS

En esta prueba, se generan las entradas de flujos y se establece conexión con el controlador OpenFlow tal y como se ha visto en la Figura A.14. La creación de flujos se realiza con el método *switch_features_handler*, que es el que responde al evento *EventOFPSwitchFeatures*, y se establece la configuración inicial del QoS mediante el uso *config.sh*. Ahora, se pretende implementar en el controlador el método *send_queue_stats_request* para obtener información sobre las estadísticas de las colas, suscribiéndose al evento *EventOFPQueueStatsReply*. Para ello, se hará uso de la librería de *python threading*, y de esta manera crear un hilo de ejecución paralelo que envíe peticiones sobre las estadísticas de las colas con cierta frecuencia.

El *thread* se genera en el propio inicializador de la clase, tal y como se muestra en la Figura A.16.

Se ha probado que el hilo se genera correctamente y el método al que llama el *thread* será el que se utilice para que el controlador obtenga información sobre las estadísticas y pueda tomar las decisiones sobre cómo aplicar las reglas de QoS. En una primera prueba, el método realiza un *print* de los *datapath_id* del *switch*, ya que el método *send_queue_stats_request* lo requiere como parámetro.

```
loading app prueba.py
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
instantiating app prueba.py of L2Switch
Valid Connection: True
VSctlCommand(args=['br0'],command='add-br',options=[],result=None)
VSctlCommand(args=['br0', 'eth1'],command='add-port',options=['--may-exist'],result=None)
VSctlCommand(args=['br0', 'eth2'],command='add-port',options=['--may-exist'],result=None)
VSctlCommand(args=['br0', 'eth3'],command='add-port',options=['--may-exist'],result=None)
VSctlCommand(args=['br0', 'eth4'],command='add-port',options=['--may-exist'],result=None)
VSctlCommand(args=['br0', 'tcp:10.0.2.15:6633'],command='set-controller',options=[],result=None)
THREAD ARRANCADO
self.dprefs.items(): dict_keys([])
instantiating app ryu.topology.switches of Switches
instantiating app ryu.controller.ofp_handler of OFPHandler
########################################
OFPSwitchFeatures received:
version=0x4,msg_type=0x6,msg_len=0x20,xid=0x8e065992,
OFPSwitchFeatures(auxiliary_id=0,capabilities=79,datapath_id=99232526687042,n_buffers=0,n_tables=254)
########################################
Received switch enter:
SEND QUEUE STATS
port_no=3 queue_id=0 tx_bytes=0 tx_packets=0 tx_errors=0
port_no=3 queue_id=1 tx_bytes=684 tx_packets=2 tx_errors=0
port_no=3 queue_id=2 tx_bytes=735804 tx_packets=486 tx_errors=0

########################################
OFPSwitchFeatures received:
version=0x4,msg_type=0x6,msg_len=0x20,xid=0xfe217baa,
OFPSwitchFeatures(auxiliary_id=0,capabilities=79,datapath_id=86730037578566,n_buffers=0,n_tables=254)
########################################
Received switch enter:
SEND QUEUE STATS
port_no=3 queue_id=0 tx_bytes=684 tx_packets=2 tx_errors=0
port_no=3 queue_id=1 tx_bytes=684 tx_packets=2 tx_errors=0
port_no=3 queue_id=2 tx_bytes=736597 tx_packets=490 tx_errors=0

SEND QUEUE STATS
port_no=3 queue_id=0 tx_bytes=684 tx_packets=2 tx_errors=0
port_no=3 queue_id=1 tx_bytes=5860942 tx_packets=3949 tx_errors=21126
port_no=3 queue_id=2 tx_bytes=736597 tx_packets=490 tx_errors=0

SEND QUEUE STATS
port_no=3 queue_id=0 tx_bytes=684 tx_packets=2 tx_errors=0
port_no=3 queue_id=1 tx_bytes=10746 tx_packets=10746 tx_errors=57451
port_no=3 queue_id=2 tx_bytes=5786796 tx_packets=3837 tx_errors=0
```

Figura A.18: Ryu *controller*: *send_queue_stats_request*

En la Figura A.17 cuando el *thread* arranca se puede observar cómo se muestra por consola, sin embargo, aún no le ha dado tiempo al controlador a recibir los *datapath_id* del *switch*. El *thread* está configurado para hacer la lectura del contenido de los *datapath_id* cada 5 segundos y se puede ver cómo se van mostrando con el paso del tiempo.

Al igual que se ha comprobado durante la prueba del controlador, por algún motivo al levantar el *switch* usando el *script*, se crean dos entradas asociadas al mismo *switch* considerando el puerto *br0* con su propio *datapath_id*

En la Figura A.18 se muestra la salida por consola del estado de las colas, verificando que funciona correctamente el *thread* para mostrar las estadísticas de las colas mediante eventos Openflow. El siguiente paso es por lo tanto buscar la forma de obtener el estado de ocupación de las colas ya que es en definitiva el parámetro que estamos buscando para realizar ajustes en el QoS. Para ello se han realizado pruebas de ancho de banda con *iperf* donde el *user-1* trata de enviar tráfico UDP a 300 Mbps hacia *user-3*. De esta prueba se obtienen las siguientes estadísticas:

En base a estas estadísticas se ha realizado el siguiente diagrama para dar visibilidad a lo que está ocurriendo en el *switch*:

```
/ # ovs-ofctl dump-flows br0
cookie=0x0, duration=207.450s, table=0, n_packets=709486, n_bytes=1057093797, priority=10,in_port=eth1
actions=set_queue:1,NORMAL
cookie=0x0, duration=207.450s, table=0, n_packets=49, n_bytes=5322, priority=0 actions=NORMAL
/ # ovs-ofctl dump-ports br0 eth1
OFPST_PORT reply (xid=0x4): 1 ports
  port eth1: rx pkts=709486, bytes=1057093797, drop=0, errs=0, frame=0, over=0, crc=0
             tx pkts=50, bytes=4874, drop=0, errs=0, coll=0
/ # ovs-ofctl dump-ports br0 eth3
OFPST_PORT reply (xid=0x4): 1 ports
  port eth3: rx pkts=29, bytes=2734, drop=0, errs=0, frame=0, over=0, crc=0
             tx pkts=40233, bytes=59879100, drop=0, errs=0, coll=0
/ # ovs-appctl qos/show eth3
QoS: eth3 linux-htb
max-rate: 50000000
Default:
    burst: 12512
    min-rate: 12000
    max-rate: 50000000
    tx_packets: 16
    tx_bytes: 1723
    tx_errors: 0
Queue 1:
    burst: 12512
    min-rate: 12000
    max-rate: 20000000
    tx_packets: 40209
    tx_bytes: 59876806
    tx_errors: 669274
Queue 2:
    burst: 12512
    min-rate: 12000
    max-rate: 10000000
    tx_packets: 0
    tx_bytes: 0
    tx_errors: 0
/ #
```

Figura A.19: Prueba 2 - Ryu *controller* Estadísticas *buffers* OVS

```
/ # ovs-vsctl show
8648ec4d-d340-403d-b2d9-523ac88a062a
    Bridge br0
        Controller "tcp:10.0.2.15:6633"
        Port eth3
            Interface eth3
        Port eth2
            Interface eth2
        Port br0
            Interface br0
                type: internal
        Port eth4
            Interface eth4
        Port eth1
            Interface eth1
    ovs_version: "2.17.8"
/ # ovs-ofctl dump-flows br0
cookie=0x0, duration=18.800s, table=0, n_packets=0, n_bytes=0, priority=10,in_port=eth1
actions=set_queue:1,NORMAL
cookie=0x0, duration=18.800s, table=0, n_packets=0, n_bytes=0, priority=10,in_port=eth2
actions=set_queue:2,NORMAL
cookie=0x0, duration=18.800s, table=0, n_packets=0, n_bytes=0, priority=0 actions=NORMAL
/ #
```

Figura A.20: Ryu *controller*: Estadísticas OVS

En la siguiente prueba se obtienen los primeros resultados a partir del estado de ocupación de las colas. Mediante el método

A.3. Prueba 3: ingress policing

Open v*vSwitch* también permite crear políticas de *policing*, de manera que un interfaz puede ser configurado para descartar paquetes que superen la tasa establecida:

```
##########################################
----> UPDATING BUFFER STATISTICS for 187915219188041
Printing flow_stats 187915219188041:
{
    "1": {
        "packet_count": 27494,
        "time": 1707338060.6991777
    },
    "2": {
        "packet_count": 0,
        "time": 1707338060.6991801
    }
}
Printing queue_stats 187915219188041:
{
    "0": {
        "time": 1707338060.6984537,
        "tx_errors": 0,
        "tx_packets": 0
    },
    "1": {
        "time": 1707338060.6984565,
        "tx_errors": 10046,
        "tx_packets": 16947
    },
    "2": {
        "time": 1707338060.6984582,
        "tx_errors": 0,
        "tx_packets": 0
    }
}
Buffer actualizado para dp_id=187915219188041, queue_id=1: 501
Buffer actualizado para dp_id=187915219188041, queue_id=2: 0
##########################################
```

Figura A.21: Ryu *controller*: Estadísticas controlador

```
/ # tc -s qdisc show dev eth3
qdisc htb 1: root refcnt 2 r2q 10 default 0x1 direct_packets_stat 0 direct_qlen 1000
 Sent 25230908 bytes 16965 pkt (dropped 10046, overlimits 16946 requeues 0)
 backlog 731800b 496p requeues 0
/ #
```

Figura A.22: Ryu *controller*: Estadísticas QoS basadas en TC (*Traffic Control* Linux)

```
$ ovs-vsctl set interface eth1 ingress_policing_rate=10000
$ ovs-vsctl set interface eth1 ingress_policing_burst=8000
```

Mediante esta secuencia de comandos se configura un QoS que impide que todo el tráfico que entre por el interfaz eth1 con una tasa superior a 10 Mbps sea descartado.

Hacer *policing* del tráfico no es la mejor recomendación según la documentación oficial de OVS, ya que puede interactuar mal con algunos protocolos de red y no comportarse de la manera esperada.

Usando la librería *vsctl* de ryu se puede configurar *ingress policing*:

```
command = vsctl.VSCtlCommand('set',['Interface','eth1','ingress_policing_rate=10000'])
ovs_vsctl.run_command([command])
print(command)
command = vsctl.VSCtlCommand('set',['Interface','eth1','ingress_policing_burst=8000'])
ovs_vsctl.run_command([command])
print(command)
```

Se ha comprobado que el tráfico que entra por el interfaz eth1 no supera los 10 Mbps como era de esperar, pudiendo limitar de esta manera el tráfico que genera, en este caso, el *user-1*. Para comprobar que funciona se han realizado pruebas con *iperf*:

Como se puede ver en la Figura A.23 establecer *policing* no se guarda en OVSDB como un QoS aunque

```
/ # ovs-ofctl dump-ports br0 eth1
OFPST_PORT reply (xid=0x4): 1 ports
  port eth1: rx pkts=953048, bytes=1419908083, drop=0, errs=0, frame=0, over=0, crc=0
             tx pkts=470637, bytes=700957103, drop=0, errs=0, coll=0
/ # ovs-ofctl dump-ports br0 eth3
OFPST_PORT reply (xid=0x4): 1 ports
  port eth3: rx pkts=470523, bytes=700938673, drop=0, errs=0, frame=0, over=0, crc=0
             tx pkts=33298, bytes=49319017, drop=0, errs=0, coll=0
/ # ovs-vsctl list qos
/ #
```

Figura A.23: Ryu *controller*: Estadísticas *ingress policing*

sí se ha verificado que cumple su función y no permite tráfico superior a 10 Mbps en el interfaz eth1. La prueba se ha realizado usando el mismo escenario de la Figura A.9 de manera que *user-1* intenta transmitir tráfico UDP a 300 Mbps hacia *user-3*. En las estadísticas recogidas en los puertos eth1 y eth3 se puede ver que el número de paquetes tx por el interfaz eth1 se corresponde con el número de paquetes recibidos por eth3. Sin embargo, el número de paquetes recibidos por el interfaz eth1 es muy superior, por lo que el número de paquetes descartados se corresponde con la resta:

$$eth1_{drop\ pkts} \approx eth1_{rx\ pkts} - eth1_{tx\ pkts} \approx 953048 - 470637 = 482411$$

Apéndice B

Controlador Ryu

```
#!/usr/bin/env python3
import time
import logging
import sys
import threading
from threading import Lock
import numpy as np
from ryu import cfg
from ryu.lib.ovs import vsctl
from ryu.lib.ovs import bridge
import common as c
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.topology import event
from ryu.controller.handler import MAIN_DISPATCHER, CONFIG_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
import pprint

CONF = cfg.CONF
CONF.register_opts([cfg.IntOpt('ovsdb-timeout', default=2, help='ovsdb timeout')])

log = logging.getLogger()
log.setLevel(logging.INFO)

max_qos_rate = 7000000  # 7 Mbps
offset = 500000  # 0.5 Mbps
max_queue_rate = max_qos_rate - 2 * offset
max_qos_rate_logs = 7
umbral = 20 # Paquetes

queues_eq = [
    {'max-rate': str(max_qos_rate)},
    {'max-rate': str(int(0.5 * max_qos_rate))},
    {'max-rate': str(int(0.5 * max_qos_rate))}
]

OVSDB_ADDR = 'tcp:192.168.122.200:6640'
```

Figura B.1: Librerías y definiciones iniciales

```
class L2Switch(app_manager.RyuApp):
OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

def __init__(self, *args, **kwargs):
    super(L2Switch, self).__init__(*args, **kwargs)
    # Estructuras de datos
    self.datapathsInfo = {}
    self.dprefs = {}
    self.macTable = {}
    self.sw_dp_id = {}
    self.flow_stats = {}
    self.queue_stats = {}
    self.buffer_stats = {}
    self.offset = {}
    self.awaiting_flow_stats = {}
    self.awaiting_queue_stats = {}
    self.last_buffer_stats = {}
    self.average_buffer_stats = {}
    self.mutex_lock = Lock()
    self.delay_buckets = {}
    self.last_tx_packets = {}

    # Iniciar hilos
    threading.Thread(target=self.send_buffer_stats_request, daemon=True).start()
    threading.Thread(target=self.send_queue_load_balance_request, daemon=True).start()
```

Figura B.2: Ryu - Creación de la clase *L2Switch*

```
    @set_ev_cls(event.EventSwitchEnter)
    def switch_enter_handler(self, ev):
        dp = ev.switch.dp
        ofp_parser = dp.ofproto_parser
        req = ofp_parser.OFPPortDescStatsRequest(dp)
        dp.send_msg(req)
        self.dprefs[dp.id] = dp
        self.flow_stats[dp.id] = {}
        self.buffer_stats[dp.id] = {}
        with self.mutex_lock:
            self.last_buffer_stats[dp.id] = {}
            self.delay_buckets[dp.id] = {}
        self.average_buffer_stats[dp.id] = {}
        self.last_tx_packets[dp.id] = {}
        self.awaiting_flow_stats[dp.id] = True
        self.awaiting_queue_stats[dp.id] = True

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        msg = ev.msg
        log.info('########################################')
        log.info('OFPSwitchFeatures received: ')
        log.info(msg)
        log.info('########################################')
        dp = ev.msg.datapath
        ofproto = dp.ofproto
        parser = dp.ofproto_parser
        # Flujo por defecto: NORMAL
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_NORMAL)]
        c.add_flow(datapath=dp, match=match, actions=actions, priority=0)
        # Flujos para asignar cola QoS
        match = parser.OFPMatch(in_port=1)
        actions = [parser.OFPActionSetQueue(1), parser.OFPActionOutput(ofproto.OFPP_NORMAL)]
        c.add_flow(datapath=dp, match=match, actions=actions, priority=10)
        match = parser.OFPMatch(in_port=2)
        actions = [parser.OFPActionSetQueue(2), parser.OFPActionOutput(ofproto.OFPP_NORMAL)]
        c.add_flow(datapath=dp, match=match, actions=actions, priority=10)

    @set_ev_cls(ofp_event.EventOFPPortDescStatsReply, MAIN_DISPATCHER)
    def port_desc_stats_reply_handler(self, ev):
        dp = ev.msg.datapath
        log.info('########################################')
        print('Received Port description of switch {} '.format(dp.id))
        self.datapathsInfo[str(dp.id)] = {}
        port_count = 0
        for port in ev.msg.body:
            self.datapathsInfo[str(dp.id)][str(port.name)] = {
                'port_no': port.port_no,
                'hw_addr': port.hw_addr,
                'curr_speed': port.curr_speed
            }
            port_count += 1
        if port_count > 2:
            self.sw_dp_id[dp.id] = True
        log.info('########################################')
```

Figura B.3: Ryu - Suscripción a eventos

```
    @set_ev_cls(ofp_event.EventOFPQueueStatsReply, MAIN_DISPATCHER)
    def queue_stats_reply_handler(self, ev):
        dp_id = ev.msg.datapath.id
        if dp_id not in self.queue_stats:
            self.queue_stats[dp_id] = {}
            self.offset[dp_id] = {}
            self.last_tx_packets[dp_id] = {}
            for stat in ev.msg.body:
                queue_id = stat.queue_id
                self.offset[dp_id][queue_id] = {
                    'offset_tx_packets': stat.tx_packets,
                    'offset_tx_errors': stat.tx_errors
                }
                current_time = time.time()
                self.queue_stats[dp_id][queue_id] = {'tx_packets': 0, 'tx_errors': 0, 'time':
                    current_time}
                self.last_tx_packets[dp_id][queue_id] = 0
            self.awaiting_queue_stats[dp_id] = False
            self.check_and_update_buffer_stats(dp_id)
        else:
            for stat in ev.msg.body:
                queue_id = stat.queue_id
                current_tx = stat.tx_packets -
                    self.offset[dp_id][queue_id]['offset_tx_packets']
                tx_errors = stat.tx_errors - self.offset[dp_id][queue_id]['offset_tx_errors']
                current_time = time.time()
                self.queue_stats[dp_id][queue_id] = {'tx_packets': current_tx, 'tx_errors':
                    tx_errors, 'time': current_time}
                # Calcular el delta de tx_packets
                if queue_id not in self.last_tx_packets[dp_id]:
                    self.last_tx_packets[dp_id][queue_id] = current_tx
                    delta_tx = current_tx
                else:
                    delta_tx = current_tx - self.last_tx_packets[dp_id][queue_id]
                    self.last_tx_packets[dp_id][queue_id] = current_tx
                self.queue_stats[dp_id][queue_id]['delta_tx'] = delta_tx
            self.awaiting_queue_stats[dp_id] = False
            self.check_and_update_buffer_stats(dp_id)

    @set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
    def flow_stats_reply_handler(self, ev):
        dp_id = ev.msg.datapath.id
        for stat in ev.msg.body:
            queue_id = None
            if stat.instructions and stat.instructions[0].actions[0].type ==
                ofproto_v1_3.OFPAT_SET_QUEUE:
                queue_id = stat.instructions[0].actions[0].queue_id
            packet_count = stat.packet_count
            if queue_id is not None:
                current_time = time.time()
                self.flow_stats[dp_id][queue_id] = {'packet_count': packet_count, 'time':
                    current_time}
        self.awaiting_flow_stats[dp_id] = False
        self.check_and_update_buffer_stats(dp_id)

    def send_queue_stats_request(self, datapath):
        dp_id = datapath.id
        ofp = datapath.ofproto
        ofp_parser = datapath.ofproto_parser
        req = ofp_parser.OFPQueueStatsRequest(datapath, 0, ofp.OFPP_ANY, ofp.OFPQ_ALL)
        datapath.send_msg(req)
        self.awaiting_queue_stats[dp_id] = True

    def send_flow_stats_request(self, datapath):
        dp_id = datapath.id
        ofp = datapath.ofproto
        ofp_parser = datapath.ofproto_parser
        cookie = cookie_mask = 0
        req = ofp_parser.OFPFlowStatsRequest(datapath, 0, ofp.OFPTT_ALL,
                ofp.OFPP_ANY, ofp.OFPG_ANY, cookie, cookie_mask)
        datapath.send_msg(req)
        self.awaiting_flow_stats[dp_id] = True
```

Figura B.4: Ryu - Suscripción a eventos estadísticas

```
    def manage_packet_times_timestamp(self, buckets, dp_id, queue_id, tx_packets,
        buffer_value):
        # Obtener el ultimo valor del buffer registrado
        prev_buffer_stats = self.last_buffer_stats.get(dp_id, {}).get(queue_id, [(0, 0)])
        prev_buffer_value = prev_buffer_stats[-1][0] if prev_buffer_stats else 0
        # Correccion de inconsistencias en tx_packets
        if buffer_value == 0:
            tx_packets = max(tx_packets, prev_buffer_value)  # Si buffer es 0, aseguramos
                transmision total
        elif tx_packets > prev_buffer_value:
            tx_packets = prev_buffer_value  # No puede haber mas transmisiones que paquetes
                en buffer
        # Si el buffer disminuye sin transmision, restar la diferencia mas antigua
        if tx_packets == 0 and buffer_value < prev_buffer_value:
            diff = prev_buffer_value - buffer_value
            for i in range(len(buckets)):
                if diff <= 0:
                    break
                count, arrival_time = buckets[i]
                if count > diff:
                    buckets[i] = (count - diff, arrival_time)
                    diff = 0
                else:
                    diff -= count
                    buckets[i] = (0, arrival_time)
        # Reducir paquetes transmitidos desde los buckets mas antiguos
        for i in range(len(buckets)):
            if tx_packets <= 0:
                break
            count, arrival_time = buckets[i]
            if count > tx_packets:
                buckets[i] = (count - tx_packets, arrival_time)
                tx_packets = 0
            else:
                tx_packets -= count
                buckets[i] = (0, arrival_time)
        # Eliminar solo los buckets vacios
        buckets = [(cnt, ts) for cnt, ts in buckets if cnt > 0]
        # Asegurar que la suma de los buckets es igual al buffer
        current_sum = sum(count for count, _ in buckets)
        delta_new = buffer_value - current_sum
        # Si el buffer ha aumentado, agregar un nuevo bucket
        if delta_new > 0:
            buckets.append((delta_new, time.time()))
        return buckets
```

Figura B.5: Ryu - Gestión de los retardos acumulados

```
    def send_buffer_stats_request(self):
        while True:
            time.sleep(0.0010)
            for datapath_id in self.dprefs.keys():
                datapath = self.dprefs[datapath_id]
                self.send_queue_stats_request(datapath)
                self.send_flow_stats_request(datapath)

    def send_queue_load_balance_request(self):
        while True:
            time.sleep(0.010)
            for datapath_id in self.dprefs.keys():
                self.queue_load_balance(datapath_id)

    ##########################################################
    def check_and_update_buffer_stats(self, dp_id):
        if not self.awaiting_flow_stats[dp_id] and not self.awaiting_queue_stats[dp_id]:
            self.update_buffer_stats(dp_id)
```

Figura B.6: Ryu - Métodos de los *threads*

```
    def update_buffer_stats(self, dp_id):
        if dp_id in self.sw_dp_id:
            if dp_id in self.flow_stats and dp_id in self.queue_stats:
                for queue_id, flow_stat in self.flow_stats[dp_id].items():
                    if queue_id in self.queue_stats[dp_id]:
                        queue_stat = self.queue_stats[dp_id][queue_id]
                        packet_count = flow_stat.get('packet_count', 0)
                        tx_packets = queue_stat.get('tx_packets', 0)
                        tx_errors = queue_stat.get('tx_errors', 0)
                        if packet_count == tx_packets + tx_errors + 2:
                            packet_count -= 2  # Corrige el offset fantasma si lo detecta
                        # Calcula la ocupacion actual del buffer
                        buffer_value = max(packet_count - (tx_packets + tx_errors), 0)

                        buffer_timestamp = time.time()
                        self.buffer_stats[dp_id][queue_id] = {'buffer_value': buffer_value,
                            'Timestamp': buffer_timestamp}

                        # Registro en log de ocupacion
                        if queue_id == 1:
                            file_path = 'buffer_stats_queue1.txt'
                        elif queue_id == 2:
                            file_path = 'buffer_stats_queue2.txt'
                        else:
                            continue
                        with open(file_path, 'a') as file:
                            file.write(f'{buffer_value} {buffer_timestamp}\n')

                        # Actualizar la lista historica de ocupacion
                        with self.mutex_lock:
                            if queue_id not in self.last_buffer_stats[dp_id]:
                                self.last_buffer_stats[dp_id][queue_id] = []
                            self.last_buffer_stats[dp_id][queue_id].append((buffer_value,
                                buffer_timestamp))
                            self.last_buffer_stats[dp_id][queue_id] =
                                self.last_buffer_stats[dp_id][queue_id][-5:]

                            # Actualizar los buckets de retardo usando marcas de tiempo
                            if dp_id not in self.delay_buckets:
                                self.delay_buckets[dp_id] = {}
                            if queue_id not in self.delay_buckets[dp_id]:
                                self.delay_buckets[dp_id][queue_id] = []
                            # En lugar de pasar directamente buffer_value, calculamos lo
                                "nuevo" a agregar
                            current_bucket_sum = sum(count for count, _ in
                                self.delay_buckets[dp_id][queue_id])
                            additional = buffer_value - current_bucket_sum  # Diferencia para
                                ajustar a la ocupacion actual
                            # Llamamos a manage_packet_times_timestamp usando el delta de tx
                                (delta_tx)
                            delta_tx = self.queue_stats[dp_id][queue_id].get('delta_tx', 0)
                            delta_tx = max(delta_tx, 0)
                            updated_buckets = self.manage_packet_times_timestamp(
                                self.delay_buckets[dp_id][queue_id],
                                dp_id,
                                queue_id,
                                delta_tx,  # Este es el numero de paquetes transmitidos
                                    (calculado en queue_stats_reply_handler)
                                buffer_value  # Este es el valor actual del buffer
                            )
                            self.delay_buckets[dp_id][queue_id] = updated_buckets

            else:
                log.warning(f"No hay estadisticas suficientes para dp_id={dp_id} para
                    actualizar buffer_stats.")
```

Figura B.7: Ryu - Método *update_buffer_stats()*

```
    def queue_load_balance(self, dp_id):
        inicio = time.time()
        if dp_id in self.sw_dp_id:
            with self.mutex_lock:
                last_values_queue_1 = [val[0] for val in self.last_buffer_stats[dp_id].get(1,
                    [])]
                last_values_queue_2 = [val[0] for val in self.last_buffer_stats[dp_id].get(2,
                    [])]
                avg_queue_1 = np.mean(last_values_queue_1) if last_values_queue_1 else 0
                avg_queue_2 = np.mean(last_values_queue_2) if last_values_queue_2 else 0

                # Calculamos el retardo para cada cola
                current_time = time.time()
                # Queue 1:
                delay_buckets_q1 = self.delay_buckets[dp_id].get(1, [])
                total_packets_q1 = sum(cnt for cnt, _ in delay_buckets_q1)
                # Retardo promedio de la cola 1
                mean_delay_q1 = (sum(cnt * (current_time - ts) for cnt, ts in
                    delay_buckets_q1) / total_packets_q1
                                 if total_packets_q1 > 0 else 0)
                # Retardo acumulado cola 1
                total_delay_q1 = sum(cnt * (current_time - ts) for cnt, ts in
                    delay_buckets_q1)
                # Retardo maximo cola 1
                max_delay_q1 = max((current_time - ts) for cnt, ts in delay_buckets_q1) if
                    delay_buckets_q1 else 0

                # Queue 2:
                delay_buckets_q2 = self.delay_buckets[dp_id].get(2, [])
                total_packets_q2 = sum(cnt for cnt, _ in delay_buckets_q2)
                # Retardo promedio cola 2
                mean_delay_q2 = (sum(cnt * (current_time - ts) for cnt, ts in
                    delay_buckets_q2) / total_packets_q2
                                 if total_packets_q2 > 0 else 0)
                # Retardo acumulado cola 2
                total_delay_q2 = sum(cnt * (current_time - ts) for cnt, ts in
                    delay_buckets_q2)
                # Retardo maximo cola 2
                max_delay_q2 = max((current_time - ts) for cnt, ts in delay_buckets_q2) if
                    delay_buckets_q2 else 0

                fin_shared_dictionary = time.time()

            # Criterio de decision
            .
            .
```

Figura B.8: Ryu - Método *queue_load_balance()*

```
    def queue_load_balance(self, dp_id):
        inicio = time.time()
        if dp_id in self.sw_dp_id:
            .
            .
            # Criterio de decision
            metrica_queue1 = total_delay_q1
            metrica_queue2 = total_delay_q2

            if metrica_queue1 == avg_queue_1:
                if avg_queue_1 == 0 and avg_queue_2 == 0:
                    avg_1, avg_2 = 0.5, 0.5
                    action_taken = "Queues_eq"
                    queue_config = queues_eq
                else:
                    if avg_queue_1 + avg_queue_2 <= umbral:
                        avg_1, avg_2 = 0.5, 0.5
                        action_taken = "Queues_eq"
                        queue_config = [
                                {'max-rate': str(max_qos_rate)},
                                {'max-rate': str(int(avg_1 * max_qos_rate))},
                                {'max-rate': str(int(avg_2 * max_qos_rate))}
                            ]
                    else:
                        total = avg_queue_1 + avg_queue_2
                        avg_1 = avg_queue_1 / total if total != 0 else 0
                        avg_2 = avg_queue_2 / total if total != 0 else 0
                        queue_config = [
                            {'max-rate': str(max_qos_rate)},
                            {'max-rate': str(int(avg_1 * max_queue_rate + offset))},
                            {'max-rate': str(int(avg_2 * max_queue_rate + offset))}
                        ]
                        action_taken = "QueuesAdaptative"
            else:
            # Caso1
                if metrica_queue1 == 0 and metrica_queue2 == 0:
                    avg_1, avg_2 = 0.5, 0.5
                    action_taken = "QueuesAdaptative"
                    queue_config = queues_eq
                # Caso2
                else:
                    total = metrica_queue1 + metrica_queue2
                    avg_1 = metrica_queue1 / total if total != 0 else 0
                    avg_2 = metrica_queue2 / total if total != 0 else 0
                    queue_config = [
                        {'max-rate': str(max_qos_rate)},
                        {'max-rate': str(int(avg_1 * max_queue_rate + offset))},
                        {'max-rate': str(int(avg_2 * max_queue_rate + offset))}
                    ]
                    action_taken = "QueuesAdaptative"

            try:
                inicio_bridge = time.time()
                ovs_bridge = bridge.OVSBridge(self.CONF, datapath_id=dp_id,
                    ovsdb_addr=OVSDB_ADDR)
                inicio_ovsdb = time.time()
                ovs_bridge.set_qos(port_name='eth3', type='linux-htb',
                    max_rate=str(max_qos_rate), queues=queue_config)
            except Exception as e:
                log.error(f"Error setting QoS: {str(e)}")
```

Figura B.9: Ryu - Método *queue_load_balance()* - Modificación OVSDB