

UNIVERSIDAD DE VALLADOLID



E.T.S.I. TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

**Integración en OMNeT++ de módulos
desarrollados en Python: Aplicación a un
simulador de redes ópticas pasivas**

Autor:

Gorka Sainz-Ezquerria Calvo

Tutor:

**Dña. Noemí Merayo Álvarez
D. Ignacio de Miguel Jiménez**

TÍTULO: Integracion en OMNeT++ de módulos desarrollados en Python: Aplicación a un simulador de redes ópticas pasivas
AUTOR: Gorka Sainz-Ezquerria Calvo
TUTOR: Dña. Noemí Merayo Álvarez y D. Ignacio de Miguel Jiménez
DEPARTAMENTO: Teoría de la Señal y Comunicaciones e Ingeniería Telemática

TRIBUNAL

PRESIDENTE: Ignacio de Miguel Jiménez
SECRETARIO: Noemí Merayo Álvarez
VOCAL: Ramón J. Durán Barroso
SUPLENTE: J. Carlos Aguado Manzano
SUPLENTE: Evaristo J. Abril Domingo

FECHA:

CALIFICACIÓN:

Resumen

En este Trabajo Fin de Grado (TFG), se ha llevado a cabo un estudio de investigación centrado en la integración de Python en el entorno de simulación OMNeT++, originalmente basado en C++.

Para ello, se ha hecho uso de la biblioteca desarrollada por Marcos Modesini, Omnetpy. Dicha biblioteca está basada en PyBind11 y se encontraba inicialmente proporcionada a través de un contenedor Docker con una versión de OMNeT++ concreta.

Con el objetivo de permitir el uso de la biblioteca Omnetpy junto con diferentes versiones de OMNeT++, se ha desarrollado un proceso para extraer la biblioteca del contenedor e integrarla en otra versión de OMNeT++ distinta.

Por otra parte, se ha realizado un estudio del estado del arte de las redes 10G-EPON (*Ethernet Passive Optical Network*) a partir de un simulador de este tipo de redes desarrollado por el Grupo de Comunicaciones Ópticas de la Universidad de Valladolid.

El estudio se ha centrado en el análisis de diversos algoritmos de asignación dinámica de ancho de banda (*Dynamic Bandwidth Allocation*, DBA) entre los que destaca el uso de una red neuronal para controlar, de forma dinámica, los parámetros de sintonización de un controlador PID (*Proportional Integral Derivative*). Además, se ha añadido un módulo desarrollado en Python a dicho simulador para evaluar las posibilidades y limitaciones de Omnetpy.

Palabras clave

PON (Red Óptica Pasiva), OMNeT++, Python, Omnetpy, Docker, 10G-EPON, DBA (Asignación Dinámica de Ancho de Banda), PID, Red neuronal

Abstract

In this Final Degree Project, a research study has been carried out. It has focused on Python integration in OMNeT++ which is originally based on C++.

In order to achieve that objective, the library developed by Marcos Modenesi, Omnetpy, has been used. Said library is based in PyBind11 and initially found on a Docker container in a precise OMNeT++ version.

Chasing the goal of allowing Omnetpy run in different OMNeT++ versions, a process has been developed to extract the library from the container and integrate it into a different one.

On the other hand, a study of the state of the art of 10G-EPON (*Ethernet Passive Optical Network*) has been carried out using a simulator developed by the Optical Communications Group of Universidad de Valladolid.

The study focused on the analysis of various DBA (Dynamic Bandwidth Allocation) algorithms, among which the use of a neural network to dynamically control the tuning parameters of a PID (Proportional Integral Derivative) controller stands out. In addition, a module developed in Python has been added to the simulator to evaluate the possibilities and limitations of Omnetpy.

Keywords

PON (*Passive Optical Network*), OMNeT++, Python, Omnetpy, Docker, 10G-EPON, DBA (*Dynamic Bandwidth Allocation*), PID, Neural Network

Agradecimientos

A mi familia por apoyarme y ayudarme diariamente, permitiéndome dedicarle a mis estudios todo el tiempo que los mismos han requerido, lo cual, de otra forma, hubiera sido impensable.

A mi pareja, Inés, por animarme siempre y darme la energía y la determinación para seguir adelante.

A mis tutores, Noemí e Ignacio, por su inestimable ayuda, consejos y enseñanzas así como su dedicación a lo largo de toda la realización de este trabajo.

A mis amigos, Daniel, Álvaro y Sergio “Pichi”, por haber estado siempre dispuestos a ayudarme cuando lo he necesitado.

La investigación desarrollada en este Trabajo Fin de Grado ha sido financiada por el Ministerio de Ciencia, Innovación y Universidades en el marco del proyecto ONOFRE-2 (TEC2017-84423-C3-1- P) y la red de investigación Go2Edge (RED2018-102585-T), por la Consejería de Educación de la Junta de Castilla y León en el marco de los proyectos ROBIN (VA085G19) y ARTEMIS (VA231P20), y por el Fondo Europeo de Desarrollo Regional FEDER, tanto a través de ARTEMIS, como del proyecto DISRUPTIVE del Programa Interreg V-A España-Portugal (POCTEP) 2014-2020 (0667_DISRUPTIVE_2_E). Las opiniones son de exclusiva responsabilidad del autor que las emite.

Índice

Agradecimientos	vii
Índice	viii
Índice de figuras	xi
Índice de tablas	xiv
1 Introducción	1
1.1 Motivación.....	1
1.2 Objetivos.....	2
1.2.1 Objetivos Generales	2
1.2.2 Objetivos específicos	2
1.3 Fases y metodología	3
1.3.1 Fase de Análisis	3
1.3.2 Fase de Implementación	3
1.3.3 Fase de Pruebas.....	4
1.4 Estructura de la Memoria del TFG	4
2 Entorno de Trabajo	6
2.1 Introducción.....	6
2.2 OMNeT++	6
2.3 Omnetpy como vía de integración entre Python y OMNeT++	7
2.4 Docker.....	8
2.5 Conclusiones.....	9
3 Descripción del simulador de redes PON (<i>Passive Optical Networks</i>) en OMNeT++	10
3.1 Introducción.....	10
3.2 Redes de Acceso Ópticas Pasivas (PON, <i>Passive Optical Networks</i>).....	10

3.3	Simulador de redes EPON en OMNeT++	12
3.4	Conclusiones.....	15
4	Integración de Python en OMNeT++	16
4.1	Introducción.....	16
4.2	Integración de Python y OMNeT++ en un contenedor Docker.....	16
4.3	Extracción de Omnetpy del contenedor Docker. Utilización de la biblioteca con otras versiones de OMNeT++.....	18
4.3.1	Instalación de PyBind11	19
4.3.2	Instalación de Omnetpy	20
4.3.3	Instalación de la versión 5.6.2 de OMNeT++	21
4.3.4	Incorporación de Omnetpy a OMNeT++	22
4.4	Incorporación del simulador de redes ópticas pasivas EPON a OMNeT++	24
4.5	Conclusiones.....	26
5	Implementación, simulación y validación de algoritmos DBA en el simulador 10G-EPON	27
5.1	Introducción.....	27
5.2	Implementación del algoritmo de <i>polling</i> IPACT	28
5.2.1	Descripción del algoritmo IPACT	29
5.2.2	Configuración del escenario de simulación.....	30
5.3	Implementación del algoritmo de <i>polling</i> con PID.....	34
5.3.1	Descripción de un controlador PID	35
5.3.2	Descripción del algoritmo SPID (Service Level Agreement PID).....	36
5.3.3	Configuración del escenario de simulación.....	37
5.4	Implementación del algoritmo NN-SPID (<i>Neural Network-SPID</i>) basado en redes neuronales.....	41
5.4.1	Descripción de la red neuronal	42
5.4.2	Configuración del escenario de simulación.....	43
5.5	Conclusiones.....	53

6 Integración de un módulo programado en Python al simulador 10G-EPON de OMNeT++.....	55
6.1 Introducción.....	55
6.2 Incorporación del nuevo módulo Python al proyecto OMNeT++.....	55
6.2.1 Definición del fichero .NED.....	56
6.2.2 Definición del fichero .cc.....	57
6.2.3 Definición del fichero makefrag.....	58
6.2.4 Definición del fichero Python.....	59
6.3 Pruebas realizadas y limitaciones encontradas.....	59
6.3.1 Intercambio de mensajes con el nuevo módulo.....	60
6.3.2 Integración de parámetros a los mensajes.....	61
6.3.3 Envíos retardados.....	64
6.3.4 Solución de compromiso.....	64
6.4 Propuesta de integración final Python-OMNeT++.....	66
6.5 Conclusiones.....	68
7 Conclusiones y líneas futuras.....	69
7.1 Conclusiones.....	69
7.2 Líneas futuras.....	70
8 Bibliografía.....	71

Índice de figuras

Figura 1. Arquitectura de la red EPON simulada en OMNeT++.	13
Figura 2. Módulo OLT.	14
Figura 3. Módulo ONU.	14
Figura 4. Error Obtenido al intentar iniciar el IDE de OMNeT++ tras la instalación de la versión “6.0 preview 8”.	19
Figura 5. Rutas a los diferentes directorios y subdirectorios que componen el proyecto simulador PON y que han de incluirse en la sección <code>Paths and Symbols</code> .	25
Figura 6. Retardo medio frente a la carga de la ONU para el algoritmo IPACT con un SLA y una sola cola.	32
Figura 7. Tamaño medio de la cola en bytes frente a la carga de la ONU para el algoritmo IPACT con un SLA y una sola cola.	33
Figura 8. Retardo medio frente a la carga de la ONU para el algoritmo IPACT con un SLA y tres colas.	34
Figura 9. Ecuación que describe la variable de control de un PID.	36
Figura 10. Diagrama de bloques que ilustra el proceso de funcionamiento del PID en la asignación dinámica de ancho de banda en redes PON.	37
Figura 11. Ecuación empleada durante la inicialización de los valores de ancho de banda máximo asignados a cada ONU.	37
Figura 12. Evolución en tiempo real del ancho de banda medio asignado para una ONU perteneciente al SLA ₀ con un algoritmo de <i>polling</i> controlado mediante un PID.	40
Figura 13. Cantidad media de ancho de banda asignado frente al tiempo para una ONU perteneciente al SLA ₀ con un algoritmo de <i>polling</i> controlado mediante un PID. <i>Zoom</i> de los primeros 60 segundos.	41
Figura 14. Diagrama de flujo de la red neuronal empleada para controlar los parámetros de sintonización del PID.	43
Figura 15. Evolución en tiempo real del ancho de banda garantizado para una ONU perteneciente a cada uno de los tres SLAs con el algoritmo de <i>polling</i> NN-SPID en el escenario 1.	46
Figura 16. Evolución en tiempo real del ancho de banda garantizado para una ONU perteneciente a cada uno de los tres SLAs con el algoritmo de <i>polling</i> NN-SPID en el escenario 1. (<i>Zoom</i> 60 s).	47

Figura 17. Evolución temporal del parámetro K_p para NN-SPID bajo el escenario 1.	48
Figura 18. Evolución temporal del parámetro T_i para NN-SPID bajo el escenario 1.	48
Figura 19. Evolución temporal del parámetro T_d para NN-SPID bajo el escenario 1.	49
Figura 20. Evolución en tiempo real del ancho de banda garantizado para una ONU perteneciente a cada uno de los tres SLAs con el algoritmo NN-SPID para el escenario 2.	50
Figura 21. Evolución en tiempo real del ancho de banda garantizado para una ONU perteneciente a cada uno de los tres SLAs con el algoritmo NN-SPID para el escenario 2 (<i>Zoom</i> 60 s).	51
Figura 22. Evolución temporal del parámetro K_p para NN-SPID bajo el escenario 2.	52
Figura 23. Evolución temporal del parámetro T_i para NN-SPID bajo el escenario 2.	52
Figura 24. Evolución temporal del parámetro T_d para NN-SPID bajo el escenario 2.	53
Figura 25. Código del fichero “.ned” del nuevo módulo de prueba.	56
Figura 26. Líneas de código añadidas a la sección “gates:” del fichero MAC_OLT.ned para incorporar dos nuevas puertas al módulo MAC_OLT.	57
Figura 27. Líneas de código añadidas a la sección “connections:” del fichero OLT.ned para reflejar la conexión bidireccional entre el nuevo módulo de prueba y el módulo MAC_OLT.	57
Figura 28. Fichero OLT.ned visualizado en modo <i>Design</i> para mostrar la estructura interna del módulo OLT.	57
Figura 29. Código del fichero “.cc” del nuevo módulo.	58
Figura 30. Código del fichero “makefrag”.	58
Figura 31. Código del fichero de extensión “.py” inicial cuya única función es notificar el momento de su inicialización.	59
Figura 32. Código del fichero de extensión “.py” que envía mensajes hacia el módulo MAC_OLT.	60
Figura 33. Código del fichero de extensión “.py” en el que se define el nuevo tipo de mensaje extendiendo de la clase cPacket.	61
Figura 34. Código del fichero _cmessage.py localizado en ~/Escritorio/omnetpy-master/omnetpy/bindings/pyopp.	63
Figura 35. Código necesario para la incorporación de variables al nombre del mensaje de la clase cMessage en el módulo MAC_OLT.cc.	65
Figura 36. Código necesario para la extracción de variables del nombre del mensaje de la clase cMessage en el módulo Python.	65

Figura 37. Código necesario para la incorporación de variables al nombre del mensaje de la clase <code>cMessage</code> en el módulo <code>Python</code>	65
Figura 38. Código necesario para la extracción de variables del nombre del mensaje de la clase <code>cMessage</code> en el módulo <code>MAC_OLT.cc</code>	65
Figura 39. Fichero <code>SFNet_small2.ned</code> visualizado en modo <i>Design</i> para mostrar la estructura de la red tras la incorporación del nuevo módulo <code>Python</code>	67

Índice de tablas

Tabla 1. Parámetros empleados en las simulaciones del algoritmo de <i>polling</i> IPACT	31
Tabla 2. Parámetros empleados en las simulaciones de un algoritmo de <i>polling</i> con un PID	39
Tabla 3. Parámetros empleados en las simulaciones de un algoritmo de <i>polling</i> con un controlador PID y una red neuronal.....	45

1

Introducción

1.1 Motivación

La cantidad de recursos de red que demandan los usuarios sigue creciendo cada día por lo que la fibra óptica, por ser la mejor solución a la hora de satisfacer dicha demanda, se sigue extendiendo cada vez a más hogares.

La fibra se extiende a todos los tramos de la red. En el segmento de acceso se localizan las redes que proporcionan ancho de banda a los usuarios finales. En este tipo de redes destacan las redes PON (*Passive Optical Network*). Dichas redes han sufrido una gran evolución para poder satisfacer la demanda actual.

La generación de redes PON encargada de suceder a las redes PON (*Gigabit PON*) tradicionales se conocen como redes 10G-PON, las cuales soportan tasas de transmisión de hasta 10 Gbit/s.

El entorno de simulación empleado durante estos años por el Grupo de Comunicaciones Ópticas de la Universidad de Valladolid para el desarrollo de diversos simuladores de redes ópticas es OMNeT++. Dicho entorno está basado en el lenguaje de programación C++.

El objetivo principal que se persigue en este trabajo es la integración de módulos programados en lenguaje Python en el entorno de simulación OMNeT++. Esto se debe a que Python ofrece una gran cantidad de bibliotecas de utilidad en este ámbito como pueden serlo aquellas asociadas al aprendizaje automático.

Para ello, se llevará a cabo un análisis de la biblioteca Omnetpy, desarrollada por Marcos Modenesi, ya que ésta podría ser una herramienta que permita la ya mencionada integración entre módulos programados en Python y el entorno OMNeT++.

Por otra parte, se pretende hacer un estudio sobre el comportamiento de una red 10G-EPON (*Ethernet* PON) a través de un simulador desarrollado en OMNeT++. Esta parte del trabajo se centrará en la aplicación de diversos algoritmos de asignación dinámica de ancho de banda sobre diferentes escenarios de red en el simulador desarrollado 10G-EPON.

1.2 Objetivos

1.2.1 Objetivos Generales

Como ya se ha adelantado en la sección de motivación, los objetivos generales de este trabajo son dos.

En primer lugar, se realizará un estudio centrado en analizar la forma de integrar módulos desarrollados en el lenguaje Python en el entorno OMNeT++. Se hará, para ello, especial hincapié en el estudio de la biblioteca Omnetpy como posible forma de lograr la consecución de este objetivo.

En segundo lugar, se pretende analizar las prestaciones de diferentes algoritmos en una red 10G-EPON mediante el proyecto desarrollado en OMNeT++. Con ello, se estudiará el comportamiento de la red 10G-EPON centrandolo estudio en diferentes algoritmos de asignación dinámica de ancho de banda implementados en el simulador. Además, se ha añadido un módulo desarrollado en Python a dicho simulador para evaluar las posibilidades y limitaciones de Omnetpy.

1.2.2 Objetivos específicos

Se describen, a continuación, los objetivos específicos necesarios para la consecución de los objetivos generales ya comentados.

1. Estudiar la viabilidad de Omnetpy a la hora de integrar módulos programados en Python en el entorno OMNeT++.

2. Desarrollar un procedimiento para instalar y utilizar la biblioteca Omnetpy en diferentes versiones de OMNeT++ puesto que, actualmente, ésta se encuentra en un contenedor Docker ligada a una versión concreta de OMNeT++ (Omnet 6.0pre8).
3. Desarrollar un módulo básico en Python y comunicarlo con otros módulos programados en C++ dentro de OMNeT++.
4. Actualizar el simulador de redes 10G-EPON desarrollado previamente por el Grupo de Comunicaciones Ópticas de la Universidad de Valladolid para que pueda utilizarse en la última versión estable de OMNeT++ (versión 5.6.2).

1.3 Fases y metodología

Se procede a detallar las fases seguidas y la metodología llevada a cabo en cada una de ellas durante la realización de este trabajo.

1.3.1 Fase de Análisis

En esta primera fase se pretende adquirir los conocimientos necesarios para un desarrollo adecuado de este Trabajo Fin de Grado:

- Estudio de los principales aspectos relacionados con la inteligencia artificial
- Repaso de la topología y principales características de una red PON
- Estudio del funcionamiento de la herramienta de virtualización Docker
- Documentación y estudio de la biblioteca Omnetpy
- Familiarización con los lenguajes de programación C++ y Python
- Análisis del simulador de redes PON desarrollado en OMNeT++

1.3.2 Fase de Implementación

Durante esta fase se llevará a cabo la instalación de la biblioteca Omnetpy [1] en OMNeT++ fuera del contenedor Docker así como su integración en el simulador de redes

PON.

Una vez instalada la biblioteca, se procederá con la integración de un módulo programado en Python en el proyecto. Para ello, se utilizará como guía los pasos indicados en [2] en los que se ilustra la creación de un módulo de ejemplo programado en Python.

1.3.3 Fase de Pruebas

En esta última fase, se llevarán a cabo diversas pruebas sobre el nuevo módulo desarrollado en Python e integrado en la red del simulador con el objetivo de comprobar la operabilidad que presenta la biblioteca Omnetpy a día de hoy.

Paralelamente, se realizarán pruebas, sobre diferentes algoritmos de asignación dinámica de ancho de banda en diversos escenarios de red dentro del simulador de redes PON, escalado para que su tasa de transmisión total llegue a los 10 Gbit/s, esto es, bajo tecnologías 10G-PON.

1.4 Estructura de la Memoria del TFG

El Capítulo 2 describe las herramientas de trabajo que serán empleadas a lo largo de todo el trabajo.

En el Capítulo 3 se comienza describiendo las principales características de las redes ópticas pasivas para proseguir, a continuación, con una explicación del simulador de redes de este tipo desarrollado en OMNeT++ por el Grupo de Comunicaciones Ópticas de la Universidad de Valladolid.

En el Capítulo 4 se indican los pasos a seguir con el objetivo de integrar Python en OMNeT++ haciendo uso, para ello, de la biblioteca Omnetpy que se encuentra, inicialmente, proporcionada en un contenedor Docker. Se detallarán, además, los pasos seguidos a la hora de extraer dicha biblioteca del contenedor Docker permitiendo, así, su utilización junto con diferentes versiones de OMNeT++.

En el Capítulo 5 se analizarán los diferentes escenarios de red y algoritmos simulados en OMNeT++ en relación con la asignación dinámica de ancho de banda en redes 10G-EPON.

El Capítulo 6 relata todas las pruebas llevadas a cabo sobre el módulo programado en Python que se pretende integrar al proyecto del simulador de redes PON en OMNeT++. También se incluye en este capítulo una descripción detallada de las limitaciones encontradas durante las pruebas realizadas, así como de los pasos seguidos para su obtención.

El Capítulo 7 recoge las conclusiones generales de este Trabajo Fin de Grado y las líneas futuras que podrían suceder a este estudio.

Por último, en el Capítulo 8 se encuentran las referencias bibliográficas que han servido de apoyo para la realización de este trabajo.

2

Entorno de Trabajo

2.1 Introducción

Este capítulo recoge información sobre las herramientas empleadas a lo largo de la realización de este trabajo.

En primer lugar, se procede a describir las principales características de OMNeT++ que es la biblioteca de simulación y entorno de trabajo empleados en este estudio para la simulación de la red de acceso óptica pasiva.

A continuación, se recalca el interés por integrar módulos programados en lenguaje Python [3] en OMNeT++ [4] así como la utilización de la biblioteca Omnetpy [1] como herramienta para lograr este objetivo. Se prosigue, entonces, con algunos comentarios que resumen los rasgos de dicha biblioteca.

El capítulo concluye con una breve descripción de Docker [5] puesto que, inicialmente, Omnetpy se proporciona a través de un contenedor Docker junto con una versión concreta de OMNeT++ (OMNeT++ 6.0 Preview 8).

2.2 OMNeT++

OMNeT++ es una biblioteca de simulación de redes de eventos discretos basada en el lenguaje de programación C++. Entiéndase por eventos discretos que la red únicamente cambia de estado en instantes discretos de tiempo y que no ocurre nada entre dos eventos discretos consecutivos (cuyo tiempo de realización dentro de la simulación es nulo).

OMNeT++ ofrece un entorno de desarrollo basado en Eclipse así como una interfaz gráfica de simulación desde la que poder realizar un seguimiento de las ejecuciones que se lleven a cabo.

Las redes creadas en OMNeT++ están formadas por módulos interconectados, típicamente, a través de enlaces físicos los cuales poseen puertas de entrada y salida en cada módulo y las conexiones que se realicen entre ellas, o bien, en algunas ocasiones, a través de enlaces directos que únicamente requieren que el módulo receptor disponga de una puerta de entrada directa.

Mientras que los módulos son programados en lenguaje C++, éstos son ensamblados a otros módulos y componentes más grandes mediante el lenguaje de alto nivel, NED (*Network Description*).

Los módulos que componen la red se comunican intercambiando mensajes o paquetes entre ellos. Éstos pueden simular tramas o paquetes de diversos protocolos de comunicaciones usados en redes de datos tales como Ethernet.

2.3 Omnetpy como vía de integración entre Python y OMNeT++

En Python se han desarrollado numerosas bibliotecas cuyas aplicaciones podrían resultar de interés en lo referente al desarrollo de algoritmos de asignación dinámica de ancho de banda en el simulador de redes PON que se está usando en este TFG, como pueden serlo aquellas asociadas al aprendizaje automático, redes neuronales, etc.

Por ello, surge la necesidad de investigar la posibilidad de integrar Python en OMNeT++, permitiendo, así, la programación de módulos en este lenguaje y el acceso a las bibliotecas que éste ofrece.

Una posible vía de integración de Python en OMNeT++ recae en la biblioteca Omnetpy [1]. Esta biblioteca, desarrollada por Marcos Modenesi, permite la programación de módulos en lenguaje Python de manera sencilla.

Para funcionar, Omnetpy extiende el intérprete de Python con clases de OMNeT++ con lo que permite que éste pueda ejecutar código escrito en C++ desde Python. Por otra parte, embebe el intérprete de Python en OMNeT++ con lo que permite que OMNeT++ pueda ejecutar código programado en Python cuando lo requiera.

Omnetpy está basado en PyBind11 [6]. PyBind11 es una biblioteca que permite crear conexiones entre código C++ ya existente y código Python que programe el usuario.

El grado de madurez que presenta la biblioteca Omnetpy es bajo puesto que hay funcionalidades básicas de OMNeT++ que no han sido aún implementadas y que resultan, por tanto, limitantes a la hora de desarrollar módulos programados en Python.

Omnetpy se proporciona en un contenedor Docker junto con la versión de OMNeT++ 6.0 Preview 8.

2.4 Docker

Docker es una plataforma abierta para desarrollar y ejecutar aplicaciones permitiendo independizar las mismas del sistema operativo de la máquina desde la que se ejecuten dichas aplicaciones. [5]

Esto se consigue a través de contenedores que son entornos aislados en los que se empaquetan y ejecutan las aplicaciones. Estos contenedores poseen también instaladas todas las dependencias necesarias para que funcionen correctamente las aplicaciones evitando tener que depender, así, de que éstas se encontrasen instaladas en la máquina *host*. Con ello, además, se tiene la certeza de que todo el que trabaje con un mismo contenedor está trabajando exactamente bajo las mismas condiciones que el resto.

El uso de Docker pasa por la creación y manipulación de objetos entre los que se encuentran las imágenes, los contenedores, los volúmenes...

Una imagen Docker es una plantilla (únicamente de lectura) con instrucciones para crear un contenedor Docker. Una imagen puede estar basada en otra a la que se ha añadido ciertas modificaciones adicionales. Es posible crear una imagen propia a través

de un fichero Dockerfile en el que se tienen que definir los pasos necesarios para crear la imagen y ejecutarla.

Un contenedor Docker es una instancia ejecutable de una imagen. Por ello, éste viene definido por su imagen, así como cualquiera de las opciones de configuración que se le proporcionen a la hora de crearlo y ejecutarlo.

Un contenedor Docker es un proceso más en la máquina en la que se esté ejecutando, el cual ha sido aislado del resto de procesos de la máquina *host*. La imagen asociada a dicho contenedor le proporciona un sistema de ficheros aislado, así como su configuración, variables de entorno, etc.

2.5 Conclusiones

En este primer capítulo de la memoria se ha establecido el marco de trabajo sobre el que se va a desarrollar el resto del estudio.

Para ello, se ha llevado a cabo, en primer lugar, una descripción superficial del entorno de simulación de redes de eventos discretos, OMNeT++, puesto que en él se realizarán la totalidad de pruebas y simulaciones recogidas en este trabajo.

Por otra parte, se ha escogido Omnetpy, biblioteca basada en PyBind11, como una vía para lograr el objetivo de integrar Python en OMNeT++.

Dicha biblioteca está proporcionada inicialmente en un contenedor Docker por lo que se ha incluido, también, en este capítulo, una introducción de su funcionamiento básico.

3

Descripción del simulador de redes PON (*Passive Optical Networks*) en OMNeT++

3.1 Introducción

En este capítulo, se va a comenzar describiendo, de forma breve, las principales características de las redes de acceso ópticas pasivas ya que en ellas se basan el estudio y las simulaciones que se van a ir desarrollando en este trabajo.

Por otra parte, se detallarán los aspectos principales de la arquitectura de la red EPON desarrollada en el simulador OMNeT++, describiendo, para ello, la topología de la red y la estructura que presentan los módulos que simularán el OLT y cada una de las ONUs.

3.2 Redes de Acceso Ópticas Pasivas (PON, *Passive Optical Networks*)

Una Red Óptica Pasiva o PON (*Passive Optical Network*) es un tipo de red de acceso que está formada únicamente por elementos ópticos pasivos entre el operador/proveedor de servicios y el cliente o abonado, interconectados mediante fibra óptica. Dichos elementos, entre los que destacan los divisores ópticos o *splitters*, reciben este nombre debido a que no requieren alimentación para su funcionamiento [7].

Las redes PON presentan una topología en árbol en las que se comunica el Terminal de Línea óptico u OLT (*Optical Line Terminal*), que se encuentra en la oficina

central del proveedor de servicios, con las Unidades de Red Ópticas u ONUs (*Optical Network Units*) a través de un *splitter* que se encarga de dividir la potencia de la señal proveniente del OLT entre cada una de las ramas que desembocan en las ONUs que forman parte de la red.

En el canal descendente (desde el OLT hacia las ONUs), la red es punto-multipunto y el OLT se encarga de transmitir información hacia las ONUs. Para ello, el *splitter* reparte dicha información entre todas las ONUs, es decir, la transmisión es de tipo difusión o *broadcast*. Es ésta última la encargada de filtrar y recibir, únicamente, el contenido destinado a ella. El OLT hace uso de Multiplexación por División en Tiempo o TDM (*Time Division Multiplexing*) para enviar la información dirigida a cada ONU en distintos instantes de tiempo. La longitud de onda empleada en este sentido es de 1490 nm (nanómetros).

Por su parte, en el canal ascendente (desde cada ONU hacia el OLT), la red es punto a punto y, en ella, la ONU transmite hacia el OLT para lo cual es necesario un mecanismo que evite las situaciones de contienda que se pudieran producir a causa del envío simultáneo desde diferentes ONUs hacia el OLT. Se emplea, en este caso, Acceso Múltiple por División en Tiempo o TDMA (*Time Division Multiple Access*). La longitud de onda usada en el canal ascendente es de 1310 nm.

En la red se utilizan algoritmos de asignación dinámica de ancho de banda o DBA (*Dynamic Bandwidth Allocation*). Dichos algoritmos son gestionados desde el OLT que es el encargado de proporcionar el ancho de banda a las ONUs tras cada ciclo de tiempo, por ejemplo, 2 ms es el máximo tiempo de ciclo establecido en el estándar EPON. Se adapta, con ello, la cantidad de ancho de banda asignada a cada ONU según diversos factores tales como la demanda en cada momento, el tráfico que se encuentra circulando en la red o los requisitos de calidad de servicio (QoS, *Quality of Service*) que hayan sido contratados por cada abonado con su proveedor de servicios.

Dentro de los algoritmos de asignación dinámica de ancho de banda, se puede realizar una clasificación en dos tipos de algoritmos. Por una parte, se encuentran los algoritmos centralizados. Se encuentran en este grupo aquellos algoritmos que asignan el ancho de banda a cada ONU al final de cada ciclo (cuando llega el mensaje *Report* de la última ONU de la red) tras conocer las demandas de ancho de banda de cada una de ellas.

Por otro lado, se presentan los algoritmos de *polling*. Este tipo de algoritmos se caracterizan por asignar el ancho de banda a cada ONU de forma independiente a través de un mensaje *Gate* tras conocer la demanda de dicha ONU mediante el mensaje de tipo *Report* recibido de esa ONU por lo que, en este caso, el algoritmo no espera a que termine el ciclo para conocer la demanda de todos los usuarios. Durante las simulaciones posteriores realizadas en este trabajo, se hará uso de algoritmos desarrollados bajo la técnica de *polling*.

3.3 Simulador de redes EPON en OMNeT++

La red EPON simulada presenta una topología en árbol en la que se interconecta el Terminal de Línea Óptico (OLT) que se encuentra en la oficina central del proveedor de servicios con 16 Unidades de Red Ópticas (ONUs) que están dentro o cerca de las dependencias del abonado. Entre el OLT y las ONUs se encuentra un *splitter* encargado de replicar la información que le llega desde el OLT hacia todas las ONUs en el sentido descendente de la comunicación y de combinar las señales de cada una de las ONUs hacia el OLT en el sentido ascendente.

Se simula una situación de Fibra hasta el hogar o FTTH (*Fiber To The Home*) en la que la fibra óptica llega hasta la vivienda del usuario final.

En la Figura 1 se puede observar la arquitectura de la red empleada en las diversas simulaciones de este trabajo.

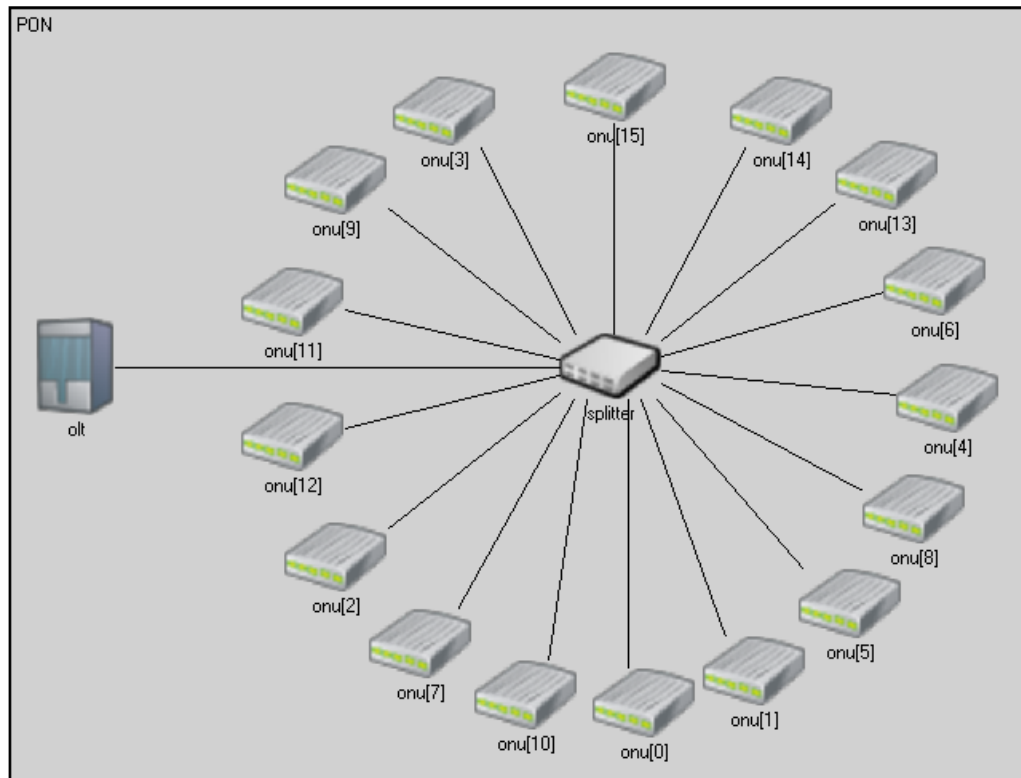


Figura 1. Arquitectura de la red EPON simulada en OMNeT++.

Por otra parte, en las Figuras 2 y 3 se presenta la estructura interna de los módulos que componen el OLT y cada ONU, respectivamente.

El módulo OLT (Figura 2) está compuesto por el submódulo *olt_mac* que simula la subcapa de control de acceso al medio (capa MAC), el submódulo *olt_wdm* encargado de separar los canales de bajada o *downstream* (hacia las ONUs) del canal de subida o *upstream* (hacia el OLT) y el submódulo *olt_table* en el que se van almacenando en una tabla los datos actualizados del ancho de banda demandado por cada ONU y el ancho de banda asignado a cada una de ellas en cada ciclo.

Este módulo contenía también, previamente, dos submódulos adicionales (*olt_rx* y *olt_tx*) que fueron suprimidos y sus funciones fueron relegadas al submódulo *olt_mac* con el objetivo de agilizar las simulaciones.

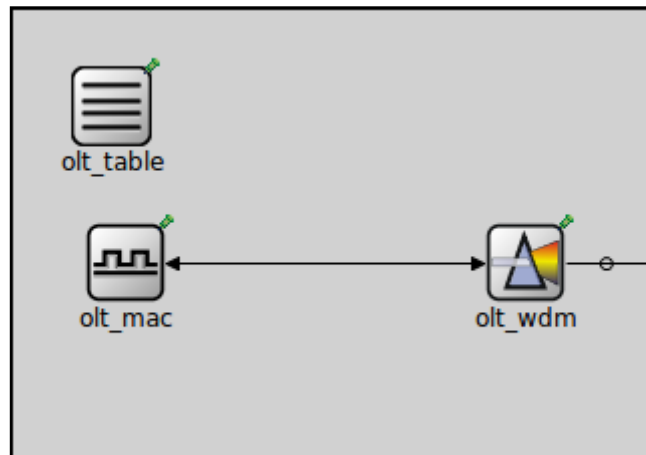


Figura 2. Módulo OLT.

El módulo ONU (Figura 3) está formado por el submódulo *onu_wdm* que tiene funciones equivalentes a las del submódulo *olt_wdm* en el OLT, el submódulo *onu_mac* que simula la capa MAC en la ONU, el submódulo *onu_source* encargado de generar tanto las tramas Ethernet como los paquetes Report, el submódulo *onu_queue* en el que se simula un sistema de colas en el que se van insertando los paquetes procedentes del submódulo *onu_source* y el submódulo *onu_table* en el que se van almacenando en una tabla los datos del ancho de banda asignado a cada ONU y el tiempo en el que ésta puede empezar a transmitir en el siguiente ciclo. Este módulo contenía también, previamente, dos submódulos adicionales (*onu_rx* y *onu_ptp*) que fueron suprimidos y sus funciones fueron relegadas al submódulo *onu_mac* con el objetivo de agilizar las simulaciones.

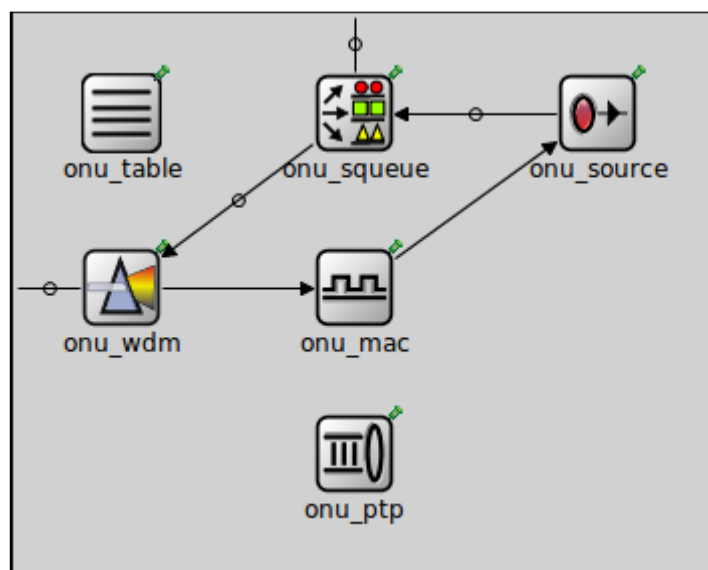


Figura 3. Módulo ONU

Una descripción más exhaustiva del simulador se encuentra en un Proyecto Fin de Carrera anterior realizado por Jose María Robledo Sáez [8] que, debido a su antigüedad, no se encuentra disponible en el repositorio de la Universidad de Valladolid.

3.4 Conclusiones

En este capítulo de la memoria se ha llevado a cabo una descripción de las redes de acceso ópticas pasivas ya que son el tipo de redes que se van a simular en secciones posteriores empleando, para ello, OMNeT++.

Después, se ha procedido con una explicación de la arquitectura y principales características del simulador de redes ópticas pasivas del que se va a hacer uso en este trabajo. Cabe destacar que, dicho simulador ha sufrido algunos cambios en la distribución de sus módulos con respecto a estudios previos. Todo ello con la finalidad de agilizar las simulaciones.

4

Integración de Python en OMNeT++

4.1 Introducción

En este capítulo de la memoria, se procederá a describir los pasos realizados para poder integrar módulos programados en lenguaje Python en OMNeT++ haciendo uso, para ello, de la biblioteca programada por Marcos Modenesi, Omnetpy.

El capítulo comenzará con un seguimiento, en detalle, del proceso de integración de Python en OMNeT++ a través de la biblioteca Omnetpy en un contenedor Docker.

Posteriormente, se describirán los pasos seguidos para poder extraer la biblioteca Omnetpy del contenedor con el objetivo de poder hacer uso de ella en otras versiones de OMNeT++. En este trabajo, se integrará Omnetpy en la versión 5.6.2 de OMNeT++ que, a día de hoy es la última versión estable.

Por último, el capítulo concluirá con la explicación de los pasos a seguir a la hora de importar el proyecto simulador de redes ópticas pasivas en OMNeT++. También se facilitarán, entonces, las consideraciones más importantes a tener en cuenta para lograr un correcto funcionamiento del simulador.

4.2 Integración de Python y OMNeT++ en un contenedor Docker

El primer paso para la consecución de este objetivo reside en la instalación de Docker. En este trabajo, se instala Docker en una máquina virtual con el sistema operativo Linux Mint 20. Para ello, siguiendo los pasos que se indican en [9], se procede a ejecutar los siguientes comandos en el terminal:

```
sudo apt-get update
```

```
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Tras esto, en [1], se detalla que la siguiente acción a tomar es lanzar el contenedor que se proporciona con la versión 6.0 Preview 8 de OMNeT++ junto con la biblioteca Omnetpy instalados en él mediante la ejecución del comando:

```
docker run --rm \  
  
-ti -e DISPLAY=$DISPLAY \  
  
-v /tmp/.X11-unix:/tmp/.X11-unix \  
  
mmodenesi/omnetpy bash
```

Nótese que los caracteres *backslash* (“\”) son empleados para indicar que el comando sigue en líneas sucesivas al terminal *bash* por lo que, si estos pasos son ejecutados en otro terminal diferente como, por ejemplo, Windows PowerShell, dicho carácter podría cambiar (en este último caso cambiaría al carácter de acento grave “`”).

Nótese, también, que el *flag* “--rm” se le está indicando al contenedor que se destruya automáticamente una vez que se cierre por lo que, en ese caso, será imposible conseguir persistencia en los datos dentro del contenedor. Basta con eliminar dicha opción del comando anterior si la persistencia en los datos fuese requerida.

Con ello, el paso restante es, simplemente, ejecutar el comando que lanza el IDE (*Integrated Development Environment*) de OMNeT++ en el terminal del contenedor:

```
omnetpp
```

Para acceder, en futuras ocasiones, al contenedor Docker en el que se encuentra la biblioteca Omnetpy junto con la versión 6.0 Preview 8 de OMNeT++, no hay que seguir ejecutando el comando “`docker run`” mostrado previamente, sino que hay que arrancar el contenedor mediante el uso del comando “`docker start`”. Para ello, es necesario pasarle como argumento a dicho comando el identificador del contenedor (otra opción sería pasándole el alias del contenedor). Este ID se puede obtener a través del comando “`docker ps -a`”. Este comando muestra en el terminal la lista de

contenedores activos en ese momento. Al añadir el *flag* “-a”, se muestran también los contenedores que estén creados pero que no se encuentren activos en ese momento.

Por último, para acceder al contenedor que ha sido activado, hay que ejecutar el comando “docker attach”. De nuevo, el argumento que requiere este comando es el identificador del contenedor (o el alias) al que se desea acceder.

El resumen de los comandos a ejecutar es el siguiente:

```
docker ps -a
```

```
docker start ID_del_contenedor
```

```
docker attach ID_del_contenedor
```

Una vez accedido al contenedor, el terminal pasará a ser el correspondiente al contenedor Docker.

4.3 Extracción de Omnetpy del contenedor Docker. Utilización de la biblioteca con otras versiones de OMNeT++

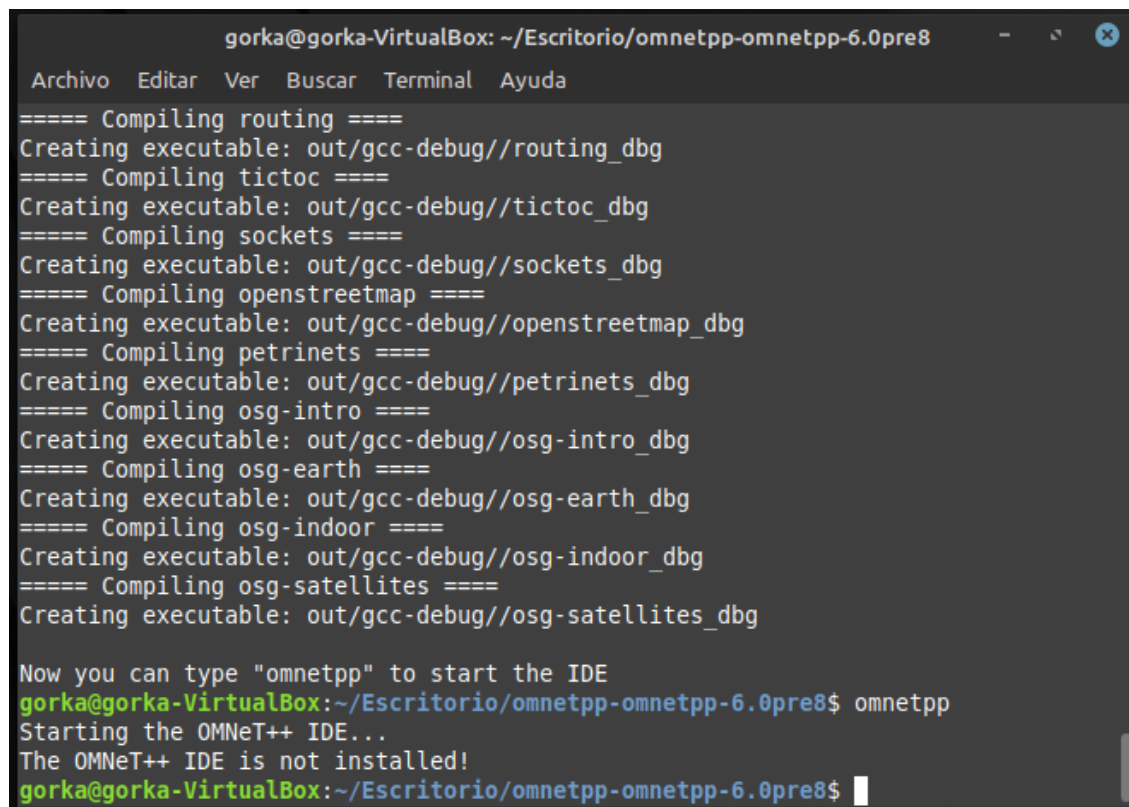
Se van a detallar, a continuación, los pasos seguidos a la hora de conseguir la extracción de la biblioteca Omnetpy del contenedor Docker con el objetivo de integrar Python y OMNeT++ en cualquier máquina y con diferentes versiones de OMNeT++.

Omnetpy es una biblioteca basada en PyBind11 [6] por lo que, en primer lugar, se procederá con la instalación de esta biblioteca que permite crear conexiones entre código en C++ y código en Python.

Una vez instalado PyBind, se proseguirá con la descarga e instalación de la biblioteca Omnetpy.

Seguidamente, se expondrá, el proceso de instalación de la versión 5.6.2 de OMNeT++. Cabe resaltar que, inicialmente, se probó a instalar la misma versión de OMNeT++ que se encontraba en el contenedor Docker, es decir, la versión “OMNeT++

6.0 Preview 8” pero, en dicha versión, resultó imposible disponer del entorno de desarrollo de OMNeT++, tal como se muestra en la Figura 4, por lo que se decidió probar a instalar una versión estable (la ya mencionada versión 5.6.2).



```
gorka@gorka-VirtualBox: ~/Escritorio/omnetpp-omnetpp-6.0pre8
Archivo Editar Ver Buscar Terminal Ayuda
==== Compiling routing ====
Creating executable: out/gcc-debug//routing_dbg
==== Compiling tictoc ====
Creating executable: out/gcc-debug//tictoc_dbg
==== Compiling sockets ====
Creating executable: out/gcc-debug//sockets_dbg
==== Compiling openstreetmap ====
Creating executable: out/gcc-debug//openstreetmap_dbg
==== Compiling petrinets ====
Creating executable: out/gcc-debug//petrinets_dbg
==== Compiling osg-intro ====
Creating executable: out/gcc-debug//osg-intro_dbg
==== Compiling osg-earth ====
Creating executable: out/gcc-debug//osg-earth_dbg
==== Compiling osg-indoor ====
Creating executable: out/gcc-debug//osg-indoor_dbg
==== Compiling osg-satellites ====
Creating executable: out/gcc-debug//osg-satellites_dbg

Now you can type "omnetpp" to start the IDE
gorka@gorka-VirtualBox:~/Escritorio/omnetpp-omnetpp-6.0pre8$ omnetpp
Starting the OMNeT++ IDE...
The OMNeT++ IDE is not installed!
gorka@gorka-VirtualBox:~/Escritorio/omnetpp-omnetpp-6.0pre8$
```

Figura 4. Error Obtenido al intentar iniciar el IDE de OMNeT++ tras la instalación de la versión “6.0 preview 8”.

Por último, la explicación se centrará en los detalles de la incorporación de la biblioteca Omnetpy a un proyecto de OMNeT++.

4.3.1 Instalación de PyBind11

Para la instalación de la biblioteca PyBind11, se siguen los pasos indicados en [10]. La versión que se va a instalar es la 2.4.3. Al ser una versión anterior a la 2.6.0, es necesario comprobar que la versión de Python instalada no sea la 3.9.0 porque podrían producirse errores.

Con el objetivo de instalar los paquetes prerequisites para poder instalar PyBind11, se procede a ejecutar en el terminal los siguientes comandos (se recuerda que dichos comandos se ejecutan en una máquina cuyo sistema operativo es Linux Mint 20):

```
apt-get install python3-dev  
  
apt-get install python3-distutils  
  
apt-get install build-essential  
  
apt-get install cmake  
  
apt-get install pybind11-dev  
  
apt-get install python3-pybind11  
  
apt-get install libeigen3-dev
```

Con ello, la biblioteca se encuentra instalada y lista para su utilización.

4.3.2 Instalación de Omnetpy

En este caso, se comienza descargando el código de [1]. Una vez descargado, se procede a su descompresión. Se puede hacer uso, para ello, del comando `unzip`.

A continuación, es necesario configurar correctamente las variables de entorno para permitir, posteriormente, una correcta instalación de la biblioteca. Para ello, hemos de modificar el fichero “.bashrc” añadiendo las siguientes líneas:

```
export OMNETPP_ROOT=~/Escritorio/omnetpp-5.6.2  
  
export OMNETPY_ROOT=~/Escritorio/omnetpy-master/omnetpy
```

Es necesario cerrar y reabrir el terminal para que las líneas añadidas surtan efecto. En las líneas anteriores, “~” se expande a la ruta personal. Nótese, también, que en mi caso tanto OMNeT++ como el código de la biblioteca Omnetpy se encuentran en el escritorio.

El siguiente paso es, entonces, ejecutar el comando `make` en el directorio en el que se encuentra el fichero `Makefile`, es decir, el directorio `omnetpy`. Tras esto, se indicará en el terminal que el módulo Python `pyopp` se ha creado con éxito por lo que se ha completado la instalación de la biblioteca.

4.3.3 Instalación de la versión 5.6.2 de OMNeT++

En lo referente a la instalación de la versión 5.6.2 de OMNeT++, se siguen las indicaciones dadas en [11]. Aunque dicha guía está originalmente diseñada para instalar la versión 5.6.1, no se ha percibido ninguna diferencia en el proceso de instalación para la versión 5.6.2 que pudiera haber producido algún error.

En primer lugar, es necesario instalar los paquetes prerequisites ejecutando para ello:

```
sudo apt-get update
```

Seguido de:

```
sudo apt-get install build-essential gcc g++ bison \  
flex perl python python3 qt5-default libqt5opengl5-dev \  
tcl-dev tk-dev libxml2-dev zlib1g-dev default-jre doxygen \  
graphviz
```

Además, es necesario instalar el paquete `openscenegraph` ejecutando para ello los siguientes comandos:

```
sudo add-apt-repository pp:ubuntugis/ppa
```

```
sudo apt-get update
```

```
sudo apt-get install openscenegraph-plugin-osgearth \  
libosgearth-dev
```

Después, se descarga el archivo comprimido de [4] y se descomprime mediante el comando:

```
tar xvfz omnetpp-5.6.2-src.tgz
```

OMNeT++ necesita que su directorio `bin/` se encuentre en el *path* por lo que, a continuación, es necesario, al igual que durante la instalación de la biblioteca Omnetpy, añadir una línea al fichero “.bashrc” tal que:

```
export PATH=$HOME/omnetpp-5.6.1/bin:$PATH
```

Es necesario cerrar y reabrir el terminal para que los cambios surtan efecto.

Posteriormente, se debe ejecutar el siguiente comando:

```
./configure
```

Con él, el *script* “configure”, detecta el *software* instalado y la configuración del sistema y escribe los resultados en el fichero “Makefile.inc” que será leído por los ficheros “Makefile” durante el proceso de compilación.

Por último, se procede a la compilación de la biblioteca a través del comando `make`. Una vez finalizado el proceso, se indicará al usuario en el terminal que todo ha transcurrido con éxito y que ahora se puede teclear el comando `omnetpp` para arrancar el IDE.

4.3.4 Incorporación de Omnetpy a OMNeT++

Una vez instalados OMNeT++ (en su versión 5.6.2) y la biblioteca Omnetpy, se prosigue con la incorporación de dicha biblioteca al simulador.

Para ello, es necesario añadir algunas líneas más al fichero “.bashrc” las cuales se muestran a continuación:

```
export LD_LIBRARY_PATH=$OMNETPP_ROOT/lib:$OMNETPY_ROOT/lib
```

```
export PYTHONPATH=$PYTHONPATH:$HOME/Escritorio/omnetpy-  
master/omnetpy/bindings
```

Se recuerda que para que estos cambios tengan efecto es necesario reiniciar el terminal desde el que se esté operando.

Los siguientes cambios requeridos para conseguir una correcta integración se llevan a cabo en el propio IDE de OMNeT++ por lo que se ha de ejecutar el comando `omnetpp` en el terminal para acceder a él.

El siguiente paso reside en instalar PyDev. Para ello, se navega hasta la sección `Help->Install New Software...` del IDE y se introduce la siguiente URL en el campo “Work With”:

<http://pydev.org/updates>

A continuación, se desmarca la casilla “Contact all update sites during install to find required software”. Se ha de hacer clic, varias veces, entonces, en “Next >”, aceptar los términos y condiciones y finalizar la instalación. Es necesario reiniciar el IDE para que los cambios tengan efecto.

Después, hay que navegar hasta la sección:

`Window->Preferences->PyDev->Python Interpreter`

En esa ventana, se ha de añadir el intérprete `python3` que se encuentra en la ruta `/usr/bin/python3`. Además, en esa misma ventana se encuentra la subsección `Environment`. Entrando en ella y haciendo clic en el botón “Add...” se debe añadir la variable de entorno `WITHIN_OMNETPP_IDE` con su valor puesto a `yes`.

Con ello, ya es posible crear proyectos en los que se adjunten ficheros con código escrito en Python en ellos con un par de salvedades que se han de considerar y que se ponen de manifiesto a continuación.

Para que no se produzcan errores a la hora de hacer un `Build Project`, se ha de añadir al proyecto un fichero `makefrag`, tal como se indica en [2] con las siguientes líneas:

```
INCLUDE_PATH += $(shell python3 -m pybind11 --include) -  
I$(OMNETPY_ROOT)/include
```

```
LIBS = -lomnetpy $(shell python3-config --libs | cut -d" " -f1)
```

```
LDFLAGS += -L$(OMNETPY_ROOT)/lib
```

Además, hay que considerar que, en el caso de que la versión de Python que se esté utilizando sea la 3.8 o posterior, se ha de reemplazar

```
python3-config --libs
```

por:

```
python3-config --libs --embed
```

Por otra parte, es necesario cambiar la configuración activa del modo debug a `release` ya que Modenesi indica en [1] que una de las limitaciones actuales de Omnetpy es que no soporta la depuración o *debug* de código Python.

Para ello, hay que navegar hasta la sección:

```
Project->Properties->C/C++ Build->Build Variables
```

En dicha ventana, se debe hacer clic en la opción “Manage configurations...” y, en ella, cambiar la configuración activa de debug a `release`.

Tras seguir todos estos pasos, se estará en disposición de crear proyectos en OMNeT++ que hagan uso de la biblioteca Omnetpy con el objetivo de poder escribir el código de los módulos en Python.

4.4 Incorporación del simulador de redes ópticas pasivas EPON a OMNeT++

En esta sección, se indican los pasos que se han de seguir para una correcta integración del proyecto del simulador EPON a OMNeT++ sobre el que se va a trabajar posteriormente.

En primer lugar, desde el IDE de OMNeT++, se debe importar el proyecto mediante:

File -> Import -> Existing Project into Workspace

A continuación, se ha de indicar la ruta en la que se encuentra el proyecto.

Después, es necesario cambiar las rutas asociadas a los diferentes directorios y subdirectorios del proyecto, cuya inclusión es necesaria, para que se correspondan con las rutas en las que se encuentran los mismos ya que, inicialmente, contendrán rutas absolutas correspondientes a otra máquina. Se muestran, en la siguiente captura (Figura 5), algunas de esas rutas tras su modificación para que el proyecto funcione en la máquina virtual empleada para la realización de este trabajo.

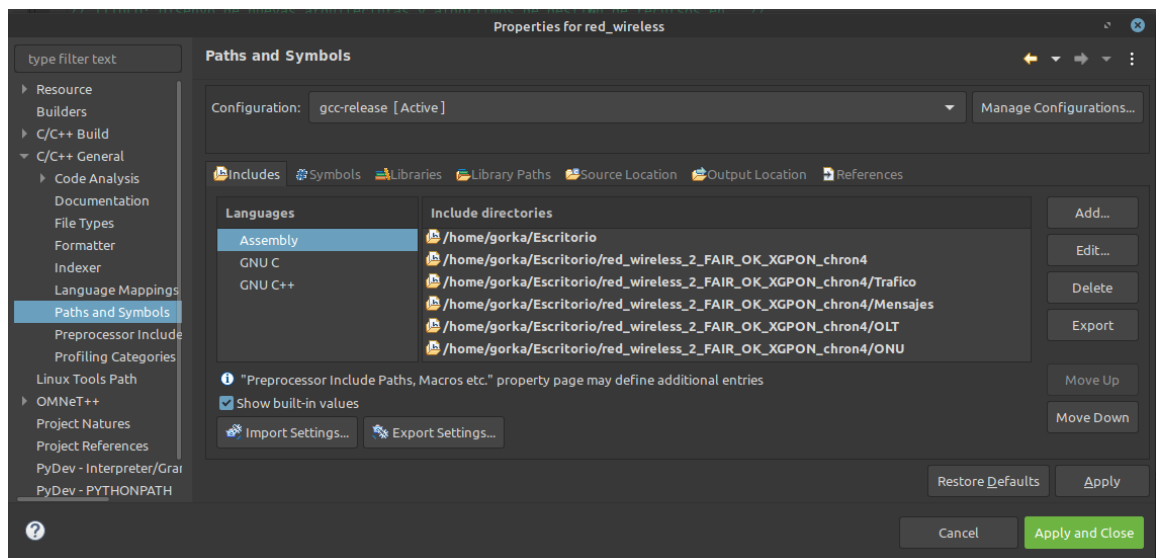


Figura 5. Rutas a los diferentes directorios y subdirectorios que componen el proyecto simulador PON y que han de incluirse en la sección Paths and Symbols.

Para llevar a cabo las mencionadas modificaciones a las rutas, se debe navegar por el IDE hasta la sección:

Project -> Properties -> C/C++ General -> Paths and Symbols

En la ventana que se muestra entonces, se deben editar las rutas en los lenguajes Assembly, GNU C y GNU C++. Para ello, basta con hacer clic en “Edit...” y escribir la dirección de la ruta correspondiente a cada directorio y subdirectorio del proyecto.

Por último, también será necesario adaptar las rutas de las directivas *Include* que se encuentran en los ficheros fuente (.cc) y de cabeceras (.h) del proyecto.

Una vez realizados los pasos anteriores, el proyecto se encontrará operativo y listo para ser ejecutado.

4.5 Conclusiones

En este capítulo se ha comenzado proporcionado una explicación, en detalle, del proceso seguido a la hora de integrar Python y OMNeT++ en un contenedor Docker haciendo uso, para ello, de la biblioteca desarrollada por Marcos Modenesi, Omnetpy.

Posteriormente, se han descrito los pasos necesarios para la extracción y utilización de dicha biblioteca fuera del contenedor Docker con el objetivo de integrarla en versiones diferentes de OMNeT++ lo cual aporta la versatilidad de poder seguir haciendo uso de ella junto con futuras actualizaciones de OMNeT++.

Por último, se indican también, en una breve sección, los pasos y consideraciones que han de tenerse en cuenta para una correcta importación del simulador EPON en OMNeT++.

Cabe resaltar que, en este capítulo, se ha podido comprobar la meticulosidad y el cuidado que ha de tener el usuario a la hora de instalar paquetes *software* puesto que siempre pueden producirse errores que impidan una correcta instalación como pueden ser las variables de entorno, dependencias de paquetes o incluso ciertas configuraciones que haya que establecer.

Nótese que aún no se ha procedido con la explicación de los pasos a seguir para poder incorporar módulos programados en lenguaje Python a proyectos de OMNeT++. Esta cuestión será tratada en detalle en el Capítulo 6 de esta memoria.

5

Implementación, simulación y validación de algoritmos DBA en el simulador 10G-EPON

5.1 Introducción

Este capítulo está dedicado a la implementación de algoritmos de asignación dinámica de ancho de banda (DBA, *Dynamic Bandwidth Allocation*) en el simulador de redes EPON en OMNeT++.

En este simulador es donde se procederá, en un capítulo posterior de esta memoria, con la integración de un módulo programado en lenguaje Python.

Previamente, en el presente capítulo, dicho simulador ha sido modificado para poder simular redes 10G-EPON, es decir, redes en las que la tasa de transmisión ha pasado de ser de 1 Gbps a 10 Gbps.

El estudio se va a llevar a cabo sobre el algoritmo de *polling* IPACT, sobre un algoritmo de *polling* controlado por un controlador PID [12] y sobre un algoritmo de *polling* con un controlador PID en el que los parámetros de sintonización de dicho PID son modificados de forma dinámica por una red neuronal [13].

En primer lugar, en cada caso, se va a comenzar describiendo, de forma breve, cada uno de los algoritmos en cuestión. A continuación, se procederá a describir la configuración de cada uno de los escenarios que se van a simular y, por último, tendrá lugar el análisis de los resultados obtenidos en cada apartado y en cada variante simulada.

5.2 Implementación del algoritmo de *polling* IPACT

En este apartado se describe, de forma resumida, el funcionamiento del algoritmo de *polling*, IPACT, empleado en la asignación dinámica de ancho de banda a cada ONU.

Se plantea, posteriormente, un escenario de simulación con dos variantes relacionadas con diferentes clases de servicio. La primera con una única prioridad de servicio para cada ONU y la segunda con tres prioridades de servicio en la que la prioridad P_0 , que se corresponde con la prioridad más alta, siempre va a suponer una carga de 0.0448. Ésta es una carga normalizada que se sustrae del total que se corresponde con la carga que se le quiera otorgar a cada ONU de cara a la simulación. La carga que le corresponde a cada ONU se define como el cociente entre la tasa media de transmisión de una ONU y la tasa de transmisión máxima que se le permite a un abonado conectado a dicha ONU [8] (en este caso, dicha tasa es de 1 Gbps).

La clase de prioridad P_0 simula un tráfico constante de videoconferencia. Su tasa de transmisión se obtiene a partir de la generación de paquetes de 70 bytes (incluyendo las cabeceras) cada 12.5 microsegundos con lo que se tiene: $(70 \times 8) / (12.5 \times 10^{-6}) = 44.8$ Mbit/s. Dicha tasa se divide entre los 1000 Mbit/s de tasa de transmisión máxima por abonado para obtener la carga de 0.0448 asociada a P_0 .

El resto de la carga con la que se haya configurado a cada ONU se reparte de forma equitativa entre las clases de servicio P_1 y P_2 que se corresponden con prioridad intermedia y prioridad baja, respectivamente.

Por ejemplo, en una simulación en la que se quiera dotar a las ONUs de una carga normalizada de 0.9, la clase de servicio P_0 contará con una carga de 0.0448 mientras que las prioridades P_1 y P_2 dispondrán de $(0.9 - 0.0448) / 2 = 0.4276$.

Cada una de las tres prioridades de servicio mencionadas se corresponde con una cola dentro de cada ONU de la red.

Por último, se analizan los resultados obtenidos en cada uno de los dos casos a través de gráficas en las que se reflejan parámetros como el retardo y la cantidad media de bytes que quedan en las colas tras cada ciclo.

5.2.1 Descripción del algoritmo IPACT

IPACT es un algoritmo de *polling* implementado en la capa MAC del OLT del simulador. Es el encargado de asignar el ancho de banda a cada una de las ONUs así como el instante de tiempo en el que cada ONU puede comenzar a transmitir en el siguiente ciclo (un ciclo tiene una duración máxima de 2 ms según el estándar EPON).

El algoritmo se ejecuta cada vez que llega un mensaje *report* al OLT (concretamente a la capa MAC del OLT) procedente de cualquiera de las ONUs. En ese momento, el algoritmo tiene en cuenta el ancho de banda demandado por la ONU y el ancho de banda máximo que se le puede asignar a esa ONU. Este último parámetro es fijo y depende del SLA (*Service Level Agreement*) al que esté asociada esa ONU. En este caso, únicamente hay un SLA para todas las ONUs por lo que dicho parámetro coincide para todas ellas.

A la hora de realizar la asignación, se hace la comparación entre el ancho de banda demandado y el ancho de banda máximo que se le puede asignar a esa ONU.

En el caso de que el ancho de banda demandado sea menor que el máximo, se le proporciona a la ONU la totalidad del ancho de banda que esté demandando.

Por otra parte, si el ancho de banda demandado por la ONU excede el máximo establecido, se le asigna este último.

Además de la asignación del ancho de banda, a cada ONU se le asigna también el instante del ciclo siguiente en el que puede comenzar a transmitir de nuevo. Para ello, el algoritmo tiene en cuenta el instante de finalización de la transmisión de la ONU inmediatamente anterior (ya que todas las ONUs transmiten en un orden preestablecido). Dicho instante temporal se compara con el instante de tiempo dado por la suma entre el instante de tiempo de simulación actual ($\text{simTime}()$), el retardo de ida de la red que se corresponde con la mitad del RTT (*Round Trip Time*) y la tasa de transmisión del mensaje *gate* creado (T_{GATE}). Se presentan, entonces, dos situaciones distintas:

- a) La primera en la que la suma de los tres términos es mayor que el instante de finalización de la ONU anterior por lo que el instante de inicio de transmisión siguiente para la ONU viene dado por dicha suma.

- b) La segunda en la que es mayor el instante de finalización de la ONU anterior, por lo que es necesario que la ONU espere a que termine de transmitir la ONU previa con lo que ese será el momento en el que la ONU podrá comenzar a transmitir en el siguiente ciclo.

5.2.2 Configuración del escenario de simulación

Se describen, a continuación, en forma de tabla, los parámetros escogidos para las simulaciones que se van a llevar a cabo en este escenario en el que se pretende simular el comportamiento del algoritmo IPACT. Todos ellos son fijados previamente a las simulaciones en el fichero de configuración `omnetpp.ini`.

En dichas simulaciones, se va a barrer la carga de cada ONU de 0.1 a 0.9 en intervalos de 0.1 en 0.1.

El tiempo de simulación escogido es de 1000 segundos para las cargas de 0.1 a 0.5 y de 2000 segundos para las cargas de 0.6 a 0.9.

Cabe destacar que, los parámetros que han cambiado con respecto a simulaciones de estudios anteriores son los relativos al escalado de la red hacia una red 10G-EPON. A saber:

- La tasa de transmisión de la red (`txrate`) pasa de 1 Gbps a 10 Gbps.
- La tasa de transmisión máxima a la que puede transmitir un abonado conectado a una ONU (`USER_Line_rate`), que pasa de 100 Mbit/s a 1 Gbit/s (se aumenta un orden de magnitud).
- El tamaño total del *buffer* para almacenar las tramas Ethernet en cada cola de cada ONU (`tambuffer`) pasa de 10 Mbytes a 100 Mbytes (se aumenta un orden de magnitud).
- Se escogen 32 *streams*, es decir, 32 fuentes de tráfico de tipo *Self-Similar* (de naturaleza rafagosa) generadas por cada ONU, ya que, tal como se mencionaba en el PFC de Jose María Robledo [8], es el valor habitual escogido en la literatura por lo que permite una mejor comparación de los resultados.

Parámetros de Simulación	Valores
Número de ONUs	16 ONUs
Número de longitudes de onda	1 longitud de onda
Tasa de transmisión de la red	10 Gbit/s
Periodo del ciclo	2 milisegundos
Tiempo de guarda	5 microsegundos
Tamaño de los paquetes	Paquetes de 64, 594 y 1500 bytes con generación de paquetes trimodal
Longitud <i>pon1</i> (OLT-Splitter)	10 km
Longitud <i>pon2</i> (Splitter-ONU)	Longitud aleatoria del canal que une el <i>Splitter</i> con cada ONU. Valores entre 0 y 10 km
Tamaño de <i>buffer</i>	100 Mbytes
Algoritmo implementado	Algoritmo de <i>polling</i> IPACT
Método de inserción de paquetes	Método de prioridad de colas
Método de extracción de paquetes	Método de extracción de colas de prioridad
Número de <i>streams</i>	32 <i>streams</i>
Número de servicios de prioridad (número de colas)	<ul style="list-style-type: none"> - 1 (P_0) para la primera variante del escenario - 3 (P_0, P_1 y P_2) para la segunda variante del escenario

Tabla 1. Parámetros empleados en las simulaciones del algoritmo de *polling* IPACT

5.2.2.1 Evaluación de IPACT considerando una clase de servicio

En este caso, se van a generar dos gráficas a partir de los ficheros de resultados obtenidos en las simulaciones que se han llevado a cabo.

Todas las gráficas incluidas en la memoria de este trabajo han sido diseñadas y realizadas en Python a través de un procesamiento *a posteriori* de los datos obtenidos en esos ficheros de resultados.

En primer lugar, en la Figura 6, se analiza la evolución del retardo medio que sufren las tramas Ethernet desde que son generadas en cada ONU hasta que son recibidas en el OLT respecto a la carga de la ONU.

Se aprecia en ella que el retardo se mantiene en niveles muy parecidos a los publicados, previamente, en estudios en los que la tasa de transmisión de la red era de 1 Gbit/s [8], lo cual permite ratificar el correcto escalado de la red a 10G-EPON.

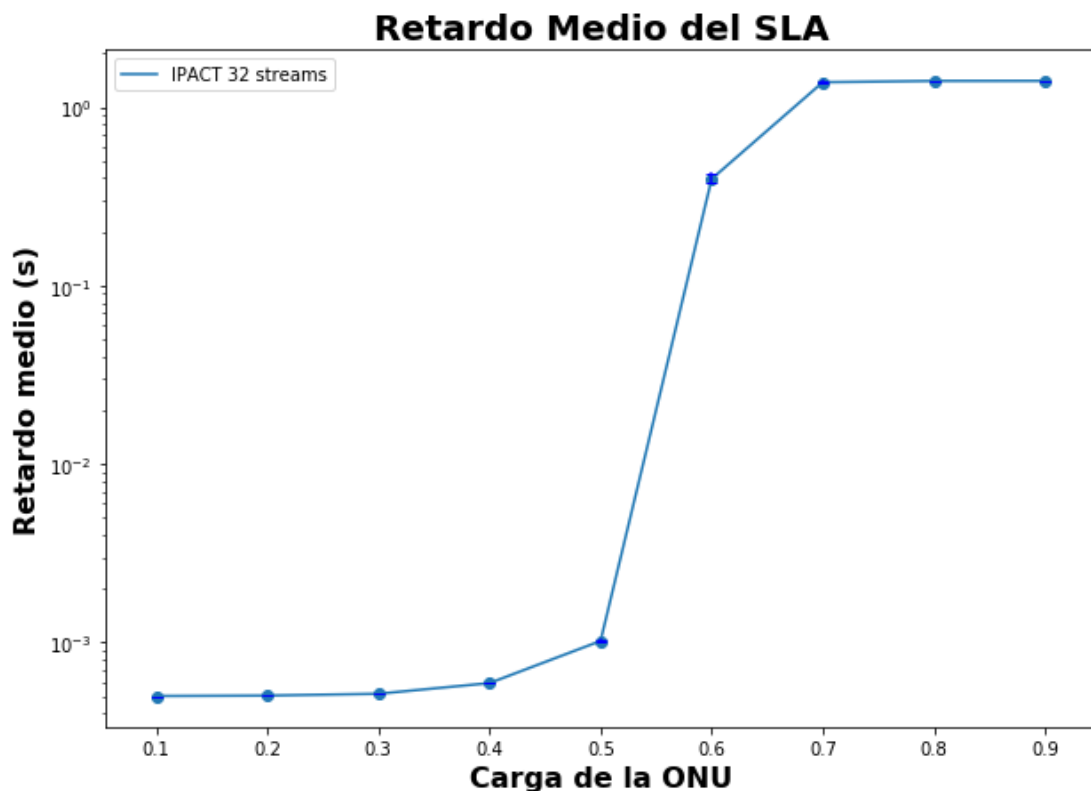


Figura 6. Retardo medio frente a la carga de la ONU para el algoritmo IPACT con un SLA y una sola cola.

Por otra parte, se presenta también, a continuación (Figura 7), la gráfica resultante del análisis de la cantidad media de bytes que quedan en las colas de una ONU tras cada ciclo de 2 ms frente a la carga de la ONU.

En este caso, si comparamos la gráfica con la obtenida en [8], podemos observar que, tal como se esperaba, la cantidad media de bytes que quedan en las colas tras cada ciclo se ha escalado un orden de magnitud hacia arriba.

Con ello, vemos como, para cargas altas en las que las colas se llenan, los valores se estancan en torno a 100 MB lo cual se corresponde con el nuevo tamaño de los *buffers* de cada ONU.

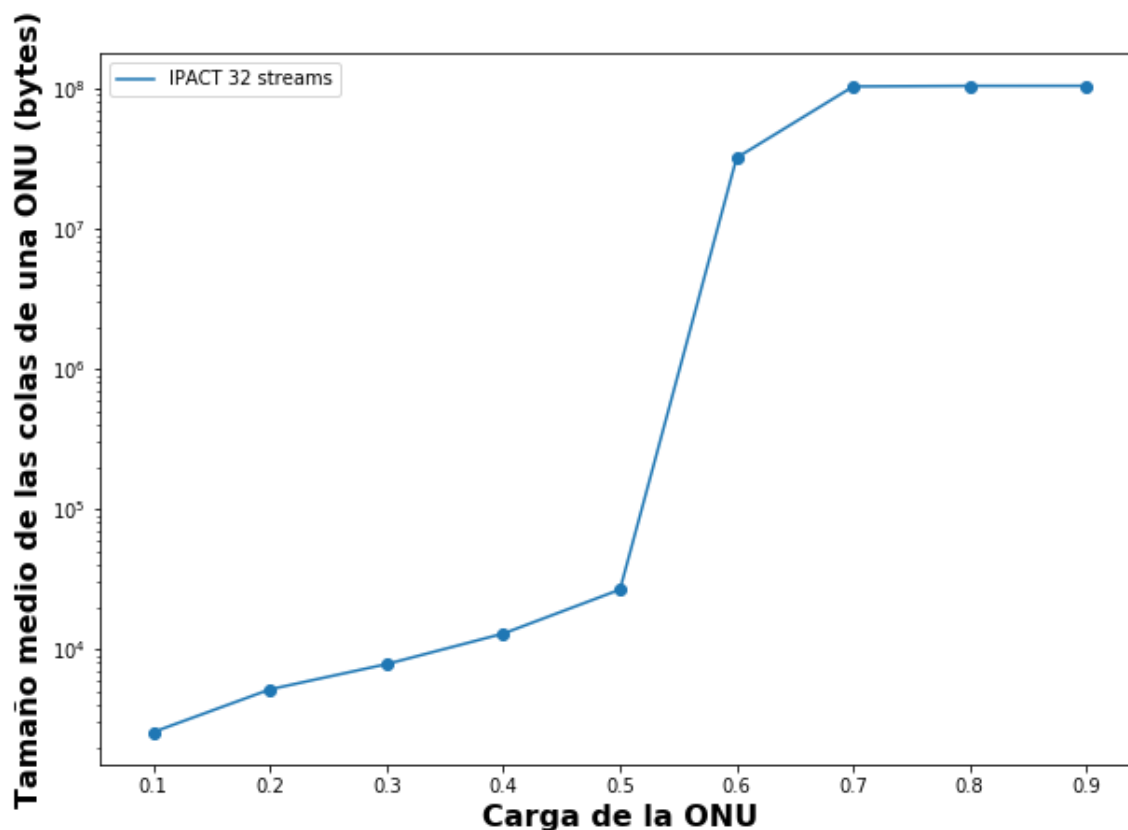


Figura 7. Tamaño medio de la cola en bytes frente a la carga de la ONU para el algoritmo IPACT con un SLA y una sola cola.

5.2.2.2 Evaluación de IPACT considerando 3 clases de servicio

En este caso, al tener tres prioridades de servicio de las cuales P_0 , correspondiente al servicio de máxima prioridad, presenta una carga fija de 0.0448. Por otro lado, la carga que se ha de indicar al parámetro de configuración del fichero `omnetpp.ini` es la mitad de la carga restante ya que ésta es la que corresponde tanto a la cola P_1 (servicio de prioridad media) como a la cola P_2 (servicio de más baja prioridad).

Por ello, el parámetro de carga se barre desde el valor 0.0276 hasta el valor 0.4276 en incrementos de 0.05 en 0.05.

Se muestra, a continuación (Figura 8), la gráfica que recoge el retardo medio que presentan cada una de las tres colas de una de las ONUs. En ella, se puede apreciar, tal como sucedía para el caso anterior con una única cola, que el retardo se mantiene en niveles similares a los publicados en [8]. Esto reafirma la correcta implementación del escalado de la red a 10G-EPON.

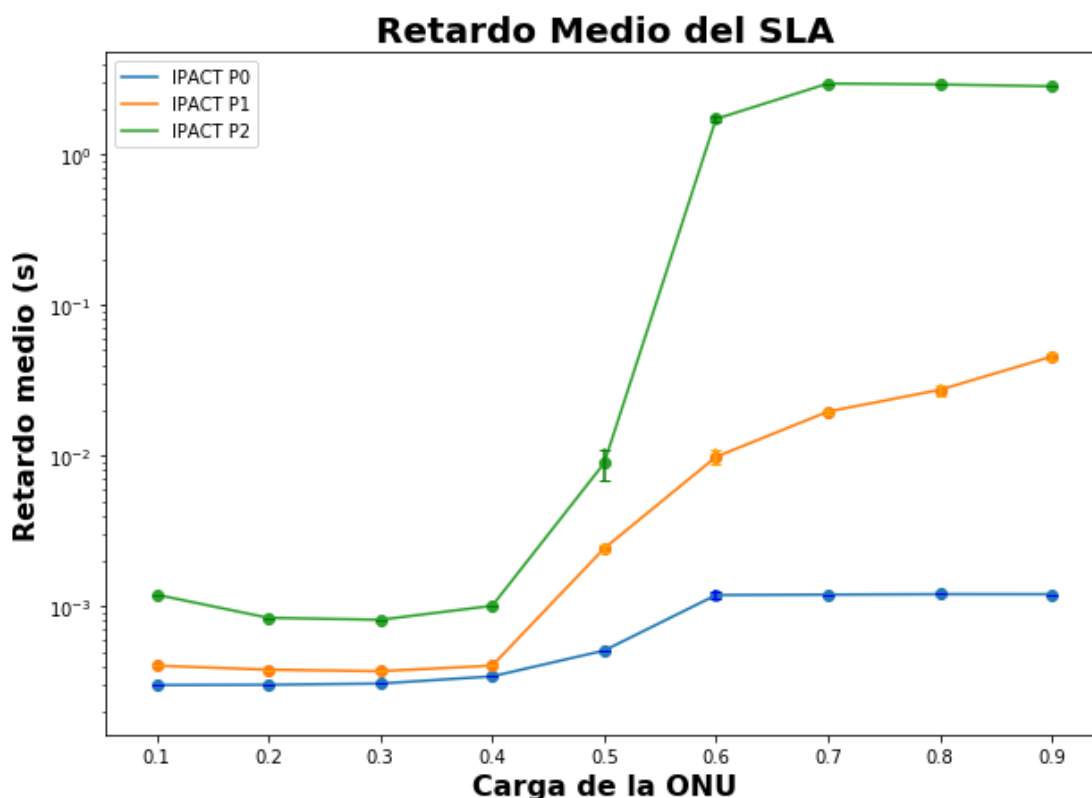


Figura 8. Retardo medio frente a la carga de la ONU para el algoritmo IPACT con un SLA y tres colas.

5.3 Implementación del algoritmo de *polling* con PID

Las siguientes simulaciones que se van a llevar a cabo, pretenden ilustrar el comportamiento de un algoritmo de *polling* controlado por un controlador PID desarrollado en trabajos de investigación anteriores.

Para ello, este apartado comienza con una pequeña explicación sobre los controladores PID así como sus principales características.

De forma análoga al apartado anterior, se proseguirá con la presentación del escenario de simulación del que, esta vez, se disgregarán tres variantes. Cada una de esas variantes presenta unos pesos diferentes para cada uno de los 3 SLAs presentes en la red simulada. De forma concreta, cada SLA representa un perfil de abonado con un ancho de banda garantizado que debe ser proporcionado por el proveedor de servicios.

Para finalizar, se muestran las gráficas de resultados obtenidas junto con la información que se puede extraer a partir de ellas. En este caso, las gráficas se centran en el ancho de banda ofrecido y demandado por cada ONU a lo largo de los 500 segundos simulados para una carga elevada en cada ONU.

5.3.1 Descripción de un controlador PID

Un PID (*Proportional Integral Derivative*) es un controlador simple basado en un bucle realimentado. Su objetivo es mantener una o más variables lo más cerca posible de un valor deseado [12].

El sistema tiene una entrada, la cual es una variable de control que se va modificando para conseguir que la variable de salida del sistema, la cual se corresponde con el parámetro que se pretende ajustar, se acerque lo más posible a dicho valor.

Para ello, se calcula la diferencia entre el valor actual de la variable y el valor deseado. Dicha diferencia se denota como el error del sistema.

Dicho error es introducido en un término proporcional, en un término con una integral y en un término con una derivada. Los tres términos son sumados para obtener la variable de control tal como se muestra, a continuación, en la siguiente ecuación (Figura 9) obtenida de [12]:

$$u(t) = K_p \left(e(t) + \frac{1}{T_i} \int_0^t e(\tau) d\tau + T_d \frac{de(t)}{dt} \right),$$

Figura 9. Ecuación que describe la variable de control de un PID.

Cabe destacar que los términos K_p , T_i y T_d son los parámetros de sintonización del controlador y éstos deben ser puestos a punto mediante algún método concreto para que el sistema sea estable.

5.3.2 Descripción del algoritmo SPID (Service Level Agreement PID)

El algoritmo empleado para asignar ancho de banda a cada ONU es un algoritmo de *polling* con un esquema limitado denominado SPID (Service Level Agreement) [12]. Esto quiere decir que el ancho de banda máximo total que se puede asignar en cada ciclo está limitado por el ancho de banda máximo que se puede transmitir en un solo ciclo cuya duración máxima es de 2 ms en el estándar EPON.

Mediante el uso del PID, que se ejecuta periódicamente cada T segundos, se controla el proceso de asignación de ancho de banda. En este caso, el error a la entrada del controlador PID, se corresponde con la diferencia entre el ancho de banda garantizado para una ONU perteneciente a un SLA determinado y el ancho de banda medio asignado a dicha ONU en el último ciclo.

La salida del PID es, entonces, el ancho de banda medio asignado a cada ONU en el siguiente ciclo, cuyo valor se quiere mantener lo más próximo posible al ancho de banda garantizado para esa ONU según el SLA al que pertenece.

El comportamiento descrito se puede apreciar, de forma visual, en el siguiente esquema (Figura 10) obtenido de [12]:

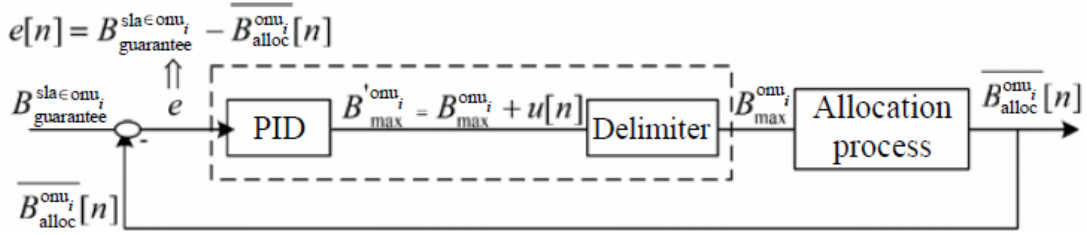


Figura 10. Diagrama de bloques que ilustra el proceso de funcionamiento del PID en la asignación dinámica de ancho de banda en redes PON.

Por otra parte, es necesario asignar unos valores iniciales a los anchos de banda máximos para cada ONU. Esto se realiza a partir de los anchos de banda garantizados correspondientes a cada SLA y a unos pesos asociados a cada SLA (W_{SLA_i}) tal como se muestra en la siguiente ecuación (Figura 11) obtenida de [12]:

$$B_{max}^{onu_i} = \frac{B_{cycle_available} \cdot W^{sla_k/ONU_i \in sla_k}}{\sum_j W^{sla_j} \cdot N_{onUs}^{sla_j}}.$$

Figura 11. Ecuación empleada durante la inicialización de los valores de ancho de banda máximo asignados a cada ONU

En ella, se reparte el ancho de banda total disponible en un ciclo ($B_{cycle_available}$) entre cada una de las ONUs según el peso asociado a cada una de ellas.

5.3.3 Configuración del escenario de simulación

Se describen, a continuación, en forma de tabla, los parámetros escogidos para las simulaciones que se van a llevar a cabo en este escenario en el que se pretende simular el comportamiento de un algoritmo de *polling* controlado por un PID.

En estas simulaciones, se va a presentar una situación de carga elevada en cada ONU. Se escoge un valor de 0.9. El tiempo de simulación escogido es de 500 segundos para todas las variantes del escenario.

Se muestran en la tabla los valores de los parámetros de sintonización K_p , T_i y T_d empleados en las simulaciones. Dichos valores han sido escogidos a partir del estudio realizado en [12] y realizado mediante el método manual de Ziegler-Nichols.

Parámetros de Simulación	Valores
Número de ONUs	16 ONUs
Carga de cada ONU	0.9
Número de longitudes de onda	1 longitud de onda
Tasa de transmisión de la red	10 Gbit/s
Periodo del ciclo	2 milisegundos
Tiempo de guarda	1 microsegundo
Tamaño de los paquetes	Paquetes de 64, 594 y 1500 bytes con generación de paquetes trimodal
Longitud <i>pon1</i> (OLT-Splitter)	10 km
Longitud <i>pon2</i> (Splitter-ONU)	Longitud aleatoria del canal que une el <i>Splitter</i> con cada ONU. Valores entre 0 y 10 km
Tamaño de <i>buffer</i>	100 Mbytes
Algoritmo implementado	SPID
Método de inserción de paquetes	Método de prioridad de colas
Método de extracción de paquetes	Método de extracción de colas de prioridad
Número de <i>streams</i>	32 <i>streams</i>
Número de servicios de prioridad (número de colas)	3 (P_0 , P_1 y P_2)
Número de SLAs	3
Número de ONUs asociadas al SLA_0	1
Número de ONUs asociadas al SLA_1	5

Número de ONUs asociadas al SLA ₂	10
K _p	0,5
T _i	11
T _d	2,75
BW garantizado para el SLA ₀ (Mbps)	1000
BW garantizado para el SLA ₁ (Mbps)	750
BW garantizado para el SLA ₂ (Mbps)	500

Tabla 2. Parámetros empleados en las simulaciones de un algoritmo de *polling* con un PID

En este caso, se van a simular tres variantes de este escenario. En la primera variante (escenario 1), los pesos asociados a cada uno de los tres SLAs permanecen iguales entre sí ($W_{SLA0}=1$, $W_{SLA1}=1$, $W_{SLA2}=1$). En la segunda (escenario 2), se le da un menor peso al SLA₀ y un mayor peso al SLA₂ ($W_{SLA0}=1$, $W_{SLA1}=2$, $W_{SLA2}=3$). Por último, en la tercera variante (escenario 3), se le otorga un mayor peso al SLA₀ y un menor peso al SLA₂ ($W_{SLA0}=3$, $W_{SLA1}=2$, $W_{SLA2}=1$). Estas variantes pretenden ilustrar cómo se adapta el algoritmo en tiempo real a la hora de asignar anchos de banda a las ONUs acorde al SLA contratado por cada abonado (ONU) y, por lo tanto, a su ancho de banda garantizado.

5.3.3.1 Evaluación de resultados

La Figura 12 recoge las gráficas obtenidas mediante las simulaciones de las tres variantes del escenario para el algoritmo SPID. En ella se puede ver como, a pesar de los diferentes pesos iniciales asignados a cada SLA, el tiempo que se tarda en alcanzar el ancho de banda garantizado prácticamente no se ve alterado. Se recuerda que, en este caso, a causa del escalado de la red, el ancho de banda garantizado para el SLA₀ es de 1000 Mbit/s.

Se observa, además, que el comportamiento es similar al publicado en [12] pero con los valores escalados a un orden de magnitud por encima.

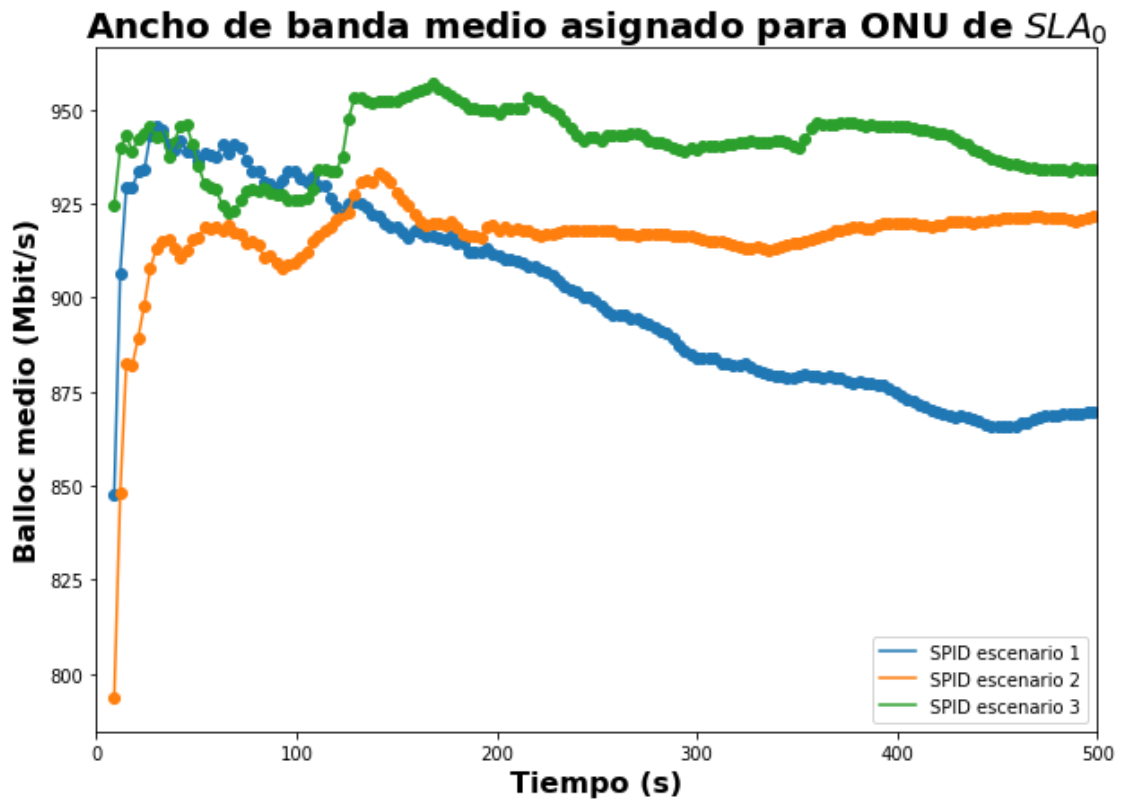


Figura 12. Evolución en tiempo real del ancho de banda medio asignado para una ONU perteneciente al SLA_0 con un algoritmo de *polling* controlado mediante un PID.

En la Figura 13, se presenta la misma gráfica centrada, esta vez, en los primeros 60 segundos para poder observar mejor como, a pesar de partir de diferentes pesos para el SLA_0 , en los tres casos se alcanza el ancho de banda medio asignado para ese nivel de servicio en un intervalo de tiempo similar.

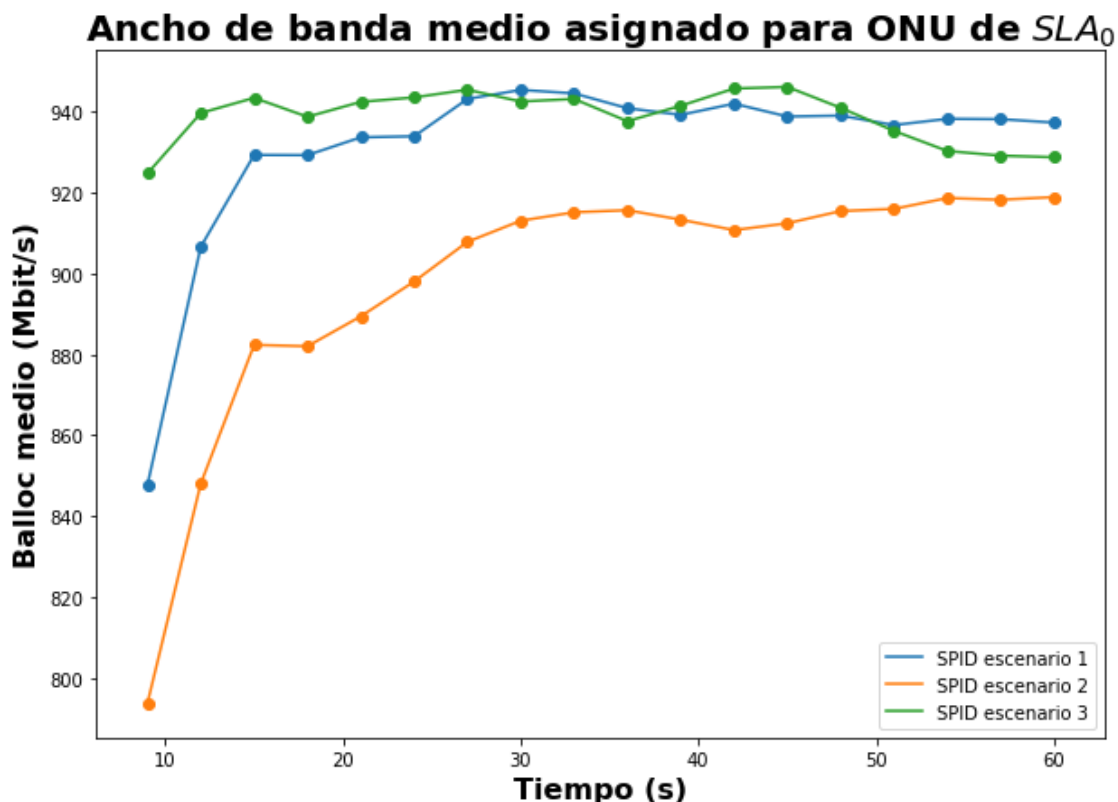


Figura 13. Cantidad media de ancho de banda asignado frente al tiempo para una ONU perteneciente al SLA_0 con un algoritmo de *polling* controlado mediante un PID. Zoom de los primeros 60 segundos.

5.4 Implementación del algoritmo NN-SPID (*Neural Network-SPID*) basado en redes neuronales

Las siguientes simulaciones pretenden ilustrar el comportamiento de un algoritmo de *polling* controlado por un controlador PID y cuyos parámetros de sintonización son actualizados de forma dinámica por una red neuronal con capacidad de auto aprendizaje. El algoritmo se denomina NN-SPID (Neural Network-SPID) [13].

El apartado comienza con una descripción de la red neuronal diseñada e incorporada al algoritmo de asignación dinámica de ancho de banda para gestionar la calidad de servicio de una red PON. Además, se hace especial hincapié en las ventajas que proporciona su integración para la gestión automatizada de los parámetros de sintonización de un PID, frente al método manual de sintonización (Ziegler-Nichols) que incorporaba el algoritmo SPID.

Al igual que en los casos anteriores, se someten a estudio dos variantes del escenario considerado. En cada una, se otorga un ancho de banda mínimo garantizado diferente a cada SLA.

Para concluir el apartado, se presentan las gráficas de resultados obtenidas junto con el análisis realizado a partir de ellas. En este caso, las gráficas se centran en el ancho de banda ofrecido y demandado por cada ONU, de nuevo, para una carga elevada en el tráfico que genera cada una de las ONUs.

5.4.1 Descripción de la red neuronal

Las redes neuronales son modelos matemáticos simplificados que tratan de imitar el comportamiento del sistema nervioso humano para resolver problemas complejos.

Una red neuronal está formada por una serie de capas en las que se encuentran las neuronas artificiales. Generalmente, las redes neuronales son multicapa aunque lo más habitual es que posea, únicamente, una sola capa intermedia denominada capa oculta [13].

Una neurona se puede considerar como un sistema que tiene tantas entradas como neuronas tenga la capa anterior (más una entrada adicional de sesgo o control, *bias*) y que produce una salida a partir de una función de activación. Las entradas a la neurona son pesadas por unos factores denominados “pesos sinápticos”. Dichos pesos definen la intensidad de la conexión entre dos neuronas.

En este trabajo, la red neuronal empleada posee tres capas. Una primera capa de entrada con tres neuronas en ella, una capa oculta con cinco neuronas y una capa de salida con otras tres neuronas.

Las entradas que se conectan a las tres neuronas de la capa de entrada se corresponden con los tres últimos errores calculados.

Por otra parte, las salidas que producen las tres neuronas de la capa de salida del sistema se corresponden con los tres parámetros de sintonización del PID que se desean modificar de forma dinámica mediante la red neuronal.

Se muestra, a continuación, en la Figura 14, un esquema con el aspecto de la red neuronal bajo estudio, obtenida de [14].

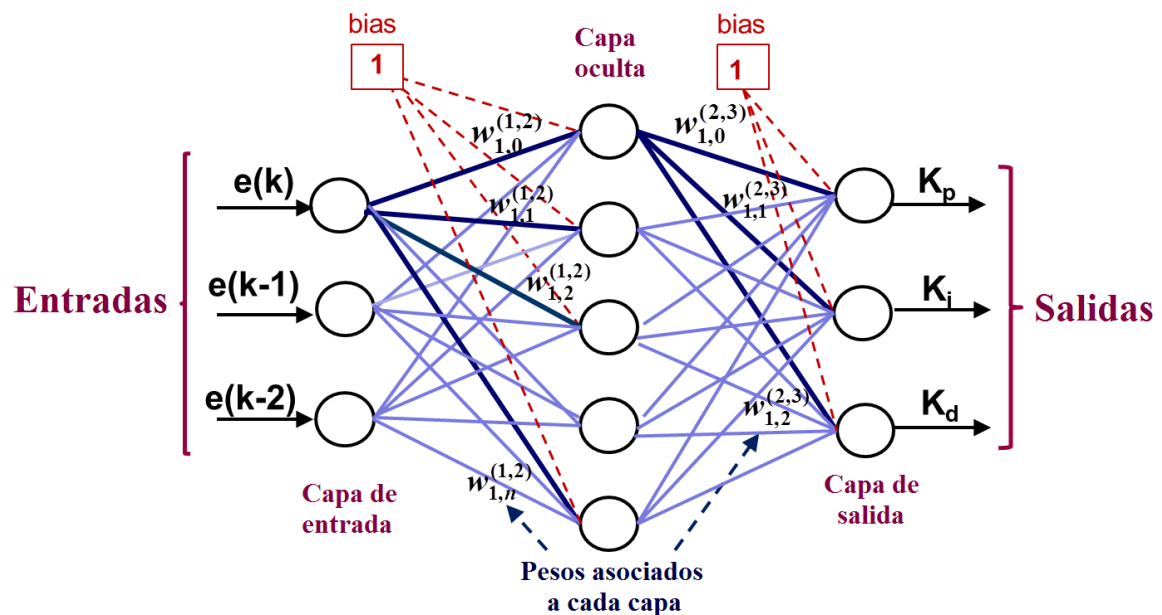


Figura 14. Diagrama de flujo de la red neuronal empleada para controlar los parámetros de sintonización del PID.

La gran ventaja de la red neuronal incorporada al algoritmo de gestión dinámica de ancho de banda, es precisamente la capacidad para modificar de forma dinámica los parámetros de sintonización del PID ya que, en estudios previos, dicha puesta a punto se llevaba a cabo de forma estática (bien a través de la técnica de Ziegler-Nichols o, posteriormente, con la implementación de un algoritmo genético), lo cual desembocaba en una peor adaptación a cambios en tiempo real de las condiciones de la red puesto que, una vez seleccionados los valores de los parámetros, estos se mantenían fijos o, en caso de querer cambiarse, debían volverse a ejecutar las técnicas de sintonización que eran procesos demasiado lentos.

5.4.2 Configuración del escenario de simulación

Se describen, a continuación, en forma de tabla, los parámetros escogidos para las simulaciones que se van a llevar a cabo en este escenario en el que se pretende simular el comportamiento de un algoritmo de *polling* controlado por un PID y cuyos parámetros de

sintonización (K_p , T_i y T_d) son actualizados dinámicamente mediante una red neuronal de tipo *feed-forward*.

En estas simulaciones, se va a presentar una situación de carga elevada en cada ONU. Se escoge un valor de 0.9.

El tiempo de simulación escogido es de 500 segundos para todas las variantes del escenario.

Parámetros de Simulación	Valores
Número de ONUs	16 ONUs
Carga de cada ONU	0.9
Número de longitudes de onda	1 longitud de onda
Tasa de transmisión de la red	10 Gbit/s
Periodo del ciclo	2 milisegundos
Tiempo de guarda	1 microsegundo
Tamaño de los paquetes	Paquetes de 64, 594 y 1500 bytes con generación de paquetes trimodal
Longitud <i>pon1</i> (OLT-Splitter)	10 km
Longitud <i>pon2</i> (Splitter-ONU)	Longitud aleatoria del canal que une el <i>Splitter</i> con cada ONU. Valores entre 0 y 10 Km
Tamaño de <i>buffer</i>	100 Mbytes
Algoritmo implementado	Algoritmo de <i>polling</i> DBA_polling_PID junto con la red neuronal
Método de inserción de paquetes	Método de prioridad de colas
Método de extracción de paquetes	Método de extracción de colas de prioridad
Número de <i>streams</i>	32 <i>streams</i>
Número de servicios de prioridad (número de	3 (P_0 , P_1 y P_2)

colas)	
Número de SLAs	3
Número de ONUs asociadas al SLA0	1
Número de ONUs asociadas al SLA1	5
Número de ONUs asociadas al SLA2	10

Tabla 3. Parámetros empleados en las simulaciones de un algoritmo de *polling* con un controlador PID y una red neuronal

Este último algoritmo DBA cuenta con dos variantes. En la primera de ellas (escenario 1), el ancho de banda garantizado a las ONUs pertenecientes a cada SLA se distribuye de la siguiente manera:

- $BW_{\text{Garantizado-SLA0}} = 1000 \text{ Mbit/s}$
- $BW_{\text{Garantizado-SLA1}} = 750 \text{ Mbit/s}$
- $BW_{\text{Garantizado-SLA2}} = 500 \text{ Mbit/s}$

Por su parte, en la segunda variante (escenario 2), los anchos de banda garantizados son:

- $BW_{\text{Garantizado-SLA0}} = 800 \text{ Mbit/s}$
- $BW_{\text{Garantizado-SLA1}} = 600 \text{ Mbit/s}$
- $BW_{\text{Garantizado-SLA2}} = 600 \text{ Mbit/s}$

Con ello, se pretende estudiar el tiempo de sintonización necesario para que la red proporcione los niveles de ancho de banda garantizados para cada uno de los tres SLAs.

5.4.2.1 Evaluación de resultados para el escenario 1

En este primer escenario de red se puede observar (Figura 15) como con el algoritmo NN-SPID se alcanza, rápidamente, el ancho de banda garantizado para cada SLA excepto para el SLA₀. Esto se debe a que el ancho de banda garantizado para dicho

SLA es de 1000 Mbit/s pero, al realizarse las simulaciones sobre una carga de 0.9, la ONU demanda, de media, 900 Mbit/s que es lo que le proporciona el algoritmo una vez se estabiliza.

Se observa, también, que las tres curvas representadas se asemejan a las publicadas en [13] pero, de nuevo, con el correspondiente escalado de un orden de magnitud hacia arriba al tratarse, ahora, de una red 10G-EPON.

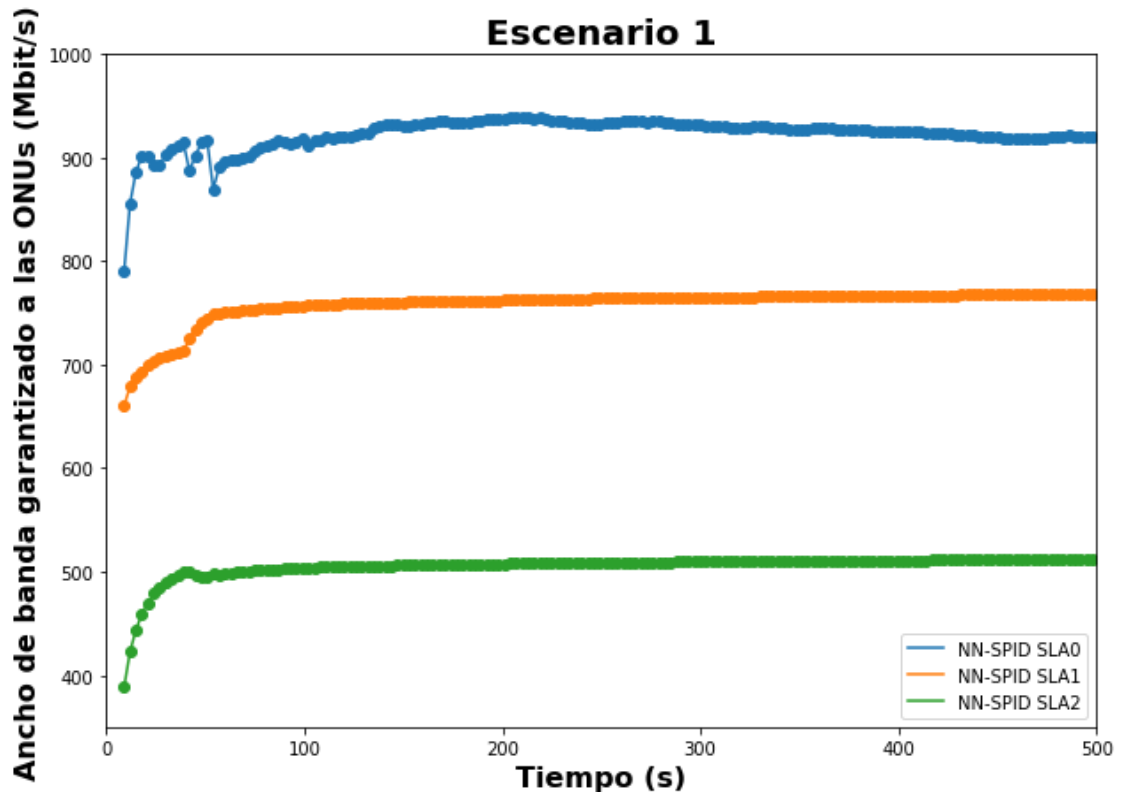


Figura 15. Evolución en tiempo real del ancho de banda garantizado para una ONU perteneciente a cada uno de los tres SLAs con el algoritmo de *polling* NN-SPID en el escenario 1.

Se muestra, a continuación, en la Figura 16, los primeros 60 segundos de la gráfica anterior con lo que se puede apreciar mejor la evolución que presenta el ancho de banda garantizado a las ONUs pertenecientes a cada uno de los 3 SLAs al inicio de la simulación.

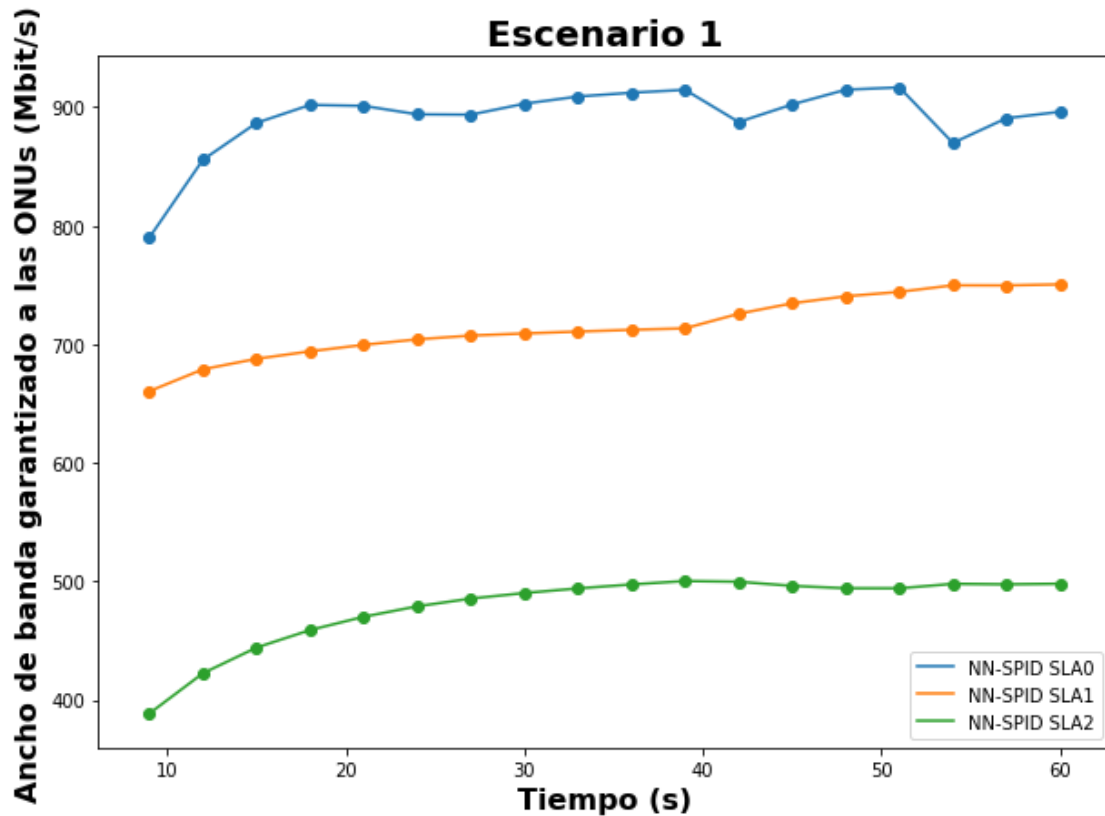


Figura 16. Evolución en tiempo real del ancho de banda garantizado para una ONU perteneciente a cada uno de los tres SLAs con el algoritmo de *polling* NN-SPID en el escenario 1. (Zoom 60 s).

En las siguientes figuras (Figura 17, Figura 18 y Figura 19), se muestran las gráficas asociadas a la evolución temporal de los valores de los parámetros de sintonización del PID. En ellas, se pueden apreciar las variaciones que sufren estos parámetros a causa del control dinámico que ofrece la red neuronal.

Se puede apreciar como, en todas ellas, los valores se estabilizan en un instante temporal que ronda los 100 segundos, coincidiendo con el instante en el que se alcanzan los niveles de ancho de banda garantizados esperados, tal como se ha visto en las gráficas anteriores.

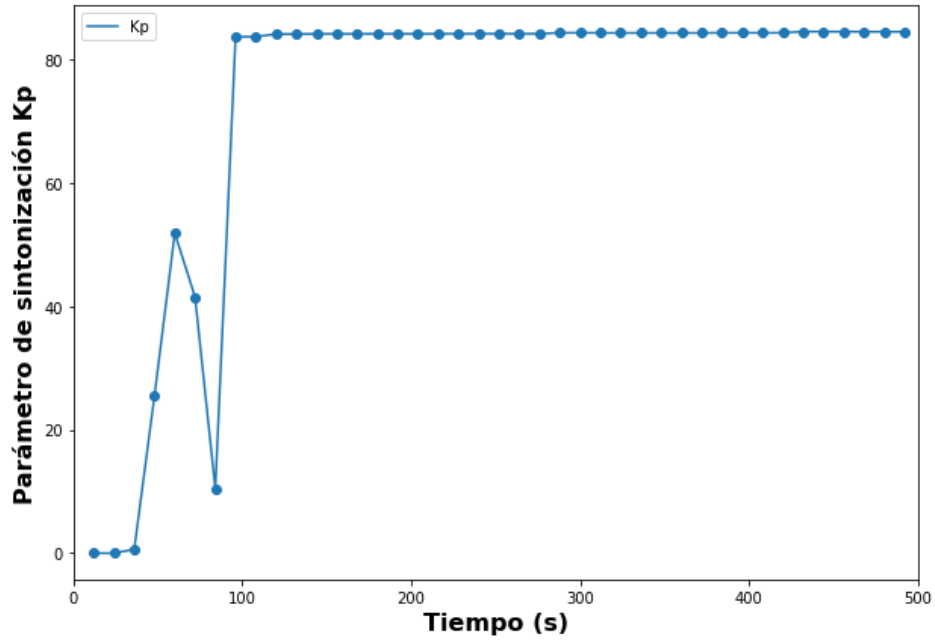


Figura 17. Evolución temporal del parámetro K_p para NN-SPID bajo el escenario 1.

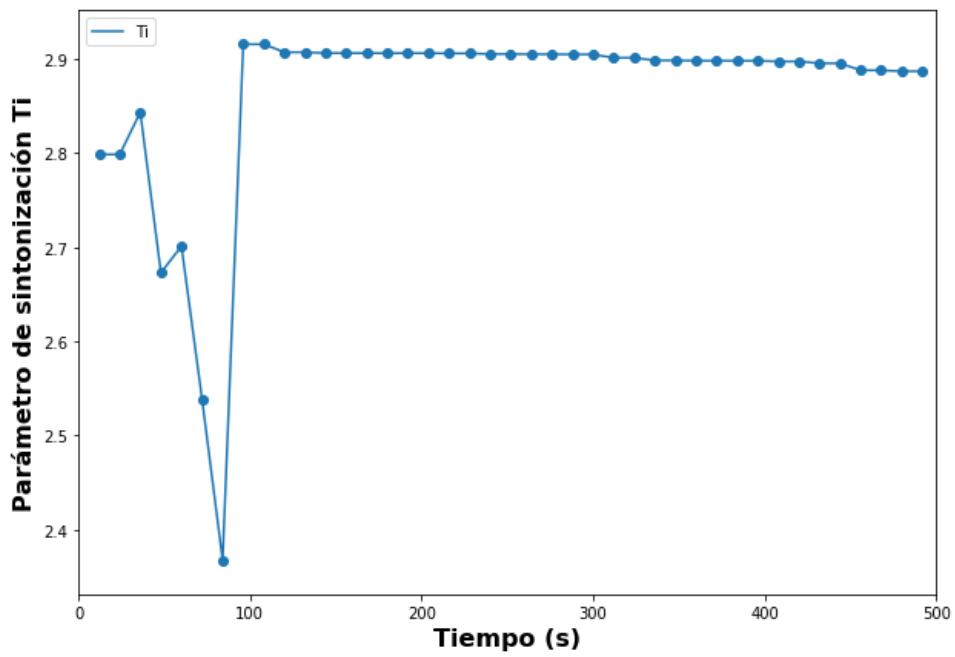


Figura 18. Evolución temporal del parámetro T_i para NN-SPID bajo el escenario 1.

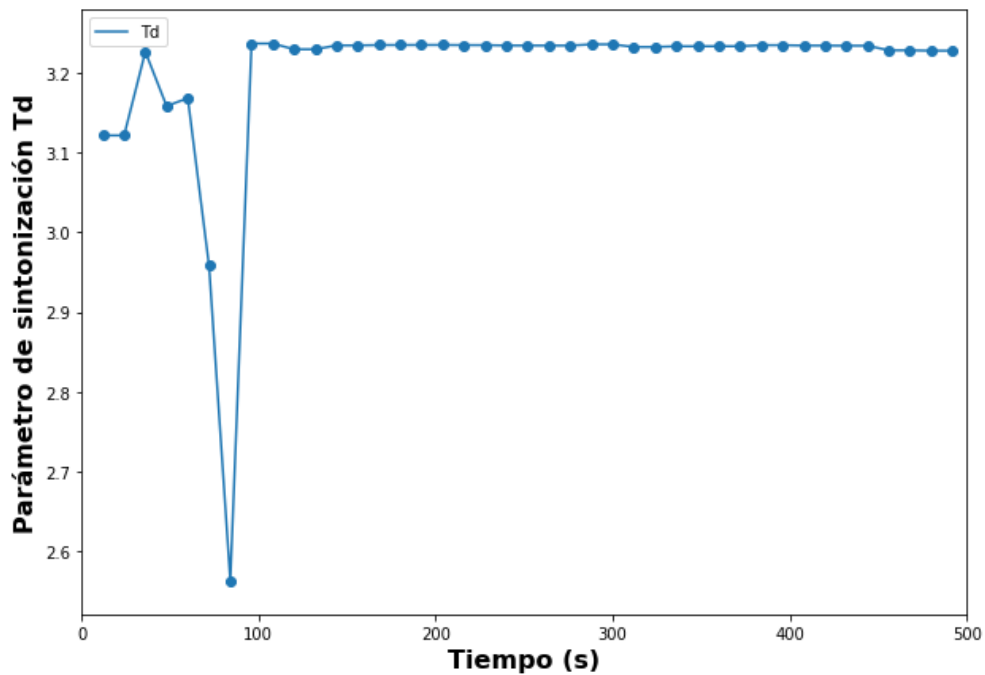


Figura 19. Evolución temporal del parámetro T_d para NN-SPID bajo el escenario 1.

5.4.2.2 Evaluación de resultados para el escenario 2

En este segundo escenario (Figura 20), al tener la suma de todos los anchos de banda garantizados un valor inferior a los 10 Gbps de capacidad máxima del canal, el OLT puede otorgar a las ONUs todo el ancho de banda que demandan (se recuerda que la carga de cada una de las ONUs es de 0.9). Por ello, las tres curvas se estabilizan en torno a esos valores de ancho de banda garantizado respectivos a su SLA correspondiente.

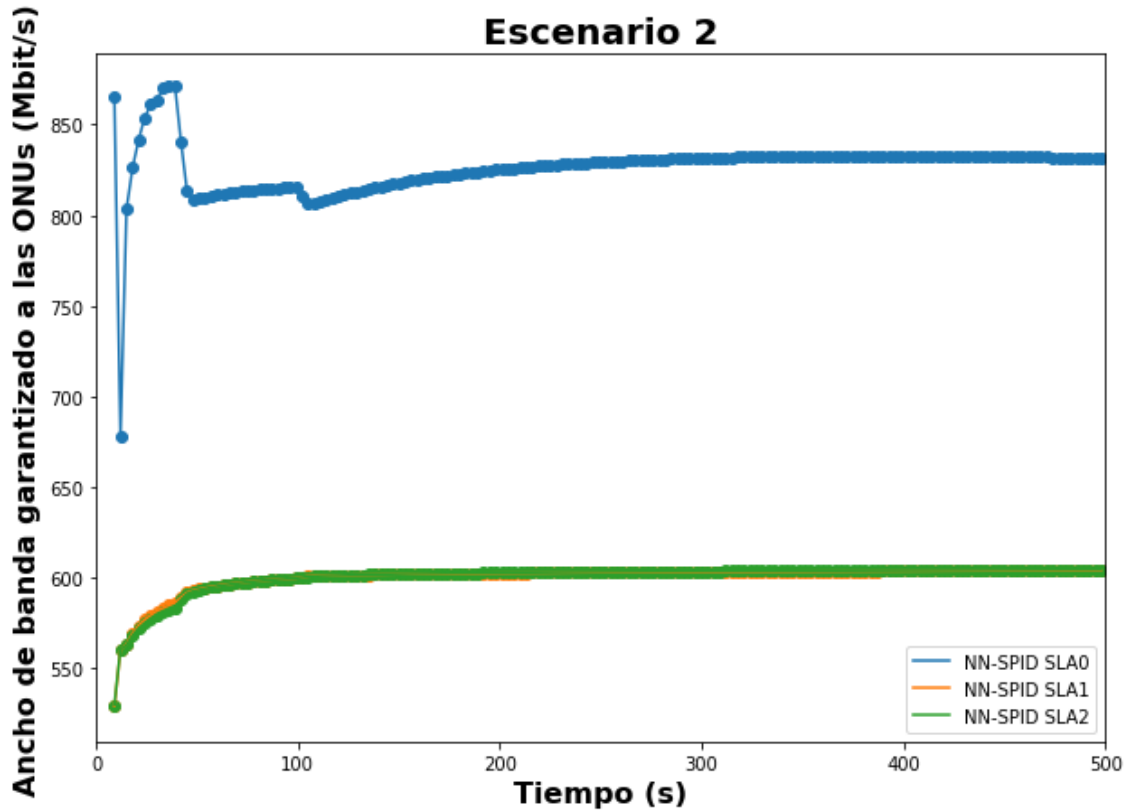


Figura 20. Evolución en tiempo real del ancho de banda garantizado para una ONU perteneciente a cada uno de los tres SLAs con el algoritmo NN-SPID para el escenario 2.

Se muestra, de nuevo, en la Figura 21, la misma gráfica recortada a los primeros 60 segundos con el objetivo de analizar de forma más precisa la evolución que sufren cada uno de los anchos de banda garantizados a las ONUs pertenecientes a cada uno de los 3 SLAs.

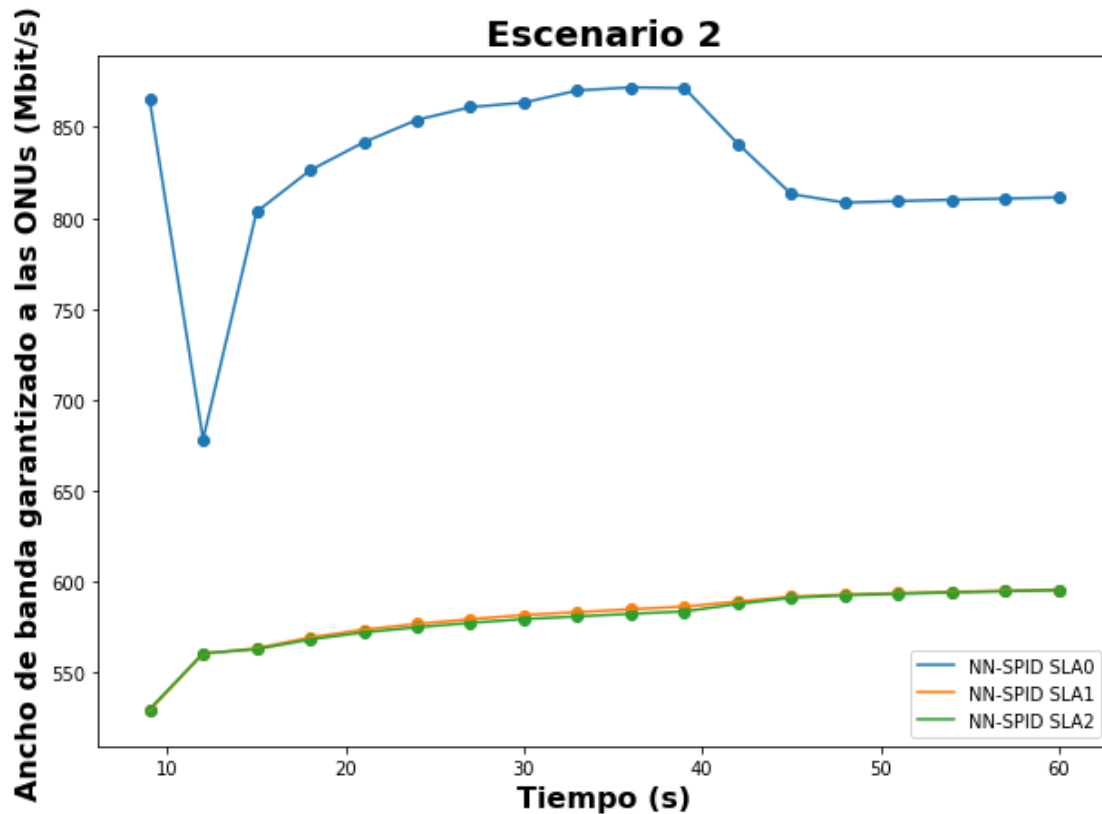


Figura 21. Evolución en tiempo real del ancho de banda garantizado para una ONU perteneciente a cada uno de los tres SLAs con el algoritmo NN-SPID para el escenario 2 (Zoom 60 s).

En las siguientes figuras (Figura 22, Figura 23 y Figura 24), se muestran las gráficas asociadas a la evolución temporal de los valores de los parámetros de sintonización del PID (K_p , T_i y T_d).

En ellas, se puede observar como, de nuevo, tal como sucedía en el primer escenario, las curvas se estabilizan pasado un tiempo cercano a los 100 segundos de simulación. Con ello, se puede concluir que, aunque las condiciones de ancho de banda garantizado para cada SLA sean diferentes, el comportamiento de la red es siempre similar gracias a la capacidad de adaptación de forma dinámica de la red neuronal.

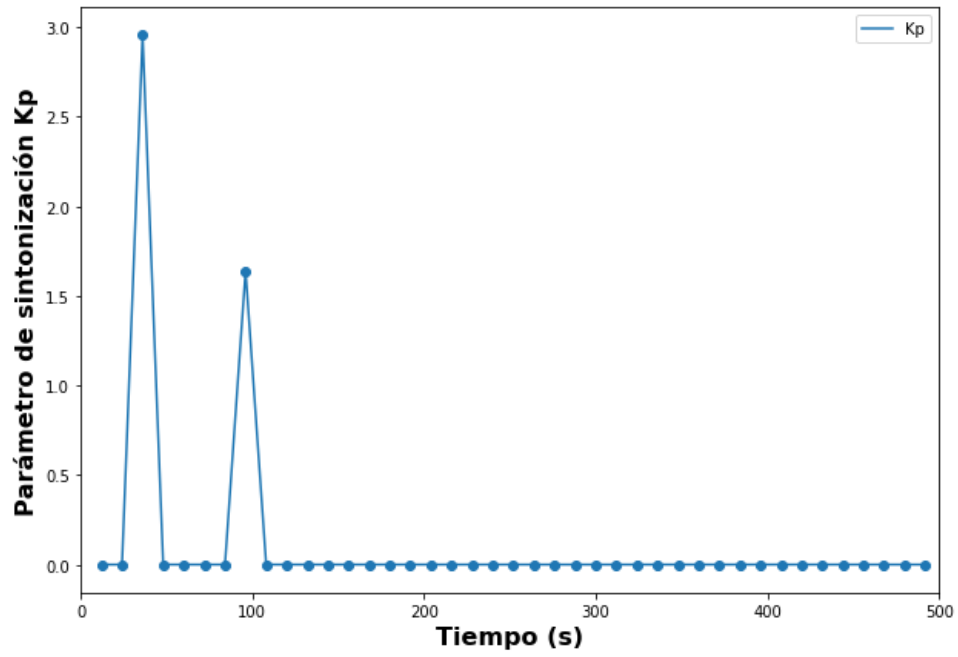


Figura 22. Evolución temporal del parámetro K_p para NN-SPID bajo el escenario 2.

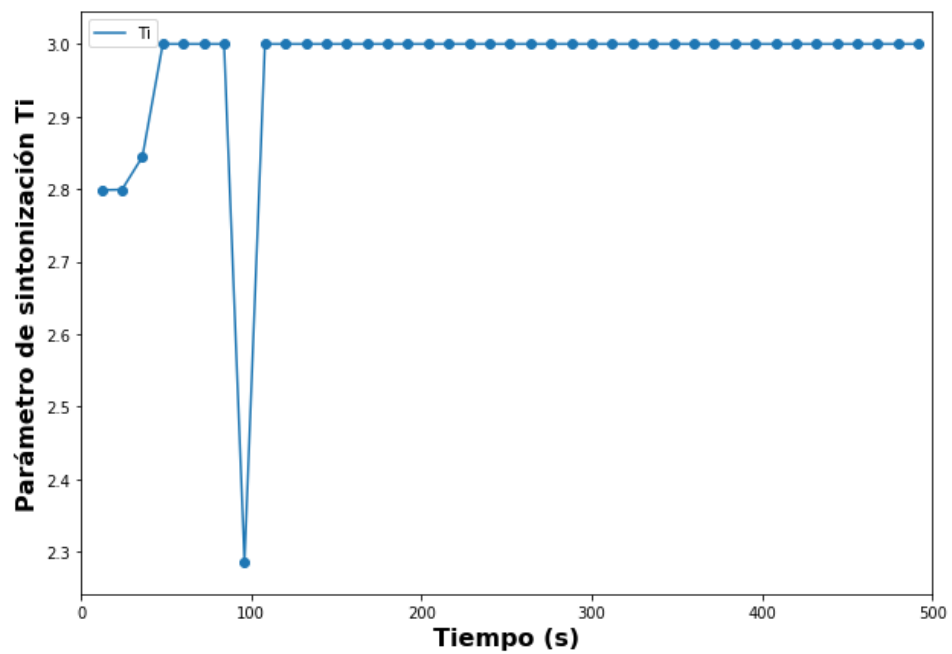


Figura 23. Evolución temporal del parámetro T_i para NN-SPID bajo el escenario 2.

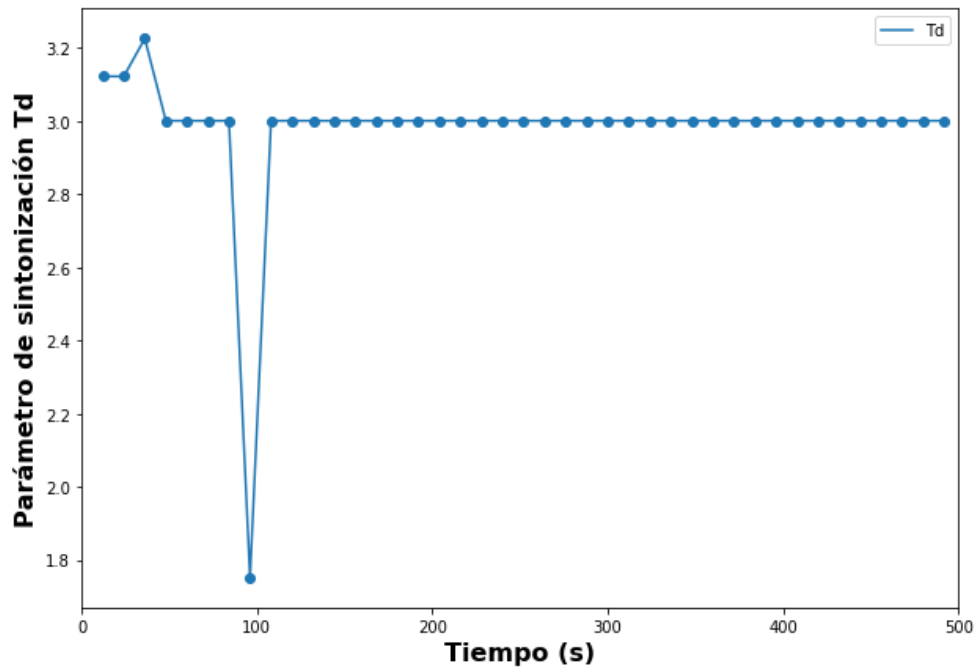


Figura 24. Evolución temporal del parámetro T_d para NN-SPID bajo el escenario 2.

5.5 Conclusiones

En este capítulo de la memoria se ha podido comprobar, a través del simulador de redes PON implementado en OMNeT++, el comportamiento que presenta una red de tipo 10G-EPON. La red del simulador se ha escalado para ofrecer una tasa de transmisión total que ha pasado de ser de 1 Gbit/s a 10 Gbit/s.

Para comprobar como afecta el escalado al comportamiento de la red, se han llevado a cabo simulaciones con tres algoritmos distintos de asignación dinámica de ancho de banda, todos ellos basados en la política de *polling*.

En el primer caso, se ha comenzado con un sencillo algoritmo de *polling*, a modo de introducción y toma de contacto, como lo es IPACT.

En segundo lugar, se ha incorporado un controlador PID al algoritmo con las mejoras que ello conlleva en cuanto a rapidez y estabilidad a la hora de asignar ancho de banda de forma dinámica a cada ONU.

Por último, se ha añadido una red neuronal de tipo *feed-forward* al simulador, con lo que se ha podido comprobar la mejoría que ello supone a la asignación dinámica de

ancho de banda a cada ONU gracias a que la red neuronal trabaja de forma dinámica permitiendo una adaptación más rápida a cambios que puedan producirse en la red.

Por tanto, para estos tres casos, se ha podido comprobar a través de [8], [12] y [13] que el comportamiento exhibido se asemeja al de una red EPON pero en la que, en este caso, se ha escalado la tasa de transmisión a 10 Gbps. Por ello, queda validado el simulador de red 10G-EPON.

6

Integración de un módulo programado en Python al simulador 10G-EPON de OMNeT++

6.1 Introducción

En este capítulo se documentan las acciones y pasos necesarios para incorporar un módulo programado en Python al simulador de redes 10G-EPON con el que se ha estado trabajando en la sección anterior.

El objetivo inicial de añadir un módulo programado en Python es el de comprobar el grado de desarrollo y operabilidad actual de la biblioteca Omnetpy.

El lenguaje de programación Python ofrece numerosas bibliotecas que pueden resultar de gran utilidad en el desarrollo de futuros trabajos basados en OMNeT++ como pueden serlo aquellas bibliotecas asociadas al aprendizaje automático.

Se va a describir, a continuación, los detalles del proceso seguido a la hora de integrar un nuevo módulo programado en Python al simulador 10G-EPON de OMNeT++ y, posteriormente, para finalizar el capítulo, se documentan todas las pruebas que se han llevado a cabo para examinar las limitaciones que pueden presentarse.

6.2 Incorporación del nuevo módulo Python al proyecto OMNeT++

El nuevo módulo Python se va a integrar dentro de la estructura del módulo OLT ya que se pretende que este nuevo módulo actúe como la red neuronal empleada en el

algoritmo NN-SPID y, para ello, debe conectarse al sub-módulo que representa la capa MAC del OLT. Este módulo tendrá una entrada y una salida que permitirán una comunicación bidireccional entre el módulo MAC_OLT y el nuevo módulo de la red neuronal programado en Python.

Con este objetivo, en primer lugar, se llevará a cabo la integración de un módulo de “prueba” programado en Python que posea la estructura mencionada pero con la única funcionalidad de mostrar un mensaje tras la correcta inicialización del módulo al comenzar la simulación.

Para ello, se pasará a describir, a continuación, los ficheros que se han de añadir, así como su contenido y las modificaciones que se han de realizar en ficheros ya existentes en el proyecto del simulador.

6.2.1 Definición del fichero .NED

Se añade el *Network Description File* en el que se ha de indicar las puertas y parámetros que se quiere que posea el nuevo módulo. En este caso, el módulo cuenta con una puerta de entrada y una de salida tal como se muestra en la siguiente captura (Figura 25).

```
package red_wireless.OLT;
simple PyOLT_prueba
{
    parameters:
    gates:
        input pruebaIn;
        output pruebaOut;
}
```

Figura 25. Código del fichero “.ned” del nuevo módulo de prueba.

Con ello, el nuevo módulo se conecta al módulo MAC_OLT a través de la implementación de nuevas conexiones en el fichero del módulo OLT, “OLT.ned”. Para ello, es necesario crear dos nuevas puertas en el módulo MAC_OLT tal como se muestra en la Figura 26.

```
input macrxPruebaIn;  
output mactxPruebaOut;
```

Figura 26. Líneas de código añadidas a la sección “gates:” del fichero MAC_OLT.ned para incorporar dos nuevas puertas al módulo MAC_OLT.

A continuación (Figura 27), se muestran las líneas incorporadas a dicho fichero.

```
olt_prueba.pruebaOut --> olt_mac.macrxPruebaIn;  
olt_mac.mactxPruebaOut --> olt_prueba.pruebaIn;
```

Figura 27. Líneas de código añadidas a la sección “connections:” del fichero OLT.ned para reflejar la conexión bidireccional entre el nuevo módulo de prueba y el módulo MAC_OLT.

Una vez realizadas las conexiones, el nuevo módulo queda conectado al módulo MAC_OLT tal como se muestra en la Figura 28.

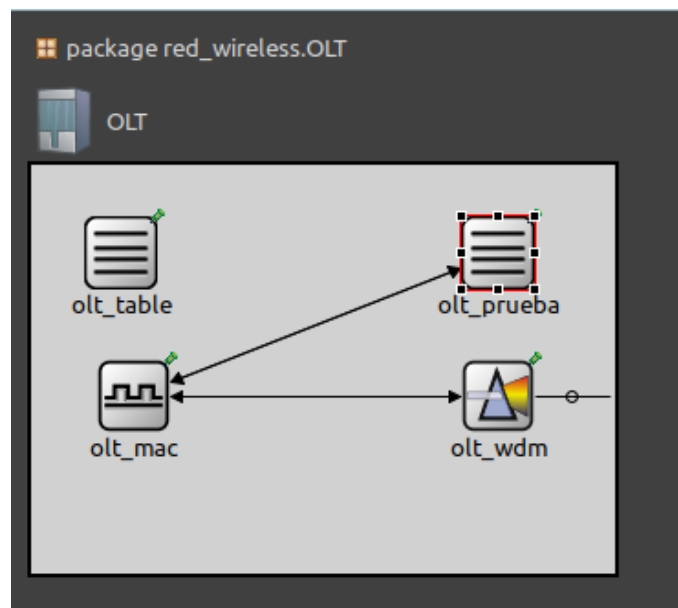


Figura 28. Fichero OLT.ned visualizado en modo *Design* para mostrar la estructura interna del módulo OLT.

6.2.2 Definición del fichero .cc

Este fichero es el único fichero que debe seguir escribiéndose en lenguaje C++ (en lo referente a la incorporación del nuevo módulo). En él, únicamente hay que añadir una llamada al fichero de cabecera de la biblioteca Omnetpy y definir el módulo como en

otras ocasiones pero, esta vez, en lugar de emplear la función “Define_Module”, se llama a “Define_Python_Module” para indicar que el módulo se encuentra en el fichero programado en Python que se agregará más adelante. En la Figura 29 se muestra una captura de este fichero “.cc”.

```
#include </home/gorka/Escritorio/omnetpy-master/omnetpy/include/omnetpy.h>
Define_Python_Module("OLT.OLT_prueba", "PyOLT_prueba");
```

Figura 29. Código del fichero “.cc” del nuevo módulo.

Nótese que la ruta de la primera del código está en formato absoluto aunque bien podría encontrarse como una ruta relativa con el mismo resultado.

Cabe resaltar también, que en el primer argumento de la llamada a “Define_Python_Module” ha sido necesario indicar que el nuevo módulo se encuentra dentro del subdirectorio OLT mediante “OLT.”.

6.2.3 Definición del fichero makefrag

Como ya se anticipaba en el Capítulo 3 de esta memoria, se ha de añadir un fichero, denotado como “makefrag”, al proyecto con las líneas mostradas en la captura de la Figura 30.

```
INCLUDE_PATH += $(shell python3 -m pybind11 --include) -I$(OMNETPY_ROOT)/include
LIBS = -lomnetpy $(shell python3-config --libs --embed | cut -d" " -f1)
LDFLAGS += -L$(OMNETPY_ROOT)/lib
```

Figura 30. Código del fichero “makefrag”.

Dichas líneas serán incorporadas automáticamente al fichero Makefile en el momento de ejecutar la orden make.

6.2.4 Definición del fichero Python

Por último, se añade el fichero de extensión “.py” en el que se puede programar las características y el comportamiento del módulo tal como se hacía originalmente en los ficheros “.cc”.

Para que sea posible hacer uso de métodos tales como `initialize` o `handleMessage`, es necesario importar al proyecto la clase `cSimpleModule` tal que:

```
from pyopp import cSimpleModule
```

Con ello se hace posible la programación en Python de dicha clase ya que la biblioteca se encarga de hacer la traducción entre los métodos que se programen y el código en C++ que entiende OMNeT++.

En caso de ser necesarias otras funcionalidades de las clases de OMNeT++, basta con importar las clases requeridas tales como `cMessage`, `cPacket`, `simTime`...

En una primera instancia, el aspecto del fichero de extensión “.py” creado tiene el aspecto mostrado, a continuación, en la Figura 31.

```
from pyopp import cSimpleModule, EV
class PyOLT_prueba(cSimpleModule):
    def initialize(self):
        EV << "Módulo de prueba programado en Python inicializándose\n"
    def handleMessage(self, msg):
```

Figura 31. Código del fichero de extensión “.py” inicial cuya única función es notificar el momento de su inicialización.

6.3 Pruebas realizadas y limitaciones encontradas

Una vez añadidos todos los ficheros necesarios para la correcta incorporación del módulo Python al proyecto, se dispone de un módulo capaz de inicializarse de forma satisfactoria y correctamente integrado en la red del simulador.

Se procede, entonces, a la realización de diversas pruebas que permitan comprobar el alcance y el estado actual de la biblioteca Omnetpy.

6.3.1 Intercambio de mensajes con el nuevo módulo

La primera prueba que se realiza consiste en el intercambio de mensajes entre el módulo MAC_OLT y el nuevo módulo Python. Con ello, se pretende conseguir que cada módulo pueda interactuar con las variables del otro módulo mediante el contenido de estos mensajes puesto que el acceso directo a variables ha resultado estar limitado.

Se crean, para ello, nuevos mensajes de la clase `cMessage` (en cada uno de los dos módulos) y se procede a su envío mediante el método `send`. Se puede observar, a continuación, en la Figura 32, las líneas de código añadidas al fichero “.py” para el envío de un mensaje de vuelta hacia el módulo MAC_OLT cada vez que se reciba un nuevo mensaje.

```
from pyopp import cSimpleModule, cMessage, EV
class PyOLT_prueba(cSimpleModule):
    def initialize(self):
        EV << "Módulo de prueba programado en Python inicializándose\n"
    def handleMessage(self, msg):
        self.msg = cMessage("mensaje de prueba hacia MAC_OLT")
        self.msg.setKind(15)
        self.send(msg, 'pruebaOut')
```

Figura 32. Código del fichero de extensión “.py” que envía mensajes hacia el módulo MAC_OLT.

En este caso, se comprueba que los mensajes llegan correctamente al otro módulo sin complicaciones.

El siguiente objetivo, entonces, pasa por conseguir añadir parámetros que transmitan la información de las variables de cada módulo que se desee a los mensajes intercambiados.

6.3.2 Integración de parámetros a los mensajes

Existen dos formas de incorporar parámetros a los mensajes en OMNeT++. La primera de ellas, consiste en crear un nuevo tipo de mensaje, es decir, una nueva clase que extienda de la clase `cMessage` (o de la clase `cPacket`) a la que se le añadan parámetros adicionales en los que poder introducir las variables que se quieran transmitir al otro módulo. Para ello, en C++, es necesario crear un fichero con extensión “.msg” en el que se crea la nueva clase con los atributos que se quiere que posea. A partir de dicho fichero, se generan los ficheros “.cc” y “.h” asociados. Es necesario añadir un “include” con la ruta al fichero “.h” del nuevo tipo de mensaje desde el fichero en el que se quiera crear este nuevo tipo de mensaje.

Por su parte, en Python, indagando en la documentación de Omnetpy se puede apreciar que la forma de añadir un nuevo tipo de mensaje cambia ligeramente. En este caso, el fichero que hay que crear es un fichero de extensión “.py”. En él se han de definir las clases que se corresponderán con los nuevos tipos de mensaje que se deseen crear. De esta forma, se pueden transmitir mensajes con parámetros incluidos en ellos entre módulos programados en Python en OMNeT++. El objetivo, entonces, es hacer uso de este método para intentar lograr la transmisión de mensajes con parámetros desde un módulo programado en Python a un módulo de la red programado en C++.

En la Figura 33 se muestra el fichero creado, en este caso, para poder añadir tres nuevos parámetros a los mensajes enviados desde el módulo programado en Python.

```
from pyopp import cPacket

class PRUEBAmsg(cPacket):

    def __init__(self, name, kind=15):
        cPacket.__init__(self, name, kind)
        self.Kp = None
        self.Ti = None
        self.Td = None
```

Figura 33. Código del fichero de extensión “.py” en el que se define el nuevo tipo de mensaje extendiendo de la clase `cPacket`.

Una vez seguidos estos pasos, se intenta crear y transmitir uno de estos nuevos mensajes con un parámetro contenido en ellos. El resultado es que la creación se lleva a

cabo de forma satisfactoria, pero, a la hora de su transmisión, el otro módulo (ya sea el módulo programado en Python como el módulo programado en C++, ya que el comportamiento es el mismo en ambos sentidos de transmisión) no reconoce el tipo de los nuevos mensajes por lo que los parámetros contenidos en ellos se pierden haciendo imposible su extracción en el módulo destino.

El segundo método está ya obsoleto en las versiones actuales de OMNeT++ pero se ha decidido intentar, también, para seguir trazando un cerco en el alcance de las funcionalidades de la biblioteca Omnetpy. Este segundo método consiste en modificar, directamente, el parámetro de la clase `cMessage`, `cParList`. Dicho parámetro estaba originalmente diseñado para incorporar en él distintas variables que se quisieran añadir al mensaje, pero, debido a la ralentización en el tiempo de ejecución que esto suponía, este método fue reemplazado por la creación de nuevas clases de mensaje que extiendan de `cMessage` (o `cPacket`) tal como se ha descrito previamente.

Para añadir nuevas variables a la lista, basta con hacer uso del método `addPar()` tal que:

```
msg->addPar("Variable");
```

Y para asignarle un valor a la nueva variable contenida en el mensaje (supongamos que se le quiere dar a la variable el valor 2,7) se procede tal que:

```
msg->par("Variable").setDoubleValue(2.7);
```

El problema es que, una vez añadida la variable y enviado el mensaje con ella, el otro módulo no consigue recibirla adecuadamente, sino que únicamente recibe un mensaje de tipo `cMessage` con sus atributos habituales como pueden ser el nombre del mensaje, su tipo, etc.

Estas pruebas han sido realizadas tanto en C++ haciendo que el módulo `MAC_OLT` sea el emisor del mensaje como en Python haciendo que el nuevo módulo sea el que transmite el mensaje. Indagando más profundamente para determinar el origen de este comportamiento, se navegó hasta los ficheros de la biblioteca Omnetpy que, en el fichero que realiza las vinculaciones relativas a `cMessage` (Figura 34). Tal y como se aprecia en dicho código se observa que algunas de sus funcionalidades no han sido

implementadas, entre ellas el método `addPar()` necesario para la incorporación de parámetros a los mensajes.

```

"""Bindings related to cMessage."""
from . import _pybind
from . _refstore import _RefStore
from . _utils import no_binding_for_method

class cMessage(_pybind._cMessage):

    def __init__(self, name=None, kind=0):
        _pybind._cMessage.__init__(self, name, kind)
        _RefStore.save(self)

    def __del__(self):
        print('__del__', self.__class__.__name__, self.getName())

    def dup(self):
        copy = super().dup()
        _RefStore.save(copy)
        return copy

    @no_binding_for_method
    def setContextPointer(self, *args, **kwargs):
        pass

    @no_binding_for_method
    def getContextPointer(self, *args, **kwargs):
        pass

    @no_binding_for_method
    def getParList(self, *args, **kwargs):
        pass

    @no_binding_for_method
    def addPar(self, *args, **kwargs):
        pass

    @no_binding_for_method
    def findPar(self, *args, **kwargs):
        pass

    @no_binding_for_method
    def hasPar(self, *args, **kwargs):
        pass

    @no_binding_for_method
    def addObject(self, *args, **kwargs):
        pass

```

Figura 34. Código del fichero `_cmessage.py` localizado en `~/Escritorio/omnetpy-master/omnetpy/bindings/pyopp`.

6.3.3 Envíos retardados

Otra de las pruebas que ha sido probada consiste en la programación de un mensaje para que sea enviado en un instante de simulación determinado, con lo que un módulo podría llevar a cabo una tarea en el instante que se desee.

Este objetivo, en C++, puede hacerse mediante el uso del método `sendDelayed()` con el que se envía un mensaje en un instante de la simulación concreto indicado en uno de sus argumentos.

Otra forma de hacerlo es la utilización del método `scheduleAt()` con el que propio módulo se envía un automensaje en el momento de simulación indicado, de nuevo, en uno de sus argumentos.

El problema surgido al intentar implementar cualquiera de estos dos métodos en Python es que en el momento en que debería llegar el mensaje retardado al módulo Python, se produce un error y la simulación termina de forma abrupta.

Se ha podido comprobar, también, que en el caso de que el retardo indicado (ya sea mediante el uso de `sendDelayed()` o de `scheduleAt()`) sea cero, no se produce ningún error.

6.3.4 Solución de compromiso

Una vez comprobadas las limitaciones de la biblioteca Omnetpy en cuanto a la interoperabilidad de las variables entre un módulo programado en C++ y otro programado en Python, se decide tratar de enviar el contenido de las variables dentro de la cadena del nombre del mensaje de tipo `cMessage`, que se sabe que sí está soportada.

Nótese que las indicaciones que se van a dar a continuación se basan en el caso de que las variables a transmitir sean tres y éstas se correspondan con los parámetros de sintonización del PID (K_p , T_i y T_d) modificados dinámicamente mediante la red neuronal.

Para ello, es necesario convertir las variables que se quieran transmitir en una sola variable de tipo “`const char*`” en el caso de la programación en el módulo en C++

así como la separación, de nuevo, en variables independientes en el módulo Python tal como se muestra en las capturas de la Figura 35 y Figura 36.

```
stringstream ss;
ss << Kp << ' ' << Ti << ' ' << Td;
string s = ss.str();
const char* str = s.c_str();
cMessage *msg15 = new cMessage (str);
```

Figura 35. Código necesario para la incorporación de variables al nombre del mensaje de la clase cMessage en el módulo MAC_OLT.cc.

```
def handleMessage(self, msg):
    datos = re.findall(r"[-+]?[d*\.d+|\d+", msg.getName()) #separo los parametros
```

Figura 36. Código necesario para la extracción de variables del nombre del mensaje de la clase cMessage en el módulo Python.

Por otra parte, en la Figura 37 y Figura 38 se muestra el código necesario para realizar los pasos análogos cuando el mensaje es creado en el módulo Python y recibido por el módulo MAC_OLT.cc.

```
frase = str(newKp) + ' ' + str(newTi) + ' ' + str(newTd)
self.msg = cMessage(frase)
```

Figura 37. Código necesario para la incorporación de variables al nombre del mensaje de la clase cMessage en el módulo Python.

```
stringstream ss(msg->getName());
ss >> Kp;
ss >> Ti;
ss >> Td;
```

Figura 38. Código necesario para la extracción de variables del nombre del mensaje de la clase cMessage en el módulo MAC_OLT.cc.

Con ello, se hace posible el intercambio de mensajes entre un módulo programado en lenguaje Python y otro programado en C++ en el que los mensajes contengan parámetros dentro de la cadena de caracteres del nombre del mensaje.

Sin embargo, durante la implementación de la solución de compromiso descrita, se detectó un nuevo problema asociado al intercambio de mensajes entre el módulo MAC_OLT.cc programado en C++ y el nuevo módulo programado en Python.

Se hace patente que los mensajes que previamente eran transmitidos y recibidos sin ningún problema durante la fase de inicialización de la red (en el instante de simulación cero) provocan que la simulación concluya de forma abrupta cuando éstos son, ahora, enviados en otro instante temporal distinto de cero.

6.4 Propuesta de integración final Python-OMNeT++

Con el objetivo de conseguir un correcto intercambio de mensajes que contengan parámetros entre el nuevo módulo programado en Python y otros módulos del simulador programados en C++, se llevan a cabo unas últimas pruebas que permitan identificar y clasificar de forma más precisa las posibles causas que provocan que las simulaciones aborten cada vez que se intenta realizar este tipo de comunicación entre los módulos.

En primer lugar, se creó un nuevo proyecto en OMNeT++ sencillo que contaba únicamente con dos módulos, uno programado en Python y el otro programado en C++, que se intercambian mensajes con parámetros en ellos. El proyecto se ejecuta correctamente y durante las simulaciones se puede observar el intercambio satisfactorio de los mensajes entre ambos módulos.

Tras llevar a cabo esta prueba, se procedió a crear un nuevo módulo programado en Python en el proyecto simulador de redes 10G-EPON, pero, esta vez, localizado en el nivel más superficial de la red, es decir, a la altura de módulos como el OLT. El resultado se muestra en la Figura 39, y tal y como se observa en nuevo módulo Python ya no está integrado dentro del módulo OLT sino a su misma altura.

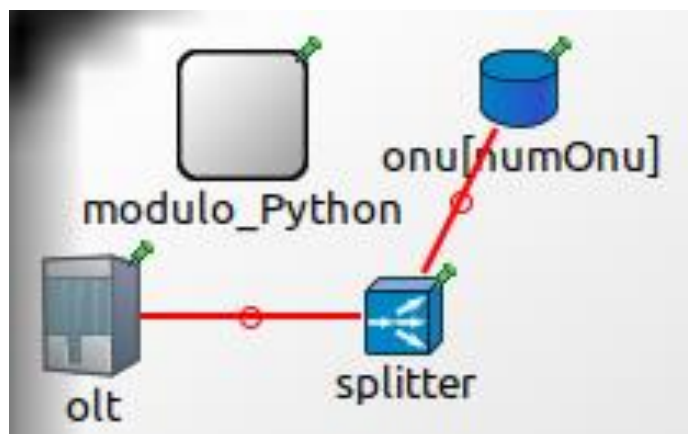


Figura 39. Fichero SFNet_small2.ned visualizado en modo *Design* para mostrar la estructura de la red tras la incorporación del nuevo módulo Python.

Este nuevo módulo se conecta al módulo MAC_OLT (tal como se hacía con el módulo creado previamente en la sección 6.2). En este caso, la conexión no es física ya que ambos módulos se encuentran en diferentes niveles por lo que se comunicarán mediante puertas directas.

Una vez creado el módulo, se realiza una simulación en la que se intercambian mensajes con parámetros (contenidos en la cadena del nombre de cada mensaje) entre ambos módulos a través de mensajes directos y, esta vez, no ocurre ningún problema que haga que la simulación se detenga.

Por ello, esta solución se presenta como la forma viable de integrar un módulo programado en Python en OMNeT++ permitiendo el intercambio de mensajes con parámetros contenidos en ellos bajo las condiciones de que el módulo a integrar se encuentre en el nivel más superficial de la red y que, la información que se quiera agregar a los mensajes transmitidos se incluya dentro de la cadena de caracteres del nombre del mensaje.

En este caso, en el que el módulo programado en Python se encuentre en el nivel más superficial de la red, los otros métodos que se conocen para crear mensajes que contengan parámetros siguen sin funcionar ya que, a la hora de ser transmitidos hacia el otro módulo (ya sea en la transmisión del módulo Python hacia el módulo programado en C++ como en el caso contrario), los parámetros se pierden.

Por otra parte, ahora, sí que funcionan los envíos retardados de mensajes, ya sea en la generación retardada de automensajes del módulo programado en Python a través de `scheduleAt`, como en los mensajes intercambiados entre el módulo Python y el módulo programado en C++ (MAC_OLT).

6.5 Conclusiones

Una vez llevadas a cabo todas las pruebas, se ha podido constatar que el grado de operabilidad de la biblioteca Omnetpy es bastante limitado.

El objetivo inicial de incorporar al simulador de redes 10G-EPON un módulo programado en lenguaje Python con el que poder, por ejemplo, programar la red neuronal empleada en el proyecto, se ha visto limitado por las dificultades encontradas a la hora de enviar información de las variables necesarias de un módulo a otro.

Mediante la solución de compromiso, se ha intentado implementar un “parche” que hubiera permitido, al menos, continuar realizando algunas pruebas más asociadas al intercambio de mensajes con variables contenidas en ellos, pero, una vez probado, se ha observado un nuevo problema asociado a los envíos de mensajes entre el módulo en C++ (MAC_OLT) y el módulo Python. Al parecer, cuando esas transmisiones se producen en un instante temporal de la simulación distinto de cero, se produce una finalización abrupta de la ejecución en ese preciso instante.

Por otra parte, finalmente, se ha llevado a cabo una última prueba en la que el módulo programado en Python se encuentra en el nivel más superficial de la red (a la altura del OLT, las ONUs...) y, en este caso, no ocurre ningún problema a la hora de comunicar este nuevo módulo con el módulo MAC_OLT a través de mensajes directos. Por tanto, esta vía se presenta como la solución a seguir ya que permite la integración de un módulo Python con el que poder intercambiar mensajes que contengan parámetros en ellos, aunque éstos se encuentren incluidos en la cadena del nombre del mensaje. Además, en este caso, se hace posible el uso de envíos retardados de mensajes.

7

Conclusiones y líneas futuras

7.1 Conclusiones

Las principales conclusiones una vez finalizado este Trabajo Fin de Grado están relacionadas con el estado actual de la biblioteca Omnetpy cuya funcionalidad principal radica en la integración de Python en OMNeT++ al permitir la programación de los módulos en este lenguaje.

Tras las numerosas pruebas realizadas a lo largo de este trabajo, se ha podido comprobar que, a día de hoy, la biblioteca aún presenta algunas limitaciones importantes que impiden la correcta integración de módulos Python en un proyecto de cierta envergadura, como así ha sido en el simulador de redes 10G-EPON que tenemos desarrollado en OMNeT++.

Se ha conseguido crear un módulo programado en lenguaje Python que se comunica con otro módulo programado en C++ a través del envío de mensajes, pero, como ya se ha mencionado, con ciertas limitaciones como la dificultad a la hora de incorporar información de variables a dichos mensajes.

También se ha desarrollado un procedimiento que permite la utilización de la biblioteca Omnetpy fuera del contenedor Docker en el que se encontraba inicialmente proporcionada y con versiones diferentes de OMNeT++ a la que se encontraba ligada.

Por otra parte, se ha validado la correcta actualización del simulador 10G-EPON desarrollado en OMNeT++ a través de las pruebas realizadas con un conjunto de algoritmos de asignación dinámica de ancho de banda y escenarios.

Este escalado se ha realizado con el objetivo de adaptarse a la creciente demanda de ancho de banda en la actualidad. Con ello, se hace posible la simulación de nuevos escenarios basados en las redes de tasa de transmisión de 10 Gbit/s.

7.2 Líneas futuras

Una posibilidad que se abre tras la realización de este trabajo, a la vista de los resultados obtenidos, es la profundización en el estudio de la biblioteca Omnetpy, así como en la biblioteca en la que ésta se fundamenta, Pybind11.

Con ello, se podría investigar la forma de habilitar ciertos aspectos de la integración de módulos programados en Python a OMNeT++ que, en el momento de la realización de este trabajo, se encuentran, aún, limitados.

Una forma de abordar esto, podría residir en trabajar en la ampliación de las funcionalidades de la biblioteca Omnetpy, programando ciertas funciones, como la adición de parámetros a la clase cMessage (o cPacket), que no se encuentran, todavía, implementadas.

Por otra parte, al disponer, ahora, de un simulador de redes 10G-EPON, se hace posible el estudio del comportamiento de diferentes algoritmos de asignación dinámica de ancho de banda en redes que transmitan a esta tasa de transmisión aumentada de 10 Gbit/s.

Por supuesto, se abre la puerta al desarrollo de nuevos algoritmos de asignación dinámica de ancho de banda en este tipo de redes basados en técnicas de Machine Learning.

8

Bibliografía

- [1] Omnetpy Repository, [En línea]. Available: <https://github.com/mmodenesi/omnetpy> [Último acceso: 29 agosto 2021]
- [2] Omnetpy Getting Started, «How to create a simulation using Python» [En línea]. Available: <https://github.com/mmodenesi/omnetpy/tree/master/getstarted> [Último acceso: 29 agosto 2021]
- [3] Python Tutorial, [En línea]. Available: <https://docs.python.org/3/tutorial/> [Último acceso: 8 septiembre 2021]
- [4] OMNeT++ main page, [En línea]. Available: <https://omnetpp.org/> [Último acceso: 29 agosto 2021]
- [5] Docker Documentation, «Get started with Docker» [En línea]. Available: <https://docs.docker.com/get-started/overview/> [Último acceso: 29 agosto 2021]
- [6] PyBind repository, [En línea]. Available: <https://github.com/pybind/pybind11> [Último acceso: 29 agosto 2021]
- [7] G. Kramer and G. Pesavento, "Ethernet passive optical network (EPON): building a next-generation optical access network," in *IEEE Communications Magazine*, vol. 40, no. 2, pp. 66-73, Feb. 2002, doi: 10.1109/35.983910.
- [8] Jose María Robledo Sáez (2012), "Implementación de un simulador de

redes de acceso pasivas en OMNeT++”, *Proyecto Fin de Carrera en Ingeniería Técnica de Telecomunicación en Sistemas de Telecomunicación*, E.T.S.I. de Telecomunicación, Universidad de Valladolid.

- [9] Docker Install Guide, «Install Docker Engine On Ubuntu» [En línea]. Available: <https://docs.docker.com/engine/install/ubuntu/> [Último acceso: 29 agosto 2021]
- [10] PyBind11 Documentation, [En línea]. Available: https://pybind11.readthedocs.io/_/downloads/en/latest/pdf/ [Último acceso: 29 agosto 2021]
- [11] OMNeT++ Installation Guide, [En línea]. Available: <https://doc.omnetpp.org/omnetpp/InstallGuide.pdf> [Último acceso: 29 agosto 2021]
- [12] T. Jiménez, N. Merayo, P. Fernández, R.J. Durán, I. de Miguel, R.M. Lorenzo, E.J. Abril (2012). Implementation of a PID controller for the bandwidth assignment in long-reach PONs. *Journal of Optical Communications and Networking*, 4(5), 392-401.
- [13] N. Merayo, D. Juárez, J.C. Aguado, I. de Miguel, R.J. Durán, P. Fernández, P., R.M. Lorenzo, E.J. Abril (2017). PID controller based on a self-adaptive neural network to ensure QoS bandwidth requirements in passive optical networks. *Journal of Optical Communications and Networking*, 9(5), 433-445.
- [14] David Juárez Estévez (2017), «Desarrollo de algoritmos de gestión de recursos en redes PON y NGPON2», Trabajo Fin de Grado, Grado en Ingeniería de Tecnologías de Telecomunicación. E.T.S.I. de Telecomunicación, Universidad de Valladolid. Disponible en: <https://uvadoc.uva.es/handle/10324/27581>