Universidad Nacional de San Martín
Escuela de Ciencia y Tecnología
Ingeniería en Telecomunicaciones

UNIVERSIDAD
NACIONAL DE
SAN MARTÍN

Proyecto Final Integrador

Desarrollo y análisis de mecanismos de calidad de servicio sobre redes definidas por software

Joaquín Gonzalez
joagonzalez@gmail.com

Tutora:
Dra. María Claudia Abeledo
mabeledo@unsam.edu.ar

27 de octubre de 2020

Resumen

Garantizar la calidad de los servicios que se ofrecen sobre redes de comunicaciones es una problemática vigente en la industria. El marco de arquitectura propuesto por las redes definidas por software (SDN) y el esfuerzo de consorcios de empresas líderes en el rubro han permitido comenzar a estandarizar las interfaces que exponen los elementos de red que componen estas infraestructuras.

En este proyecto se diseña, desarrolla y analiza una aplicación de software que permite implementar funcionalidades de calidad de servicio y admisión de llamadas en tiempo real para comunicaciones realizadas por usuarios de una central telefónica IP. Para ello, se utiliza el modelo de arquitectura y protocolos propuestos por el modelo SDN así como controladores de red que exponen interfaces programables a través de las cuales se comunican intenciones a la red en tiempo real.

También, se ha desarrollado un laboratorio que implementa una topología de red virtualizando componentes para evaluar a la aplicación en un escenario productivo.

Este trabajo tiene por objetivo evaluar, a través de un caso práctico concreto, la arquitectura SDN a través de la implementación de dos funcionalidades específicas vinculadas a la calidad de servicio sobre redes IP. Además, se hará hincapié en cómo los componentes de red comienzan a ofrecerse como software. Esta coyuntura abre las puertas a metodologías que afectan tanto la operación, como el desarrollo y la implementación de las redes y sus funcionalidades. Se utiliza el marco metodológico de DevOps/NetOps durante el ciclo de vida completo de la aplicación.

Índice

1. Anteproyecto

Previa a la descripción del desarrollo de este trabajo que se realiza en el marco del Proyecto Final Integrador para la carrera de grado de Ingeniería en Telecomunicaciones se describen los lineamientos donde se identifican y describen partes esenciales del análisis anterior a su realización. Entre ellas, se encuentra el acta constitutiva, el desglose de tareas a realizar (*work breakdown structure*), el diagrama Gantt con el detalle de las tareas y los tiempos que estas conllevan y el análisis de los riesgos principales que deben ser considerados durante la realización del proyecto. Por último, se detalla un análisis de costos para el desarrollo de la aplicación.

1.1. Acta constitutiva

1.1.1. Justificación

Las empresas proveedoras de servicios de telecomunicaciones invierten un gran porcentaje de sus ingresos en la operación y el mantenimiento de sus infraestructuras. Las redes de datos tradicionales, una vez desplegadas, tienen altos costos de mantenimiento (OPEX [1]) debido a que responden a un modelo tecnológico poco versátil que no se ha adaptado a las nuevas necesidades y servicios que ofrecen las redes convergentes. Las Redes Definidas por Software (SDN) proponen un cambio de paradigma en el modelo de operación tradicional, así como la introducción de nuevos protocolos y especificaciones que permiten la implementación de redes más versátiles y automatizables. Actualmente, las tecnologías y estándares propuestos por las Redes Definidas por Software se encuentran en constante evolución a través de consorcios como el Open Networking Foundation (ONF) en donde participan empresas como Google, Microsoft, Intel, Huawei, Dell, Juniper, Cisco e instituciones como Stanford University, IEEE, entre otras. La tecnología SDN propone la separación del plano de control del plano de datos existente en los equipamientos de telecomunicaciones, así como el desarrollo de interfaces programables abiertas (APIs) para la comunicación entre la capa de control y las aplicaciones que utilizarán recursos en una red. De esta manera, se define el rol de controlador dentro de las arquitecturas SDN, esta entidad ejecutará el sistema operativo de la red y será el nexo de las comunicaciones entre las aplicaciones y el plano de datos. El controlador tendrá definidas tres interfaces principales:

- Interfaz norte: Usualmente interfaces API Rest o WebSocket que permiten interacción en tiempo real entre el controlador y las aplicaciones que utilizarán la red (VoIP, Chat, HTTP, monitoreo, etc)
- Interfaz sur: Comunicación entre el controlador y el plano de datos de la red. El protocolo que se ha impuesto hasta el momento es OpenFlow
- Interfaz este/oeste: Comunicación entre distintos controladores. Permite obtener una visión homogénea de la red, así como la posibilidad de tener arquitecturas redundantes y escalables en el plano de control

Una de las funcionalidades más interesantes que permiten las arquitecturas SDN es la implementación de mecanismos de calidad de servicio en tiempo real. Es decir, la reserva de recursos de ancho de banda en las interfaces de los equipamientos de red podrá disponibilizarse en el momento en el que es requerido por una aplicación, optimizando la distribución y el uso del ancho de banda total de la red. Esto tiene como ventaja, además, la reducción de fuentes de error humano en la configuración de estos mecanismos. Esto permitirá mejorar la experiencia final de los usuarios que utilicen la infraestructura y reducir costos debido a la optimización de los recursos de la red.

En soluciones de Voz sobre IP, los mecanismos de calidad de servicio suelen complementarse con una funcionalidad llamada Call Admission Control. Esta aplicación restringe de manera granular la cantidad de sesiones concurrentes que pueden establecerse entre distintas subredes. Este umbral de sesiones concurrentes está emparejado con el ancho de banda asignado a las colas de audio y video de los mecanismos de calidad de servicio. Estas dos funcionalidades, en conjunto, solo permitirán llamadas cuya calidad pueda garantizarse, el resto serán rechazadas ya que generaran un desborde en las colas de audio y video configuradas y comenzarán a experimentar pérdida de paquetes.

Con este proyecto se pretende diseñar, analizar e implementar una aplicación de Call Admission Control y calidad de servicio en tiempo real para aplicaciones de Voz sobre IP sobre una arquitectura de red SDN utilizando interfaces API Rest y el protocolo OpenFlow [39]. La misma será estudiada a través de simulaciones para escenarios de red específicos utilizando Asterisk [1], OpenVirtualSwitch [6], Mininet [37], Docker Inc. [36] y Ryu [11], todos productos de software OpenSource maduros que han sido utilizados en diversas investigaciones por la comunidad científica y en diversos productos por la industria de las telecomunicaciones. Este tipo de soluciones permitirán reducir costos de operación e implementación en comparación con arquitecturas de redes convencionales.

[1] OPEX: Gastos operativos.

1.1.2. Objetivo General

Diseño, análisis e implementación de una aplicación de Call Admission Control y calidad de servicio en tiempo real para aplicaciones de Voz sobre IP sobre una arquitectura de red SDN.

1.1.3. Alcance

El alcance del proyecto se enfoca en el diseño, análisis e implementación de un prototipo de aplicación desarrollada en Python [42] que contará con los módulos necesarios para poder interactuar con aplicaciones de Voz sobre IP y controladores SDN. La aplicación contará, además, con un módulo de Front End para poder visualizar los cambios realizados en la red y realizar configuraciones específicas como el umbral de llamadas concurrentes permitido por la política de Call Admission Control. La aplicación será testeada sobre maquetas de simulación utilizando Docker Containers [36] y Mininet [37] con el objetivo de verificar funcionalidades, performance y analizar viabilidad de portabilidad a hardware.

Se define una topología de red sobre la cual será testeada la aplicación y se analizará el tráfico así como la señalización de las llamadas para evaluar su comportamiento. Además, se validarán las configuraciones impactadas por la aplicación utilizando consultas a las APIs del controlador y de las central IP y se realizarán pruebas sobre el funcionamiento del *shaping* realizado por los elementos de red.

El software de switch SDN a utilizar es OpenVirtualSwitch [6], el cual permite portabilidad a whiteboxes [2] por lo que se espera obtener una aplicación funcional en ambientes físicos. Si bien la aplicación tendrá la capacidad de interactuar con switches SDN implementados en hardware, no se encuentra dentro del alcance de este proyecto el análisis de estos escenarios.

1.1.4. Grupos de interés

Cliente directo:	Newtech Multimedia S.A
Clientes indirectos:	Clientes y Partners de Newtech Multimedia S.A
Team Member:	Joaquin Gonzalez
Tutora:	Dra. María Claudia Abeledo

[2] Dispositivos de hardware genéricos.

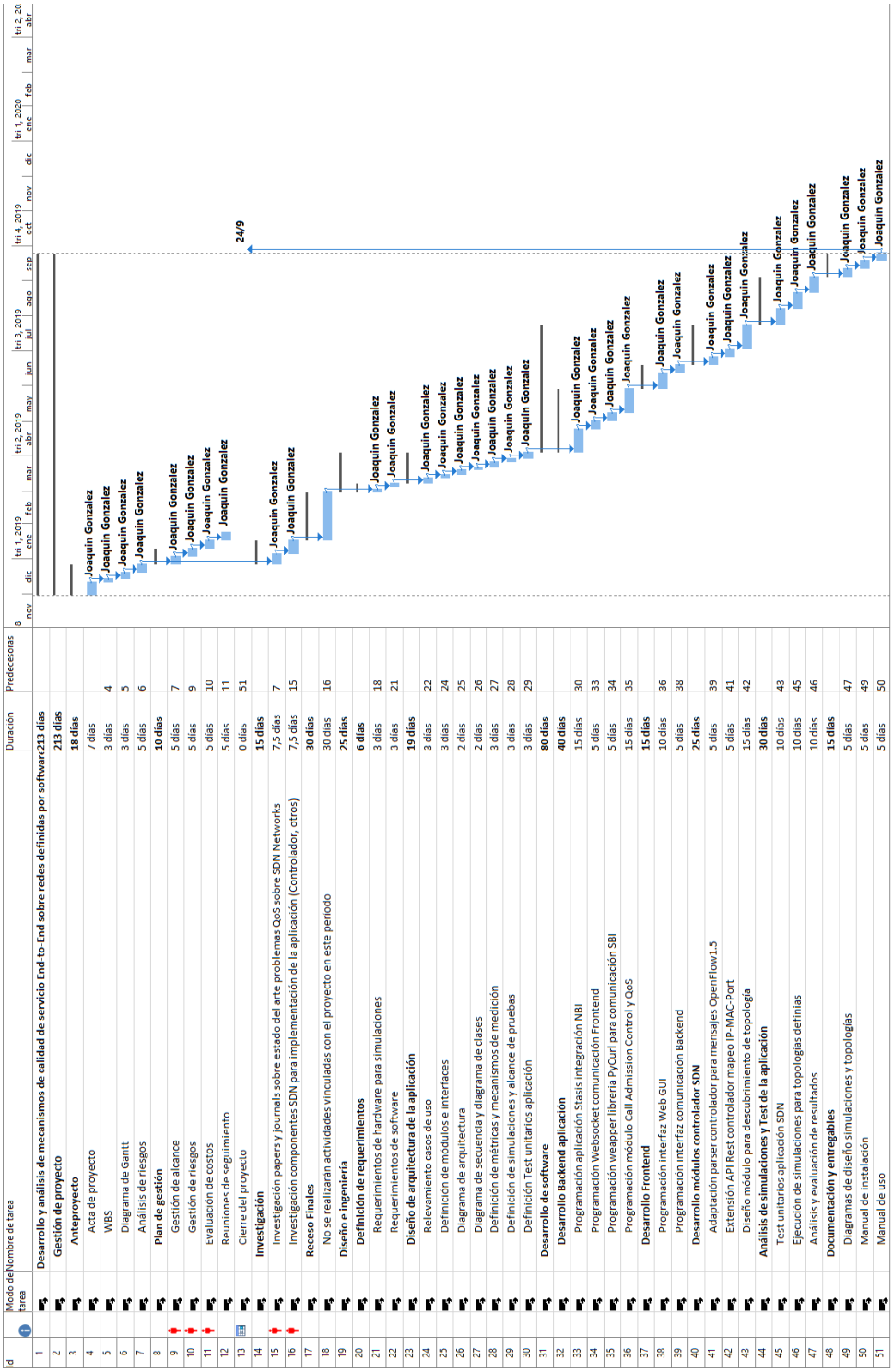
1.2. WBS

El WBS (Work Breakdown Structure) es una herramienta que realiza una descomposición jerárquica del trabajo a ser ejecutado por el equipo del proyecto que está orientada al producto entregable. Descompone el proyecto en tareas y subtareas que permite organizar la realización del proyecto así como dimensionar el esfuerzo que este conlleva.

Desarrollo y análisis de mecanismos de calidad de servicio sobre redes SDN	
1. Gestión de Proyecto	
1.1	**Anteproyecto**
1.1.1	Acta de proyecto
1.1.2	WBS
1.1.3	Diagrama de Gantt
1.1.4	Análisis de Riesgos
1.2	**Plan de Gestión**
1.2.1	Gestión de alcance
1.2.2	Gestión de riesgos
1.3	**Evaluación de costos**
1.4	**Cierre del proyecto**
2. Investigación	
2.1	Investigación papers y journals sobre estado del arte QoS en SDN
2.2	Investigación componentes SDN para aplicación
2.2	Investigación mecanismos QoS sobre redes tradicionales
3. Diseño e ingeniería	
3.1	**Definición de requerimientos**
3.1.1	Requerimientos de hardware para simulaciones
3.1.2	Requerimientos de software
3.2	**Diseño de arquitectura de la aplicación**
3.2.1	Relevamiento de casos de uso
3.2.2	Definición de módulos e interfaces
3.2.3	Diagrama de arquitectura
3.2.4	Diagrama de secuencia y diagramas de clase
3.2.5	Definición de métricas y mecanismos de medición
3.2.6	Definición de simulaciones y alcance de pruebas
3.2.7	Evaluación de implementación de testing
4. Desarrollo de software	
4.1	**Desarrollo Backend aplicación**
4.1.1	Programación aplicación Stasis integración NBI
4.1.2	Programación Websocket comunicación Frontend
4.1.3	Programación wrapper librería Pycurl para comunicación SBI
4.1.4	Programación módulo Call Admission Control y QoS
4.2	**Desarrollo Frontend**
4.2.1	Programación interfaz Web GUI
4.2.2	Programación interfaz comunicación Backend
4.2.2	Integración módulo topología de controlador SDN
4.3	**Desarrollo módulos controlador SDN**
4.3.1	Extensión API Rest controlador
4.3.2	Extensión aplicación para soporte de colas QoS
5. Análisis de simulaciones y Test de aplicación	
5.1	Test unitarios aplicación SDN
5.2	Ejecución de simulaciones para topología definida
5.3	Análisis y evaluación de resultados
6. Documentación y entregables	
6.1	Diagramas de diseño de la aplicación
6.2	Diagramas de diseño simulaciones y topologías
6.3	Documentación análisis, evaluación y conclusiones de resultados obtenidos
6.4	Manual de instalación
6.5	Manual de uso

1.3. Gantt

Utilizando el WBS desarrollado en la sección anterior, se construye el diagrama de Gantt. De esta manera, se expone la planificación temporal de las tareas involucradas en el desarrollo del proyecto.



Id	Modo de tarea	Nombre de tarea	Duración	Predecesoras
1		**Desarrollo y análisis de mecanismos de calidad de servicio End-to-End sobre redes definidas por software**	**213 días**	
2		**Gestión de proyecto**	**213 días**	
3		**Anteproyecto**	**18 días**	
4		Acta de proyecto	7 días	
5		WBS	3 días	4
6		Diagrama de Gantt	3 días	5
7		Análisis de riesgos	5 días	6
8		**Plan de gestión**	**10 días**	
9		Gestión de alcance	5 días	7
10		Gestión de riesgos	5 días	9
11		Evaluación de costos	5 días	10
12		Reuniones de seguimiento	5 días	11
13		Cierre del proyecto	0 días	51
14		**Investigación**	**15 días**	
15		Investigación papers y journals sobre estado del arte problemas QoS sobre SDN Networks	7,5 días	7
16		Investigación componentes SDN para implementación de la aplicación (Controlador, otros)	7,5 días	15
17		**Receso Finales**	**30 días**	
18		No se realizarán actividades vinculadas con el proyecto en este período	30 días	16
19		**Diseño e ingeniería**	**25 días**	
20		**Definición de requerimientos**	**6 días**	
21		Requerimientos de hardware para simulaciones	3 días	18
22		Requerimientos de software	3 días	21
23		**Diseño de arquitectura de la aplicación**	**19 días**	
24		Relevamiento casos de uso	3 días	22
25		Definición de módulos e interfaces	3 días	24
26		Diagrama de arquitectura	2 días	25
27		Diagrama de secuencia y diagrama de clases	2 días	26
28		Definición de métricas y mecanismos de medición	3 días	27
29		Definición de simulaciones y alcance de pruebas	3 días	28
30		Definición Test unitarios aplicación	3 días	29
31		**Desarrollo de software**	**80 días**	
32		**Desarrollo Backend aplicación**	**40 días**	
33		Programación aplicación Stasis integración NBI	15 días	30
34		Programación Websocket comunicación Frontend	5 días	33
35		Programación weapper libreria PyCurl para comunicación SBI	5 días	34
36		Programación módulo Call Admission Control y QoS	15 días	35
37		**Desarrollo Frontend**	**15 días**	
38		Programación interfaz Web GUI	10 días	36
39		Programación interfaz comunicación Backend	5 días	38
40		**Desarrollo módulos controlador SDN**	**25 días**	
41		Adaptación parser controlador para mensajes OpenFlow1.5	5 días	39
42		Extensión API Rest controlador mapeo IP-MAC-Port	5 días	41
43		Diseño módulo para descubrimiento de topología	15 días	42
44		**Análisis de simulaciones y Test de la aplicación**	**30 días**	
45		Test unitarios aplicación SDN	10 días	43
46		Ejecución de simulaciones para topologías definias	10 días	45
47		Análisis y evaluación de resultados	10 días	46
48		**Documentación y entregables**	**15 días**	
49		Diagramas de diseño simulaciones y topologías	5 días	47
50		Manual de instalación	5 días	49
51		Manual de uso	5 días	50

1.4. Análisis de riesgos

En esta sección se identifican los posibles riesgos inherentes a un proyecto de desarrollo de software. Se analizarán posibles riesgos existentes en las etapas de relevamiento (análisis funcional y no funcional), diseño, desarrollo e implementación así como documentación.

Se tendrán en consideración dos métricas que darán la información necesaria para realizar el análisis.

- Probabilidad de ocurrencia
- Impacto

Valor	Grado
1	Muy baja
2	Baja
3	Media
4	Alta
5	Muy alta

Cuadro 1: Probabilidad de ocurrencia de un riesgo

Valor	Nivel	Descripción
1	Insignificante	Impacta levemente en el desarrollo del proyecto
2	Menor	Impacta en el desarrollo del proyecto
3	Moderado	Impacta en la operatividad del marco del proyecto
4	Mayor	Impacta en la operatividad de los procesos del proyecto
5	Desastroso	Impacta fuertemente en la operatividad

Cuadro 2: Impacto de los riesgos en el proyecto

1.4.1. Cuantificación y evaluación de riesgos

Se identifican y cuantifican los diversos riesgos asociados a la ejecución del proyecto.

Riesgos	**Prob.**	**Impacto**	**Descripción**	**Plan de mitigación**
Estimación de esfuerzo	3	4	Mala estimación en el esfuerzo o tiempos en la ejecución de tareas	Aplicar metodología de trabajo que permita revisión y retroalimentación constante sobre el estado de las tareas
Cambios de alcance y requerimientos	3	3	Ante la complejidad de delimitar alcance del proyecto pueden surgir cambios no previstos en etapas de relevamiento	Revisión de priorización de tareas ante cambios para no afectar curso normal del proyecto
Baja productividad	2	3	Asociado al desempeño del equipo de trabajo a cargo del desarrollo	Contratación de personal asumiendo cambio en costo
Fallas/Bugs en tiempos de desarrollo	4	1	Fallas no previstas que ocasionan un comportamiento inesperado de la aplicación	Trabajar con sistemas de versionado y ambientes de desarrollo y producción separados que permitan rápida aplicación de correcciones
Fallas y errores técnicos en infraestructura crítica	1	4	Infraestructura imprescindible para las tareas de desarrollo	Utilización de herramientas que permitan montar ambiente de desarrollo de manera ágil como docker y git
Cambios en definiciones de protocolos utilizados	3	1	Protocolos y estándares sufren modificaciones que pueden desencadenar modificaciones en módulos de la aplicación	Desarrollar interfaces conectores/adaptadores que permitan una rápida adaptación a cambios de tecnologías fuera del núcleo de la aplicación
Subestimar complejidades técnicas del desarrollo	2	3	No contemplar dificultad de problemas técnicos a resolver para el desarrollo de la aplicación	Inversión en estudio y relevamiento de las tecnologías a utilizar. Trabajar realizando simulaciones que permitan entender factibilidad técnica de los diseños
Uso de tecnologías nuevas y poco probadas	4	1	Tecnologías que aún están siendo definidas pueden generar comportamientos inesperados	Utilizar versiones estables e implementar cambios luego de ejecución de test sobre nuevas releases de tecnologías
Promedio	2.75	2.50		

1.4.2. Análisis y conclusiones

Los riesgos del proyecto se encuentran en una zona media. Esto indica que la realización del proyecto es viable siempre y cuando los mismos sean tenidos en cuenta con el objetivo de mitigar sus efectos lo antes posible.

1.5. Análisis de costos

En esta sección se desarrolla un resumen de los costos del presente proyecto.

Para el análisis de costos se contempla una inversión inicial a través de un préstamos bancario que ayudará a costear gastos operativos iniciales y las horas de trabajo del desarrollador asignado a la programación de la aplicación. Este préstamo tendrá costos de financiación que son tenidos en cuenta en este análisis.

Los costos operativos del proyecto están asociados al alquiler de infraestructura necesaria para la realización de tareas de desarrollo y ejecución de simulaciones. Por otro lado, con el objetivo de generar más interés en el producto, se ofrece un kit de desarrollo junto con el software para poder realizar pruebas de la aplicación con hardware real, estos costos son considerados a la hora de calcular el precio del producto con un margen de ganancia de 35 % por unidad vendida.

A continuación, se adjuntan los detalles del análisis de costos así como una proyección de ingresos para los primeros 5 años de vida del producto.

	Año 1	**Año 2**	**Año 3**	**Año 4**	**Año 5**
Proyección de venta	0	50	100	200	300
Ingreso de ventas	u$s 0	u$s 33.750	u$s 67.500	u$s 135.000	u$s 202.500
Costo B/S	u$s 16.641	u$s 21.250	u$s 42.500	u$s 85.000	u$s 127.500
Margen Bruto	u$s -16.641	u$s 12.500	u$s 25.000	u$s 50.000	u$s 75.000
Gs. Operativos	u$s 0	u$s 11.621	u$s 11.621	u$s 11.621	u$s 11.621
Margen Operativo	u$s -16.641	u$s 878	u$s 13.378	u$s 38.378	u$s 63.378
Gs. Financieros	u$s 16.641	u$s 0	u$s 0	u$s 0	u$s 0
Interés	u$s 6350	u$s 0	u$s 0	u$s 0	u$s 0
Margen Neto	u$s -39.632	u$s 878	u$s 13.378	u$s 38.378	u$s 63.378

Precio producto	**Costo producto**
u$s 675	u$s 425

Gastos operativos	u$s 11.621
Gastos financieros	u$s 22.991
Repago	28 meses (operación)

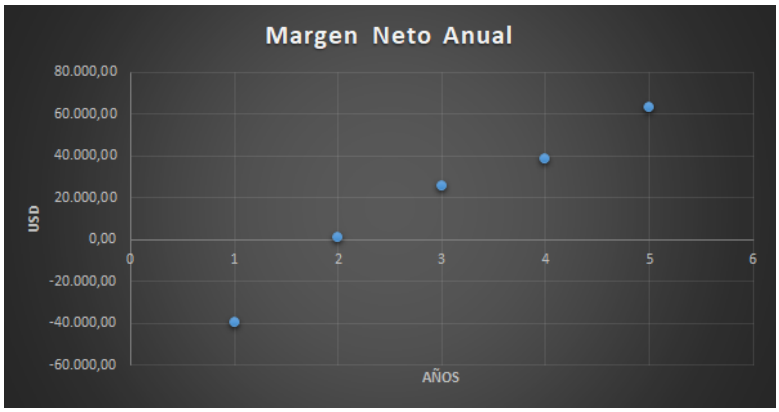


Figura 1: Margen neto anual del proyecto.

El primer año no habra ventas del producto, se lo dedicará exclusivamente al desarrollo y maduración del mismo. Por otro lado, el margen de ganancia se eligió para recuperar la inversión al poco tiempo de finalizar el segundo año de operación según las proyecciones de ventas realizadas. Esto puede observarse en la Figura 1.

2. Introducción

Este capítulo tiene por objetivo dar una introducción teórica al estado del arte de las redes definidas por software así como de los mecanismos de calidad de servicio que serán considerados en el desarrollo de la aplicación que este trabajo tiene por objetivo.

2.1. Redes definidas por software

Las redes definidas por software (SDN: Software Defined Networking) han generado un nuevo paradigma cambiando el enfoque sobre anteriores conceptos. Debido a la multiplicidad de cambios propuestos éstas pueden definirse de múltiples maneras. Las definiciones consideradas a lo largo de este trabajo corresponden a [32] y son:

SDN es una arquitectura L2/L3 en la cual un dispositivo centralizado controla el comportamiento de conmutación de elementos de red distribuidos

SDN es un framework conceptual en donde las redes son tratadas como una abstracción que funciona sobre hardware programable. Estas redes son controladas a través de aplicaciones que exponen interfaces estándar para interactuar con sus elementos. Esta situación permite miniminzar la interacción directa con elementos de red individuales para su operación

Estas definiciones apuntan a una idea central en la propuesta de las redes definidas por software, que es la centralización de la lógica de la red. En las redes tradicionales, cada dispositivo tiene un plano de control donde residen funciones implementadas por software, como el proceso de MAC [3] learning en un switch. Además, cada dispositivo tiene un plano de datos que se encarga de la conmutación de los paquetes. Esto hace que cada dispositivo tenga su propia visión sesgada de la red.

SDN desacopla el plano de control del plano de datos de los dispositivos. De esta manera los elementos de red, que realizan el forwarding de los paquetes, recibirán instrucciones de un controlador externo. Esto genera una serie de ventajas como:

- Las aplicaciones de red en el plano de control conocen y tienen acceso a todos los dispositivos en el plano de datos
- No hay tiempos de convergencia
- Los dispositivos pueden tomar diversas funciones, solo alcanza con ejecutar otra aplicación en el controlador
- El controlador expone interfaces hacia aplicaciones que utilizan recursos red

Esta idea de la separación entre el plano de control y el plano de datos será explicada con mayor detalle en párrafos posteriores.

Por otro lado, como tecnología, SDN disponibiliza una arquitectura que habilita una administración de infraestructura de comunicaciones más ágil y orientada a la programabilidad, con el objetivo de mejorar performance, monitoreo y escalabilidad.

Este trabajo considera cuatro áreas críticas en que la tecnología SDN puede marcar una diferencia considerable dentro de la industria y que servirán para elaborar los conceptos necesarios para la realización de la aplicación propuesta.

1. Programabilidad de la red: SDN permite que el comportamiento de la red sea controlado por el software que reside más allá de los dispositivos de red que proporcionan conectividad física. Como resultado, los operadores de red pueden adaptar el comportamiento de sus redes para soportar nuevos servicios, e incluso clientes individuales. Al desacoplar el hardware del software, los operadores pueden introducir nuevos servicios innovadores y diferenciados rápidamente, sin las limitaciones de las plataformas cerradas y patentadas.
2. Centralización y control: SDN se basa en topologías de red centralizadas lógicamente, que permiten el control inteligente y la gestión de los recursos de la red. Los dispositivos funcionan de forma autónoma con un conocimiento limitado del estado de la red. Con el tipo de control centralizado que proporciona una red basada en SDN, la administración del ancho de banda, la restauración, la calidad de servicio y las políticas pueden ser altamente inteligentes y optimizadas, y una organización obtiene una visión integral de la red.

[3] MAC: Medium Access Control. Refiere al direccionamiento de capa 2 del protocolo Ethernet.

3. Abstracción de la red: los servicios y aplicaciones que se ejecutan en tecnología SDN se extraen de las tecnologías y el hardware subyacentes que proporcionan conectividad física desde el control de la red. Las aplicaciones interactuarán con la red a través de API's (Aplication Programming Interface), en lugar de interfaces de administración estrechamente acopladas al hardware.

4. Estandarización: las arquitecturas de SDN marcan el comienzo de una nueva era de apertura: habilitan la interoperabilidad de múltiples proveedores y fomentan un ecosistema agnóstico a marcas propietarias. El uso de un protocolo estándar para la comunicación con los elementos de red programables brinda un contexto sin vendor lock-in. Por otro lado, interfaces programables (API) abiertas en los controladores admiten una amplia gama de aplicaciones, como puede ser la integración con sistemas OSS/BSS, ofrecer aplicaciones de red en formato SaaS, entre otras. Definamos estos conceptos:

   a) Los Sistemas de Soporte de Operaciones (OSS) se refieren principalmente a los sistemas de red que están directamente vinculados con la operación de la misma, por ejemplo, configuración de los elementos, detección temprana de fallas, mantenimiento, etcétera. Básicamente, es lo que permite a los operadores de telecomunicaciones mantener sus servicio en funcionamiento. El Sistema de Soporte de Negocios (BSS), por su parte, es el elemento complementario y se encarga de la administración de los elementos del negocio para los operadores de telecomunicaciones. Estos incluyen herramientas para atención al cliente, cobro, facturación, entre otros.

   b) El software como servicio (SaaS) permite a los usuarios conectarse a aplicaciones basadas en la nube a través de Internet y usarlas. SaaS ofrece una solución de software integral que se adquiere de un proveedor de servicios en la nube mediante un modelo de pago por uso. Toda la infraestructura subyacente, el middleware, el software y los datos de las aplicaciones se encuentran en el centro de datos del proveedor. El proveedor de servicios administra el hardware y el software y, con el contrato de servicio adecuado, garantizará también la disponibilidad y la seguridad de la aplicación y de los datos.

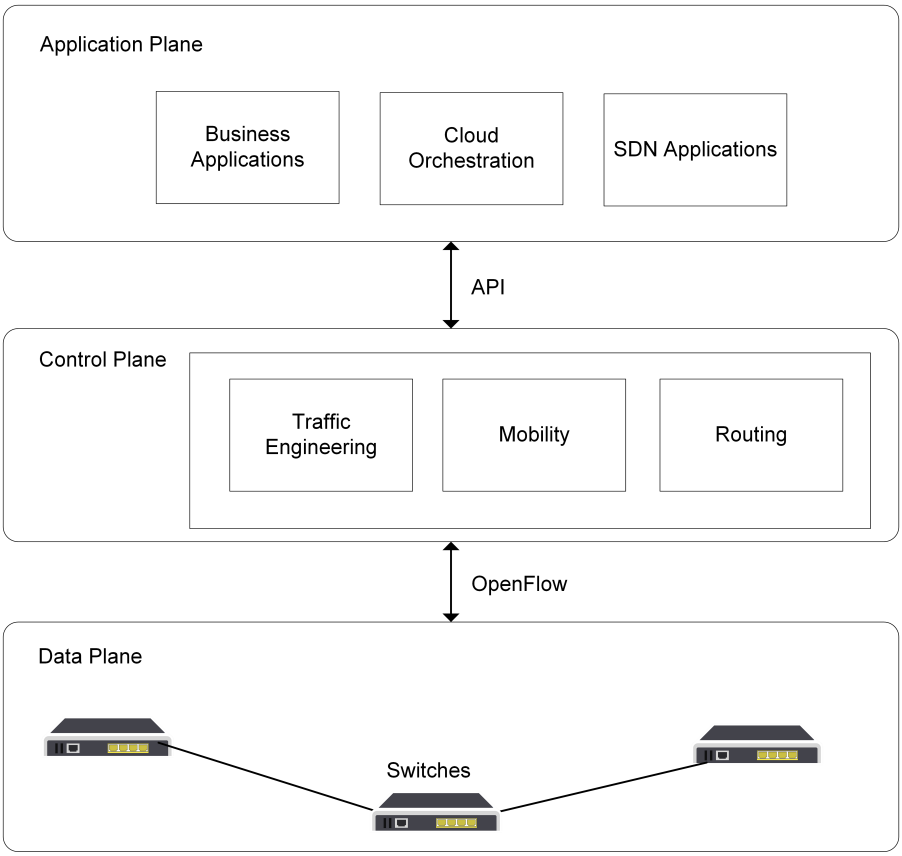


Figura 2: Estructura lógica de una red SDN. (Fuente: Elaboración propia [29])

A esto debemos agregarle las aplicaciones de red críticas para cada empresa u organización que dependerán de cada entorno.

Como puede observarse en la Figura 2 el modelo de arquitecura propuesto por SDN tiene tres planos principales.

- **Application Plane:** Plano de aplicaciones que interactuan con la red a través de las northbound-interfaces expuestas por el controlador. Estas interfaces son, preferentemente, interfaces REST [4] sobre protocolo HTTP [5], con el objetivo de utilizar mecanismos de comunicación estándar y aceptados en la industria del desarrollo de software.
- **Control Plane:**Plano de control, donde se ejecutan las aplicaciones del controlador SDN. Estas aplicaciones toman las decisiones de conmutación que luego serán comunicadas a través de las southbound-interfaces a los elementos de red programables. Estas interfaces son estándar y agnósticas a los fabricantes. Es decir, cada vendor tendrá una implementación de estas interfaces y realizará una traducción de las instrucciones comunicadas para realizar los cambios necesarios en sus dispositivos. El protocolo por defecto para esta funcionalidad es OpenFlow.
- **Data Plane:** Plano de datos, es donde residen los elementos de red programables que realizarán las operaciones de conmutación de los datos que transiten en la red. Hablamos de conmutación en un aspecto genérico, ya que los flujos del protocolo OpenFlow pueden utilizar como criterio de switching información de capa 2 (MAC-address), capa 3 (IP address), capa 4 (tupla puerto, protocolo), entre otros.

Como puede observarse, otro aspecto interesante de esta arquitectura, es que los dispositivos de red ya no están asociados a una funcionalidad específica. Dependiendo de la aplicación que resida en el controlador SDN, el dispositivo podrá funcionar como switch, router, firewall, load balancer, MPLS edge/switch (P/PE), etcétera. De esta manera, los dispositivos de red pasan a ser *cajas* programables que se adaptan a la necesidad coyuntural de la red o de las aplicaciones que utilizan la red.

Las aplicaciones que requieran modificar condiciones de la red para un mejor funcionamiento, podrán solicitar recursos en tiempo real a través de las north-bound interfaces. Estas interfaces exponen funcionalidades de las aplicaciones que se ejecutan en el controlador y que pueden orquestar cambios en los dispositivos de red a través de un protocolo como OpenFlow. De esta manera, los canales de comunicación entre las aplicaciones y los elementos de red se encuentran disponibles.

[4]Representational State Transfer: Arquitectura para el desarrollo de interfaces programables [14].
[5]Hypertext Transfer Protocol: Protocolo de capa 7 utilizado para exponer servicios web en redes de datos IP.

2.2. Mecanismos de calidad de servicio

Este trabajo toma como marco de referencia para las definiciones de los mecanismos y tecnologías de calidad de servicio el trabajo de Gerometta Oscar [20].

2.2.1. Calidad de servicio

Inicialmente, el objetivo de las infraestructuras de comunicaciones estuvo focalizado en la conectividad [20]. En este contexto, las permisas de operación de la red eran:

- El primero que solicita recursos de la red, es el primero en servirse
- Todos los tipos de tráfico tienen, de forma indistinta, el mismo tratamiento
- La diponibilidad de recursos de red para un usuario dependen de la cantidad de usuarios conectados a la red
- Aplicaciones con requerimientos específicos de red usan redes independientes

Sin embargo, con el avance de la integración del uso de las redes de datos en la operación de negocio de las empresas se ha desarrollado progresivamente el concepto de redes convergentes. En las redes convergentes se brinda servicio a diferentes aplicaciones (voz, video, datos), cada una de ellas con requerimietnos diferentes.

Algunos de los desafíos más importantes que deben resolverse en las redes convergentes son:

- Preservar el ancho de banda
- Limitar latencia extremo a extremo
- Limitar Jitter
- Reducir al mínimo posible la pérdida de paquetes

La calidad de servicio (QoS) pone a disposición herramientas tecnológicas que permiten garantizar condiciones específicas de disponibilidad de ancho de banda, latencia y pérdida de paquetes para cada uno de los diferentes tipos de tráfico.

Esto se logra identificando los diferentes tipos de tráfico que conviven en la red, agrupando estos tipos de tráfico considerando aquellas aplicaciones o servicios que tienen requerimientos semejantes y definiendo políticas que se aplicarán a cada una de las clases definidas. Estas políticas podrán impactar sobre distintas características del tráfico como pueden ser:

- Descarte preventivo
- Traffic shaping
- Compresión
- Queuing y scheduling

Es importante destacar que, distintos tipos de tráfico tendrán diferentes tratamientos en el manejo de su calidad y distintos requerimientos para su buen funcionamiento. A continuación se adjunta una tabla que describe los requerimientos de tres tipos de tráfico que se encuentran presentes en todas las redes modernas [20].

	Voz	**Video**	**Datos**
Latencia	<150 ms	<150 ms	Depende aplicación
Jitter	<30 ms	<30 ms	Depende aplicación
Pérdida de paquetes	<1 %	<1 %	Depende aplicación
Características	Tráfico UDP-RTP de paquetes pequeños. Debe considerarse tráfico asociado a la señalización. Reserva de recursos constante	Tráfico UDP-RTP de paquetes medianos. Debe considerarse tráfico asociado a la señalización. Reserva de recursos constante	Aplicaciones sobre TCP son tolerantes a pérdida de paquetes. Deben clasificarse según requerimientos y criticidad como ▪ Misión crítica ▪ Transaccional ▪ Best-effort

Algo interesante a tener en cuenta es que, si bien las comunicaciones de voz y video suelen utilizar protocolos no orientados a la conexión como UDP [6] para su transporte, las implementaciones modernas de este tipo de aplicaciones tiene cierta tolerancia a la pérdida de paquetes gracias técnicas como Packet Loss Concealment [5] que implementan, a su vez, mecanismos como Interleaving, Forward Error Correction, u otros.

Dentro de los modelos de implementación de QoS [7] podemos destacar los siguientes:

- **Best-effort:** No se discrimina ningún tipo de tráfico. Las aplicaciones que utilizan la red compiten por igual por los recursos disponibles
- **IntServ:** Los recursos de la red son reservados por las aplicaciones bajo demanda, los mismos estarán reservado al canal de comunicación solicitado mientras la aplicación lo requiera
- **DiffServ (differenciated services):** La infraestructura de red reconoce los distintos tipos de tráfico y aplica políticas diferenciadas para cada clase de tráfico.

En el campo de las redes de datos y, más específicamente, las redes de paquetes conmutados, Calidad de Servicio refiere a las tecnologías que permiten priorizar el tráfico en una red o reservar recursos en distintos componentes de una red (RSVP o Resource Reservation Protocol).

Este trabajo se centra en el modelo de servicios diferenciados (DiffServ) por ser de los mecanismos más ampliamente utilizados en las redes modernas. Algunas de las características de este modelo de calidad de servicio que lo han hecho tan popular respecto a los demás se listan a continuación.

1. No requiere señalización previa
2. No permite garantizar condiciones de tráfico extremo a extremo
3. Es muy flexible y escalable
4. Divide el tráfico en clases en función de los requerimientos de la red
5. Cada paquete recibe el tratamiento que se ha definido para la clase a la cual ese paquete pertenece
6. A cada clase se le puede asignar un diferente nivel de servicio y con ello diferentes recrusos
7. La asignación de recrusos se hace salto por salto en cada dispositivo de la red y no para una ruta específica
8. El mecanismo de implementación es relativamente complejo

Este trabajó pone especial atención en los puntos 2 y 8 anteriormente listados. Se intentará mostrar cómo en arquitecturas SDN y con las interfaces programables correspondientes expuestas por controladores y aplicaciones la complejidad de la implementación y garantizar las condiciones de tráfico extremo a extremo son una posibilidad.

2.2.2. Modelo de servicios diferenciados (DiffServ)

Cuando se implementa el modelo de servicios diferenciados o DiffServ se procura brindar determinadas condiciones de servicio a cada paquete que atraviesa la red independientemente de su pertenencia o no a determinado flujo de datos. Se establecen condiciones específicas para cada paquete y los recursos se asignan paquete por paquete.

A continuación, se describe terminología de este modelo que será importante para el desarrollo de este trabajo.

- **DSCP:** Differentiated Service Code Point es un valor decimal obtenido a partir de 6 bits del campo Type of Services (ToS) del encabezado IP utilizado para identificar distintos tipos de servicios dentro del flujo de datos general. Toma valores decimales entre 0 y 63.
- **BA:** Behavior Aggregate es un conjunto de paquetes con un mismo DSCP que atraviesan un enlace en una determinada dirección. Este conjunto puede estar constituido por tráfico de una o más aplicaciones diferentes.
- **PHB:** Per-Hop Behavior es el tratamiento que se aplica en un nodo de la red a un BA. Este tratamiento incluye elementos tales como scheduling, queuing, policing o shaping en un nodo específico.

[6] User Datagram Protocol: Protocolo de capa 4 no orientado a la conexión.
[7] QoS: Quality of Service o calidad de servicio. Se utilizarán indistinamente a lo largo de este trabajo.

A continuación, se describen los principios básicos de este modelo.

- El tráfico que ingresa a la red es clasificado lo más cerca posible de las fronteras de la red (origen del paquete)
- En función de la clasificación, los paquetes son incorporados a un **BA** o clase
- Todo paquete que no es asociado a un **BA**, queda asociado a la clase *default* que define el comportamiento por defecto para todo paquete que no deba recibir un tratamiento diferencial
- Cada clase de tráfico es identificada con uno o varios valores de DSCP
- Todo paquete no marcado, o no IP se asume con valor DSCP=0
- A cada clase se le aplica en cada salto de la red un **PHB** de acuerdo a las políticas definidas para esa clase
- De esta forma, un valor DSCP queda asociado a un **PHB** que se define en cada salto

De esta manera, se puede simplificar flujo de este mecanismo como se ilustra en la Figura 3.

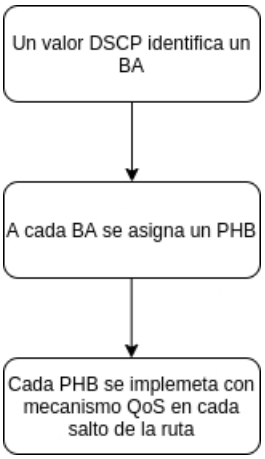


Figura 3: Modelo de servicios diferenciados. (Fuente: elaboración propia)

Una de las claves del modelo DiffServ es la utilización del campo Type of Service [25] [23] [24] o ToS del encabezado IP para marcar los paquetes clasificados que pertenecen a cada Behavior Aggregate. El campo ToS de 8bits puede verse en la Figura 4.

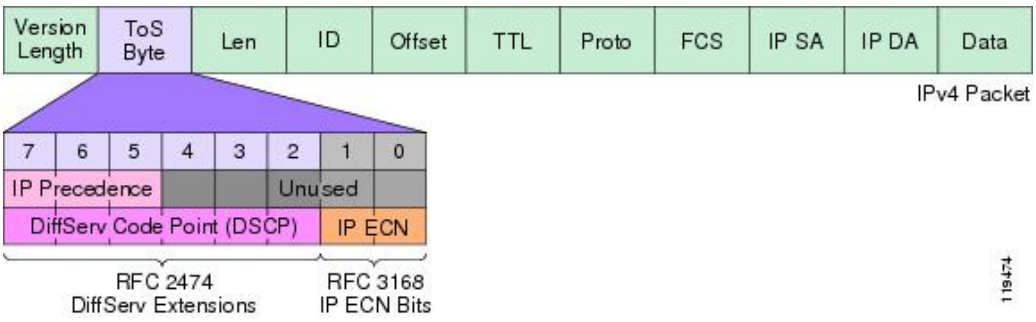


Figura 4: Campo ToS IP y DSCP. (Fuente: Cisco [9])

La definición de este campo depende de un conjunto de definiciones de la IETF, la que más nos interesa es la perteneciente al RFC2474, donde se reemplaza el campo ToS por el campo DS, utilizando los 6 bits de mayor orden (bits de 2 a 7) orden para marcar DSCP. Los dos bits de menor orden [0-1] se dedican a notificación de congestión [23].

Por último, es relevante describir para el contexto de este trabajo los distintos tipos de **PHB** disponibles así como el más relevante para el tráfico real-time. Los **PHB** es el conjunto de definiciones respecto del tratamiento que se debe dar en cada nodo a las diferentes clases de tráfico. La IETF define 4 **PHB** básicos.

- **Default:** Define servicio **best-effort**. Es el tratamiento que se da a todo tráfico no IP, no identificado, o identificado con DSCP=0. El valor de los bits DSCP [5-7] es 000.
- **Expedited Forwarding (EF):** Tratamiento aplicado a servicio que requieren baja latencia. El valor de los bits DSCP [5-7] es 101.
- **Assured Forwarding (AF):** Tratamiento aplicado a servicios que requieren un ancho de banda garantizado. El valor de los bits DSCP [5-7] es 001, 010, 011 o 100. Consecuentemente, existen cuatro categorías de AF.
- **Class Selector (CS):** Clasificación utilizada para mantener compatibilidad hacia atrás con dispositivos que soportan IP Precedence [24] y no DSCP. En este caso, el valor de los bits 2 a 4 es específicamente 000.

El objetivo de este trabajo es, basándose en la arquitectura de aplicaciones e interfaces propuesta por SDN, desarrollar una aplicación que permita configurar priorización de los paquetes de voz de comunicaciones VoIP en tiempo real. Esta aproximación podrá ser extendida en una instancia futura con el objetivo de realizar estas configuraciones de manera desatendida extremo a extremo.

Estas capacidades se vuelven posibles en el contexto de una arquitectura que propone estandarizar interfaces de comunicación entre las aplicaciones y las redes de datos, situación que no es posible en contextos de redes legacy [8].

Se hace foco especialmente en la configuración de los PHB y, además, se tendrá en cuenta como extender el uso de DSCP en futuras versiones.

[8] Redes legadas o tradicionales. Refiere a redes no SDN.

2.2.3. Call Admission Control

Call Admission Control (CAC) es una funcionalidad que permite limitar la cantidad de hilos de comunicación concurrentes en un segmento de red VoIP para evitar la congestión de los enlaces. CAC funciona en la fase de call-setup de la llamada. El objetivo principal de esta funcionalidad es el de garantizar la calidad de las comunicaciones, evitando aquellas cuya calidad no pueda garantizarse.

Esta capacidad es especialmente útil en enlaces WAN, donde el ancho de banda es restringido y el tráfico se voz compite con otras aplicaciones de misión crítica. En este sentido, CAC funciona en conjunto con *Calidad de Servicio*, y la cantidad de llamadas concurrentes se limitan a la cantidad de tráfico que puede soportar la cola correspondiente.

Un esquema que ejemplifica la implementación de CAC puede verse en la Figura 5. El tráfico VoIP compite con tráfico *Best Effort*. En escenarios con CAC, la cantidad de llamadas permitidas no puede superar un umbral establecido en conjunto con el tamaño de la cola EF (DSCP46). En escenario sin CAC, no hay umbral predefinido que limite la cantidad de comunicaciones concurrentes, en este caso al sobrepasar el tamaño de la cola de calidad de servicio las comunicaciones comenzarán a perder paquetes, afectando la calidad o incluso, la posibilidad de las mismas.

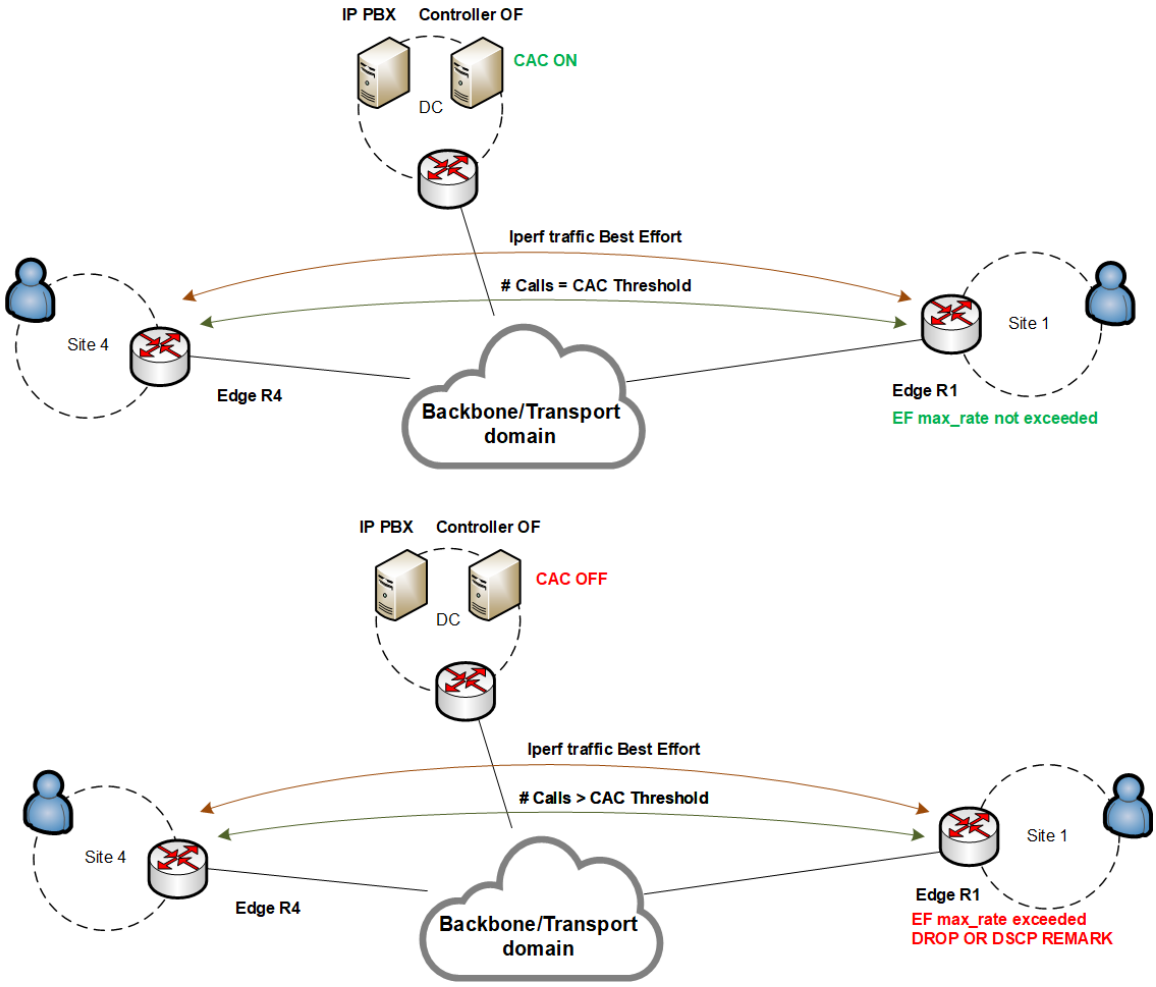


Figura 5: Escenarios CAC y no CAC. (Fuente: Elaboración propia)

Esta funcionalidad es implementada en soluciones VoIP de fabricantes como Cisco Call Manager [8] y Microsoft Skype for Business [34].

El problema de este mecanismo es, que al funcionar en la fase de call-setup, en escenarios multi-fabricante existe imposibilidad de implementarla de manera consistente, ya que una central Cisco no verá las comunicaciones concurrentes de una central Microsoft o Avaya. Una mejora considerable a esta situación, sería implementar la funcionalidad fuera de las centrales en una aplicación que sea consiente de todas las llamadas que se encuentran activas en la red. Este trabajo propone una resolución a esta problemática con el enfoque antes mencionado.

2.3. Metodología

2.3.1. DevOps/NetOps

DevOps es un marco metodológico que surge en la industria del desarrollo de software. De acuerdo con [19] [18], algunos de los conceptos fundamentales de estas metodologías son:

- Metodología en donde la operación de servicios se encuentra fuertemente integrada al desarrollo de software
- **Nuevo paradigma:** El software cambia continuamente con el objetivo de generar cambios rápidos en producción
- **Nueva forma de trabajo:** Minimizar los efectos y treas relacionadas con la actualización de software mediante la automatización y la orquestación
- **Nueva cultura:** comunicación, colaboración e integración entre equipos de desarrollo, implementación y operaciones

DevOps pone especial atención en la integración entre las áreas de desarrollo y operaciones. Esto se debe a que los tiempos de entrega de software son cada más agresivos, asi como la cantidad de actualizaciones que deben implementarse, esto hace que el trabajo entre las distintas áreas esté fuertemente relacionado. Además, con el objetivo de obtener un feedback constante sobre el desarrollo, acortar los tiempos de puesta en producción y de implementación de las aplicaciones permite reaccionar de manera rápida a errores o cambios en los requerimientos. Para lograr estos objetivos, la metodología DevOps se apoya fuertemente en herrramientas tecnológicas de automatización y orquestación así como en prácticas como Integración Continua y Delivery Continuo.

Al separar los planos de control, gestión y datos en la arquitectura SDN, la posibilidad de desarrollar aplicaciones de red o interfaces que permitan interactuar con aplicaciones de red presenta el desafío de comenzar a pensar en términos de desarrollo de software, ya que tanto la lógica de red como la infraestructura comienza a escribirse con lenguajes de programación de distintos niveles de abstracción que deben ser versionados, implementados, actualizados y monitoreados como cualquier aplicación.

A raíz de lo expuesto en el párrafo anterior, surge el concepto de NetOps. Algunos de los argumentos que movilizan esta metodología son:

- **Evolución del servicio:**
  - El modelo de trabajo actual está limitado por ventanas de mantenimiento y disponibilidad de ingenieros con conocimientos en cada uno de los temas.
  - El tráfico, los servicios, los elementos de red y la complejidad de los mismos está creciendo exponencialmente
- **Evolución tecnológica:**
  - El modelo de trabajo actual (High level design, Low level design, MOP, Ventana, Optimización) está diseñado para generar cambios controlados en una red estática
  - NFV[9], SDN y 5G Network Slicing son nativamente soluciones automatizadas y dinámicas.
- **Evolución del negocio:**
  - Los tiempos en el desarrollo de un nuevo producto/servicio, incluso una vez que se decide su despliegue son sumamanete extensos. Migraciones de servicios a equipamiento de un nuevo fabricante o desplegar una nueva tecnología en redes grandes puede llevar meses o años.
  - La velocidad del mercado y las alternativas abiertas pueden poner en riesgo el negocio de grandes operadores.
- **Escala de la red:**
  - Tradicionalmente la cantidad de elementos de red es conocida, agregar un nuevo elemento en la red es un proceso definido y que involucra múltiples etapas, incluyendo, por ejemplo, el alta en el inventario y la integración con un OSS o gestor.
  - La tendencia es una red donde orquestadores van a generar miles de máquinas virtuales y contenedores para brindar servicios, de forma dinámica, escalable, distribuida y automatizada.

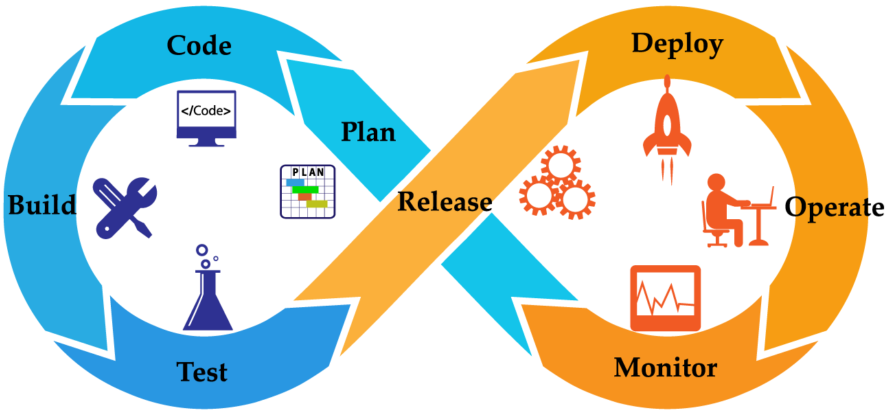


Figura 6: Ciclo de vida de aplicaciones DevOps/NetOps. (Fuente: Eweek [12])

Es por esto, que la metodología DevOps evoluciona para su funcionamiento en contextos de redes de datos donde se hace foco tanto en metodologías de trabajo como en herramientas tecnológicas de orquestación y automatización. El flujo iterativo propuesto se observa en la Figura 6.

Así como en el desarrollo de software, la operación de redes tendrá etapas de planificación y diseño, programación, construcción de imágenes para distribución de software, testing, implementación, operación y monitoreo. Este proceso es iterativo y contiene el ciclo de vida completo de los servicios y elementos de infraestructura que funcionan sobre una red.

2.3.2. Infraestructura como código

Comenzar a trabajar sobre dispositivos de hardware genéricos cuya funcionalidad es aprovisionada por controladores u orquestadores de red hace que el vendor lock-in disminuya y permita utilizar infraestructura virtualizada. Aquí, la virtualización y la containerización [10] juegan un papel central. En este sentido, pueden utilizarse herramientas de infraestructura como código que permitan definir topologías productivas y laboratorios con lenguajes de programación de alto nivel. Algunas de las herramientas tecnológicas que brindan esta capacidad son:

- Ansible
- Terraform
- Mininet
- GNS3

La infraestructura como código tiene la ventaja de poder versionar los distintos estadíos de nuestra infraestructura y poder utilizar herramientas de automatización para el despiegue y operación.

Este trabajo utiliza algunas de las herramientas listadas anteriormente para la realización de laboratorios.

[9] NFV: Network Functions Virtualization.
[10] Utilizar virtualización basada en contenedores.

3. Alcance Tecnológico: Protocolos y herramientas de software

Este capítulo tiene por objetivo describir tecnologías, metodologías y protocolos que serán utilizados a lo largo de este trabajo.

3.1. Interfaces SDN

3.1.1. OpenFlow

OpenFlow es un protocolo que funciona como interfaz de comunicación entre los controladores SDN y los dispositivos de red. El protocolo se define en OpenFlow Switch Specification, publicado por Open Networking Foundation (ONF). ONF es un consorcio de proveedores de software, redes de distribución de contenido y proveedores de hardware de redes cuyo objetivo es promover la evolución de infraestructura de comunicaciones asi como de los modelos de negocio de la industria. Dentro de sus funciones más relevantes, podemos mencionar que incuba y potencia proyectos de software como controladores SDN, soluciones de SD-WAN [11], protocolos como OpenFlow, entre otros.

Este trabajo se basa en una especificación de OpenFlow, versión 1.3., del 25 de junio de 2012. La especificación original, 1.0, se desarrolló en la Universidad de Stanford [39] y se implementó ampliamente. OpenFlow 1.2 fue el primer lanzamiento de ONF después de heredar el proyecto de Stanford. OpenFlow 1.3 expande significativamente las funciones de la especificación. Es probable que la versión 1.3 se convierta en la base estable sobre la cual se construirán futuras implementaciones comerciales para OpenFlow.

Como puede verse en la Figura 7, Openflow es un protocolo que ha ido evolucionando con el tiempo y que ha sufrido algunos cambios drásticos entre versiones. Hasta fines del 2018 OpenFlow se encontraba en la versión 1.5, pero la versión 1.3 es la que se tomará como referencia para explicar las características principales del protocolo y la que se utiliza en el marco de este trabajo.

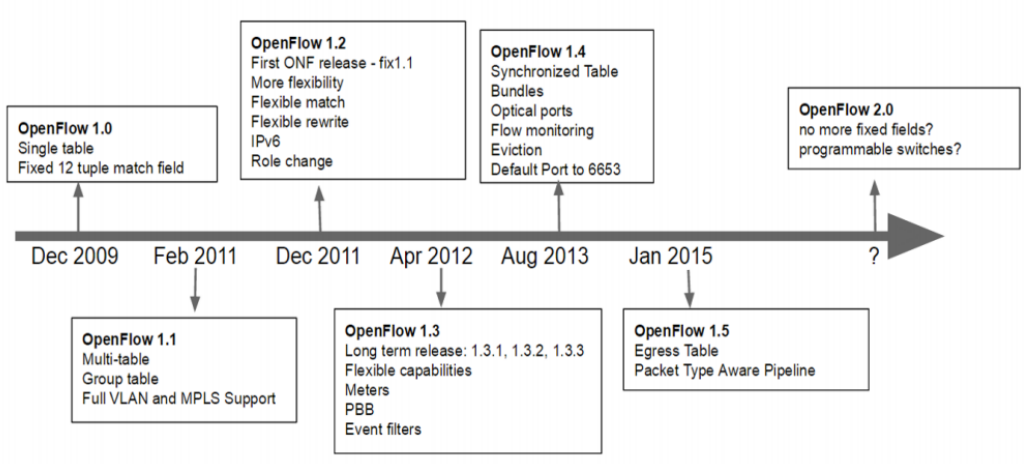


Figura 7: Evolución del protocolo OpenFlow. (Fuente: kspviswa [31])

Debido a que SDN sigue siendo un modelo de referencia en constante evolución, las tecnologías y las interfaces cambian con el tiempo. Actualmente la ONF ha comenzado a trabajar y a incentivar otros proyectos como Stratum y OpenConfig para operar sobre el plano de datos por lo que el futuro de OpenFlow es incierto en este aspecto. En gran parte, se puede señalar que esta evolución dependerá de la adopción y aceptación que estas tecnologías tengan en la industria. No obstante, la motivación principal de este trabajo, independientemente de los nombres propios que tengan las tecnologías al momento de redactarlo, es poner a prueba las ventajas y posibilidades de una nueva arquitectura.

3.1.2. Switch OpenFlow

Los switches OpenFlow son elementos de red programables que poseen una implementación del protocolo interna. A continuación, se describen los componentes principales de estos switches asi como las características

[11] Software Define WAN. Tecnología para gestión y operación de vínculos WAN.

de implementación del protocolo.

Flujos: El flujo tiene una expresión de evaluación que es utilizada para analizar los paquetes entrantes en un switch. Esta expresión puede evaluar propiedades de distintas capas del paquete. Por ejemplo, se puede construir un flujo que evalúe direcciones origen/destino de capa 2, capa 3, analizar la tupla puerto/protocolo del paquete, atributos del campo Type of Service (TOS) en caso de que el paquete esté marcado para ser tratado con alguna preferencia específica con respecto otros tipos de tráfico, etcétera. Los flujos tiene, además, un **action set**. Esta propiedad define qué acciones deben tomarse con el paquete una vez que ha tenido un matching con un flujo específico. Puede haber muchos flujos que hagan match con un paquete, en estos casos el flujo que tenga prioridad más alta es el que aplicará. Hay muchos tipos de acciones definidas, entre las más relevantes podemos mencionar:

1. Enviar paquete al controlador
2. Enviar paquete hacia otro puerto/puertos del switch
3. Enviar paquete para un nuevo análisis a otra tabla del
4. Modificar headers del paquete
5. Marcar el paquete con una calidad de servicio distinta

Los flujos OpenFlow, como puede observarse en la Figura 11, existen en los elementos de red (switches).

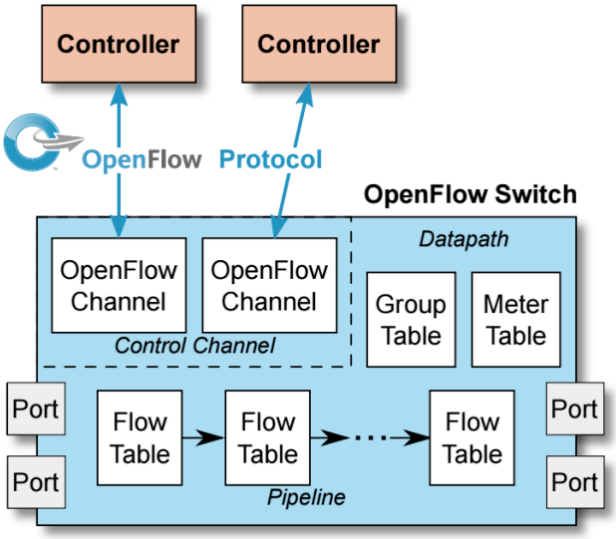


Figura 8: Componentes de un switch OpenFlow. (Fuente: ONF ONOS [40])

Tablas de flujos Los flujos se agrupan en tablas similares a las de enrutamiento. Cuando se recibe un paquete, se analiza la tabla de flujos elemento a elemento para comprobar si se cumplen los requisitos de matching en alguna de ellas. En la versión 1.0 de Openflow, únicamente existía una tabla de flujos. Sin embargo, a partir de OpenFlow 1.1 se soportan tablas anidadas. OpenFlow sigue un proceso general perfectamente definido a la hora de recorrer las tablas de flujos en el switch. Este proceso, o pipeline, es el siguiente:

Las tablas de flujos en el switch se ordenan por números, empezando en 0. La tabla 0 debe existir siempre, ya que debe haber al menos una tabla de flujos en el switch. El proceso empieza siempre en esta primera tabla. Cuando entra un paquete se intenta asociar con alguna entrada de la tabla 0. En caso de encontrar una coincidencia, se añaden las instrucciones asociadas a ese flujo al denominado "action set" del paquete. El action set es el conjunto de acciones que se aplican al paquete entrante una vez que han acabado de recorrerse las tablas de flujos. Si entre las instrucciones de la entrada se encuentra la de pasar a otra tabla (Instrucción go to), se avanza a esa tabla y se repite el mismo proceso. Nótese que siempre debe avanzarse a tablas de numeración más alta, nunca más baja. Dicho de otra forma, siempre debe avanzarse hacia delante, y no puede volverse nunca a tablas anteriores. Cuando no quedan más tablas que recorrer, es decir, cuando el "match" de una tabla no incluye la instrucción "go to" indicando que debe avanzarse a otra tabla, se ejecuta el action set asociado al paquete. El action set se ejecutará en un orden preestablecido.

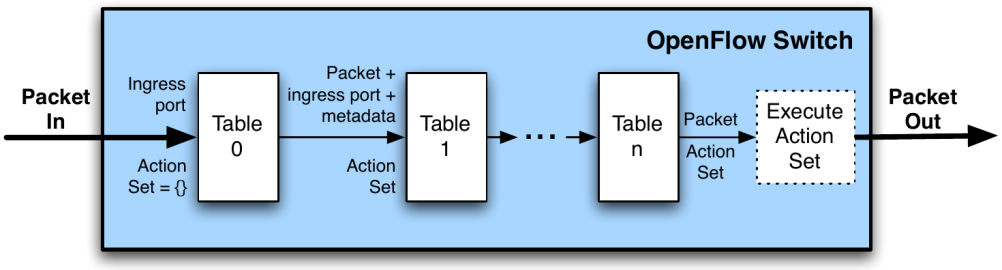


Figura 9: Análisis de un paquete en un pipeline de tablas. (Fuente: ONF ONOS [40])

En caso de que no exista una entrada asociada al paquete entrante en una tabla, hay un flujo especial al final de la tabla de prioridad 0 ("table miss") cuyo action set determinará qué hacer con esos paquetes. Las opciones son:

1. Enviar el paquete al controlador para que este se encargue de encaminarlo.
2. Pasar el paquete a otra tabla del switch
3. Descartar el paquete (drop)

Nótese que el orden de las entradas de los flujos influye en su prioridad, de forma similar a las reglas de un firewall o las listas de acceso de un router. Esto implica que podría darse el caso de que existan dos entradas en la tabla flujos que pudieran estar asociadas a un paquete entrante, sin embargo, al recorrerse en orden descendente la tabla, solo se ejecutarán las instrucciones asociadas a la primera entrada que tenga coincidencia, las de mayor prioridad. Este flujo se puede observar en la Figura 10.

En OpenFlow 1.5 se modifica el pipeline. La parte vista hasta ahora pasa a denominarse Ingress Processing (procesamiento de entrada) y se incluye el Egress Processing (procesamiento de salida). Fundamentalmente, el Egress Processing sigue el mismo funcionamiento que el Ingress Processing, se recorren las tablas buscando equiparaciones válidas para el paquete y se ejecutan las instrucciones de la tabla asociadas a la entrada. Su función es permitir mayor granularidad y organización en las entradas de las tablas de flujos. Se trata de un procesamiento opcional y por tanto no es necesario que el switch lo implemente para poder encaminar tráfico correctamente.

Instrucciones, actions y action sets

Como ya se ha establecido, cada paquete entrante tiene asociado un action set y cada entrada en las tablas de flujos contiene un conjunto de instrucciones. Es importante entender esta distinción. El action set asociado al paquete entrante contiene las acciones que se ejecutarán sobre el paquete al acabar de recorrer las tablas. Las instrucciones asociadas a la entrada de un flujo en la tabla de flujos se ejecutan cuando un paquete entrante equipara en esa entrada.

Primero se hablará sobre las instrucciones. Existen dos tipos, obligatorias y no obligatorias. Estas últimas son implementadas según criterio del fabricante del equipo:

Obligatorias

1. **Write-actions [actions]:** Inserta una acción o conjunto de acciones en el action set del paquete entrante. Si alguna de las acciones que se pretenden insertar ya está en el action set, se sobrescribe.
2. **Goto-table [table id]:** Indica cuál es la siguiente tabla que debe consultarse.

No obligatorias

1. **Meter [metric id]:** Aplica una limitación de throughput a los paquetes que tengan match con esta instrucción.
2. **Apply-Actions [actions]:** Se aplica la acción especificada sobre el paquete inmediatamente, sin esperar a que se ejecute el action set.
3. **Clear-Actions [actions]:** Elimina todas las acciones contenidas en el action set.
4. **Write-Metadata [metdata/mask]:** Actualiza los metadatos

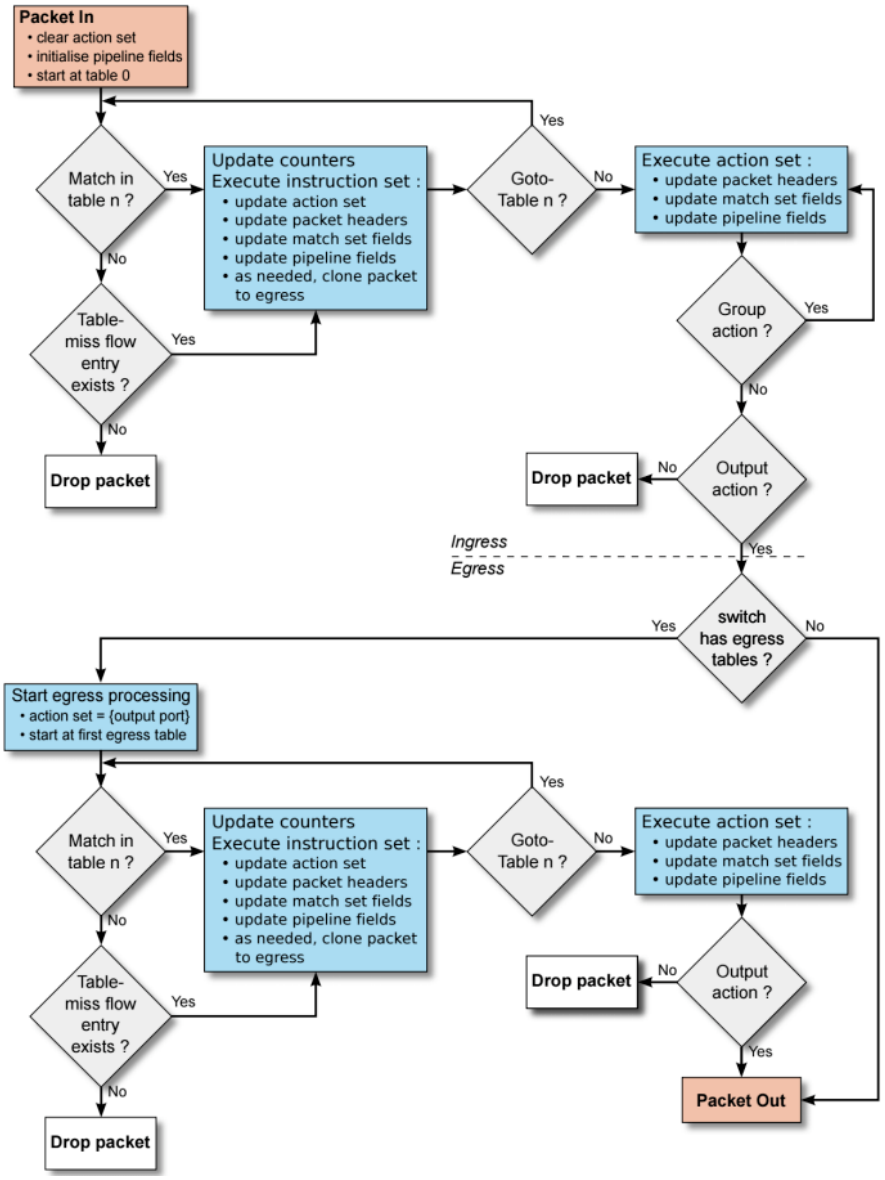


Figura 10: Flujo para procesamiento de un paquete en OpenFlow. (Fuente: ONF ONOS [40])

Las acciones son el elemento atómico que dictan que se hará con el paquete. Un action-set es un conjunto de acciones. Hay acciones para interactuar con dinstintos protocolos como se oberva en la Figura 12

Cada entrada en la tabla de flujos sólo puede tener una instrucción de cada tipo asociada y se ejecutan en el orden en el que están definidas en dicha entrada. Esto, en la práctica, y aplicado a las instrucciones obligatorias, implica que Goto-table siempre va después de Write-actions, ya que de lo contrario se avanzaría de tabla antes de escribir las acciones en el action set del paquete entrante. Los action sets, por otro lado, son el conjunto de acciones que se ejecutan sobre el paquete al acabar el pipeline. Aunque las acciones se añaden al action set a medida que se van recorriendo las sucesivas tablas de flujos, éstas no se ejecutan en orden cronológico de llegada. A continuación, se definen las acciones en orden de prioridad de ejecución:

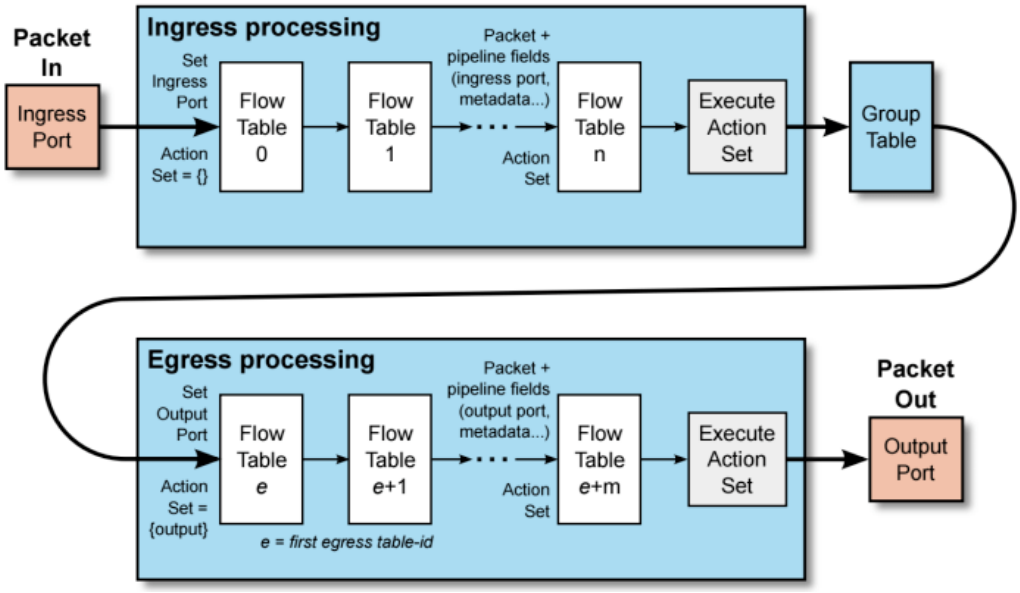


Figura 11: Pipelines en OpenFlow 1.5. (Fuente: ONF ONOS [40])

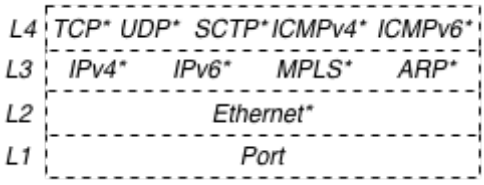


Figura 12: Actions stack. (Fuente: Flowgrammable [16])

- Empty: Descarta el paquete
- Group: Aplica el procesamiento del grupo [group id]
- Output: Conmuta el paquete hacia puerto/s especifico/s
- MPLS
- ARP
- IPv4
- IPv6
- TCP
- UDP
- SCTP
- ICMPv4
- ICMPv6
- Queue: aplica queue

Esta dinámica entre instrucciones, actions y actions set que propone el protocolo OpenFlow puede verse resumida en la Figura 13.

Las instrucciones se encuentran dentro del paquete, estas pueden agregar/quitar acciones del action set original de la tabla para ese paquete o bien pueden modificar el pipeline processing con instrucciones como goto para continuar el procesamiento en otra tabla o apply para ignorar los action set y ejecutar acciones específicas de forma inmediata.

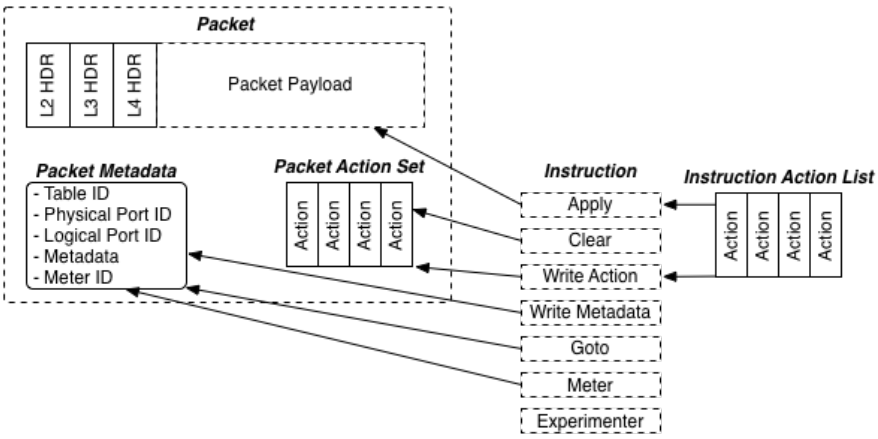


Figura 13: Procesamiento de instrucciones y acciones en OpenFlow. (Fuente: Flowgrammable [16])

3.1.3. APIs

Para exponer funcionalidades en el plano de control hacia las aplicaciones que requieren interactuar con la red, se definen las northbound-interfaces (NI). Estas interfaces dependerán del controlador, pero hay una fuerte tendencia a implementar APIs (application programming interfaces) REST sobre protocolo HTTP. De esta manera, se adopta un mecanismo de comunicación estándar en la industria del software que facilita enormemente la tarea de escribir aplicaciones que interactúen con controladores SDN.

REST (Representational State Transfer) [14] es un modelo de arquitectura para el diseño de interfaces que se puede observar en la Figura 14. Algunos conceptos importantes de estas interfaces son:

- Modelo cliente-servidor
- Interfaces que utilizan como base al protocolo http
- Interfaces basadas en recursos, no acciones
- Consistencia entre los verbos del protocolo http y las acciones a realizar sobre los recursos
- Stateless: Es decir, cada consulta es independiente de las otras. Toda la información necesaria para realizar una consulta la provee el cliente (autenticación, recurso a consultar, filtros, etcéctera)
- Cache: El servidor puede sugerir cachear información que, a priori, sepa que no se actualizará. La responsabilidad final de hacerlo queda en el cliente
- Multicapa: El acceso al servicio no es directo, puede haber abstracciones o elementos de conectividad intermedios. Esta situación es transparente para el cliente que consume el servicio a través de la interfaz
- El payload de las consultas se formatea en JSON

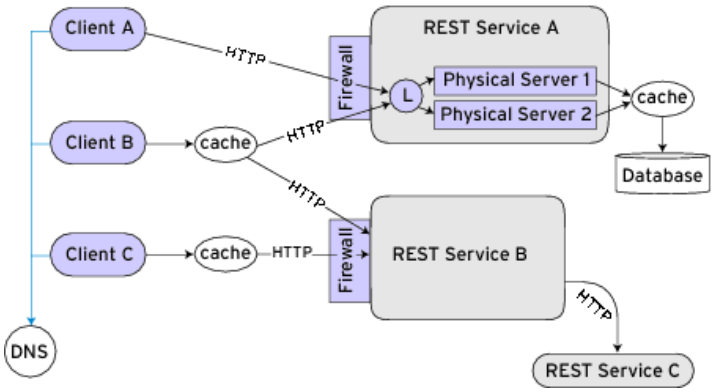


Figura 14: Arquitectura API REST. (Fuente: Web Development [10])

3.2. Open Virtual Switch (OVS)

OpenVirtualSwitch (OVS) [6] es un switch virtual (software) multi-capa open source con soporte para OpenFlow. Es un proyecto maduro con muchos años en la industria y es utilizado por otros proyectos como OpenStack, OVirt, VirtualBox, entre otros en su stack de networking. Además de ser utilizado en escenarios SDN, un caso de uso muy común donde se lo encuentra es como switch distribuido para interconectar máquinas virtuales que se ejecutan en distintos servidores con tecnologías como VXLAN como se observa en la Figura 15.

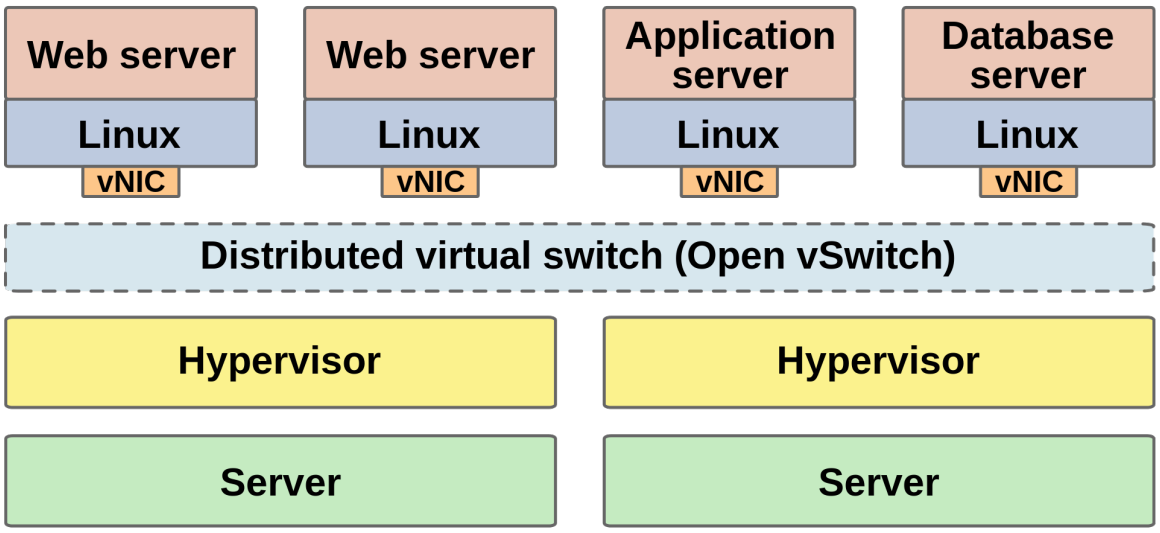


Figura 15: OpenVirtualSwitch como switch distribuido entre hypervisores. (Fuente: Wikipedia [47])

Algunas de las tecnologías que soporta OVS son:

- VLAN tagging y 802.1Q trunking
- Spanning tree (802.1D)
- LACP (link aggregation)
- port mirroring (SPAN/RSPAN)
- Flow Export (sflow, netflow, ipfix)
- tunneling (GRE, VLAN, IPSEC)
- OpenFlow
- QoS

En este trabajo, se utiliza OpenVirtualSwitch como elemento de red programable. Dispositivos OVS se conectarán con un controlador SDN vía OpenFlow para construir un ambiente que permita testear las funciones de la aplicación desarrollada.

3.3. Ryu Controller

Ryu es el controlador SDN utilizado en la realización de este trabajo. Ryu soporta distintas versiones de OpenFlow como v1.3, v1.4 y v1.5. Por otro lado, cuenta con soporte para diversos modelos de switches OpenFlow que existen en la industria como:

Dispositivo	**OpenFlow 1.3**	**OpenFlow 1.4**
Allied Telesis x510	Supported	
Allied Telesis x930	Supported	
Centec V350	Supported	
CpQd	Supported	
Edge-Core AS4600-54T	Supported	
HP 2920	Supported	
IBM RackSwitch G8264	Supported	
Indigo Virtual Switch	Supported	
Lagopus	Supported	
LINC	Supported	Supported
NEC PF5220	Supported	
NoviFlow NoviKit200	Supported	
Open vSwitch	Supported	Supported
Open vSwitch (netdev)	Supported	Supported
Pica8 P-3290	Supported	Supported
Trema Switch	Supported	

Además, ofrece no solo la capacidad de controlador sino, también, un framework con el cual se pueden desarrollar aplicaciones de red. Esto es posible gracias a un conjunto de APIs que el controlador expone y de librerías con las que se pueden manipular mensajes del protocolo OpenFlow.

Ryu tiene una arquitectura que expone clases para las aplicaciones a desarrollar asi como clases internas para gestionar los eventos de la red a través de OpenFlow que harán de nexo con nuestra aplicación a través de eventos. El **Event Loop** principal se ejecuta en un thread dedicado. Este loop estará a la espera de eventos que son encolados en el **Event Queue** a través de mensajes OpenFlow como PacketIn y PacketOut. Al recibir un evento, el programa principal cargará el evento y llamará al handler asociado. Estos handlers son definidos por el programador decorando la clase que se utiliza para desarrollar la aplicación. Esta lógica puede observarse en la Figura 16.

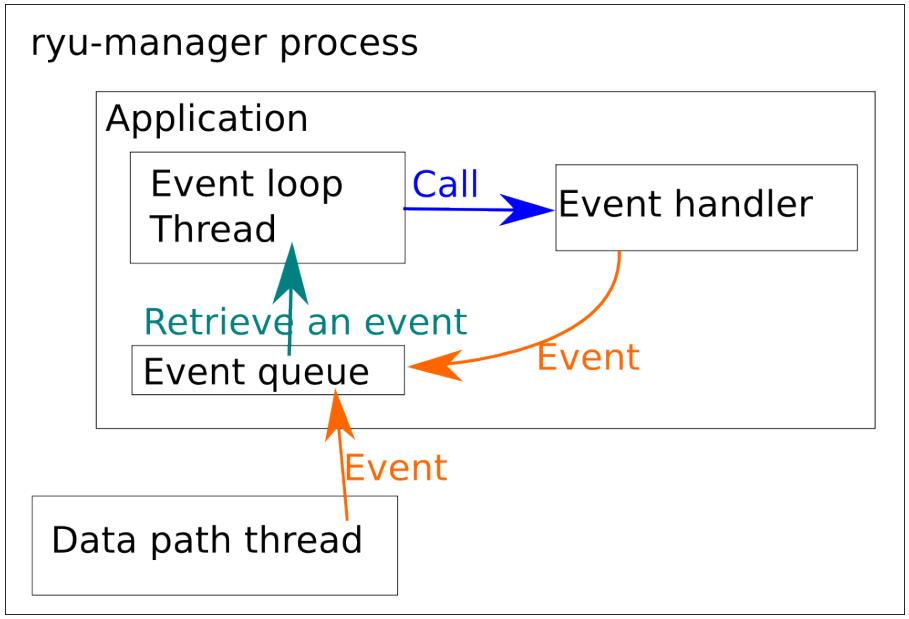


Figura 16: Arquitectura interna del controlador SDN Ryu. (Fuente: Ryu Docs [11])

A continuación, se adjunta una muestra del API que expone Ryu a la hora de desarrollar aplicaciones de red. Como puede observarse, se crea una clase de aplicación que hereda de **app_manager.RyuApp**. Luego, todos los métodos internos de la clase pueden ser decorados con el evento OpenFlow que debería ejecutar esa porción de código que implementa una lógica específica para el tráfico de la red. En este caso puntual, puede verse como ante un evento de PacketIn se ejecuta la función packet_in_handler que realiza una operación de FLOOD a todos los puertos del switch que envío el mensaje al controlador.

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_0

class L2Switch(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_0.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(L2Switch, self).__init__(*args, **kwargs)

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
        msg = ev.msg
        dp = msg.datapath
        ofp = dp.ofproto
        ofp_parser = dp.ofproto_parser

        actions = [ofp_parser.OFPActionOutput(ofp.OFPP_FLOOD)]
        out = ofp_parser.OFPPacketOut(
            datapath=dp, buffer_id=msg.buffer_id, in_port=msg.in_port,
            actions=actions)
        dp.send_msg(out)
```

3.4. Python

Python [42] es un lenguaje de programación interpretado de alto nivel y de propósito general. Soporta diversos paradigmas de programación como Object Oriented Programming, Structured programming, Functional Programming, entre otros. Python fué creado por Giuido Van Rossum y es un lenguaje que enfatiza la legibilidad del código, característica que puede observarse en su sintaxis. Además, python cuenta con una gran comunidad que aporta constantemente al desarrollo de librerías que se encuentran disponibles a través de su sistema de gestión de paquetes (pip).

Este trabajo utiliza python como lenguaje de programación para el desarrollo de las aplicaciones. Esto se decidió por el hecho de que el controlador SDN Ryu está desarrollado en este lenguaje y que el API de la central telefónica elegida tiene versiones en python. Esta elección simplifica el ecosistema de tecnologías del trabajo. Por otro lado, python es un lenguaje popular[12], lo que asegura una fácil adopción para extender funcionalidades en un desarrollo futuro por otras personas.

3.5. Git

Git [21] es un sistema de control de versiones distribuido (*version control system o VCS*) desarrollado por Linus Torvalds en el año 2005. A diferencia de otros VCS como Subversion y Perforce git es distribuido, no requiere de un servidor centralizado y tiene un footprint bajo a la hora de crear branches. Estas características hicieron a git el sistema de facto en la actualidad a la hora de versionar y mantener software.

Si bien git es un sistema distribuido, soporta diferentes workflows para su uso. Por ejemplo, como se observa en la Figura 17, se puede implementar un formato centralizado utilizando un repositorio de referencia. Este esquema es el que utilizan proyectos de hosting como Github y Gitlab y el adoptado durante la realización de este trabajo.

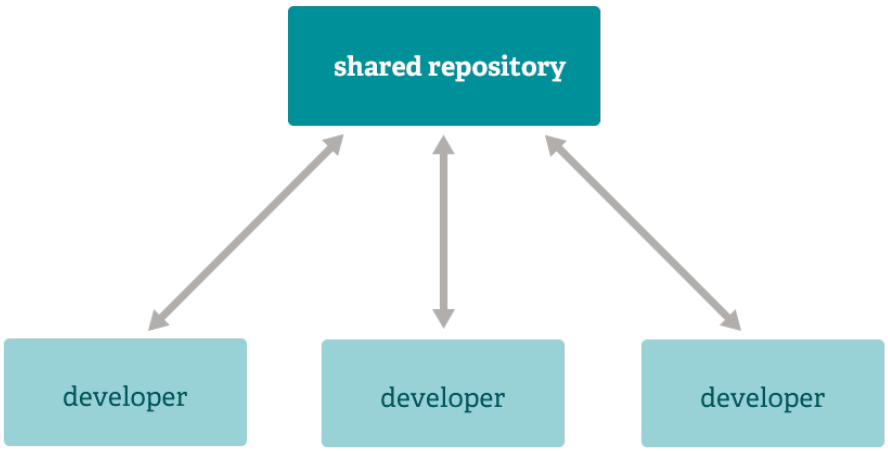


Figura 17: Workflow centralizado usando git. (Fuente: Medium [7])

Para el desarrollo de este proyecto se utiliza git para el versionado y mantenimiento del código.

Otros aspectos y usos de esta tecnología serán discutidos en el capítulo de metodologías.

[12]Lenguajes más buscados: https://pypl.github.io/PYPL.html.

3.6. Docker

Docker es una tecnología de contenedores. Permite realizar lightweight-virtualization[13] a nivel de sistema operativo. Esta tecnología utiliza herramientas del kernel del sistema operativo para aislar recursos de procesamiento, memoria, filesystem, networking con el objetivo de ejecutar aplicaciones en contextos dedicados. Además, brinda herramientas que permiten empaquetar aplicaciones en un formato denominado *imagen* que facilita la distribución del software.

Como tecnología de distribución, ejecución y empaquetamiento de software, docker se ha vuelto un estándar en la industria del desarrollo de software. Además de las ventajas y características mencionadas, podemos destacar que docker tiene un overhead [14] mucho menor que la virtualización de máquinas virtuales, debido a que no debe virtualizar componentes de hardware ni ejecutar un sistema operativo completo para poder funcionar, esto puede observarse en la Figura 18.

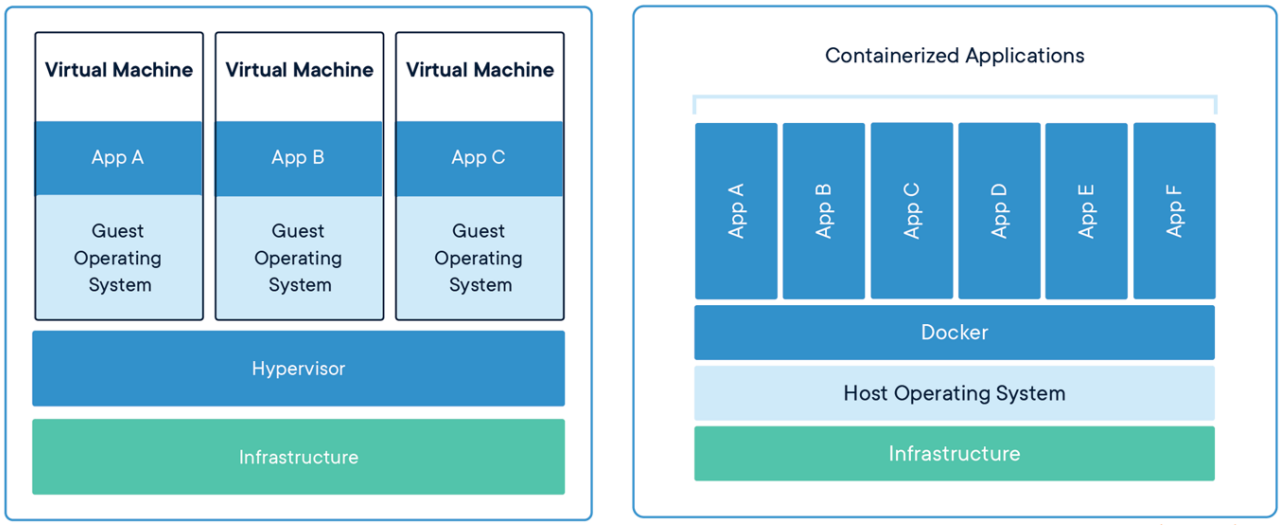


Figura 18: Comparación entre contenedores y máquinas virtuales. (Fuente: Medium [33])

Como herramienta para distribuir y ejecutar la aplicación desarrollada en el contexto de este trabajo, se optó por docker debido a las ventajas antes mencionadas.

[13] Virtualización *liviana*. Consume menos recursos que la virtualización de hardware tradicional al reutilizar el sistema operativo del host.

[14] Consumo de recursos base del hypervisor necesarios para realizar la virtualización.

3.7. Mininet

Mininet es un software para emular redes que permite desplegar ambientes representativos de topologías reales fácilmente. Utiliza OVS como virtual switch y soporta cualquier controlador que tenga una implementación de OpenFlow. Mininet puede desplegarse en máquinas virtuales, hardware físico o contenedores. Todas estas características la convierte en una plataforma ideal para tareas de investigación. Además, posee un API en python que permite programar escenarios complejos de simulación sin la necesidad de interactuar de forma manual.

En el contexto de este trabajo, se utilizó mininet y, puntualmente, el API en python para automatizar escenarios de pruebas de calidad de servicio en tiempo real. Estas pruebas permitieron reducir el tiempo de implementar un laboratorio complejo para realizar pruebas de integración entre el controlador y los switches elegidos. Estas pruebas serán tratadas en detalle en el capítulo 5.

A continuación, se observa el uso del API que ofrece mininet [37]. En este caso, se implementa una red con topología estrella compuesta de un switch OVS con tres hosts. El switch se conecta a un controlador local utilizando OpenFlow 1.3. Lo interesante de esta API, es que permite no solo automatizar el despliegue de un escenario puntual de red sino, también, escenarios de pruebas de tráfico.

```
1  from mininet.cli import CLI
2  from mininet.log import setLogLevel
3  from mininet.net import Mininet
4  from mininet.topo import Topo
5  from mininet.node import RemoteController, OVSSwitch
6
7  class MinimalTopo( Topo ):
8      "Minimal topology with a single switch and three hosts"
9
10     def build( self ):
11         # Create two hosts.
12         h1 = self.addHost( 'h1' )
13         h2 = self.addHost( 'h2' )
14         h3 = self.addHost( 'h3' )
15
16         # Create a switch
17         s1 = self.addSwitch( 's1' )
18
19         # Add links between the switch and each host
20         self.addLink( s1, h1 )
21         self.addLink( s1, h2 )
22         self.addLink( s1, h3 )
23
24 def runMinimalTopo():
25     "Bootstrap a Mininet network using the Minimal Topology"
26
27     # Create an instance of our topology
28     topo = MinimalTopo()
29
30     # Create a network based on the topology using OVS and controlled by
31     # a remote controller.
32     net = Mininet(
33         topo=topo,
34         controller=lambda name: RemoteController( name, ip='ryu', port=6653 ),
35         switch=OVSSwitch,
36         autoSetMacs=True )
37
38     # Actually start the network
39     net.start()
40
41     # hosth1 = net.get('h1')
42     # hosth2 = net.get('h2')
```

```
43      switch1 = net.get('s1')
44      switch1.cmdPrint('ovs-vsctl set-manager ptcp:6632')
45      switch1.cmdPrint('ovs-vsctl set Bridge s1 protocols=OpenFlow13')
46      switch1.cmdPrint('ovs-vsctl show')
47      switch1.cmdPrint('ovs-ofctl -O OpenFlow13 show s1')
48      net.pingAll()
49      # hosth1.cmdPrint('ping ' + hosth2.IP())
50
51      # Drop the user in to a CLI so user can run commands.
52      CLI( net )
53
54      # After the user exits the CLI, shutdown the network.
55      net.stop()
56
57  if __name__ == '__main__':
58      # This runs if this file is executed directly
59      setLogLevel( 'info' )
60      runMinimalTopo()
61
62  # Allows the file to be imported using `mn --custom <filename> --topo minimal`
63  topos = {
64      'minimal': MinimalTopo
65  }
```

3.8. Asterisk (IP PBX)

Asterisk es un framework open-source que permite construir aplicaciones de comunicaciones. El caso de uso más emblématico de asterisk en la industria es el de aplicación IP PBX, es decir, como central telefónica por software con soporte para el protocolo SIP, entre otros. Además asterisk permite, hardware mediante, el uso de interfaces analógicas y digitales como pueden ser puertos FXS/FXO, tramas digitales ISDN, entre otros.

Como plataforma para construir aplicaciones, asterisk crea un conjunto de interfaces y abstracciones que serán usadas para el desarrollo de las aplicaciones de calidad de servicio. A continuación se describen estos componentes en detalle.

3.8.1. APIs

Asterisk posee, además, interfaces programables o APIs que permite obtener información del estado de las comunicaciones en tiempo real asi como poder registrar aplicaciones propias que modifiquen el flujo estándar de las comunicaciones o, también, crear flujos nuevos inexistentes.

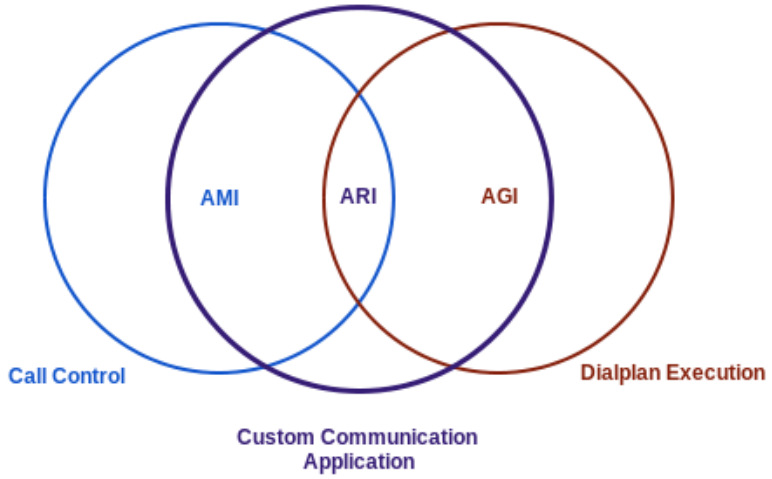


Figura 19: APIs de Asterisk. (Fuente: Asterisk [4])

- AGI: Interfaz síncrona que bloquea el thread [15] de aplicación cuando una acción es realizada en un canal. Permite manipular los canales de una comunicación.
- AMI: Interfaz asíncrona, basada en eventos. No permite realizar acciones sobre los canales pero si obtener información sobre los estados de los mismos

Como puede observarse, dada la naturaleza de las primeras APIs que Asterisk ofreció, una aplicación que requiera manipular comportamientos de un canal debe coordinar el uso de dos interfaces muy distinas. Además, al ser interfaces desarrolladas en estadíos tempranos del proyecto, ofrecen tecnologías que han quedado en desuso y no contemplan conceptos JSON-RPC/REST. Para resolver este inconveniente, los desarrolladores del producto liberaron una interfaz específica para el desarrollo de aplicaciones de comunicaciónes que utilizan de base el motor de canales y los protocolos que ofrece asterisk. Esta interfaz se llama ARI, y es el acrónimo de *Asterisk RESTful Application*. ARI fué desarrollada con el objetivo de disponibilizar, en una sola interfaz, todo lo necesario para poder desarrollar aplicaciones basadas en estados y que permitan manipular los canales de comunicación de asterisk como se observa en la Figura 19. Los componentes principales de ARI son:

- Componente asíncrono RESTful que permite a un cliente controlar los recursos de asterisk
- Eventos de los recursos controlados a través de websocket en JSON [16]
- Aplicación Stasis, que toma control de canal/canales en asterisk en base a configuración del dialplan

[15] Flujo de ejecución utilizado por la aplicación que utiliza recursos de procesamiento.
[16] Javascript Object Notation.

3.8.2. Canales

En asterisk, los canales son vías de comunicación entre un cliente y la central telefónica. Los canales manejan tanto la señalización como el tráfico multimedia de la llamada. Cuando un canal es creado, asterisk asigna un ID único que lo distingue del resto de los canales y que permite manipularlo mientras dure la comunicación. En la Figura 20 puede observarse el canal de comunicación entre un endpoint SIP, este recibe un ID único junto con un nombre, dado por la tupla *(protocolo, usuario)*. Como se explicó en la sección de APIs Asterisk ofrece, a través de ARI, una interfaz RESTful para interactuar con los canales en tiempo real [3].

Hay un tipo especial de canal llamado *Local Channel*. Estos canales son un componente interno que permite vincular dos canales de comunicación. Para que el tráfico multimedia perteneciente a los canales de dos usuarios distintos pueda intercambiarse se utilizan esos canales que hacen de hub. Durante el desarrollo de este trabajo se utilizaran componentes similares llamados *puentes* que serán explicados más adelante en este mismo capítulo.

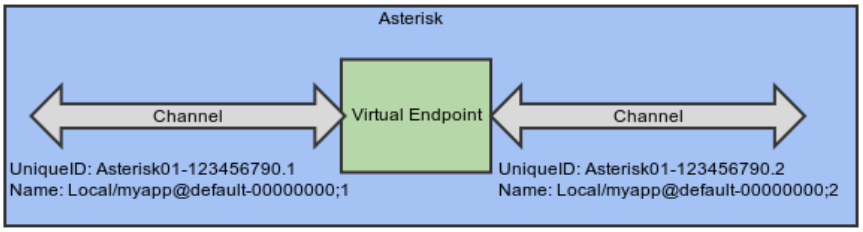


Figura 20: Local Channel actúa como hub entre dos canales de comunicación. (Fuente: Asterisk [3])

3.8.3. Puentes

Los puentes [2] pueden pensarse como un contenedor de canales que provee caminos para comunicar a estos canales entre sí. A diferencia de los *Local Channels* explicados en la sección anterior, los puentes pueden contener un número indeterminado de canales. Este tipo de componente podría utilizarse para construir aplicaciones de conferencias, IVRs, etcétera.

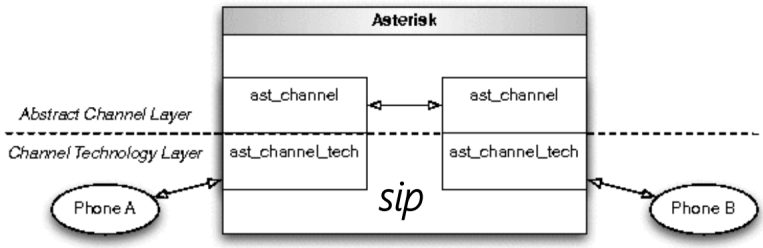


Figura 21: Comunicación entre distintos canales utilizando puentes. (Fuente: Asterisk [2])

Existen distintos tipos de puentes, entre los más importantes tenemos los siguientes:

- **mixing**: Permite que el tráfico multimedia sea compartido entre todos los canales del puente
- **holding**: Hay dos roles en este tipo de puente para los canales. El rol *participante* permite solo recibir tráfico multimedia. El rol *anunciante* realiza un broadcast del tráfico multimedia a todos los *participantes*

En el caso de los puentes de tipo *mixing*, asterisk se encarga de encontrar la forma más optima de transcoding en el caso de que existan canales utilizando tecnologías de audio distintas.

3.8.4. Aplicaciones Stasis

Las aplicaciones *Stasis* son aplicaciones externas a Asterisk que utilizan las interfaces que provee la plataforma para agregar funcionalidades nuevas o modificar el comportamiento de flujos de comunicación existente. Se podría utilizar como ejemplo una aplicación Stasis llamada *application*. Para utilizarla se vincula el dial plan de una o varias extensiónes nuestra aplicación, en cuanto una llamada entrante llegue a esa extensión el control del canal del cliente pasa a tomarlo la aplicación Stasis. En ese momento, *application* implementará la lógica necesaria apoyándose en las interfaces que ofrece asterisk. A través de un websocket podrá recibir eventos de

los canales (StasisStart, StasisEnd) y reaccionar ante ellos utilizando la REST API para cortar una llamada, redireccionarla, inyectar un audio específico, o cualquier otra acción que ARI disponibilice. Esta dinámica puede observarse en la Figura 22.

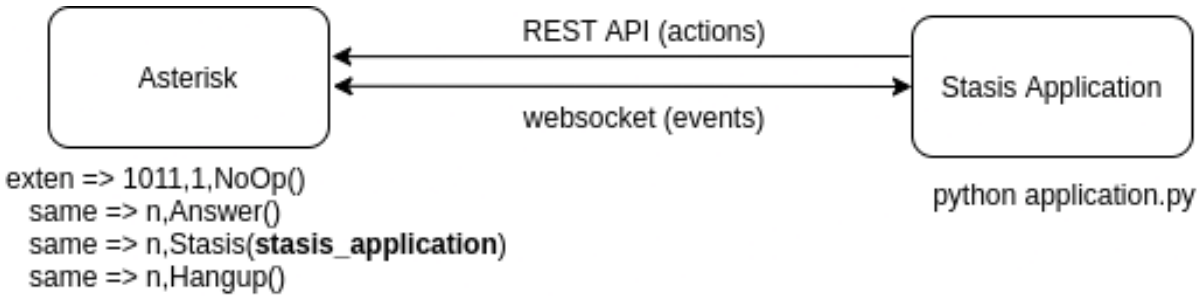


Figura 22: Conexión entre extensión y aplicación Stasis. (Fuente: Elaboración propia)

Asterisk ofrece APIs para distintos lenguajes de programación, entre ellos, una versión para Python. Esta API ofrece una abstracción para interactuar con el websocket y la REST API de ARI de manera simplificada para el desarrollo de aplicaciones. A continuación, se muestra un ejemplo de una aplicación llamada *channel-dump* que muestra, en tiempo real, las llamadas entrantes a una extensión (1011) y las llamadas que se cortan. Esto se hace a través de callbacks, son funciones que se ejecutan ante la llegada de eventos de tipo **StasisStart** y **StasisEnd** a través del websocket.

```
#!/usr/bin/python2.7

import ari
import logging

ARI = '192.168.10.51'

logging.basicConfig(level=logging.ERROR)

client = ari.connect('http://{}:8088'.format(ARI), 'asterisk', 'asterisk')

current_channels = client.channels.list()
if len(current_channels) == 0:
    print("No hay canales registrados :-(")
else:
    print("Canales registrados:")
    for channel in current_channels:
        print(channel.json.get('name'))

def stasis_start_cb(channel_obj, ev):
    """Handler for StasisStart event"""

    channel = channel_obj.get('channel')
    print("Channel %s ha entrado a la aplicacion" % channel.json.get('name'))

    for key, value in channel.json.items():
        print("%s: %s" % (key, value))

def stasis_end_cb(channel, ev):
    """Handler for StasisEnd event"""

    print("%s ha dejado la aplicacion" % channel.json.get('name'))

client.on_channel_event('StasisStart', stasis_start_cb)
client.on_channel_event('StasisEnd', stasis_end_cb)

client.run(apps='channel-dump')
```

La aplicación comienza mostrando los canales actuales y luego entra en el *application loop*. Este loop ejecutará los callbacks mencionados ante la recepción de los eventos de los canales.

```
(python) jgonzalez@cerealkiller:~/dev/python/src/asterisk(master)$ python2.7 channels-dump.py
No hay canales registrados :-(
Channel SIP/alice-00000003 ha entrado a la aplicacion
accountcode:
name: SIP/alice-00000003
language: en
caller: {u'name': u'', u'number': u'alice'}
creationtime: 2020-10-03T23:33:11.134+0000
state: Up
connected: {u'name': u'', u'number': u''}
dialplan: {u'priority': 3, u'exten': u'1011', u'context': u'default'}
id: 1601767991.8
SIP/alice-00000003 ha dejado la aplicacion
```

Figura 23: Aplicación Stasis funcionando. (Fuente: Elaboración propia)

Puede observarse que esta estructura de interfaces permite comenzar a desarrollar una lógica que haga uso de todo el motor de funcionalidades que una plataforma como asterisk oferce. Durante este trabajo se utilizará una aplicación Stasis con puentes para la implementación de la funcionalidad de **call admission control** que será discutida en detalle en el capítulo 4.

3.9. Graphical Network Simulator 3

GNS3 [22] (Graphic Network Simulation) es un simulador gráfico de red que permite diseñar topologías de red complejas y poner en marcha simulaciones sobre ellas. GNS3 utiliza un conjunto diverso de proyectos OpenSource con el objetivo de dar soporte a tecnologías de distintos fabricantes. Entre los proyectos más importantes se encuentran:

- Dynamips [13], un emulador de IOS que permite a los usuarios ejecutar binarios imágenes IOS de Cisco Systems.
- Dynagen [15], un front-end basado en texto para Dynamips
- Qemu [45], un emulador de PIX.GNS3 es una excelente herramienta complementaria a los verdaderos laboratorios para los administradores de redes de Cisco o las personas que quieren pasar sus CCNA, CCNP, CCIE DAC o certificaciones.

En el contexto de este trabajo, GNS3 se utiliza para montar un ambiente de laboratorio representativo de escenarios productivos.

3.10. SIPp

SIPp [38] es una herramienta OpenSource que permite generar patrones de tráfico para el protocolo SIP. Además, incluye plantillas de escenarios que simulan User Agents y realizan comunicaciones SIP con tráfico multimedia RTP y todos los métodos esperados durante una comunicación (INVITE, BYE, SESSION PROGRESS, etc). Estos escenarios pueden ser parametrizados en archivos XML y cargados en tiempo de ejecución.

En este trabajo se utiliza SIPp para emular patrones de llamada que permitan realizar pruebas automatizadas de las funcionalidades de la aplicación.

3.11. iPerf

iPerf [30] es una herramienta que permite realizar mediciones sobre el ancho de banda de enlaces en redes IP a través de la simulación de tráfico sobre los mismos. Soporta configuración granular sobre parámetros varios del tráfico a simular relacionados a timing, buffers, protocolos (TCP, UDP, SCTP sobre ipv4 e ipv6).

En el contexto de este trabajo, IPerf se usa para realizar mediciones en el ancho de banda de los enlaces sobre los cuales se configure calidad de servicio.

4. Proyecto

4.1. Introducción

En este capítulo se describe la realización del proyecto, partiendo del diseño de la arquitectura de la aplicación. A continuación, se explicarán los flujos de comunicación entre los distintos servicios que la componen y, luego, mediante simulaciones y laboratorios representativos, se describirán los comportamientos y resultados obtenidos.

4.2. Arquitectura de la aplicación

4.2.1. Microservicios

Para el desarrollo de la aplicación se decidió trabajar con una arquitectura basada en microservicios [35]. Es decir, en vez de consolidar todo el proyecto en una única pieza de código, también llamada monolito, se decidió desagregar las distintas partes funcionales de la aplicación en servicios atómicos e independientes.

Esta decisión fue motivada principalmente por los siguientes motivos:

- **Modularidad:**
  - Separar el desarrollo en distintos servicios permite paralelizar el trabajo.
  - Cada servicio puede estar desarrollado en tecnologías distintas mientras haya interfaces (APIs) bien definidas de comunicación, motivo por el cual se obtiene mayor versatilidad.
  - Se puede actualizar/apagar solo una parte de la aplicación sin afectar a todo el conjunto.
- **Escalabilidad:** Los servicios pueden escalar de forma independiente. Esto permite acudir a técnicas de escalamiento horizontal que proveen orquestadores de contenedores sin tener que escalar todos los componentes de la aplicación.
- **Mantenibilidad:** Mantener una pieza de código pequeña que realiza una sola función simplifica las tareas de mantenimiento y refactorización del código.

Además de estas ventajas inherentes a la arquitectura, la naturaleza del problema que necesitamos resolver implica la comunicación entre distintas piezas de software e interfaces, por lo que la elección natural a este problema fueron los microservicios.

4.2.2. Funcionalidades

Retomando lo explicado en el capítulo 2 de este trabajo respecto de Call Admission Control y Calidad de Servicio, se analizan en detalle cómo se encaran, desde la perspectiva de la aplicación desarrollada, estos dos problemas.

Una implementación de **Call Admission Control** consiste en un mecanismo central dedicado a permitir o denegar la inicialización de llamadas basado en una política o umbral preexistente. El objetivo de este mecanismo es el de permitir solamente las comunicaciones sobre las cuales se pueda garantizar una calidad dependiendo de los recursos disponibles en la red.

Un implementación de **Calidad de servicio** en el contexto de comunicaciones de voz real-time, podría basarse en la configuración de colas en los dispositivos que participan de la conmutación de paquetes de voz para que prioricen y garanticen ciertas condiciones a esos paquetes.

Estos dos mecanismos funcionan juntos de manera tal de mejorar la experiencia de los usuarios de la red. Los umbrales de **Call Admission Control** se eligen de acuerdo a los codecs a utilizar y la cantidad de ancho de banda reservado en las colas de *Express Forwarding (EF = DSCP46)*. De esta manera, se evitan situaciones en donde la cola *EF* tenga desborde y comience a perder paquetes.

$$BW_{interfaz} = BW_{G711} \times LC$$

LC hace referencia a la cantidad de llamadas concurrentes que serán permitidas por CAC. El ancho de banda de la interfaz está relacionado al ancho de banda configurado en la cola *EF*.

La aplicación desarrollada implementa tres módulos, como puede observarse en la Figura 24, que se describen a continuación.

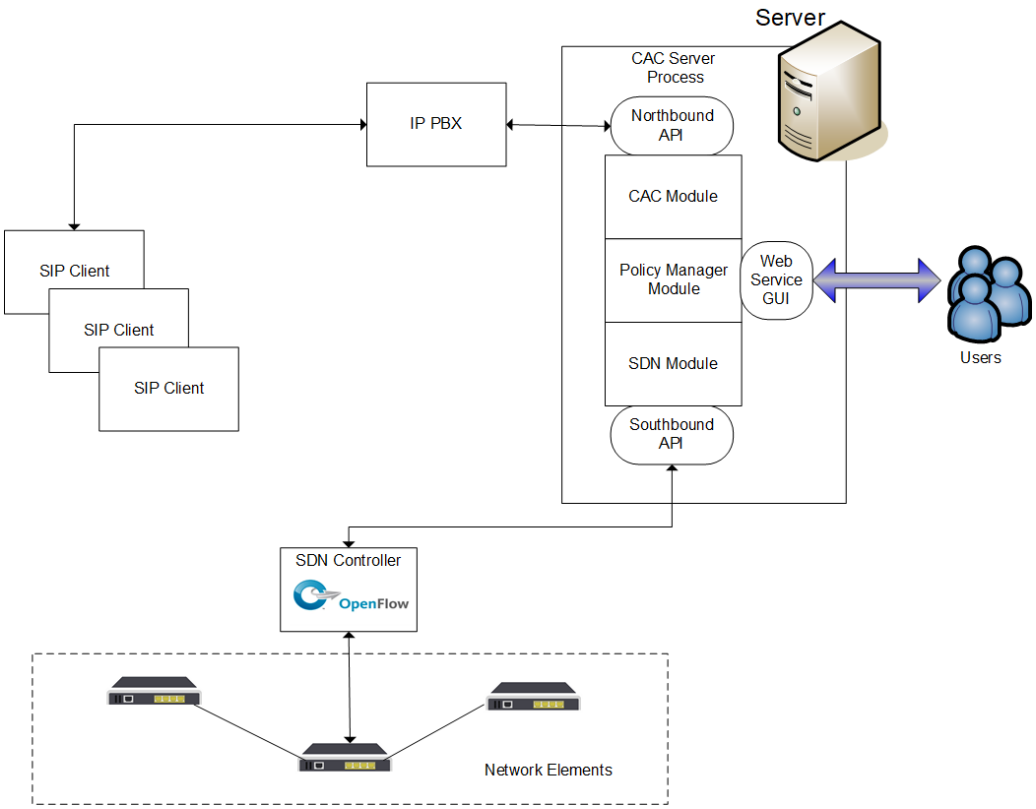


Figura 24: Módulos e interfaces de aplicación. (Fuente: Elaboración propia [29])

- **CAC Module:** Módulo que interactúa a través de northbound interfaces (API HTTP Restful, Websocket) con la central IP PBX. Este módulo implementa la lógica que permite contabilizar la cantidad de llamadas concurrentes en la red reaccionando a eventos **StasisStart** y **StasisEnd** que envía la central a través del websocket. Si el umbral se excede, entonces este módulo podrá finalizar las nuevas comunicaciones para impedir que exista tráfico de voz cuya calidad no puede ser garantizada.

- **Policiy Manager Module:** Este módulo permite la configuración, desde el frontend de la aplicación, de parámetros que modifiquen el comportamiento de las funcionalidades implementadas en backend. Para el alcance de este proyecto, los parámetros configurables son **feature flags** que permiten habilitar y deshabilitar las funcionalidades de CAC y QoS. El umbral se configura en código y actualmente no puede ser modificado en tiempo de ejecución.
- **SDN Module:** Este módulo se encarga de interactuar con la southbound interface [44] del controlador SDN (API HTTP Restful) para implementar las colas de calidad de servicio ante eventos de **StasisStart** enviados por la central IP PBX y de eliminar las colas ante eventos de **StasisEnd**

Es importante señalar que, si bien la aplicación interactúa a través de un API HTTP con el controlador SDN, el controlador, luego, traduce esas peticiones a comandos sobre el protocolo OpenFlow que luego serán comunicados a los elementos de red programables de la infraestructura que, en el caso de este trabajo, se han utilizado switches OpenVirtualSwitch como se describe en el capítulo 3.

Otro aspecto interesante a señalar, es que el funcionamiento de la aplicación está orientado a **eventos** generados por los usuarios de la central IP PBX que requieren utilizar recursos de la red. La aplicación funciona con un **event-loop** a la espera de eventos ante los cuales debe reaccionar con lógica específica para realizar una configuración sobre la red o sobre la central. Esta lógica puede observarse en el diagrama de flujo de la Figura 25 donde se puede ver cómo los eventos de llamadas disparan las distintas funcionalidades explicadas anteriormente.

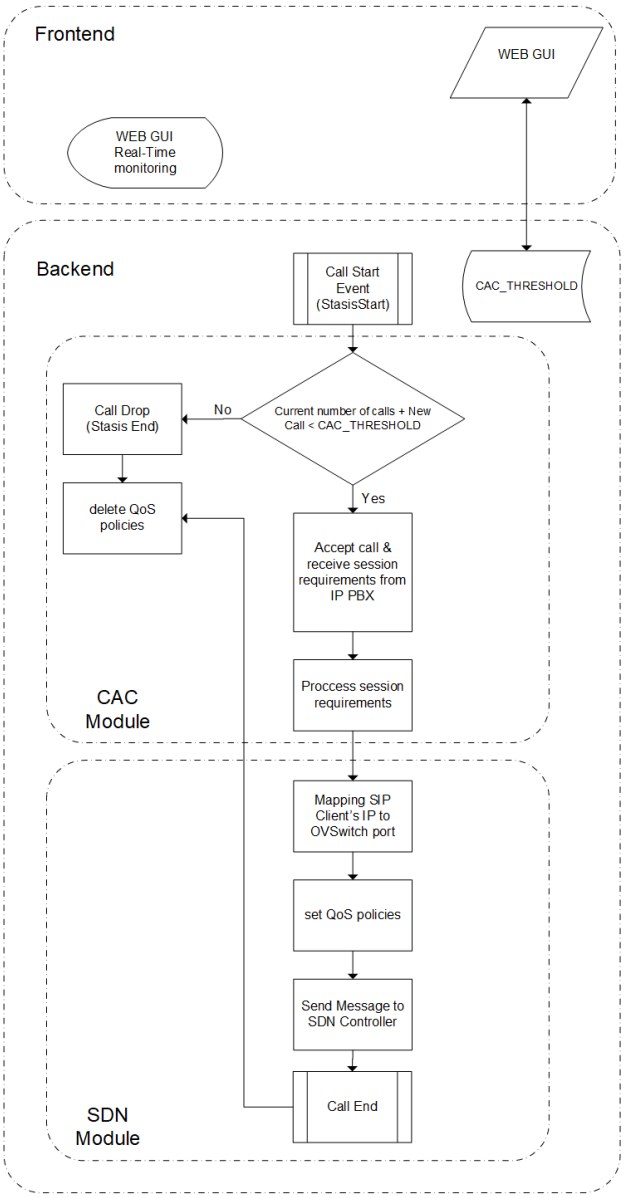


Figura 25: Diagrama de flujo de módulos CAC y SDN. (Fuente: Elaboración propia [29])

En la figura 25 se muestra el diagrama de flujo de la lógica implementada basada en eventos. Un evento **StasisStart** llega a la aplicación desde la central IP PBX a través del websocket, si el umbral de llamadas concurrentes fue alcanzado, se envía un request al API de *channels* de la central IP PBX para finalizar la llamada. Si el umbral no fue alcanzado, entonces la comunicación sigue su curso normal. Luego, se envían requests al API del controlador SDN para configurar colas que prioricen el tráfico de esa llamada reservando recursos en la interfaz. El paso *Mapping SIP Clien's IP to OVSwitch port* excede el alcance de este trabajo, pero lo contemplará como una posible mejora a realizar. Al finalizar la llamada, un evento de **StasisEnd** será enviado a la aplicación desde la central IP PBX al igual que en la inicialización. Este evento desencadenará otros requests al API del controlador SDN para eliminar las colas creadas anteriormente y, así, liberar recursos de la interfaz.

Este flujo se documenta en el diagrama de secuencia de la Figura 26.

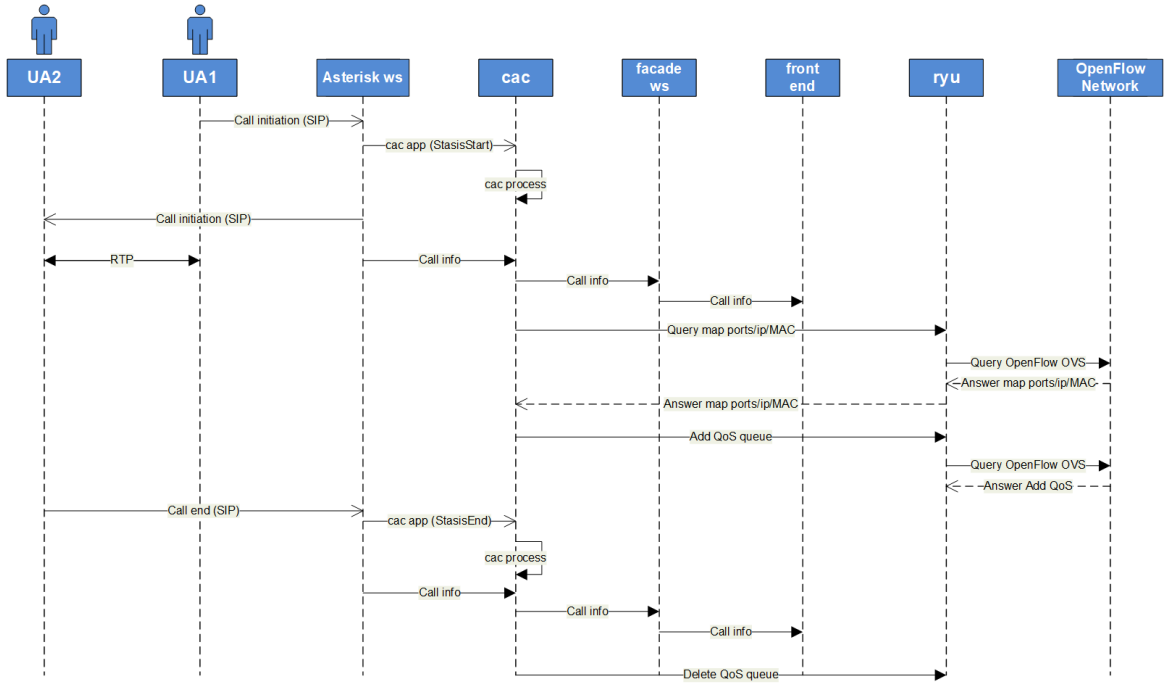


Figura 26: Diagrama de flujo de módulos CAC y SDN. (Fuente: Elaboración propia)

Este trabajo se ha centrado en explotar la arquitectura SDN y sus interfaces para desarrollar las funcionalidades detalladas en el capítulo 2. Hay funcionalidades que pueden mejorar el uso de esta aplicación para escenarios diversos y que le permitan adaptarse a diferentes topologías. Estos escenarios serán tratados en el capítulo final de este trabajo.

4.2.3. Flujos de comunicación

Como se menciona en la sección 4.2.1 de este capítulo, la aplicación se desarrolla utilizando una arquitectura basada en microservicios. A continuación, se detallan los servicios desarrollados, sus interfaces y los flujos de comunicación implementados.

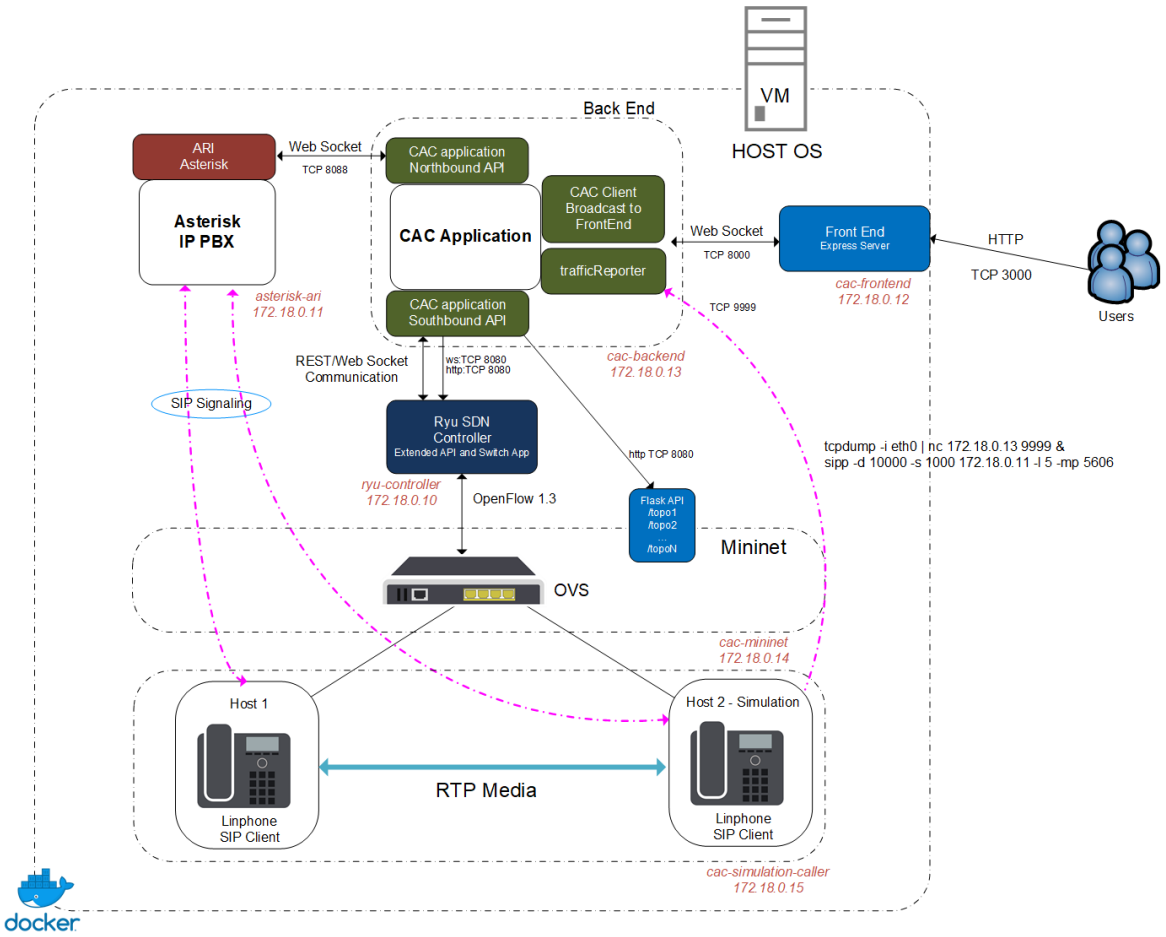


Figura 27: Microservicios y flujos de comunicación de la aplicación en el contexto de simulación. (Fuente: Elaboración propia)

En la Figura 27 pueden observarse los servicios desarrollados así como los servicios que representan componentes de infraestructura externos que han sido virtualizados con tecnología de contenedores. Además, se observan los esquemas de comunicación entre ellos.

- Backend: Implementa la lógica de los mecanismos de calidad de servicio. Mantiene canales de comunicación síncrona (API REST) y asíncrona (websocket) con la central IP PBX. Mantiene comunicación síncrona con el controlador SDN. Mantiene comunicación asíncrona (websocket) con el Fronend para enviar eventos que permitan visualización en tiempo real de lo que sucede en la red. Además, permite cambios, en forma de feature flags, en runtime de aplicación desde el Frontend.
- Frontend: Visualización del estado de llamadas concurrentes y funcionamiento de Call Admission Control. Comunicación con el backend a través de websocket.
- Asterisk IP PBX: Comunicación con backend a través de websocket. Comunicación con endpoints SIP que utilizan la central IP PBX.
- Ryu SDN Controller: Comunicación con elementos de red a través de OpenFlow. Comunicación con backend a través de canal asíncrono (API REST).
- OVS Switches: Comunicación a través de OpenFlow con controlador SDN.

Todo este esquema ha sido empaquetado en un stack de contenedores. Esto permite generar simulaciones utilizando una red overlay provista por Docker que será detallada en la sección de laboratorio.

Por otra parte, en la Figura 28, se muestra la interacción interna entre los componentes del backend de la aplicación en términos de clases de software implementadas. Por un lado, la interfaz norte, que crea el websocket con la central IP PBX, instancia una aplicación Stasis llamada **cac**. Esta aplicación manipula, a través del API REST expuesta por *ARI*, los puentes y canales de comunicación. En particular, el método *hangup_channel()* se encarga de cortar la comunicación ante un umbral alcanzado y de utilizar métodos de la clase que implementa la conexión contra el controlador para implementar o eliminar las colas en los switches.

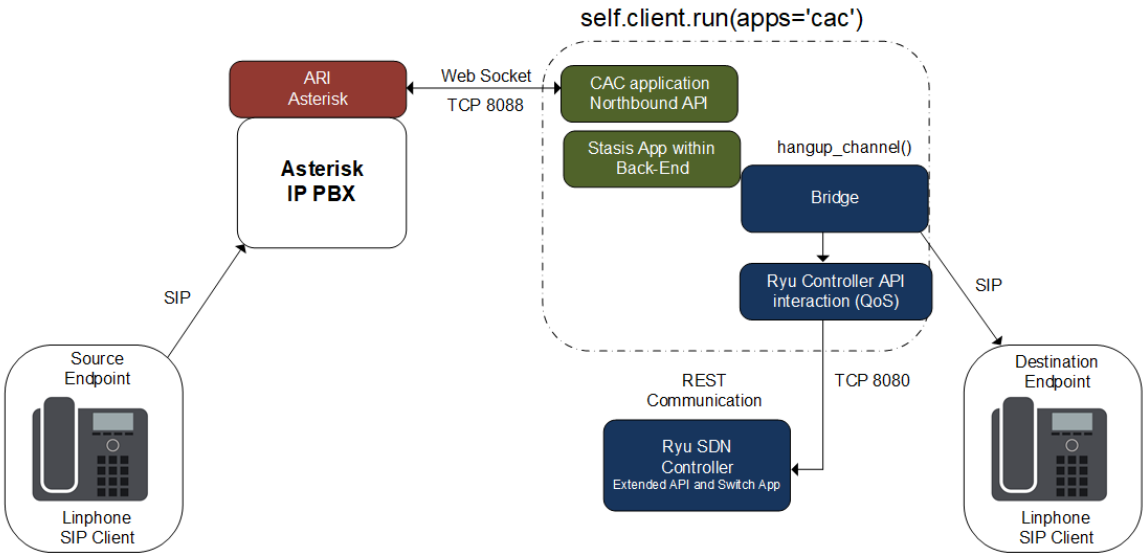


Figura 28: Componentes internos aplicación Call Admission Control. (Fuente: Elaboración propia)

4.2.4. Balanceo de carga y alta disponibilidad

Como se ha mencionado en secciones anteriores de este trabajo, la aplicación se desarrolló contemplando una arquitectura basada en microservicios. En este contexto, el empaquetamiento y la distribución de la aplicación se realiza utilizando tecnología de contenedores. Puntualmente, las tecnologías utilizadas son Docker [36] y Docker Swarm [26]. Docker, como tecnología, fue explicada en el capítulo 3 de este trabajo. Docker Swarm es una tecnología de orquestación de contenedores, permite crear clusters para la ejecución de aplicaciones. Cluster refiere a la utilización de una o mas máquinas virtuales o servidores físicos que trabajan en conjunto con el objetivo de mantener el estado deseado de los servicios que se implementan utilizando algoritmos de reconciliación que se comunican con los servidores disponibles para ejecutar los contenedores necesarios. Además, proveen la posibilidad de configurar redes overlay utilizando VXLAN como tecnología de base para que los contenedores que pertenecen a una misma aplicación puedan comunicarse entre sí en un ambiente de red consistente entre las distintas máquinas virtuales. Esto puede verse en la Figura 29, donde los contenedores c1 y c2, que pertenecen a la misma aplicación, se ven a través de una única red de capa 3 gracias a que la misma es encapsulada a través de una VXLAN entre los servidores del cluster.

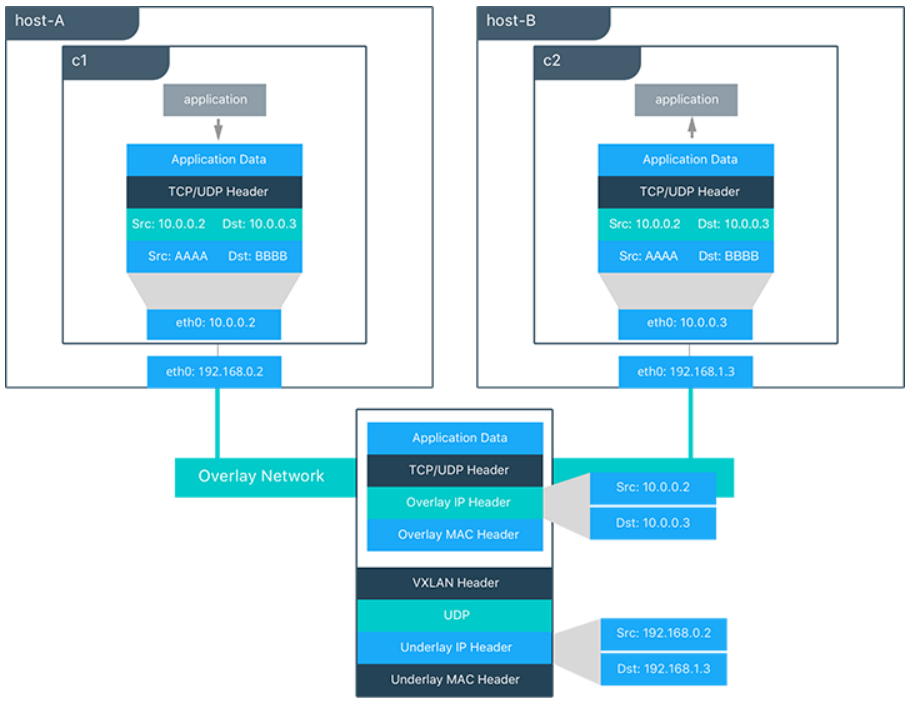


Figura 29: Red Overlay utilizando VXLAN en docker swarm. (Fuente: Docker Inc [27])

En términos de red, también podemos decir que el tráfico externo que entra al cluster es manejado a través de una *ingress network*. Esta red publica los puertos necesarios y se comporta de manera transparente entre los distintos nodos del cluster. Es decir, sin importar contra qué servidor del cluster llegue la conexión, la misma será enrutada hacia el contenedor correspondiente gracias a un sistema de balanceo de carga interno. Esto puede observarse en la Figura 30, en donde el puerto 8080 es publicado en el cluster y mapeado con el puerto 80 de una aplicación. El tráfico entrante, independientemente del nodo, será enrutado a los contenedores correspondientes.

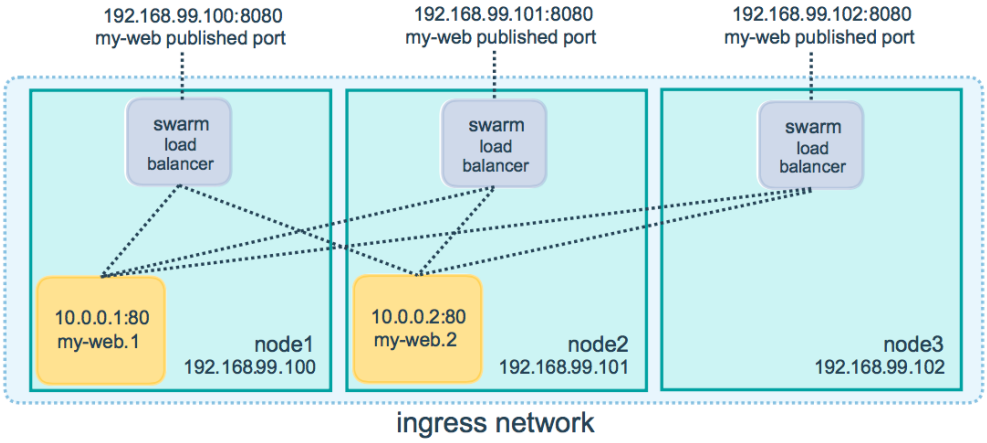


Figura 30: Ingress network en cluster de docker swarm. (Fuente: Docker Inc [28]

Además de contar con estas funcionalidades a nivel comunicación, la aplicación puede escalar horizontalmente de manera transparente. En estos escenarios los contenedores serán distribuidos de forma homogénea según el algoritmo de *scheduling* de docker swarm. En la Figura 31 se puede ver a la aplicación implementada en un cluster. En este caso, cada microservicio se implementa en un contenedor dedicado, y cada servicio puede escalar horizontalmente según sea necesario como se observa en la Figura 32 donde se escala el servicio correspondiente al controlador SDN Ryu.

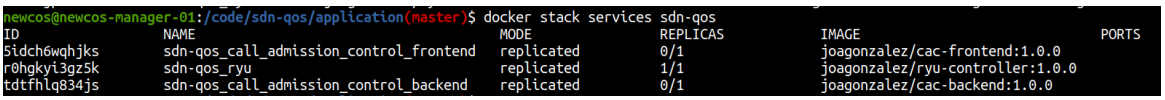


Figura 31: Estado servicio swarm. (Fuente: Elaboración propia)

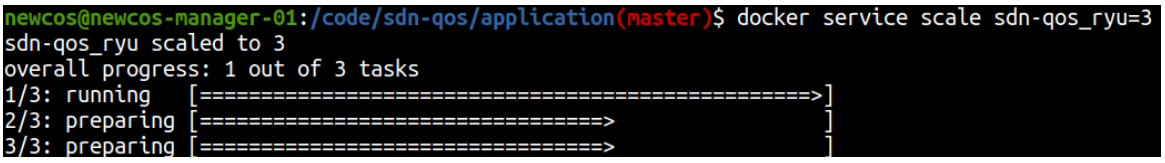


Figura 32: Escalar servicio swarm. (Fuente: Elaboración propia)

Al finalizar el proceso de *scheduling*, el cluster muestra el estado final del servicio que ahora tiene tres replicas. Cuando el tráfico entrante llegue a la ip virtual del servicio, será balanceado entre las tres replicas del mismo.

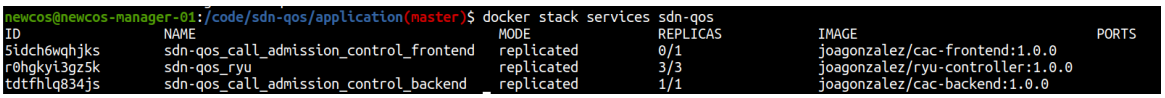


Figura 33: Estado final servicio swarm. (Fuente: Elaboración propia)

4.3. Laboratorio

En esta sección se describen las simulaciones y laboratorios realizados con el objetivo de poner a prueba el funcionamiento de la aplicación.

4.3.1. Calidad de servicio

En una primera instancia, se realizó un ejercicio de simulación para modificar colas de calidad de servicio en switches OVS a través del API del controlador SDN Ryu [43]. La clase desarrollada en esta prueba será portada, luego, a la aplicación CAC para realizar la misma funcionalidad ante callbacks asociados a eventos de tipo **StasisStart** y **StasisEnd** según corresponda. Para esta prueba, se utiliza una instalación de Ryu estándar (sin contenedores) y la aplicación mininet para crear los switches OVS. Para el desarrollo de la simulación se utiliza el API de Python que provee mininet, a la cual se le incorpora la clase antes mencionada denominada *Simulation*.

En la Figura 34 puede observarse el escenario puesto a prueba con sus componentes. Se simula una topología con dos switches conectados entre sí a través de un único enlace. Cada uno de estos switches tiene, a su vez, un host conectado. El host H1 ejecuta el software para generar tráfico IPerf en formato de servidor y escuchará en los puertos 5001, 5002 y 5003 respectivamente. Por otro lado, el host H2 ejecutará IPerf como cliente. El switch S2 actuará como elemento de borde de la red de transporte, y es donde se marcará el tráfico con distintos DSCP tags.

- Tráfico con puerto destino 5001: Se marca con DSCP 26
- Tráfico con puerto destino 5002: Se marca con DSCP 10
- Tráfico con puerto destino 5003: Se marca con DSCP 12

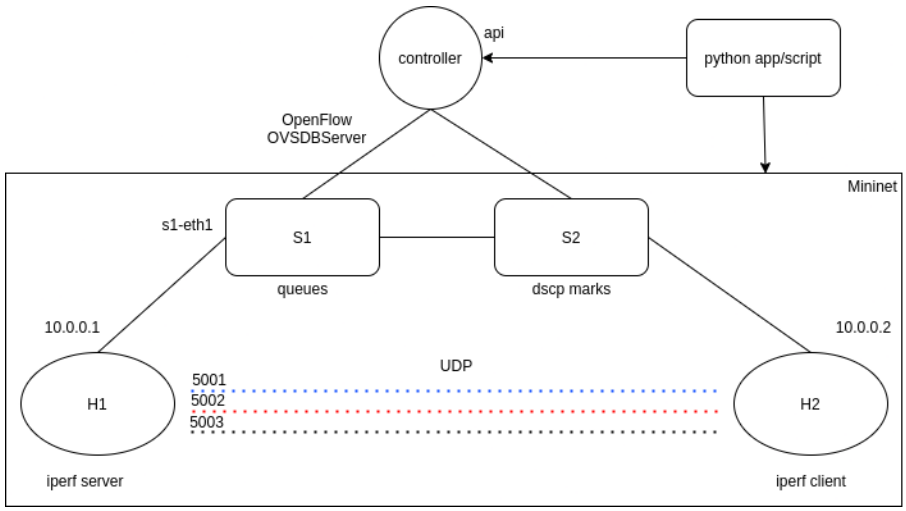


Figura 34: Esquema de laboratorio para ejecución de simulación de calidad de servicio. (Fuente: Elaboración propia)

Estas reglas se implementan con el método *dscpMark* de la clase *Simulation*. El payload se construye en formato JSON antes de ser enviado al API del controlador SDN.

```
    # Tagging de campo Type of Service con DSCP
    def dscpMark(self, switch):
        datapath = self.switch_query(switch)
        api = self.api

        # Reglas DSCP
        endpoint = self.QOS_RULES + datapath
        rules = [{"match": {"nw_dst": "10.0.0.1", "nw_proto": "UDP", \
                "tp_dst": "5001"}, "actions": {"mark": "26"}}, \
                 {"match": {"nw_dst": "10.0.0.1", "nw_proto": "UDP", \
                 "tp_dst": "5002"}, "actions": {"mark": "10"}}, \
                 {"match": {"nw_dst": "10.0.0.1", "nw_proto": "UDP", \
                 "tp_dst": "5003"}, "actions": {"mark": "12"}}
        ]

        # Queries API
        for rule in rules:
            api.post(endpoint, rule)
```

Por otro lado, en el switch S1 se crean colas para priorizar el tráfico marcado. Se crean dos esquemas de priorización que permitirán hacer cambios en *runtime* y evaluar el comportamiento del controlador.

- Escenario 1:
  - Cola 0: Max rate 100 Kbps - Match con tráfico DSCP 26
  - Cola 1: Max rate 200 Kbps - Match con tráfico DSCP 10
  - Cola 2: Max rate 300 Kbps - Match con tráfico DSCP 12
- Escenario 2:
  - Cola 0: Max rate 200 Kbps - Match con tráfico DSCP 26
  - Cola 1: Max rate 300 Kbps - Match con tráfico DSCP 10
  - Cola 2: Max rate 100 Kbps - Match con tráfico DSCP 12

Estas reglas se implementan con el siguiente código del método *qosSetup* la clase *Simulation*. El payload se construye en formato JSON antes de ser enviado al API del controlador SDN. Puede observarse que, luego de crear las colas, las mismas se aplican a tráfico con características determinadas con reglas de matcheo.

```
    def qosSetup(self, test, switch):
        '''
        Configurar qos queues en switches
        '''
        datapath = self.switch_query(switch)
        endpoint = self.OVSDB_CONF + datapath + '/ovsdb_addr'
        self.api.put(endpoint, self.OVSDB_SERVER)
        sleep(2)

        # Post Queue
        if test == 1:
            queueEndpoint = '/qos/queue/' + datapath
            queueData = {
                "port_name": "s" + str(switch) + "-eth1",
                "type": "linux-htb",
                "max_rate": "1000000",
                "queues": [{"max_rate": "100000"},
                        {"max_rate": "200000"},
                        {"max_rate": "300000"},
                        {"min_rate": "800000"}]
            }

            self.api.post(queueEndpoint, queueData)
        else:
            queueEndpoint = '/qos/queue/' + datapath
            queueData = {
                "port_name": "s" + str(switch) + "-eth1",
                "type": "linux-htb",
                "max_rate": "1000000",
                "queues": [{"max_rate": "200000"},
                        {"max_rate": "300000"},
                        {"max_rate": "100000"},
                        {"min_rate": "800000"}]
            }
            self.api.post(queueEndpoint, queueData)

        endpoint = self.QOS_RULES + datapath
        rules = [{"match": {"ip_dscp": "26"}, "actions":{"queue": "0"}},
                 {"match": {"ip_dscp": "10"}, "actions":{"queue": "1"}},
                 {"match": {"ip_dscp": "12"}, "actions":{"queue": "2"}}
        ]
        for rule in rules:
            try:
                self.api.post(endpoint, rule)
            except Exception as e:
                loggerService.error('Error configurando reglas qos en controlador')
```

Con este escenario preparado, el guión de la simulación se detalla en el siguiente bloque de código que utiliza el API de mininet. En base a un *SIMULATION_TIME* de 20 segundos, se alternará entre los escenarios de calidad de servicio que se detallaron en párrafos anteriores. De esta manera, se observará cómo la política de shaping del switch, modificará el max_rate de cada hilo de comunicación al establecido por las colas que esten aplicando en un momento dado. Para lograr una visualización clara, las tasas de transferencia de datos de los tres hilos de comunicación están seteadas a 1Mbps, por lo que el algoritmo de *shaping* siempre recortará ante el escenario propuesto.

```
    # Comenzamos simulacion
    loggerService.info( '*** Configuramos DSCP tags...' )
    simulation.dscp_mark(2)
    loggerService.info( '*** Configuramos QOS queues...' )
    simulation.qosSetup(1, 1)
    simulation.iperfTest(hosts, ports, SIMULATION_TIME)

    loggerService.info( '*** Configurando DSCP tags...' )
    simulation.dscp_mark(2)
    loggerService.info( '*** Configuramos QOS queues...' )
    simulation.qosSetup(2, 1)
    simulation.iperfTest(hosts, ports, SIMULATION_TIME)

    loggerService.info( '*** Configurando DSCP tags...' )
    simulation.dscp_mark(2)
    loggerService.info( '*** Configuramos QOS queues...' )
    simulation.qosSetup(1, 1)
    simulation.iperfTest(hosts, ports, SIMULATION_TIME)
```

Al ejecutar la simulación [17] se obtienen los resultados que se muestran en la Figura 35.

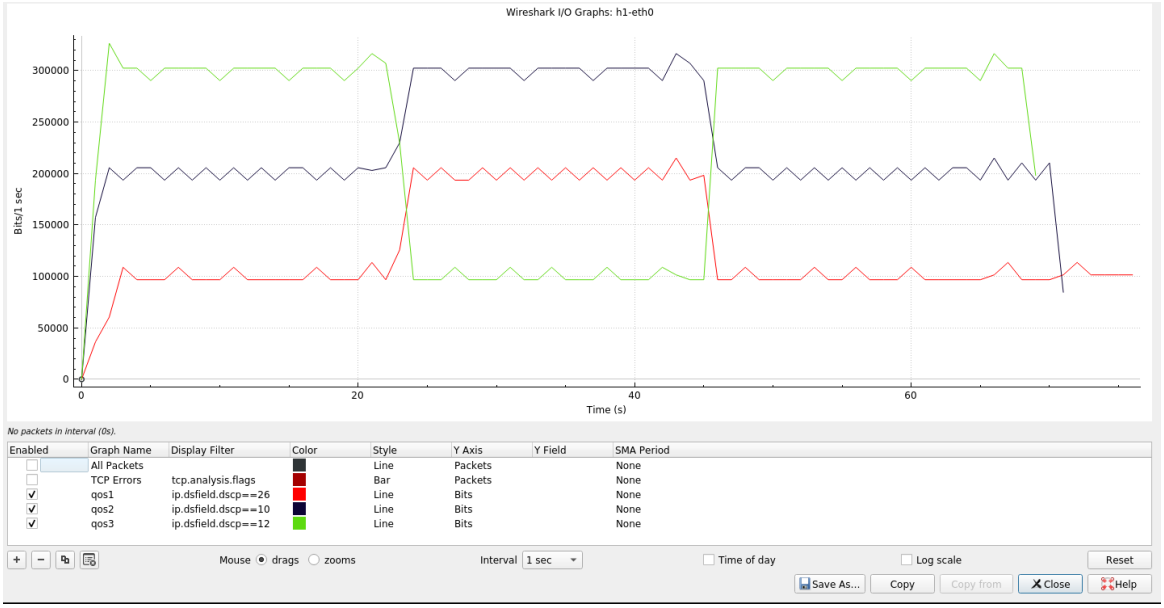


Figura 35: Resultados de laboratorio de calidad de servicio. (Fuente: Elaboración propia)

Vemos que los resultados obtenidos son consistentes con las configuraciones realizadas. Esto se consolida en el Cuadro 3. Además, se observa que los tiempos de respuesta ante la modificación de una cola en el switch a través del API son inferiores a 1 segundo, tiempos de respuesta más que aceptables teniendo en consideración el tiempo de setup de una llamada VoIP.

Escenario	**Tiempo**	**Tasa de transferencia**
1	[0-20]s	DSCP 26 = 100Kbps - DSCP 10 = 200Kbps - DSCP 12 = 300Kbps
2	[20-40]s	DSCP 26 = 200Kbps - DSCP 10 = 300Kbps - DSCP 12 = 100Kbps
1	[40-60]s	DSCP 26 = 100Kbps - DSCP 10 = 200Kbps - DSCP 12 = 300Kbps

Cuadro 3: Modificación de colas de calidad de servicio en runtime utilizando API de controlador SDN

En base a los resultados obtenidos, se llega a la conclusión de que la clase desarrollada para interactuar con las funcionalidades de calidad de servicio es apta para el caso de uso que se aborda en este trabajo.

[17] Las instrucciones para reproducir estos resultados de este laboratorio son detalladas en el Anexo I de este trabajo.

4.3.2. Call Admission Control

Con el objetivo de probar la funcionalidad de Call Admission Control, se lleva a cabo una simulación que trabaja con el mismo esquema topológico representado en la Figura 27. De esta manera, todos los servicios de la aplicación, junto con la central IP PBX se encuentran funcionando sobre una misma red. Además, se utiliza el software SIPp para simular patrones de llamadas SIP contra la central IP PBX.

El patrón de tráfico utilizado con SIPp es el siguiente. SIPp establecerá cinco llamadas concurrentes con una duración de 10 segundos. Luego, terminará las llamadas y volverá a realizar cinco llamadas con una duración de 10 segundos. Con el fin de mostrar esta simulación, se utiliza GNS3 para montar un laboratorio que ejecuta este escenario como puede verse en la Figura 36.

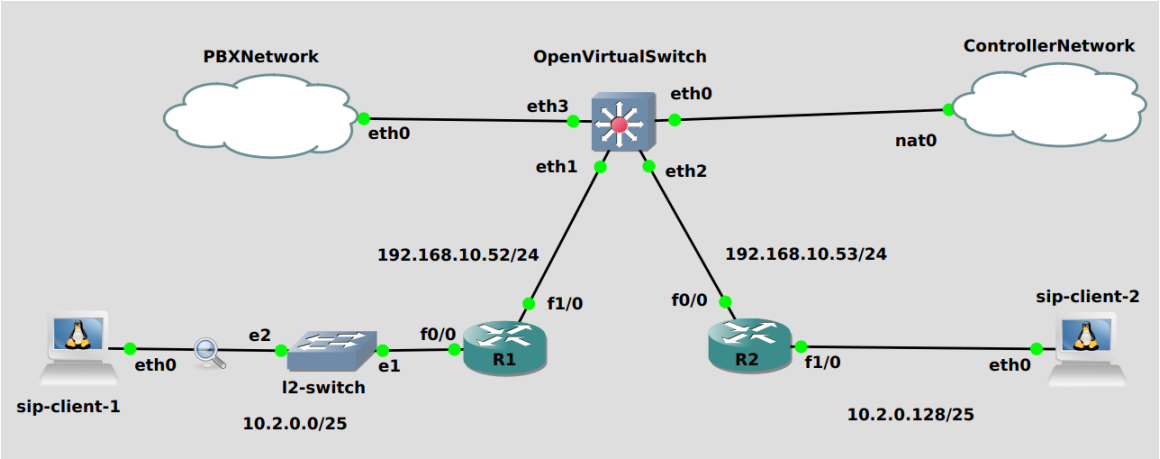


Figura 36: Laboratorio montado con GNS3 para probar patrones de tráfico de llamadas con SIPp. (Fuente: Elaboración propia)

Este laboratorio monta un contenedor linux llamado *sip-client-1* que tiene software SIPp instalado. A través de R1 y el OpenVirtualSwitch logra conectarse a una interfaz que permite conectividad contra un servidor Astersik virtualizado fuera del ambiente de GNS3 contra el cual se realizarán las llamadas. Al comenzar la simulación, se captura el tráfico en el vínculo entre *sip-client-1* y *l2-switch*. Al filtrar el tráfico RTP, se obtiene el patrón buscado como se observa en la Figura 37.

Figura 37: Patrón de llamadas obtenido con SIPp en laboratorio. (Fuente: Elaboración propia)

Con este escenario establecido, se describe la parametrización de la aplicación CAC.

Se configura un umbral de dos llamadas concurrentes para la aplicación CAC. Al ver un flujo de llamadas concurrentes superior al umbral, la aplicación estará terminando llamadas de manera constante para bajar la concurrencia de cinco a dos. Este escenario nos permitirá verificar los tiempos de respuesta y la cantidad de

tráfico RTP generado por las llamadas durante la prueba y, de esta forma, evaluar la performance de la aplicación.

Con el objetivo de poder obtener feedback de la aplicación, el servicio de frontend expone una interfaz web que nos permite ver la cantidad de llamadas concurrentes, la cantidad de llamadas finalizadas por la aplicación y los ids de las llamadas en curso como se observa en la Figura 38.

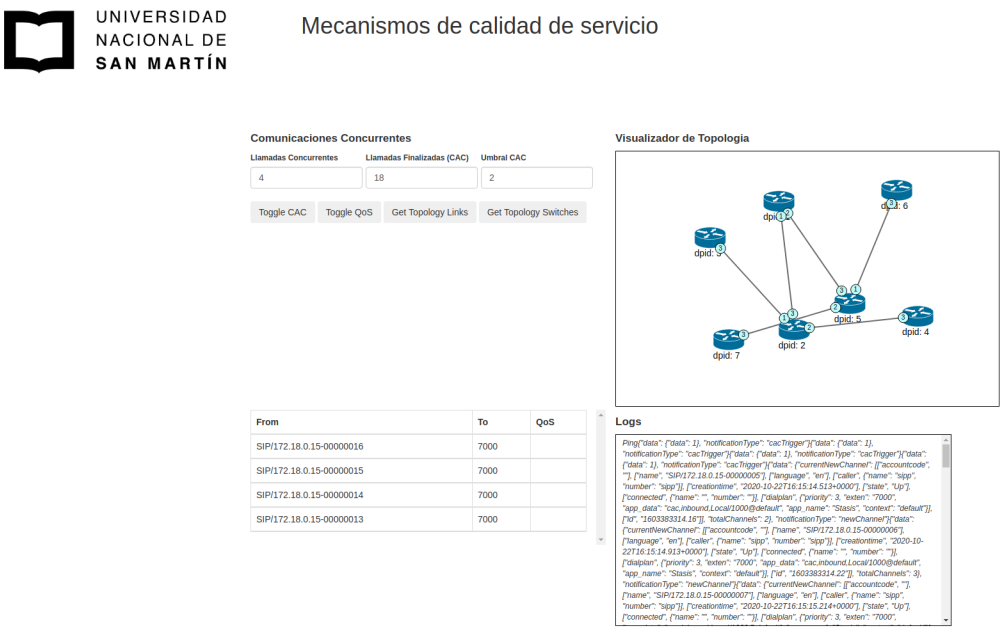


Figura 38: Interfaz web de aplicación CAC. (Fuente: Elaboración propia)

Como se menciona en la sección 4.2.2 de este capítulo, la información visualizada a través del frontend se envía a través de un websocket entre este servicio y el backend de la aplicación.

Es importante mencionar que, en la interfaz web, se muestra un esquema de la topología de switches implementados a través de los endpoints que el controlador SDN expone.

También, pueden observarse botones que nos permitirán activar o desactivar el funcionamiento de CAC y QoS en runtime. Esto es importante, ya que pueden existir coyunturas específicas en donde no se requiera cortar las llamadas que excedan la cola de *EF* y, por otra parte, pone de manifiesto la capacidad de poder activar a desactivar funciones de red sin la necesidad de interactuar con dispositivos de red de manera directa.

Los resultados obtenidos en términos de funcionalidad fueron exitosos. Las llamadas concurrentes se mantienen según el umbral establecido y la aplicación responde de manera positiva a las pruebas de stress en donde existe un software que persiste de manera sistemática llamadas por encima del umbral establecido.

En una primera instancia, el mecanismo de corte se realizó en el callback de inicio de llamada ante eventos de tipo *StasisEnd*. Puntualmente, en el hilo de comunicación saliente del bridge, esto quiere decir que el canal entrante, en ese momento, se encuentra establecido. Esto genera un delay entre el establecimiento de la llamada y el corte por parte de la aplicación, situación que da lugar al envío de tráfico RTP de audio por unos momentos. Esta situación no es ideal ya que la llamada no debe permitirse y todo tráfico de audio generado es ruido en la red y conlleva a un uso no óptimo de los recursos de ancho de banda. A continuación, se muestra el callback *outgoing_start_cb(channel_obj, ev)*.

```
    def outgoing_start_cb(channel_obj, ev):
        """StasisStart handler for our dialed channel"""
        print "{} answered; bridging with {}".format(outgoing.json.get('name'),
                                                    channel.json.get('name'))
        channel.answer()

        bridge = self.client.bridges.create(type='mixing')
        bridge.addChannel(channel=[channel.id, outgoing.id])

        CacController.connectedChannels[ bridge.id ] = True
        totalChannels = CacController.getTotalChannels()

        channel.on_event('StasisEnd', lambda *args: self.safe_hangup(outgoing, bridge))
        outgoing.on_event('StasisEnd', lambda *args: self.safe_hangup(channel, bridge))

        self.frontClient.broadcast("newChannel", {
            "currentNewChannel": channel.json.items(),
            "totalChannels": totalChannels
        })

        if self.cacEnable:
            # Hang up the channel in 4 seconds
            if totalChannels >= self.CAC_THRESHOLD:
                timer = threading.Timer(4, hangup_channel, [channel])
                self.channel_timers[channel.id] = timer
```

En la Figura 39 puede observarse el resultado obtenido. En los momentos en que la funcionalidad de CAC se encuentra activa, sigue habiendo, aunque en menor cantidad, tráfico RTP durante los segundos iniciales del SETUP de la llamada.

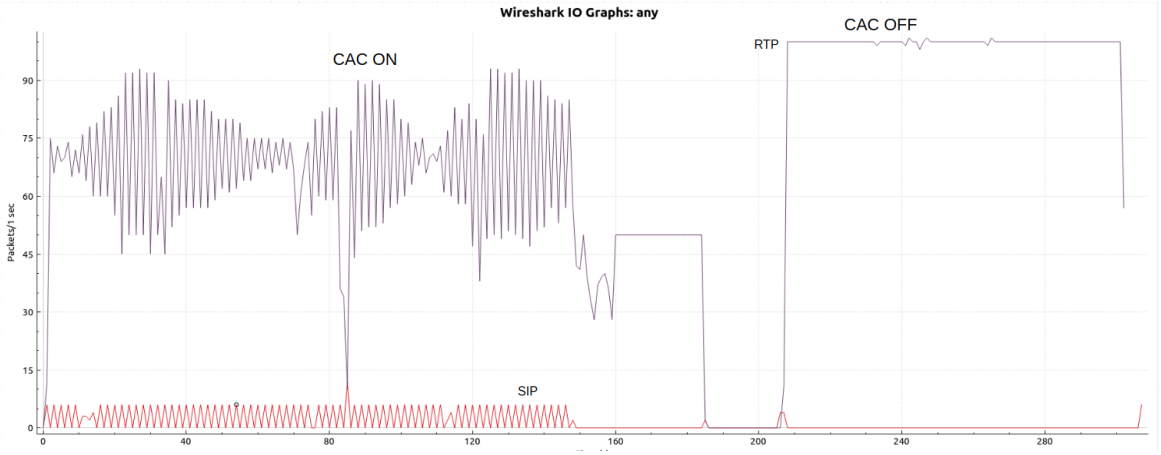


Figura 39: Version 1 de algoritmo para terminación de llamadas en aplicación CAC. (Fuente: Elaboración propia)

Con el objetivo de mejorar este comportamiento, se implementa la acción de finalización de llamada al callback del primer canal que origina la comunicación dentro del bridge. Esto permite terminar la llamada a nivel de señalización SIP. Esto conlleva a una mejora sustancial respecto del caso anterior ya que el tráfico RTP observado en este escenario se iguala con el ancho de banda consumido por SIP, el cual es prácticamente despreciable si se lo compara en términos de orden de magnitud con respecto al de una llamada de audio.

```
def onStartCallback(self, channel_obj, ev):
    ''' Hanlder for StasisStart '''

    if ev.get('args')[0] == 'inbound':
        self.incomingChannles += 1

    print '### Incoming CHANNELS:'
    print self.incomingChannels:

    channel = channel_obj.get('channel')

    if self.incomingChannels >= self.CAC_THRESHOLD:
        print 'Error: {} Call rejection by CAC mechanism!'
        channel.hangup()
        self.incomingChannels -= 1
        return
```

Puede observarse en la Figura 40 la mejora luego del refactor en el mecanismo de finalización de llamada.

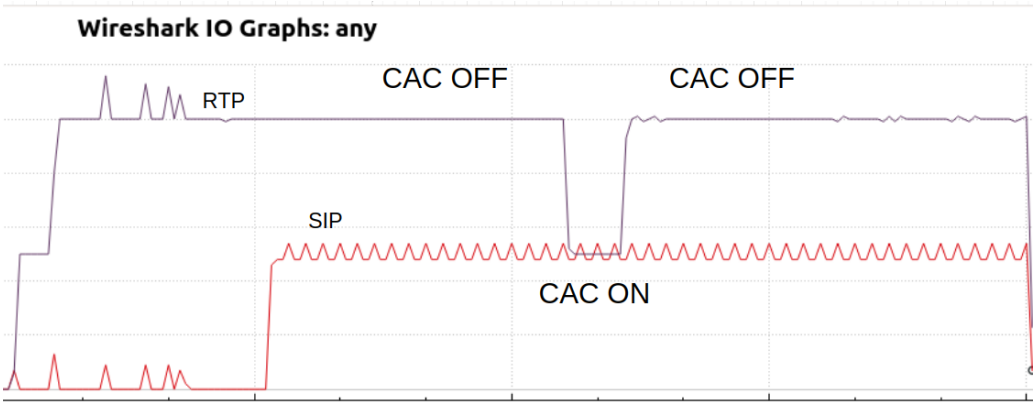


Figura 40: Version 1 de algoritmo para terminación de llamadas en aplicación CAC. (Fuente: Elaboración propia)

5. Conclusiones y nuevas líneas de trabajo

Este capítulo tiene por objetivo consolidar las conclusiones del proyecto realizado, analizar si los objetivos propuestos inicialmente fueron alcanzados y, también, discutir futuras líneas de trabajo así como mejoras que podrían realizarse sobre la aplicación desarrollada.

Las funcionalidades buscadas inicialmente en el alcance de este trabajo fueron implementadas con éxito. No obstante, debe considerarse que hay desafíos pendientes para poder adaptar el funcionamiento de la aplicación a otras topologías de red así como a otras centrales IP PBX. Por un lado, para no depender de la topología puntual donde se implemente la aplicación, deben desarrollarse mecanismos que permitan descubrir dicha topología. De esta manera, podrían desarrollarse funciones que prioricen el tráfico en todos los equipos involucrados en una comunicación punto a punto entre dos usuarios. El controlador expone endpoints [18] en su API que informan sobre los nodos y los enlaces de la red, lo cual nos garantiza factibilidad técnica al poder volcar en una estructura de datos de tipo grafo la topología de la red. Además, para poder utilizar la aplicación con otras centrales, sería interesante utilizar el patron de diseño *adapter* [17] en la clase que desarrolla la interacción con Asterisk, de esta manera, se puede trabajar con una abstracción que nos permita soportar fácilmente nuevas centrales.

Se pudo utilizar la desagregación de funcionalidades propuesta por la arquitectura SDN utilizando OpenFlow como interfaz sur y REST APIs como interfaces norte. A su vez, se logró desarrollar una aplicación de negocio que logra interactuar con la infraestructura de red a través del controlador dando nuevas funcionalidades.

Se realizó un refactor de la funcionalidad de Call Admission Control con el objetivo de mejorar la performance en términos de uso de recursos de la interfaz ya que la primera versión generaba, durante el proceso de *setup* de la llamada, tráfico RTP innecesario.

Durante el desarrollo del proyecto surgieron problemas vinculados a la necesidad de manejar eventos en tiempo real que no fueron considerados inicialmente. El uso de *threads* permitió manejar estas situaciones y obtener los resultados deseados. No obstante, es una complejidad que debe ser tenida en cuenta si se desea escalar la aplicación. En este caso, es importante destacar la ventaja de utilizar tanto interfaces asíncronas para el manejo de eventos, como los websockets, en conjunto con interfaces síncronas para realizar tareas específicas en el contexto de comunicación de microservicios para casos de uso de tiempo real como se explica en la sección 3.8 de este trabajo.

Las interfaces sur de la arquitectura SDN están en constante evolución y OpenFlow aún no se instala como un estándar de facto en la industria. Otras iniciativas como OpenConfig [41] compiten en este sentido. De todas maneras, lo interesante de una arquitectura que se centra en desacoplar componentes, hace que la modificación de una interfaz no conlleve a cambios radicales en las aplicaciones que la utilizan. Nuevamente, podría recurrirse a un patrón como *adapter* [17] para abstraer una clase de comunicación contra los elementos de red que soporte tanto OpenFlow como OpenConfig y, de esta manera, lograr abstraernos de la interfaz puntual que estemos utilizando.

Respecto a futuras líneas de trabajo y mejoras que podrían realizarse sobre este proyecto, podemos mencionar las siguientes.

Ha quedado pendiente realizar un análisis cuantitativo del overhead que la aplicación de Call Admission Control podría introducir en el tiempo de *setup* de las llamadas ya que debe ejecutarse código ante eventos de señalización SIP. Si bien durante las pruebas y simulaciones realizadas este tiempo aparenta ser despreciable, cuantificarlo podría aportar mayor claridad en términos del margen disponible al trabajar con la aplicación en distintos escenarios.

Por otro lado, una posible línea de trabajo futuro podría ser la de analizar el funcionamiento de la aplicación en escenarios multi-dominio o entre distintos sistemas autónomos como se observa en la Figura 41.

Deberían contemplarse, en estos escenarios, modificaciones como las que se listan a continuación.

- Comunicación con otros controladores utilizando interfaces este-oeste [46].
- Tablas de conversión para mantener un criterio de marcado de paquetes a través de distintos sistemas autónomos. Esto es de especial importancia cuando haya sistemas autónomos que no utilizan el mismo esquema de tags DSCP dentro de su red.

[18] Enlace con documentación de endpoints de topología: https://github.com/faucetsdn/ryu/blob/master/ryu/app/rest_topology.py.

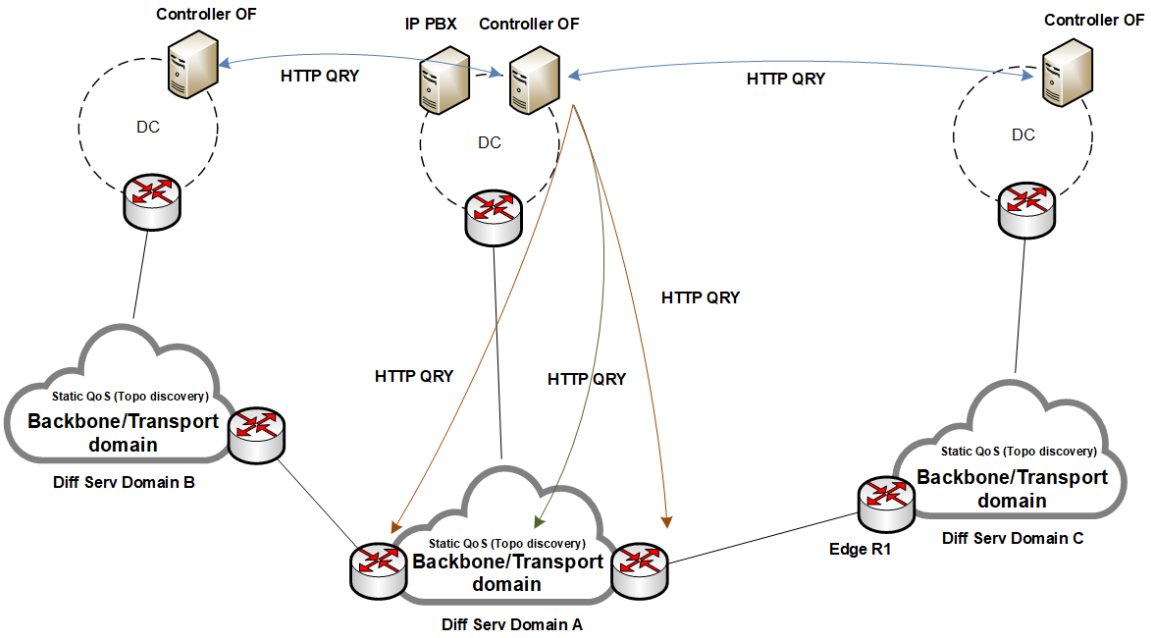


Figura 41: Aplicación en escenario con múltiples sistemas autónomos. (Fuente: Elaboración propia)

Además de los casos mencionados anteriormente, el mecanismo de Call Admission Control podría mejorarse de manera tal que tenga en cuenta el efecto de negociación dinámica de codecs. Este trabajo se focaliza en un escenario determinista en este aspecto ya que supone que el codec utilizado por los SIP *User-Agents* de la IP PBX es siempre el mismo (G711). Esta situación hace que el valor del umbral CAC sea un número estático ya que el ancho de banda requerido puede calcularse con la siguiente expresión.

$$BW_{interfaz} = BW_{G711} \times LC$$

Donde LC es la cantidad de llamadas concurrentes en la interfaz. El umbral de Call Admission Control, entonces, coincidiría con la cantidad de llamadas concurrentes deseadas, ya que el $BW_{interfaz}$ sería el ancho de banda reservado para la cola *EF*. Si, por el contrario, la negociación dinámica de codecs se encuentra habilitada, la expresión cambiaría de la siguiente forma.

$$BW_{interfaz} = \sum_{n=1}^{N} BW_i \times LC_i, \quad i \in \{G711, G729, ...\}$$

Utilizando la nueva expresión y adaptando el código que implementa el corte de las llamadas podría calcularse de forma dinámica el umbral. En este nuevo esquema, el umbral no sería algo configurable por el usuario, sino que se trabajaría de manera directa en conjunto con el ancho de banda reservado para la cola *EF* de la interfaz en cuestión.

Para finalizar, podemos destacar que la complejidad de este trabajo radica principalmente en la cantidad de interfaces y tecnologías que deben articularse, así como los aspectos de tiempo real que deben tenerse en cuenta para trabajar orientado a eventos de la central IP PBX. Además, el hecho de que la industria no tenga consenso mayoritario respecto de estándares para las interfaces sur hace que aún haya incertidumbre sobre que protocolos o APIs elegir a la hora de desarrollar una aplicación. No obstante, la versatilidad de la arquitectura propuesta por SDN abre un amplio espectro de nuevas posibilidades en lo que respecta al uso de las infraestructuras de red como la automatización y el uso, por parte de aplicaciones de negocio, de los recursos de red en tiempo real.

Bibliografía

[1] Asterisk. *Asterisk*. URL: https://www.asterisk.org/. (accessed: 27/10/2020).

[2] Asterisk. *Asterisk Bridges*. URL: https://wiki.asterisk.org/wiki/display/AST/Introduction+to+ARI+and+Bridges. (accessed: 03/10/2020).

[3] Asterisk. *Asterisk Channels API*. URL: https://wiki.asterisk.org/wiki/display/AST/Asterisk+12+Channels+REST+API. (accessed: 03/10/2020).

[4] Asterisk. *Asterisk REST Interface (ARI)*. URL: https://wiki.asterisk.org/wiki/pages/viewpage.action?pageId=29395573. (accessed: 27/10/2020).

[5] Tom Bäckström. *Speech Coding*. qos, packet loss, voip, speech, 2017. ISBN: 10.1007/978-3-319-50204-5_12. URL: https://www.researchgate.net/publication/315857445_Packet_Loss_and_Concealment.

[6] Teemu Koponen Ben Pfaff Justin Pettit. *The Design and Implementation of Open vSwitch*. VMWare, Awake Networks, 2015. ISBN: NA. URL: https://www.usenix.org/sites/default/files/conference/protected-files/nsdi15_slides_pfaff.pdf.

[7] Ben Cheng. *Distributed Version Control System — Git*. URL: https://medium.com/developer-notes/distributed-version-control-system-git-c11b6fe4ac98. (accessed: 27/10/2020).

[8] Cisco. *Call Admission Control in Call Manager*. URL: https://www.cisco.com/c/en/us/td/docs/voice_ip_comm/cucm/admin/11_5_1/sysConfig/CUCM_BK_SE5DAF88_00_cucm-system-configuration-guide-1151/CUCM_BK_SE5DAF88_00_cucm-system-configuration-guide-1151_chapter_011100.html. (accessed: 07/10/2020).

[9] Cisco. *Enterprise QoS Solution Reference Network Design Guide*. URL: https://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/WAN_and_MAN/QoS_SRND/QoS-SRND-Book/QoSIntro.html. (accessed: 27/10/2020).

[10] Web Development. *Representational state transfer, or 'REST' for short, is a less restrictive form of SOA than web services*. URL: https://www.drdobbs.com/web-development/soa-web-services-and-restful-systems/199902676. (accessed: 27/10/2020).

[11] Ryu Docs. *Application Programming Model*. URL: https://osrg.github.io/ryu-book/en/html/arch.html. (accessed: 27/10/2020).

[12] eweek. *DevOps image*. URL: https://www.eweek.com/enterprise-apps/best-practices-for-devops-in-the-cloud. (accessed: 27/10/2020).

[13] Christophe Fillot Fabien Devaux. *dynamips hypervisor*. URL: https://github.com/GNS3/dynamips. (accessed: 12/10/2020).

[14] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. UNIVERSITY OF CALIFORNIA, IRVINE, 2000. ISBN: NA. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[15] Christophe Fillot. *Dynagen*. URL: https://sourceforge.net/p/dyna-gen/wiki/Home/. (accessed: 12/10/2020).

[16] Flowgrammable. *OpenFlow actions stack*. URL: http://flowgrammable.org/sdn/openflow/actions/. (accessed: 27/10/2020).

[17] Geek For Geeks. *Adapter Pattern*. URL: https://www.geeksforgeeks.org/adapter-pattern/. (accessed: 26/10/2020).

[18] Kim Gene. *The DevOps Handbook. How to create world-class agility, reliability, security in technology organizations*. devops, metodologia, software, agiles, 2016. ISBN: 978-1942788003. URL: https://itrevolution.com/the-phoenix-project/.

[19] Kim Gene. *The Phoenix Project. A novel about IT, DevOps and Helping your Business Win*. devops, metodologia, software, agiles, 2013. ISBN: 978-1942788294. URL: https://itrevolution.com/the-phoenix-project/.

[20] Oscar Gerometta. *Introducción a Quality of Service (version 1.0)*. qos, diffserv, networks, 2015. ISBN: 978-987-05-9337-9. URL: https://www.usenix.org/sites/default/files/conference/protected-files/nsdi15_slides_pfaff.pdf.

[21] Git. *Git SCM*. URL: https://git-scm.com/. (accessed: 27/10/2020).

[22] GNS3. *GNS3*. URL: https://gns3.com/. (accessed: 12/10/2020).

[23] IETF. *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers (RFC 2474)*. URL: https://tools.ietf.org/html/rfc2474. (accessed: 12/10/2020).

[24] IETF. *INTERNET PROTOCOL (RFC 791)*. URL: https://tools.ietf.org/html/rfc791. (accessed: 12/10/2020).

[25] IETF. *Type of Service in the Internet Protocol Suite (RFC 1349)*. URL: https://tools.ietf.org/html/rfc1349. (accessed: 12/10/2020).

[26] Docker Inc. *Swarm Mode Overview*. URL: https://docs.docker.com/engine/swarm/. (accessed: 21/10/2020).

[27] Docker Inc. *Understanding docker networking drivers and their usecases*. URL: https://www.docker.com/blog/understanding-docker-networking-drivers-use-cases/. (accessed: 27/10/2020).

[28] Docker Inc. *Use swarm mode routing mesh*. URL: https://docs.docker.com/engine/swarm/ingress/. (accessed: 27/10/2020).

[29] Deolindo Zanuttini Joaquín Gonzalez. "Using Software Defined Networking for Call Admission Control and VoIP applications". En: *IEEE Xplore* (2018). DOI: https://doi.org/10.1109/CACIDI.2018.8584371.

[30] Bruce A. Mah Jon Dugan Seth Elliott. *IPerf3 traffic simulator*. URL: https://iperf.fr/. (accessed: 12/10/2020).

[31] Kspviswa. *OpenFlow Roadmap*. URL: https://kspviswa.github.io/OpenFlow_Version_Roadmap.html. (accessed: 27/10/2020).

[32] D. Marschke. *SDN: Anatomy of OpenFlow*. Lulu, 2015. ISBN: 978-1-4834-2723-2. URL: https://www.amazon.com/Software-Defined-Networking-SDN-OpenFlow/dp/1483427234.

[33] Medium. *Docker Containers vs Virtual Machines*. URL: https://medium.com/better-programming/docker-containers-vs-virtual-machines-838022906016. (accessed: 27/10/2020).

[34] Microsoft. *Call Admission Control in Skype for Business*. URL: https://docs.microsoft.com/en-us/skypeforbusiness/deploy/deploy-enterprise-voice/deploy-call-admission-control. (accessed: 07/10/2020).

[35] Microsoft. *Estilo de arquitectura de microservicios*. URL: https://docs.microsoft.com/es-es/azure/architecture/guide/architecture-styles/microservices. (accessed: 13/10/2020).

[36] NA. *Docker*. URL: https://docs.docker.com/. (accessed: 02/10/2020).

[37] NA. *Introduction to Mininet*. URL: https://github.com/mininet/mininet/wiki/Introduction-to-Mininet#apilevels. (accessed: 01/10/2020).

[38] NA. *SIPp voip traffic simulator*. URL: http://sipp.sourceforge.net/. (accessed: 12/10/2020).

[39] Tom Anderson Nick McKeown. *OpenFlow: Enabling Innovation in Campus Networks*. ACM SIGCOMM Computer Communication Review, 2008. ISBN: NA. URL: http://ccr.sigcomm.org/online/files/p69-v38n2n-mckeown.pdf.

[40] ONF Onos. *OpenFlow Protocol*. URL: https://wiki.onosproject.org/display/ONOS/OpenFlow+1.5+Implementation. (accessed: 27/10/2020).

[41] OpenConfig. *Vendor-neutral, model-driven network management designed by users*. URL: https://www.openconfig.net/. (accessed: 26/10/2020).

[42] Python. *Python*. URL: https://www.python.org/. (accessed: 27/10/2020).

[43] Ryu. *QoS with Ryu SDN controller*. URL: https://osrg.github.io/ryu-book/en/html/rest_qos.html. (accessed: 21/10/2020).

[44] Ryu. *Ryu Framework REST API documentation*. URL: https://ryu.readthedocs.io/en/latest/app/ofctl_rest.html. (accessed: 17/10/2020).

[45] Qemu Software. *What is QEMU*. URL: https://www.qemu.org/. (accessed: 12/10/2020).

[46] Coroller Stevan. *Inter-controller communication in switching SDN architecture*. URL: https://wiki.minet.net/_media/wiki/sdn/inter-controller_communication_in_switching_sdn_architecture.pdf. (accessed: 26/10/2020).

[47] Wikipedia. *Open vSwitch*. URL: https://en.wikipedia.org/wiki/Open_vSwitch#cite_note-ibm-developerworks-3. (accessed: 27/10/2020).

A. Instalación e implementación

Clonar repositorio

```
$ git clone git@github.com:joagonzalez/sdn-qos.git
$ mkproject sdn-qos
$ workon sdn-qos
$ pip install -r requirements.txt en cada servicio python
$ npm install en cada servicio node.js
```

Construir image docker de servicios de la aplicación

```
# para cada servicio
docker build -t <service> .
```

Implementar servicio

```
# para cada servicio
docker service create --network sdn-qos --publish <PORT-EXT:PORT-INT> --name <SERVICE>
```

Ejecutar simulación Call Admission Control

```
cd application
docker-compose -f docker-compose-simulation.yml up -d
docker ps
Abrir enlace http://localhost:3000
```

Ejecutar simulación QoS Inicializar controlador SDN Ryu

```
workon ryu
python3 ./bin/ryu-manager --observe-links ryu.app.rest_topology \
ryu.app.ws_topology ryu.app.ofctl_rest ryu.app.qos_simple_switch_13_CAC \
ryu.app.qos_simple_switch_rest_13_CAC ryu.app.rest_conf_switch \
ryu.app.rest_qos ryu.app.gui_topology.gui_topology
```

Inicializar simulación

```
cd application/mininet
sudo python2.7 sdn-qos-RealTimeQueues.py
```

Correr aplicación con infraestrucura de red externa

```
cd application
docker-compose -f docker-compose.yml up -d
docker ps
open http://localhost:3000
```