UNIVERSIDAD DE VALLADOLID

E.T.S.I. TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Implementación de un Simulador de Redes de Acceso Ópticas Pasivas en Python

Autor:
Victor Herrezuelo Paredes
Tutora:
Dña. Noemí Merayo Álvarez

TÍTULO: **Implementación de un Simulador de Redes de Acceso Ópticas Pasivas en Python**
AUTOR: **Victor Herrezuelo**
TUTOR: **Dña. Noemí Merayo Álvarez**
DEPARTAMENTO: **Teoría de la Señal y Comunicaciones e Ingeniería Telemática**

TRIBUNAL
PRESIDENTE:
SECRETARIO:
VOCAL:
SUPLENTE:
SUPLENTE:

FECHA:
CALIFICACIÓN:

Resumen

Este Trabajo Fin de Grado presenta un modelo de simulación desarrollado utilizando SimPy para analizar el comportamiento de Redes Ópticas Pasivas (PONs) con el protocolo de Asignación Dinámica de Ancho de Banda Intercalada con Priorización (IPACT).

Se ha utilizado la librería SimPy para modelar los eventos discretos que configuran una simulación de dicha red. En línea con otros trabajos anteriores realizados para esta Escuela, nuestra red tiene una Terminal de Línea Óptica (OLT) y 16 Unidades de Red Ópticas (ONTs).

Se ha modelado el tráfico usando la distribución de Pareto con el fin de simular el tráfico rafagoso que caracteriza este tipo de redes.

Se han realizado barridos de carga en las ONTs para tasas de transmisión de 1 Gbps y 10 Gbps; y para redes tanto con una sola clase de servicio como para redes con tres clases de servicio.

Palabras clave

PON (Red Óptica Pasiva), Python, Simpy, EPON, 10G-EPON, DBA (Asignación de Ancho de Banda Dinámica), IPACT (Asignación Dinámica de Ancho de Banda Intercalada con Priorización)

Abstract

This Bachelor's Thesis presents a simulation model developed using SimPy to analyze the behavior of Passive Optical Networks (PONs) with the Interleaved Polling with Adaptive Cycle Time (IPACT) protocol.

The SimPy library has been used to model the discrete events that configure a simulation of such a network. In line with previous work done for this School, our network has one Optical Line Terminal (OLT) and 16 Optical Network Terminals (ONTs).

Traffic has been modeled using the Pareto distribution in order to simulate the ragged traffic that characterizes this type of network.

Load sweeps have been performed on the ONTs for transmission rates of 1 Gbps and 10 Gbps; and for networks with a single class of service as well as for networks with three classes of service.

Keywords

PON (*Passive Optical Network*), Python, Simpy, EPON, 10G-EPON, DBA (*Dynamic Bandwidth Allocation*), IPACT (*Interleaved Polling with Adaptive Cycle Time*)

Agradecimientos

A mi familia por todo su apoyo incondicional.

A mi pareja, Marina, por estar a mi lado en todo momento.

A mi tutora, Noemí, por su inestimable ayuda, consejos y enseñanzas así como su dedicación a lo largo de la realización de este trabajo.

Índice

Agradecimientos 6
Índice de figuras 9
1 Introducción 12
1.1 Motivación 12
1.2 Objetivos 13
1.2.1 Objetivos Generales 13
1.2.2 Objetivos específicos 13
1.3 Fases y metodología 13
1.3.1 Fase de Análisis 13
1.3.2 Fase de Implementación 14
1.3.3 Fase de Pruebas 14
1.4 Estructura de la Memoria del TFG 14
2 Entorno de Trabajo 16
2.1 Introducción 16
2.2 Python 16
2.3 Librería Simpy 17
2.4 Jupyter 17
2.5 Visual Studio Code 18
2.6 Conclusiones 18
3 Descripción del simulador de redes EPON en Python 20
3.1 Introducción 20
3.2 Redes de Acceso Ópticas Pasivas (PON, Passive Optical Networks) 20
3.3 Simulador de redes EPON en Python 22
3.3.1 Configuración de la simulación: configuration.py y parameters.py 24
3.3.2 Arranque de la simulación: main.py y ejecutar_simulacion.py 25
3.3.3 Clases que definen los diferentes mensajes 27
3.3.3.1 Clase TramaEthernet 27
3.3.3.2 Clase MensajeGate 28
3.3.3.3 Clase MensajeReport 28
3.3.4 Clase Enlace 29
3.3.5 Clase OLT 30
3.3.5.1 Inicialización de variables y de procesos al comienzo de la simulación 30
3.3.5.2 Escucha y procesamiento de reports 31
3.3.6 Clase ONT 32
3.3.6.1 Escucha al splitter y procesado del mensaje recibido 33
3.3.7 Generación de números de Pareto: clase ParetoGenerator 33
3.3.8 Clase GeneraTrafico 38
3.3.8.1 Variables de la clase 38
3.3.8.2 Generadores de paquetes. Inicialización 39
3.3.8.3 Fuente de paquetes de Pareto 40

3.3.8.4 Fuente de paquetes uniforme 41
3.3.8.5 Método de inserción de paquetes: método de prioridad de colas 41
3.3.9 Clase EstadisticasWelford 42
3.4 Conclusiones 43
4 Implementación, simulación y validación de algoritmos DBA en el simulador PON.45
4.1 Introducción 45
4.2 Implementación del algoritmo IPACT (Interleaved Polling with Adaptive Cycle Time) 46
4.2.1 Descripción del algoritmo IPACT 47
4.2.2 Escenario de simulación en una red Ethernet PON (EPON) 48
4.2.2.1 Evaluación de IPACT considerando una clase de servicio 50
4.2.2.2 Evaluación de IPACT considerando 3 clases de servicio 53
4.2.3 Escenario de simulación en una red 10G-EPON 56
4.2.3.1 Evaluación de IPACT considerando 1 clase de servicio 57
4.2.3.2 Evaluación de IPACT considerando 3 clases de servicio 60
4.3 Conclusiones 62
5 Conclusiones y líneas futuras 64
5.1 Conclusiones 64
5.2 Líneas futuras 65
6 Bibliografía 66

Índice de figuras

Figura 1. Esquema básico de una red PON……………………………………………………...21

Figura 2. Distribución de directorios y ficheros en el proyecto………………………………23

Figura 3. Distribución de Pareto utilizada en la publicación de Kramer [5]...........................35

Figura 4. Histogramas correspondientes al barrido de medias……………………………….38

Figura 5. Retardo medio frente a la carga de la ONT para el algoritmo IPACT con un SLA y una sola cola, obtenido en las simulaciones en el trabajo de José María Robledo [15]...........50

Figura 6. Retardo medio frente a la carga de la ONT para el algoritmo IPACT con un SLA y una sola cola, obtenidos en las simulaciones de este trabajo…………………………………51

Figura 7. Tamaño medio de la cola en bytes frente a la carga de la ONT para el algoritmo IPACT con un SLA y una sola cola, obtenido con las simulaciones en el trabajo de José María Robledo [15]..52

Figura 8. Parámetros empleados en las simulaciones del algoritmo de polling IPACT para una tasa de transmisión de 10Gbps (10G-PON)..53

Figura 9. Retardo medio frente a la carga de la ONT para el algoritmo IPACT con un SLA y tres colas, obtenida con las simulaciones en el trabajo de José María Robledo [15]................54

Figura 10. Retardo medio frente a la carga de la ONT para el algoritmo IPACT con un SLA y tres colas, obtenida con las simulaciones en este trabajo……………………………………..54

Figura 11. Tamaño medio de la cola en bytes frente a la carga de la ONT para el algoritmo IPACT con un SLA y tres colas, obtenido con las simulaciones en el trabajo de José María Robledo [15]..55

Figura 12. Retardo medio frente a la carga de la ONT para el algoritmo IPACT con un SLA y una sola cola, obtenido en las simulaciones en el trabajo de Gorka Sainz-Ezquerra [10] 58

Figura 13. Retardo medio frente a la carga de la ONT para el algoritmo IPACT con un SLA y una sola cola, obtenidos en las simulaciones de este trabajo…………………………………58

Figura 14. Tamaño medio de la cola en bytes frente a la carga de la ONT para el algoritmo IPACT con un SLA y una sola cola, obtenido con las simulaciones en el trabajo de Gorka Sainz-Ezquerra [10]..59

Figura 15. Tamaño medio de la cola en bytes frente a la carga de la ONT para el algoritmo IPACT con un SLA y una sola cola, obtenido con las simulaciones en este trabajo…………59

Figura 16. Retardo medio frente a la carga de la ONT para el algoritmo IPACT con un SLA y tres colas, obtenida con las simulaciones en el trabajo de Gorka Sainz-Ezquerra [10]............60

Figura 17. Retardo medio frente a la carga de la ONT para el algoritmo IPACT con un SLA y tres colas, obtenida con las simulaciones en este trabajo……………………………………..61

Figura 18. Tamaño medio de la cola en bytes frente a la carga de la ONT para el algoritmo IPACT con un SLA y tres colas, obtenido con las simulaciones en este trabajo ……………62

Figura 19. Resultados de la prueba de 100s…………………………………..……………63

Índice de tablas

Tabla 1. Campos de una trama Ethernet……………………………………………………28

Tabla 2. Campos de un mensaje GATE ...……………………………………………………29

Tabla 3. Campos de un mensaje REPORT ..…………………………………………………29

Tabla 4. Parámetros empleados en las simulaciones del algoritmo de polling IPACT para una tasa de transmisión de 1Gbps (10G-PON)…………………………………………………....49

Tabla 5. Parámetros empleados en las simulaciones del algoritmo de polling IPACT para una tasa de transmisión de 10Gbps (10G-PON)………………………………………………..57

Tabla 6. Parámetros empleados en la prueba de 100s………………………………………..63

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++[1] . 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.

Concretamente, se ha empleado en este trabajo extensivamente la librería de Python denominada Simpy [2]. Esta librería está basada en procesos y se emplea principalmente para simular eventos discretos.

1.2 Objetivos

1.2.1 Objetivos Generales

Los objetivos generales de este trabajo son dos. En primer lugar, se realizará el diseño e implementación de un simulador de redes EPON (*Ethernet Passive Optical Networks*) en el lenguaje de programación Python.

En segundo lugar, se pretende analizar las prestaciones de diferentes algoritmos tanto en infraestructuras EPON (1G) como en infraestructuras 10G-EPON sobre el simulador desarrollado en Python. Con ello, se estudiará el comportamiento de dichos algoritmos sobre ambas infraestructuras PON (1G, 10G) implementadas en el simulador.

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 Python para desarrollar un simulador de redes PON basado en eventos discretos.

2. Diseñar y desarrollar un simulador de redes EPON a diferentes tasas de transmisión: EPON y 10G-EPON.

3. Seleccionar uno o varios algoritmos DBA (Dynamic Bandwidth Allocation Algorithms) para su implementación en el simulador de Python.

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 la topología y principales características de redes EPON y 10G-EPON.
- Documentación y estudio de la biblioteca Simpy de Python para desarrollar un simulador de eventos discretos.
- Familiarización con el lenguaje de programación Python y diferentes plataformas o frameworks de trabajo.

1.3.2 Fase de Implementación

Durante esta fase se llevará a cabo la instalación de la biblioteca Simpy [1] en Python para desarrollar el simulador de redes EPON.

Una vez instalada y analizada la biblioteca, se procederá con el diseño y programación de un simulador EPON que trabaja a diferentes tasas de transmisión, esto es, redes EPON y redes 10G-EPON. A continuación, se implementarán algoritmos DBA en el simulador EPON desarrollado.

1.3.3 Fase de Pruebas

En esta última fase, se llevarán a cabo diversas pruebas de los algoritmos de asignación dinámica de ancho de banda (DBA) en diversos escenarios de red dentro del simulador de redes PON, tanto para infraestructuras EPON (1G) como en infraestructuras de siguiente generación 10G-EPON.

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 Python.

En el Capítulo 4 se analizarán diferentes escenarios de red y algoritmos simulados en el simulador de Python desarrollado en relación con la asignación dinámica de ancho de banda en redes EPON y 10G-EPON.

El Capítulo 5 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 6 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 enunciar las principales características de Python, que es el lenguaje de programación utilizado en este trabajo.

En segundo lugar, se procede a describir las características de la librería de Python Simpy, sobre la cual se fundamentan las simulaciones realizadas en este trabajo.

Por último, realizaremos una descripción de Jupyter, framework que ha sido utilizado en las fases más tempranas de este trabajo.

2.2 Python

Python es un lenguaje de programación de alto nivel, interpretado, multipropósito y de código abierto. Algunas de las características que lo hacen popular son su sintaxis amigable y legible, su gran variedad de bibliotecas y herramientas disponibles[11].

Se trata de un lenguaje que ofrece flexibilidad en dos aspectos: en primer lugar, se trata de un lenguaje multiplataforma lo que significa que puede ejecutarse en una variedad de sistemas operativos, incluidos Windows, macOS y Linux. En segundo lugar, admite el empleo de diferentes paradigmas de programación[11].

2.3 Librería Simpy

Simpy es una librería en Python mediante cuya principal finalidad es la simulación de eventos discretos. Permite modelar y simular sistemas complejos en los que los eventos ocurren en momentos discretos en el tiempo. Esta biblioteca proporciona las herramientas necesarias para crear modelos de simulación que involucran la interacción de múltiples componentes que se envían mensajes entre sí y comparten el uso de recursos limitados.

Una de las características clave de Simpy es su capacidad para modelar eventos, procesos y recursos. Los procesos representan entidades que realizan actividades a lo largo del tiempo, mientras que los recursos son entidades que pueden ser utilizadas por los procesos para llevar a cabo estas actividades. Los eventos son variables que representan si un proceso ha terminado, o está todavía por terminar. Simpy permite definir la lógica de cómo estos procesos interactúan con los recursos y entre sí, lo que facilita la creación de modelos precisos de sistemas complejos e interconectados.

Simpy se puede utilizar para diversos sistemas, incluidos servidores, colas, sistemas de producción, redes de computadoras y sistemas de transporte, entre otros. Como cualquier otro método de simulación, nos es útil en situaciones donde es difícil o costoso realizar pruebas en el mundo real, o donde se necesita explorar diferentes escenarios sin riesgo. Además, tanto Simpy como Python proporcionan herramientas para recopilar estadísticas durante la simulación, lo que permite analizar el rendimiento del sistema y ahondar más en los detalles de la simulación.

En la Sección *3.3.Simulador de redes EPON en Python* se explica en detalle de qué forma se ha empleado Simpy para modelar la red EPON.

2.4 Jupyter

Jupyter es una aplicación web de código abierto que permite crear y compartir documentos interactivos que contienen código, visualizaciones y texto explicativo. Estos documentos se denominan "notebooks" y pueden contener tanto código ejecutable como elementos descriptivos en formato Markdown[12].

Una de las principales razones por las que hemos utilizado Jupyter en las primeras fases del desarrollo del proyecto es que nos ha permitido visualizar rápidamente diferentes escenarios y diferentes modelos.

Una vez el proyecto fue aumentando en tamaño y complejidad, decidí dejar de utilizar Jupyter para así desarrollar el proyecto en un IDE más complejo que me permitiera desarrollar, interpretar, y debuggear de una forma más precisa y cómoda. El IDE que he escogido ha sido Visual Studio Code.

2.5 Visual Studio Code

Visual Studio Code (VS Code) es un IDE (*Integrated Development Environment*, Entorno de Desarrollo Integrado) desarrollado por Microsoft que ofrece una amplia gama de funcionalidades para programadores. Se destaca por su interfaz de usuario intuitiva, su extensibilidad mediante plugins y su integración con herramientas de desarrollo populares. VS Code es multiplataforma, lo que significa que está disponible para Windows, macOS y Linux[13].

Una de las razones principales por las que VS Code ha sido útil para desarrollar este proyecto es su poderosa capacidad de edición y resaltado de sintaxis para varios lenguajes de programación, incluido Python. Esto facilita la escritura y comprensión del código, ya que resalta la sintaxis de manera visualmente clara, lo que ayuda a identificar errores y mejorar la legibilidad del código.

Por otra parte, permite emplear herramientas de debugging para poder examinar el código en profundidad y así detectar más fácilmente los errores[13].

2.6 Conclusiones

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

Para ello, en primer lugar, se ha descrito el lenguaje de programación empleado, Python, y las razones de la elección del mismo.

En segundo lugar, se han expuesto las principales características de la librería Simpy, sobre la que basa la simulación objeto de este trabajo.

En tercer lugar, se ha mencionado el uso de la aplicación Jupyter y su utilidad al inicio del proceso de desarrollo del proyecto.

En cuarto lugar, se ha explicado el cambio de herramienta de desarrollo a un IDE, concretamente Visual Studio Code, en las fases posteriores del proyecto.

3

Descripción del simulador de redes EPON en Python

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 Python, describiendo, para ello, la topología de la red y la estructura que presentan los módulos que simularán el OLT y cada una de las ONTs.

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 [14].

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* (ver Figura 1). Dicho *splitter* 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 código y en esta memoria se ha empleado el término ONTs (*Optical Network Terminals*) para referirnos a las ONUs, siendo ambos términos intercambiables.

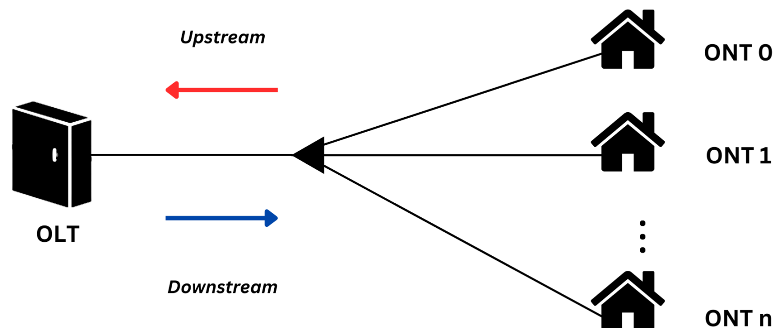

Figura 1. Esquema básico de una red PON

En el canal descendente (desde el OLT hacia las ONTs), la red es punto-multipunto y el OLT se encarga de transmitir información hacia las ONTs. Para ello, el *splitter* reparte dicha información entre todas las ONTs, 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 ONT 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 ONT hacia el OLT), la red es punto a punto y, en ella, la ONT 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 ONTs 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 ONTs 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 ONT 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 ONT al final de cada ciclo (cuando llega el mensaje *Report* de la última ONT 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 ONT de forma independiente a través de un mensaje *Gate* tras conocer la demanda de dicha ONT mediante el mensaje de tipo *Report* recibido de esa ONT 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 Python

Aunque en las fases iniciales del desarrollo el proyecto lo escribí en un único script, en las fases posteriores decidí ordenarlo en forma de proyecto. Con esta finalidad, seguí las recomendaciones de Kenneth Reitz y T. Schlusser en "The Hitchhiker 's Guide to Python" [3]. Ésto me ha permitido organizar el proyecto en un repositorio con una estructura clara y coherente.

Además, presté especial atención a la modularidad del código, separando las funciones y clases relacionadas en módulos y paquetes que reflejan la estructura lógica del proyecto. Esto permitirá una fácil navegación del código, así como una mejor escalabilidad y mantenimiento en el futuro.

Los módulos empleados en el proyecto se pueden ver desglosados en la Figura 2.

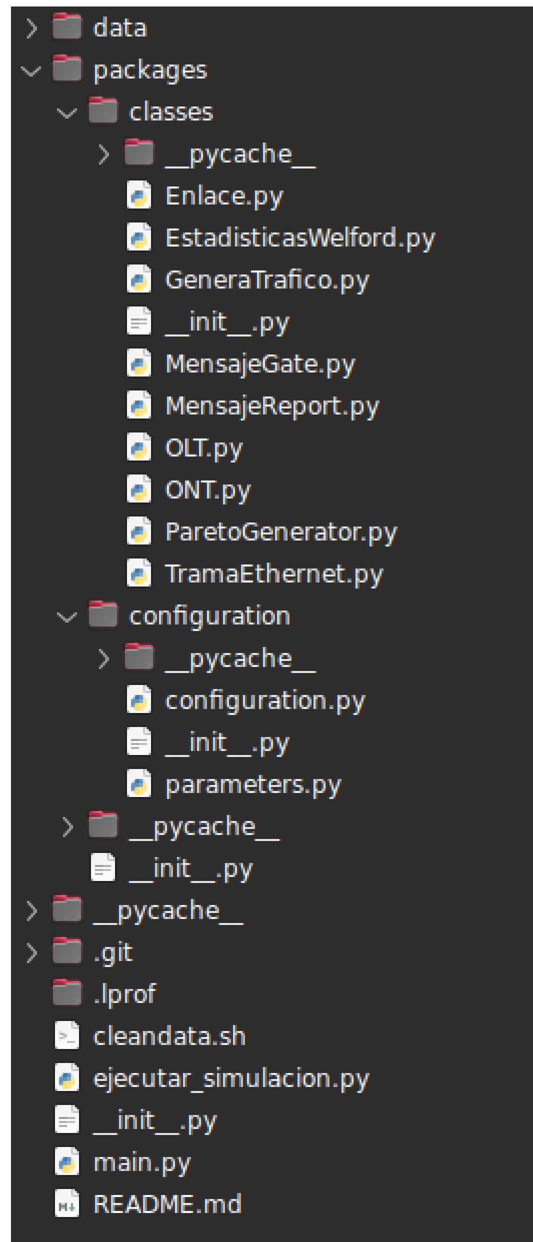

Figura 2. Distribución de directorios y ficheros en el proyecto

En el **directorio raíz** del proyecto, hemos ubicado los scripts que inician y ejecutan la simulación. En el **directorio`./configuration`** se almacenan los ficheros con los que se configura las diferentes características de la simulación. En el **directorio `./classes`** se sitúan los ficheros en los que se declaran las clases que a su vez representan los diferentes elementos de la simulación.

A continuación explicaremos el funcionamiento de los diferentes elementos del simulador.

3.3.1 *Configuración de la simulación:* `configuration.py` *y* `parameters.py`

La intención de utilizar dos ficheros diferentes para la configuración es que el fichero **`configuration.py`** contenga las variables que más frecuentemente haya que editar, mientras que en el fichero **`parameters.py`** se declaran los parámetros que no necesitaríamos ir cambiando con tanta frecuencia para las simulaciones. A continuación se muestra un fragmento de código de dicho fichero de configuración:

```python
# A través de este fichero configuramos rápidamente los parámetros relevantes de la simulación.
CONFIG_T_SIM=10
CONFIG_CARGA=[.1,.2,.3,.4,.5,.6,.7,.8,.9]

## Configuración
multiples_colas=True
insertionmethod_separatequeue0_priorityqueue1=True
watch_on=False
mostrar_progreso=True
mostrar_profiling=True# True = Mostrar estadísticas de profiling al final de la simulación
```

En el fichero **`configuration.py`** se establecen las siguientes variables de configuración:

1. `multiples_colas`: Controla si se usan tres clases de servicio (True) o una sola (False).
2. `insertionmethod_separatequeue0_priorityqueue1`: Decide entre Priority Queueing (`True`) o Separate Queues (`False`) para insertar paquetes.
1. `watch_on`: Activa (`True`) o desactiva (`False`) la observación durante la simulación. A lo largo de los ficheros hay ciertos segmentos que imprimen por pantalla el valor de

las variables si esta variable está activada. Su finalidad ha sido ayudar en el proceso de *debugging* a lo largo del desarrollo del proyecto.

2. `mostrar_progreso`: Determina si se muestra (`True`) o no (`False`) un contador de progreso en pantalla.
3. `mostrar_profiling`: Controla si se muestran (`True`) o no (`False`) estadísticas de profiling al finalizar la simulación.

El segundo archivo, **`parameters.py`**, establece parámetros específicos para la simulación. Define constantes como la velocidad de la luz, el tamaño de la red, las tasas de transmisión, y parámetros relacionados con el tráfico de datos y la simulación del sistema. También configura características como el número de colas en la ONT y el número de fuentes de tráfico de diferentes tipos. Además, utiliza caracteres ANSI para colorear el texto en la salida de la consola.

3.3.2 Arranque de la simulación: `main.py` y `ejecutar_simulacion.py`

El flujo principal de **la función `main()`** consiste en iterar sobre las diferentes cargas de configuración definidas en la lista `CONFIG_CARGA`, donde para cada carga se ejecuta la simulación llamando a la función `ejecutar_simulacion()` con la carga actual como argumento. Se implementa un manejo de excepciones para capturar cualquier error que pueda surgir durante la ejecución de la simulación, mostrando un mensaje descriptivo en caso de que se produzca alguna excepción. De esta manera, el programa gestiona de manera estructurada y controlada la ejecución de la simulación bajo diferentes condiciones de carga.

Por su parte, la **función `ejecutar_simulacion()`** va instanciando los diferentes objetos que son necesarios para ejecutar la simulación para por último representar un resumen de los datos por la consola. A continuación, se detalla paso a paso lo que esta función va realizando:

1. **Inicialización y configuración de la simulación:**
    - 1.1. Inicia un temporizador para medir el tiempo de ejecución. Para ello, se hace uso de la librería `time`.
    - 1.2. Imprime un mensaje indicando el inicio de la simulación y la carga actual.
    - 1.3. Configura la escritura de resultados en un archivo específico.

2. **Creación del entorno de simulación:** Crea una instancia de `simpy.Environment()`. La clase `Environment` de `SimPy` proporciona el entorno en el cual se desarrolla la simulación. Es esencialmente un contenedor para los eventos y procesos que ocurren durante la simulación. Permite controlar el paso del tiempo y la coordinación de las actividades de los distintos componentes simulados.
3. **Configuración de elementos de red:**
    3.1. Declara y configura los enlaces `splitter_downstream` y `splitter_upstream`. Un *splitter* en nuestro modelo de simulación actúa como un enlace bidireccional entre OLT → ONTs (en una dirección) y ONTs → OLTs (en la otra dirección). Es por ésto que debemos instanciar la clase `Enlace` como dos objetos *splitter*, uno para cada dirección.
    3.2. Crea instancias de ONTs y la OLT.
4. **Inicio de la simulación:** Llama al método env.run(until=T_SIM) para ejecutar la simulación hasta el tiempo especificado.

```python
# Iniciamos simulación
env.run(until=T_SIM)
```

5. **Escritura de estadísticas de retardos en un archivo CSV.** Se guardan en un fichero con el formato `retardos-carga_cccc-yyyMMdd_HHss.csv`, done `cccc` representa la carga.
6. **Cálculo y escritura de estadísticas generales de la simulación:**
    6.1. Calcula estadísticas como carga, bytes generados, bytes descartados, paquetes generados, paquetes descartados, etc., para cada ONT.
    6.2. Calcula el tiempo total de ejecución de la simulación.
    6.3. Escribe todas estas estadísticas en un archivo de texto con el nombre `summary-cccc-yyyMMdd_HHss.txt`
7. **Impresión de resumen** de la simulación por la consola.
8. **Imprime en la consola los resultados** de las estadísticas calculadas.
9. **Impresión de estadísticas de profiling** (si está habilitado).

En este fichero, se ha utilizado el *profiler* de Python (librería `pstats`) para medir el tiempo que tarda cada parte de la simulación en ejecutarse. Esto me ha ayudado a identificar qué partes del código causaban unos mayores retardos a la hora de simular. A continuación se muestra un extracto de la salida del *profiler* para una de las simulaciones:

```
Unset
      ## Stats de profiling
         144017533342 function calls in 74151.534 seconds

       Ordered by: internal time

       ncalls tottime percall cumtime percall filename:lineno(function)
      9460060965    7182.169          0.000    70902.540          0.000
/home/victorherrezuelo/.local/lib/python3.9/site-packages/simpy/core.py:181(step)
      6249672843    6712.243          0.000    53240.336          0.000
/home/victorherrezuelo/.local/lib/python3.9/site-packages/simpy/events.py:387(_resume)
      570370410 5100.702  0.000 5100.702  0.000 {built-in method builtins.print}
      9460060965 4689.448  0.000 4689.448  0.000 {built-in method _heapq.heappop}
      9460061538    4309.462          0.000    8378.186          0.000
/home/victorherrezuelo/.local/lib/python3.9/site-packages/simpy/core.py:164(schedule)
                               1    3248.972    3248.972    74151.512    74151.512
/home/victorherrezuelo/.local/lib/python3.9/site-packages/simpy/core.py:206(run)
...
```

3.3.3 Clases que definen los diferentes mensajes

3.3.3.1 Clase `TramaEthernet`

Con esta clase generamos una trama *Ethernet*, que cuenta con los campos mostrados en la Tabla 1 [7]:

Campo	Número de octetos
Preámbulo	3
Id paquete	4
Delimitador de inicio de trama	1
Dirección de destino (MAC)	6
Dirección de origen (MAC)	6

Longitud/Tipo	2
Datos	46-1500
Timestamp	4

Tabla 1. Campos de una trama Ethernet [7]

3.3.3.2 Clase MensajeGate

Con esta clase generamos un mensaje *Gate*, usando los siguientes campos [8]:

Campo	Número de octetos	Descripción
Dirección destino	6	Dirección del destinatario
Dirección origen	6	Dirección del origen
Longitud/Tipo = 88-08	2	Indica longitud o tipo de mensaje
Código de operación = 00-02	2	Indica si es GATE de solicitud, respuesta o confirmación.
Tiempo de creación (timestamp)	4	Tiempo de creación
Number of grants/flags	1	Número de grants o flags
Grant #1 Start time	0/5	Concesiones
Grant #1 Length	0/4	
Grant #2 Start time	0/5	
Grant #2 Length	0/4	
Grant #3 Start time	0/5	
Grant #3 Length	0/4	
Pad / reserved	12	Reservado
FCS	4	Frame Check Sequence

Tabla 2. Campos de un mensaje GATE [8]

3.3.3.3 Clase MensajeReport

Con esta clase generamos un mensaje *Report*, usando los siguientes campos [9]:

Campo	Número de octetos	
Dirección destino	6	Dirección del destinatario
Dirección origen	6	Dirección del origen
Longitud/Tipo = 88-08	2	Indica longitud o tipo de mensaje
Código de operación = 00-02	2	Indica si es GATE de solicitud, respuesta o confirmación.
Tiempo de creación (timestamp)	4	Tiempo de creación
Number of grants/flags	1	Número de grants o flags
Grant #1 Start time	0/5	Concesiones
Grant #1 Length	0/4	
Grant #2 Start time	0/5	
Grant #2 Length	0/4	
Grant #3 Start time	0/5	
Grant #3 Length	0/4	
Pad / reserved	12	Reservado
FCS	4	Frame Check Sequence

Tabla 3. Campos de un mensaje REPORT [9]

3.3.4 Clase `Enlace`

Esta clase simula un enlace de comunicación entre un dispositivo fuente y uno o más dispositivos de destino. Se trata de una abstracción que representa un enlace de comunicaciones con ciertas propiedades. El planteamiento de esta clase, junto con sus métodos enviar/get, se han inspirado en uno de los ejemplos de uso de la clase `Store` que figuran en la documentación de la librería SimPy [4].

En su inicialización, esta clase toma como parámetros:

- La **capacidad** del enlace (número de mensajes que pueden viajar por el enlace al mismo tiempo)
- El **retardo** de propagación.
- El **número de destinatarios**. Si `n_destinatarios=1`, se trata de un enlace punto a punto. Si `n_destinatarios>1`, se trata de un enlace tipo *broadcast*.

La capacidad puede ser configurada como finita o infinita, mientras que el retardo simula el tiempo necesario para que los mensajes viajen desde un extremo a otro.

La función `enviar_con_retardo` añade un retardo antes de almacenar el mensaje en el enlace, emulando así el tiempo de propagación real. La función `enviar` facilita el envío del mensaje al iniciar el proceso de envío con retardo. Por último, la función `get` permite recuperar mensajes del almacenamiento del enlace una vez que han sido recibidos.

3.3.5 *Clase* `OLT`

La clase `OLT` simula una OLT de la red. A continuación se va a describir cada una de las partes de la misma.

3.3.5.1 Inicialización de variables y de procesos al comienzo de la simulación

Al comienzo de la simulación, se declaran las siguientes variables:

- `splitter_in` y `splitter_out`: Representan los enlaces que conectan la OLT con el splitter en las direcciones ascendente (*Upstream*) y descendente (*Downstream*), respectivamente.
- `B_demand`, `B_alloc`, `B_alloc_acum`, `n_alloc`, `T_alloc`, `t_inicio_tx`, `colas_tamanos`, `w_sla`, `B_max`, y `retardos_estadisticas`: Son vectores que almacenan información relacionada con la asignación de recursos que ejecuta la OLT. Estos recursos son, respectivamente, la ventana asignada a cada ONT en bits, el número de veces que se ha asignado una ventana a cada ONT, los tiempos de transmisión asignados a cada ONT, el tiempo de inicio de transmisión de cada ONT, el tamaño de cada cola en bits, las ponderaciones de cada SLA (*Service Level Agreement*, acuerdo de nivel de servicio), el ancho de banda máximo asignado a cada ONT, y las estadísticas de retardo de cada ONT, respectivamente.

A mayores, se inicializa una función `escucha_splitter` para que desde el comienzo de la simulación la OLT escuche y recoja todos los mensajes que le lleguen de la red. Se inicializa también una función que envía un gate inicial (`envia_gate_inicial`)

Al comienzo de la simulación, enviamos mensajes GATE para que las ONTs comiencen a transmitir. Es por esto que cuando se instancia la clase `OLT` se llama al método `enviar_gate_inicial`.

3.3.5.2 Escucha y procesamiento de reports

Al comienzo de la simulación, se invoca al método `escucha_splitter` que, a través de un bucle infinito, escucha al *splitter en* sentido *Upstream* (`splitter_in`). Cuando se recibe una trama, dependiendo del tipo de ésta, se realiza un procesamiento específico: si es un objeto `MensajeReport`, se procede a actualizar el registro en la `OLT` que guarda la cola de cada ONT (llamando al método `procesa_report`). Acto seguido se envía un mensaje GATE (llamando al método `enviar_gate`). Si se trata de un objeto `TramaEthernet`, se contabiliza la trama y su tamaño y se extrae el retardo asociado a ella (llamando al método `extraer_retardo`). Dicho objeto de tipo `TramaEthernet` se sobreescribe cada vez que llega una nueva trama, de esa forma no ocupamos memoria de forma innecesaria.

El método `procesa_report` opera de la siguiente manera. Comienza extrayendo el tamaño de las colas de la ONT desde el mensaje report, así como su identificador único. Luego, recorre cada cola de la ONT y actualiza el registro en la OLT que guarda el tamaño de cada cola. Posteriormente, calcula la demanda total de la ONT sumando los tamaños de todas sus colas. Si la suma total es cero, se asigna un valor predeterminado (por si acaso entran paquetes en las colas de la ONT durante el tiempo de espera en ese ciclo). A continuación, actualiza el ancho de banda y los tiempos de transmisión permitidos a la ONT según el DBA de polling, teniendo en cuenta el tamaño del REPORT. También actualiza el tiempo de inicio de transmisión de la ONT, considerando si necesita esperar a que finalicen las transmisiones de ONTs previas. Dependiendo del caso, se calcula el tiempo de inicio de transmisión de la ONT actual y se actualizan los registros correspondientes en la OLT. Finalmente, devuelve el identificador de la ONT procesada para su posterior uso en el envío de mensajes GATE. A continuación, se muestra el código implementado para realizar dicha funcionalidad:

```python
# Caso A: La ONT no tiene que esperar a que terminen de transmitir ONTs previas
if(self.env.now + tamano_gate/R_tx + T_propagacion > self.t_inicio_tx[ont_id_prev]+self.T_alloc[ont_id_prev]+T_GUARDA):
  self.t_inicio_tx[ont_id]=self.env.now+tamano_gate/R_tx+T_propagacion
  caso='A'
# Caso B: La ONT tiene que esperar a que terminen de transmitir ONTs previas
if(self.env.now + tamano_gate/R_tx + T_propagacion <= self.t_inicio_tx[ont_id_prev]+self.T_alloc[ont_id_prev]+T_GUARDA):
        self.t_inicio_tx[ont_id] = self.t_inicio_tx[ont_id_prev] + self.T_alloc[ont_id_prev]+T_GUARDA
  caso='B'
```

Por su parte, el método `enviar_gate` simplemente encapsula una trama GATE en una instancia de la clase `MensajeGate`. En ella se incluyen los tiempos de inicio y los anchos de banda asignados en `procesa_report`. Dicho mensaje se envía a través del *splitter* hacia las ONTs.

La función `extraer_retardo` simplemente recibe una trama *Ethernet* (clase `TramaEthernet`), calcula el retardo como el tiempo entre que se crea la trama y se recibe en la `OLT`. Actualiza así el retardo medio guardado en la variable `OLT.retardos_estadisticas`. Dicha variable es una instancia de la clase `EstadisticasWelford`, que será explicada a continuación.

3.3.6 Clase `ONT`

La clase ONT modela una ONT de la red. A continuación se va a realizar una descripción de cada uno de sus componentes. Cuando se instancia la clase `ONT` al comienzo de la simulación, se inicia el proceso de escucha al *splitter*.

3.3.6.1 Escucha al splitter y procesado del mensaje recibido

La función escucha_splitter escucha continuamente el *splitter*. Cuando recibe un mensaje se llama a la función `procesa_respuesta`.

La función `procesa_respuesta` es responsable de gestionar la respuesta de una ONT a un mensaje GATE recibido. Primero, verifica si el mensaje GATE está dirigido a la misma ONT que lo ha recibido. Si es así, extrae la asignación de ancho de banda (`B_alloc`) y los tiempos de inicio de transmisión asignados (`t_inicio_tx`). Luego, calcula el tiempo de espera (`t_espera`) antes de que la ONT pueda comenzar a enviar datos. Si este tiempo de espera es positivo, la ONT espera hasta que pueda comenzar a enviar.

Para extraer paquetes, se utiliza el método de *strict priority* (prioridad estricta). Es decir, que se van extrayendo paquetes de la cola empezando por aquellos que tienen una prioridad mayor. La ONT extrae y transmite dichos paquetes hasta que se agote la ventana de asignación de ancho de banda (`B_alloc`). Durante este proceso, se actualiza continuamente el indicador de datos enviados (`B_enviado`). Si la suma de los datos enviados y el tamaño máximo potencial (1500 Bytes de *payload*) del próximo paquete supera la ventana de asignación de ancho de banda, el bucle se interrumpe para evitar exceder la ventana.

```python
def procesa_respuesta(self, env, msg):
    # Procesa el mensaje GATE recibido
    if msg.mac_dst==self.id:
        # Si el mensaje GATE se dirige a la ONT, lo procesa
        # Extraemos los grants obtenidos
        B_alloc = msg.grants_lengths
        # Extraemos los tiempos de espera
        t_initio_tx = msg.grants_start_times
        # Calculamos el tiempo de espera
        t_espera = t_initio_tx - self.env.now
        # Si el tiempo de espera es positivo, esperamos
        if t_espera > 0:
            yield env.timeout(t_espera)
        # Enviamos datos hasta que se agote la ventana
        B_enviado = 0
```

```
        while True:
            # Durante toda la ventana concedida, vamos extrayendo paquetes de
la cola
            paquete_respuesta = self.extrae_paquete_prioridad_estricta()
            # Enviamos el paquete extraído
                        yield  env.process(self.transmite_respuesta(env,
paquete_respuesta))
            # Actualizamos el indicador de datos enviados
            B_enviado += paquete_respuesta.len
                # Aquí se puede añadir una función que mire cuánto tiene el
próximo paquete?
                    if B_enviado + (max(tamano_payload) + tamano_cabecera +
tamano_report) >= B_alloc:
                # Si ya hemos enviado toda la ventana (a falta del report más
un paquete, para no excedernos), salimos del bucle
                break

        # Enviamos mensaje report
        env.process(self.envia_report(env))
        # Esperamos tiempo de guarda
        yield env.timeout(T_GUARDA)
```

Después de enviar los datos, la ONT envía un mensaje *report* y espera un tiempo de guarda (`T_GUARDA`) antes de continuar su operación. Para enviar un *report*, se llama a la función `envia_report`.

Esta función simplemente almacena en un mensaje tipo `MensajeReport` los tamaños de las colas de la ONT y envía dicho *report* para que le llegue a la OLT.

3.3.7 Generación de números de Pareto: clase `ParetoGenerator`

En un trabajo previo a éste, José María Robledo Sáez modeló una red PON usando OMNet++ y reutilizando un generador de tráfico de Pareto diseñado por el profesor Glenn

Kramer en C++[5]. Si bien nuestra intención inicial era utilizar este generador, no hemos podido encontrarlo ya que la página web de Kramer ya no es accesible y sólamente está implementado en C++. Tampoco hemos podido reproducirlo de C++ a Python por la falta de información del simulador y los métodos usados. Por lo tanto, hemos procedido a elaborar un generador de tráfico que siguiera la distribución de Pareto desde cero.

Según un documento extraído de la web personal de Kramer [5], el modelo del que este profesor ha partido para su simulador. En este documento, Kramer explica que se ha demostrado en la literatura que el tráfico de red *Self-Similar* o con dependencia a largo plazo (*Long-Range Dependant*, LRD) puede ser generado mediante la multiplexación de varias fuentes de períodos de actividad (ON) y de inactividad (OFF) distribuidos de manera Pareto. En un contexto de una red conmutada por paquetes, los períodos ON corresponden a trenes de paquetes, es decir, paquetes transmitidos uno tras otro o separados solo por un prefacio relativamente pequeño (como se define en el estándar IEEE 802.3, por ejemplo). Los períodos OFF son los momentos de silencio entre los trenes de paquetes.

Una forma de modelar las múltiples fuentes que contribuyen al tráfico sintético resultante es considerarlas como flujos particulares (conexiones), que en este trabajo denominaremos fuentes o *sources*. A continuación, voy a explicar cómo he diseñado el generador de tráfico con Python partiendo de los parámetros de *media, forma de distribución (a)* y *carga* (L_i).

La librería de Python que vamos a utilizar para caracterizar la distribución de Pareto es `numpy.random.` Tal y como se explica en la documentación de la misma [6], la función densidad de probabilidad en la que se basa es:

$$(1) \qquad p(x) = \frac{am^a}{x^{a+1}}, \qquad x \geq b$$

siendo a la forma de la distribución y m la escala. Cuando $a \leq 2$, la varianza de la distribución es infinita. cuando $a \leq 1$, la esperanza de la distribución es infinita también. Es por eso que para el tráfico *self-similar,* queremos que $1 < a < 2$. En la figura de abajo se muestra la función de densidad de probabilidad en función de diferentes a.

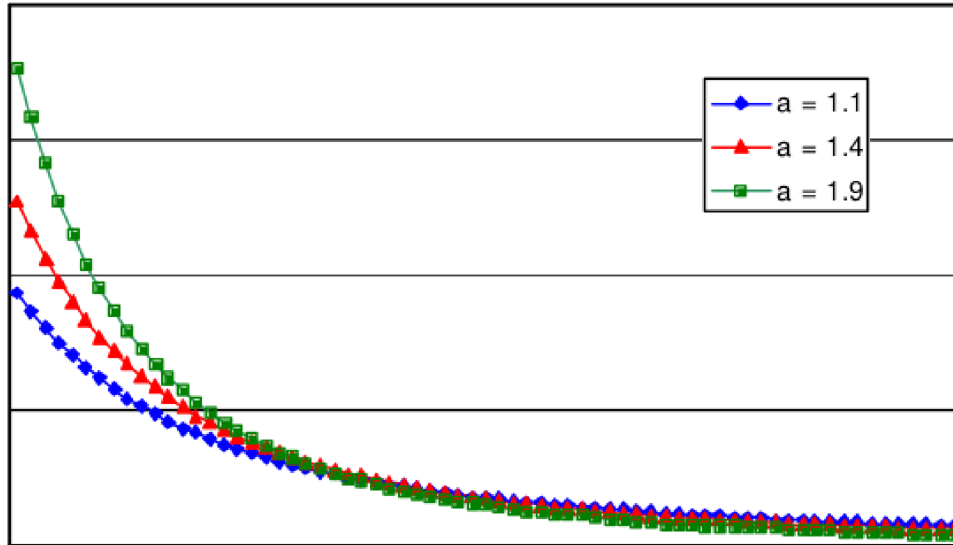

Figura 3. Distribución de Pareto utilizada en la publicación de Kramer [5]

En cuanto a los parámetros, examinemos primero el parámetro de forma, a. En el trabajo de José María, se emplea un parámetro $a = 1,4$ para las ráfagas de paquetes, y un parámetro $a = 1,2$ para los tiempos de silencio. No hay ningún parámetro que defina concretamente la escala, por lo que utilizaremos como dato de input un parámetro `media`, que será la media de la muestra de números de pareto generada.

También deberemos considerar como parámetro de entrada la carga en las ONTs. Kramer define la carga L de la fuente i como la ecuación (2):

$$(2) \qquad L_i = \frac{\overline{ON_i}}{\overline{ON_i}+\overline{OFF_i}}$$

Siendo $\overline{ON_i}$ la duración media de los tiempos de ON (transmisión) y $\overline{OFF}$ la duración media de los tiempos de OFF (silencio). Para $a > 1$, podemos considerar que la media de la distribución (µ) es la que se muestra en la ecuación (3):

$$(3) \qquad \mu = \frac{am}{a-1}$$

Despejando m resulta la ecuación (4):

$$(4) \qquad m = \frac{\mu(a-1)}{a}$$

De aquí, obtenemos el parámetro m para la función de `numpy.random` partiendo de la media. Por lo tanto podemos obtener la carga L_i a partir de la expresión (5):

$$(5) \qquad L_i = \frac{m_{ON}}{m_{OFF} + m_{ON}}$$

Siendo m_{ON} el parámetro escala de los tiempos de transmisión y m_{OFF} el parámetro forma de los tiempos de silencio. Despejando de la anterior ecuación m_{OFF} obtenemos la ecuación (6):

$$(6) \qquad m_{OFF} = m_{ON} \frac{1 - L_i}{L_i}$$

De esta forma, hemos conseguido definir la distribución de Pareto en función de los parámetros *media, forma de distribución (a)* y *carga (L_i).*

En nuestro simulador, la clase `ParetoGenerator` se define en pocas líneas, tal y como se muestra a continuación en el siguiente código:

```
Python
class ParetoGenerator:
  def __init__(self):
    # Constructor de la clase (vacío)
    pass

  def pareto_generator(self,rng, a, m):
    # Generador de números de Pareto
    # a es el parámetro de forma
    # m es el parámetro de escala
    # rng es la semilla
    while True:
      yield (rng.pareto(a) + 1) * m
```

Entre los parámetros generales de la simulación, definidos en el fichero `parameters.py`, tenemos los siguientes:

```python
N_SOURCES = [3, 5, 24]   # Número de Sources de las fuentes de Pareto
a = 1.2

media = 8e-5

# Calculamos m
m = media * (a-1)/a

# Para los periodos de ON.
m_on = m
a_on = 1.4

# Para los periodos de OFF
a_off = 1.2
```

Donde los parámetros $a_{ON} = 1,4$ y $a_{OFF} = 1,2$ son los utilizados por José María en su simulador, y m se obtiene de la ecuación (4).

Para este trabajo hemos escogido la media de 8×10^{-5} a través de un tanteo. Se ha tomado una simulación de 10s de un generador de números de pareto. Tras realizar un barrido de valores para la media en el generador de Pareto, hemos dibujado el histograma de los valores obtenidos y tanteado para observar a simple vista qué valores se asemejan más a la distribución expuesta en la Figura 3. En la Figura 4 se muestran los histogramas del barrido. Un análisis más profundo de dicho parámetro deberá realizarse para el trabajo futuro, pues 8×10^{-4} y 8×10^{-3} también podrían asemejarse bien a este comportamiento.

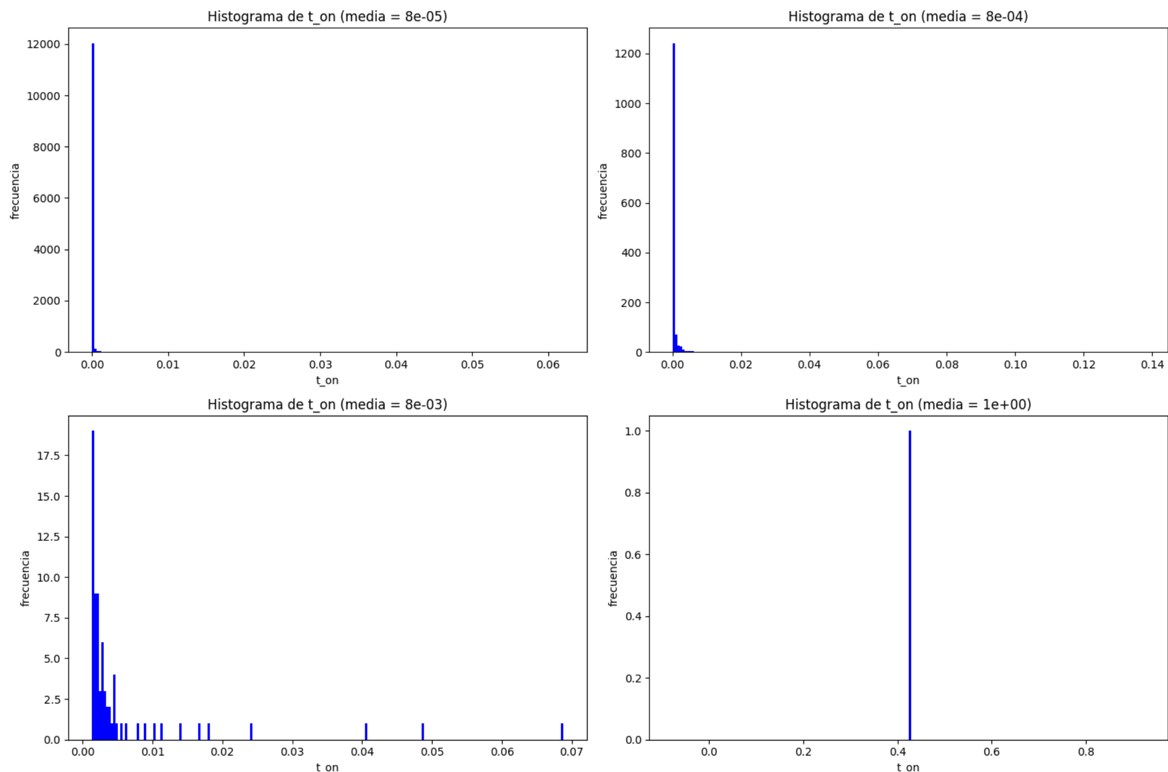

Figura 4. Histogramas correspondientes al barrido de medias.

En el siguiente apartado se explicará cómo se utiliza la clase `ParetoGenerator` para generar el tráfico en las ONTs.

3.3.8 Clase `GeneraTrafico`

Esta clase engloba varios elementos de las ONTs. Por una parte, simula la capa de aplicación que va generando paquetes en las ONTs; por otra, simula las colas de las ONTs, con los correspondientes métodos de inserción (no de extracción) en las mismas. Cada ONT contiene una sola clase `GeneraTrafico` que va simulando todas las colas contenidas en ella.

3.3.8.1 Variables de la clase

Las variables `colas`, `colas_longitudes`, contienen las colas de cada ONT junto con una variable que mide sus longitudes. Las variables `Bytes_generados` y `Bytes_descartados` miden los Bytes generados y descartados en cada ONT. Las

variables `paquetes_generados` y `paquetes_descartados` miden los Bytes generados y descartados en cada ONT.

3.3.8.2 Generadores de paquetes. Inicialización.

Para modelar los generadores, inicializamos la clase con las siguientes líneas:

```python
# Generador de números de pareto
if multiples_colas:
        carga_p1p2 = carga-.0448
self.rng_on = np.random.RandomState(seed_1) # Semilla para el generador de números de Pareto (tiempos de ráfaga)
self.rng_off = np.random.RandomState(seed_2) # Semilla para el generador de números de Pareto (tiempos de silencio)
pareto_generator_on_class = ParetoGenerator() # Clase del generador de números de Pareto (tiempos de ráfaga)
pareto_generator_off_class = ParetoGenerator() # Clase del generador de números de Pareto (tiempos de silencio)
self.generador_pareto_on = pareto_generator_on_class.pareto_generator(self.rng_on, a_on, m_on) # Instancia del generador de números de Pareto (tiempos de ráfaga)
self.generador_pareto_off = pareto_generator_off_class.pareto_generator(self.rng_off, a_off, m_on*(1-carga_p1p2)/carga_p1p2) # Instancia del generador de números de Pareto (tiempos de silencio)
```

En caso de que tengamos tres clases de servicio (`multiples_colas=True`), según el modelo de José María, tendríamos que la clase P_0 genera un tráfico constante, de 4.48 Mbit/s (paquetes de 70 Bytes, cabecera incluida, cada 125 microsegundos). El resto del tráfico se repartiría entre las dos clases restantes (P_1 y P_2) y a partes iguales. El tráfico P_0 es el más prioritario de la red, seguido en orden de prioridad por P_1 y P_2. De ahí el bucle de las primeras líneas del código del cuadro anterior.

A continuación, se inicializan las semillas aleatorias para los generadores de números de Pareto para los tiempos de ráfaga y los tiempos de silencio. En el script `ejecutar_simulacion.py` se extrae la semilla de las siguientes líneas:

```python
        # Creamos las ONTs y la OLT
        for i in range(N_ONTS):
```

```
                              capas_app_ont.append(GeneraTrafico(env,  i,  carga,
i*datetime.utcnow().microsecond // 1000))
                   onts.append(ONT(env, i, capas_app_ont[i], splitter_downstream,
splitter_upstream))
          olt = OLT(env, splitter_upstream, splitter_downstream)
```

Es decir, se multiplica en bucle la hora de ejecución del script en microsegundos y se extraen las últimas tres cifras. De esta forma generamos números pseudo aleatorios para nuestra simulación.

A continuación, se crea una instancia de la clase `ParetoGenerator` para los tiempos de ráfaga y otra para los tiempos de silencio. Por último, se llama al método `pareto_generator` de cada instancia de `ParetoGenerator` para generar secuencias de números de Pareto. Estos números representan los tiempos de ráfaga y los tiempos de silencio respectivamente, siguiendo la distribución de Pareto. Se puede observar que para los tiempos de OFF, tenemos que el parámetro de escala es `m_on*(1-carga_p1p2)/carga_p1p2`, de acuerdo con la ecuación (6). La variable es la suma de las cargas de las clases de servicio P_1 y P_2.

3.3.8.3 Fuente de paquetes de Pareto

Este método utiliza los generadores de paquete anteriormente instanciados para repartir de forma uniforme la carga entre las prioridades 1 y 2.

Para ello, se realiza una iteración perezosa (*lazy iteration*) dentro de un bucle `while` sobre el generador para sacar un tiempo de ráfaga en bruto (`i_on_rough`). A partir de ahí sacamos el número de bytes que podemos transmitir en ese tiempo de ráfaga tal y como se muestra a continuación en el siguiente código:

```python
        # Calculamos tiempo de ráfaga
        i_on_rough = next(self.generador_pareto_on)
        self.i_on_total.append(i_on_rough)

        # Calculamos tamaño de la ráfaga en Bytes
```

```
tamano_rafaga = int(i_on_rough/((tamano_cabecera+tam_paq)/R_datos))
```

De aquí se van generando tramas *Ethernet* y se van insertando en las colas según el método de prioridad de colas.

Por último se van insertando tiempos de silencio, en los que no se generan tramas sino que simplemente se realiza un `self.env.timeout` en función de la configuración de la simulación.

3.3.8.4 Fuente de paquetes uniforme

El método `generador_uniforme_paquetes` se encarga de generar paquetes de manera uniforme en el tiempo, correspondientes a la clase P_0. La frecuencia y el tamaño de los paquetes son constantes. Dentro de un bucle infinito, se espera un tiempo fijo de 125 microsegundos antes de generar cada paquete. El tamaño del *payload* del paquete se calcula restando el tamaño de la cabecera del tamaño total de un paquete de 70 Bytes.

3.3.8.5 Método de inserción de paquetes: método de prioridad de colas

El método `encolador_prioridad_colas` se encarga de la inserción o descarte de los paquetes en las colas una vez se han generado.

Este método recoge el paquete generado. Si la suma de todas las longitudes de colas, más la longitud del paquete, no supera el máximo, el paquete se añade en la cola correspondiente a su prioridad.

En cambio, si dicha suma supera el máximo, se pueden producir diferentes situaciones:

a) Si el mensaje que llega es de menor prioridad de todos, borramos dicho paquete ya que no entra en la cola.
b) Si el mensaje que llega es de prioridad mayor que la prioridad más baja, comprobamos si hay paquetes desde la menor prioridad hasta la prioridad del paquete a insertar:

i) Si hay paquetes en la cola de menor prioridad, comprobamos que la cola tiene suficientes paquetes como para acomodar el tamaño de dicho paquete. En caso negativo descartamos el paquete, en caso afirmativo vamos borrando paquetes del final de la cola hasta que podamos insertar el paquete generado en la cola correspondiente.

ii) Si, tras borrar los paquetes de esta cola de prioridad menor, no podemos aún insertar el paquete generado, pasamos a la cola inmediatamente superior a la actual. Seguimos borrando paquetes del final de la cola hasta que podamos insertar el paquete en alguna cola.

iii) Si tras borrar todos los paquetes de prioridad inferior a la del paquete, seguimos sin tener espacio suficiente, el paquete queda descartado.

3.3.9 *Clase* `EstadisticasWelford`

La clase `EstadisticasWelford` calcula la media, la varianza, la desviación estándar y el intervalo de confianza de una secuencia de números utilizando el algoritmo de Welford. Este algoritmo es una forma de calcular la media y la varianza de una secuencia de números con un barrido sobre los datos, lo que lo hace útil para el cálculo en tiempo real. En lugar de mantener todas las muestras, el algoritmo actualiza la media y la varianza a medida que cada muestra nueva se recibe. Esto lo logra manteniendo una suma acumulada de las diferencias entre cada muestra y la media actual, y utiliza esta información para calcular la varianza al final.

A continuación se muestra la implementación de dicho algoritmo, basada en [16]:

```python
from scipy.stats import norm

class EstadisticasWelford:
# Va calculando la media y la varianza de una secuencia de números
# usando el algoritmo de Welford.
def __init__(self):
  self.n = 0 # Número de muestras
  self.media = 0.0 # Va acumulando la media
  self.M2 = 0.0 # Agrega el cuadrado de las diferencias
```

```
def actualizar(self, x):
  # Actualiza la media y la varianza con una nueva muestra
  self.n += 1
  delta = x - self.media
  self.media += delta / self.n
  delta2 = x - self.media
  self.M2 += delta * delta2

def varianza(self):
  # Devuelve la varianza de las muestras
  if self.n < 2:
    return float('nan')
  return self.M2 / self.n

def desviacion_tipica(self):
  # Devuelve la desviación típica de las muestras
  return self.varianza()**0.5

def intervalo_confianza(self, nivel_confianza=.95):
  # Devuelve el intervalo de confianza de las muestras
  # Calculamos el nivel de significación
  alpha = 1.0 - nivel_confianza
  # Calculamos el Valor crítico
  valor_critico = norm.ppf(1.0 - alpha / 2.0)
  # Calculamos el intervalo de confianza
     confidence_interval          =          (self.media          -
valor_critico*self.desviacion_tipica()/self.n,          self.media          +
valor_critico*self.desviacion_tipica()/self.n)
  return confidence_interval
```

3.4 Conclusiones

En este capítulo de la memoria se ha realizado una descripción de una Red Óptica Pasiva (PON). Se ha explicado también la topología en árbol de la red y el uso de TDMA en la misma.

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

Implementación, simulación y validación de algoritmos DBA en el simulador PON

4.1 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 de acceso PON (Passive Optical Networks) desarrollado en Python.

En el presente capítulo, dicho simulador ha sido diseñado y programado para poder simular tanto redes EPON (1 Gbps) como redes de siguiente generación 10G-EPON (10nGpbs), es decir, redes en las que la tasa de transmisión es de 1 Gbps y 10 Gbps simétricos.

El estudio se va a llevar a cabo sobre el algoritmo de *polling* IPACT (Interleaved Polling with Adaptive Cycle Time), un algoritmo ampliamente conocido e implementado como política de asignación dinámica de ancho de banda, Dynamic Bandwidth Allocation (DBA), en redes EPON (Ethernet PON). La implentación se ha hecho también considerando en ambos casos diferentes tipos de servicios en las ONTs.

En primer lugar, se va a comenzar describiendo de forma breve el algoritmo 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. Cada uno de los escenarios será comparado con trabajos anteriores. Concretamente:

- Para los escenarios de tasa de transmisión de la red 1 Gbps, se va a comparar con el trabajo de José María Robledo Sáez [15]. En dicho proyecto, los resultados se obtuvieron con el simulador de redes OMNeT++ .
- Para los escenarios de tasa de transmisión de la red 10 Gbps, se va a comparar con el trabajo de Gorka Sainz-Ezquerra Calvo [10]. En dicho proyecto, los resultados se obtuvieron usando OMNeT++.

4.2 Implementación del algoritmo IPACT (*Interleaved Polling with Adaptive Cycle Time*)

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 ONT.

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 ONT 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 ONT de cara a la simulación. La carga que le corresponde a cada ONT se define como el cociente entre la tasa media de transmisión de una ONT y la tasa de transmisión máxima que se le permite a un abonado conectado a dicha ONT. Dicho criterio para caracterizar la carga es el mismo que ha sido utilizado en los trabajos de José María Robledo [15] y de Gorka Sainz-Ezquerra [10].

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: $(70\times8)/(12.5\times10^{-6}) = 44.8$ Mbit/s. Dicha tasa se divide entre los 1000 Mbit/s de tasa de transmisión (red PON) 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 ONT 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 ONTs 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 ONT 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.

4.2.1 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 ONTs así como el instante de tiempo en el que cada ONT 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 ONTs. En ese momento, el algoritmo tiene en cuenta el ancho de banda demandado por la ONT y el ancho de banda máximo que se le puede asignar a esa ONT. Este último parámetro es fijo y depende del SLA (*Sevice Level Agreement*) al que esté asociada esa ONT. En este caso, únicamente hay un SLA para todas las ONTs 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 ONT.

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

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

Además de la asignación del ancho de banda, a cada ONT 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 ONT inmediatamente anterior (ya que todas las ONTs 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, 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 ONT anterior por lo que el instante de inicio de transmisión siguiente para la ONT viene dado por dicha suma.

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

4.2.2 Escenario de simulación en una red Ethernet PON (EPON)

Se describen a continuación 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.

En dichas simulaciones, se va a barrer la carga de cada ONT 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.

Los parámetros detallados de la simulación han sido los siguientes:

Parámetros de Simulación	Valores
Número de ONTs	16 ONTs
Número de longitudes de onda	1 longitud de onda
Tasa de transmisión de la red	1 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 *pon1* (OLT-ONT)	20 km
Tamaño de *buffer*	10 Mbytes
Algoritmo implementado	Algoritmo de *polling* 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 *streams*	32 *streams*
Número de servicios de prioridad (número de colas)	- 1 (P_0) para la primera variante del escenario - 3 (P_0, P_1 y P_2) para la segunda variante del escenario

Tabla 4. Parámetros empleados en las simulaciones del algoritmo de polling IPACT

4.2.2.1 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 las Figura 5 y 6, se analiza la evolución del retardo medio que sufren las tramas Ethernet desde que son generadas en cada ONT hasta que son recibidas en el OLT respecto a la carga de la ONT.

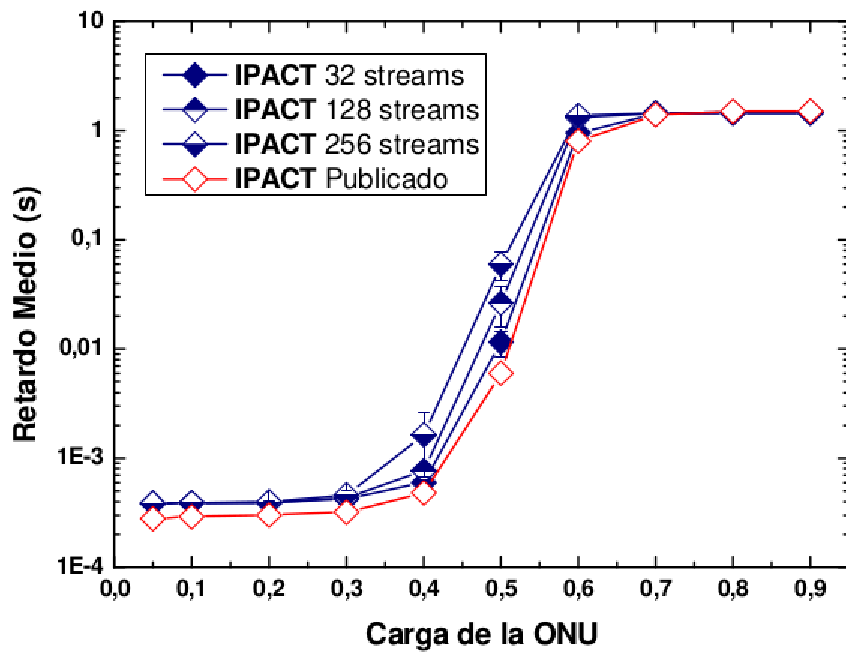

Figura 5. Retardo medio frente a la carga de la ONT para el algoritmo IPACT con un SLA y una sola cola, obtenido en las simulaciones en el trabajo de José María Robledo [15]

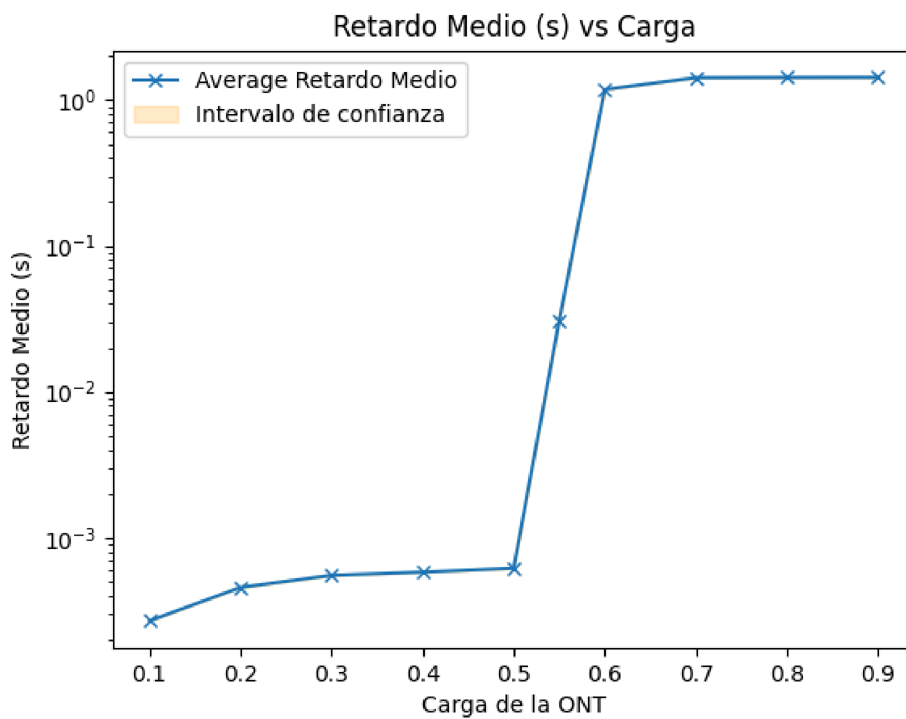

Figura 6. Retardo medio frente a la carga de la ONT para el algoritmo IPACT con un SLA y una sola cola, obtenidos en las simulaciones de este trabajo

Si se compara esta gráfica con la obtenida en el trabajo de José María Robledo [15], los resultados obtenidos entre 0.1 y 0.3 están en el mismo orden de magnitud (10^{-3}s) en ambas gráficas. Sin embargo, los resultados entre 0.4 y 0.6 no son los mismos, ya que en las simulaciones de este trabajo, el aumento del retardo medio se produce a unas cargas mayores y de forma algo más brusca. En cambio, entre los valores de carga entre 0.7 y 0.9, ambas gráficas tienen una magnitud similar (en torno a 1s).

Por otra parte, se presenta también, a continuación (Figuras 7 y 8), la gráfica resultante del análisis de la cantidad media de bytes que quedan en las colas de una ONT tras cada ciclo de 2 ms frente a la carga de la ONT.

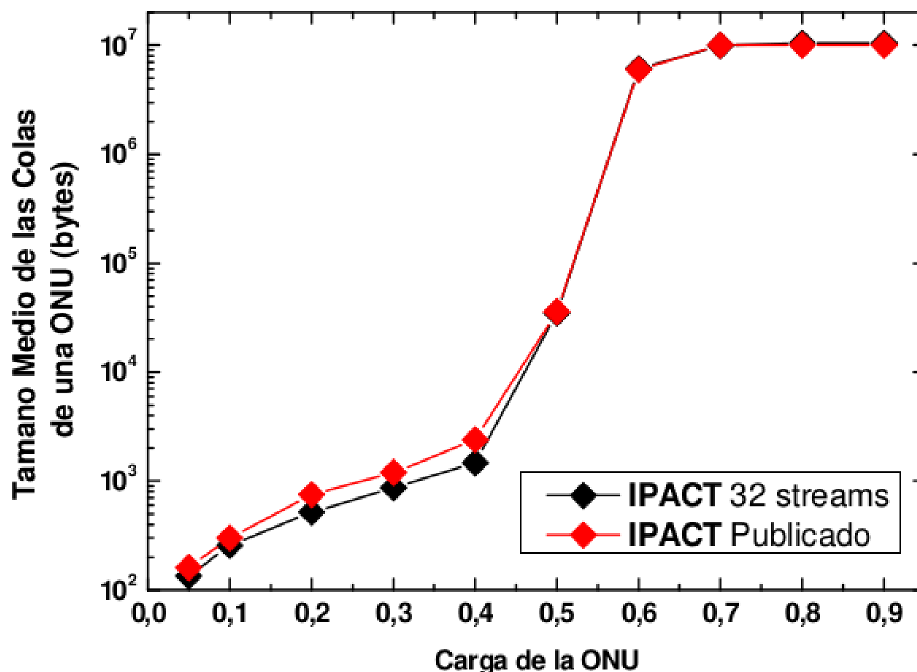

Figura 7. Tamaño medio de la cola en bytes frente a la carga de la ONT para el algoritmo IPACT con un SLA y una sola cola, obtenido con las simulaciones en el trabajo de José María Robledo [15].

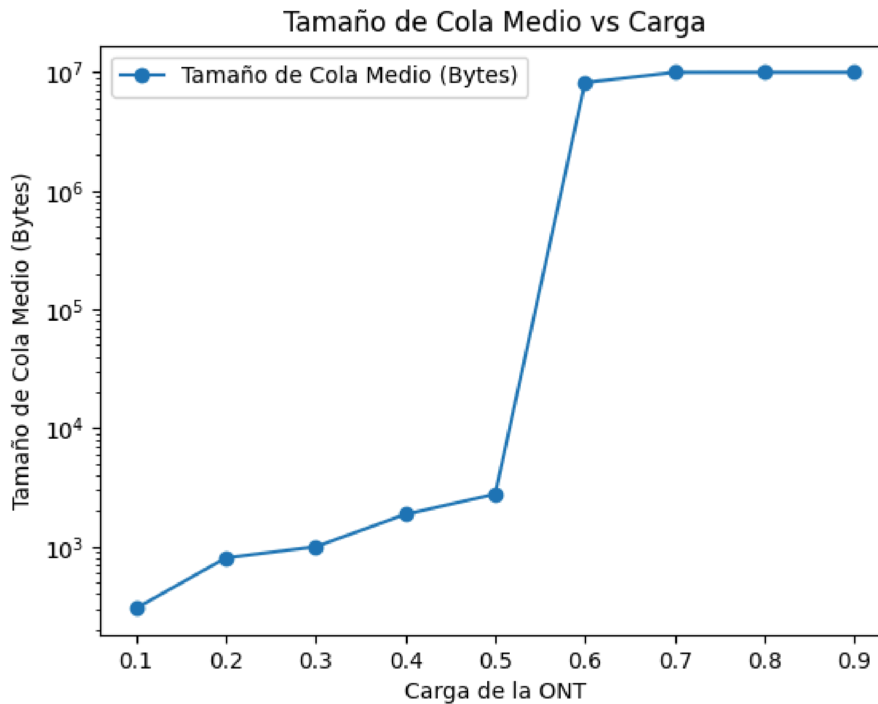

Figura 8. Parámetros empleados en las simulaciones del algoritmo de polling IPACT para una tasa de transmisión de 10Gbps (10G-PON)

Podemos observar que el aumento brusco del tamaño de colas en función de la carga se produce en nuestro caso entre 0.5 y 0.6, mientras que en el trabajo de José María Robledo [15] se produce entre 0.4 y 0.6.

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 ONT.

4.2.2.2 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. El resto de carga se distribuye a partes iguales entre P_1 y P_2

Se muestra, a continuación (Figuras 9 y 10), la gráfica que recoge el retardo medio que presentan cada una de las tres colas de una de las ONTs. 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 el trabajo de José María Robledo [15]. Esto reafirma la correcta implementación del escalado de la red a 10G-EPON.

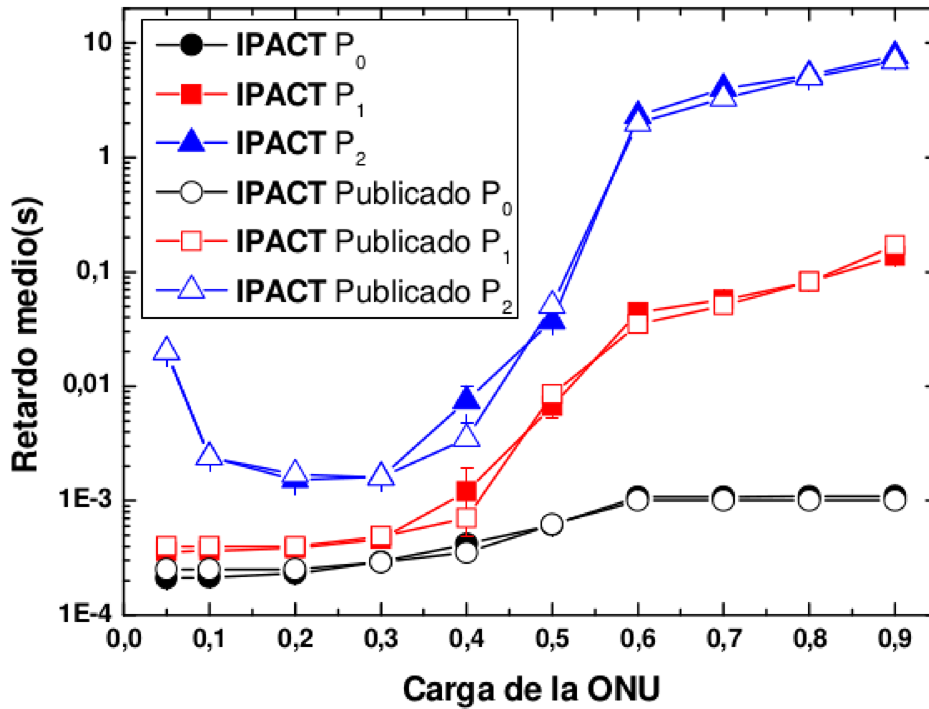

Figura 9. Retardo medio frente a la carga de la ONT para el algoritmo IPACT con un SLA y tres colas, obtenida con las simulaciones en el trabajo de José María Robledo [15].

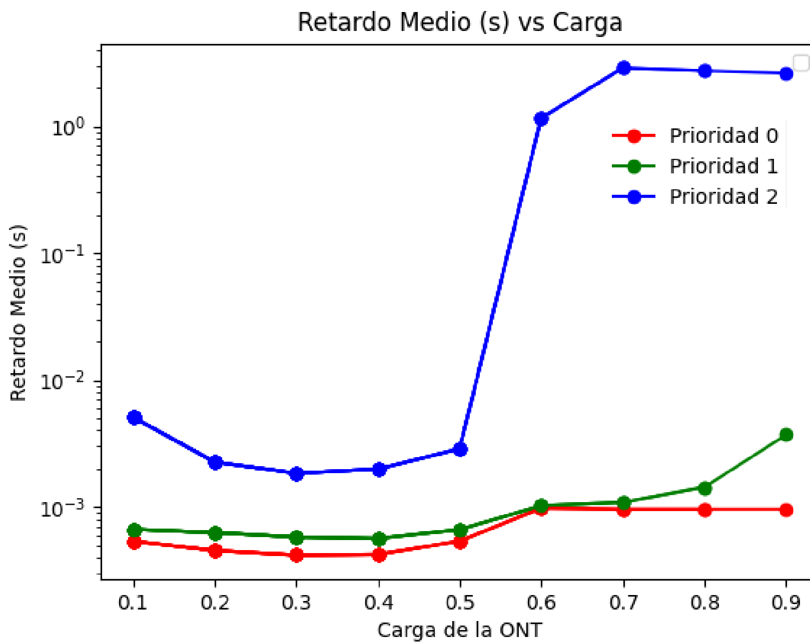

Figura 10. Retardo medio frente a la carga de la ONT para el algoritmo IPACT con un SLA y tres colas, obtenida con las simulaciones en este trabajo

Si comparamos nuestra gráfica con la obtenida en el trabajo de José María Robledo [15], obtenemos unas curvas para P_0 y P_2 muy parecidas. Sin embargo, la curva P_1 cae inesperadamente y se encuentra un orden por debajo de lo esperado para cargas superiores a 0.4.

Por otra parte, se presenta también, a continuación (Figura 11), la gráfica resultante del análisis de la cantidad media de bytes que quedan en las colas de una ONT tras cada ciclo de 2 ms frente a la carga de la ONT.

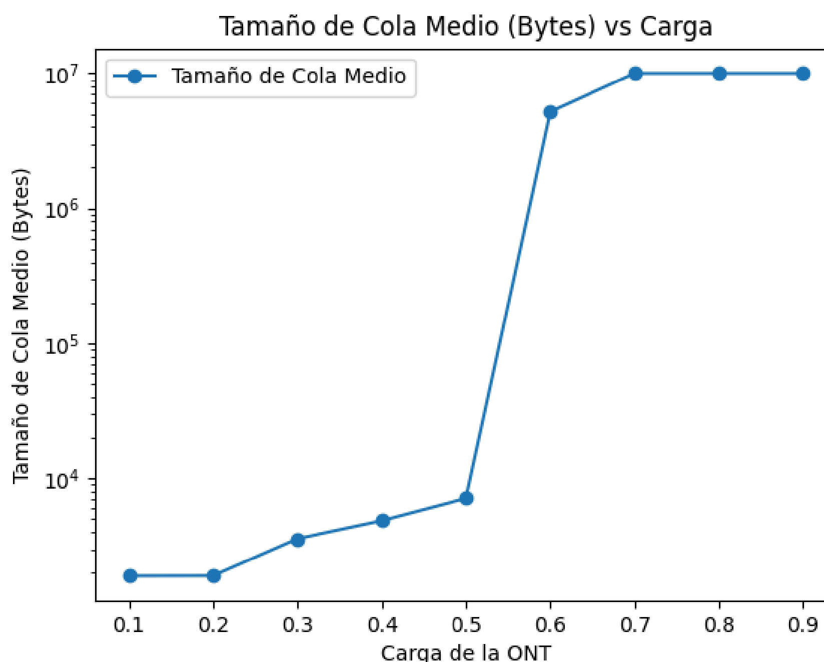

Figura 11. Tamaño medio de la cola en bytes frente a la carga de la ONT para el algoritmo IPACT con un SLA y tres colas, obtenido con las simulaciones en el trabajo de José María Robledo [15].

Puesto que en el trabajo de José María Robledo [15] no se elaboró esta gráfica, no podemos comparar sus resultados con los nuestros. Sin embargo, podemos observar que la gráfica se asemeja a la obtenida con una clase de servicio (Figura 4.4).

4.2.3 Escenario de simulación en una red 10G-EPON

Se describen a continuación los parámetros escogidos para las simulaciones, que van a ser idénticas a las anteriores, salvo por ciertos parámetros que se enuncian a continuación:

- La tasa de transmisión de la red (`txrate`) pasa de 10 Gbps.
- La tasa de transmisión máxima a la que puede transmitir un abonado conectado a una ONT (USER_Line_rate), que pasa de 1 Gbit/s.
- El tamaño total del *buffer* para almecenar las tramas Ethernet en cada cola de cada ONT (`tambuffer`) pasa de 100 MBytes.

Los parámetros detallados de las simulaciones han sido los siguientes:

Parámetros de Simulación	Valores
Número de ONTs	16 ONTs
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 *pon1* (OLT-ONT)	20 km
Tamaño de *buffer*	100 Mbytes
Algoritmo implementado	Algoritmo de *polling* 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 *streams*	32 *streams*

Número de servicios de prioridad (número de colas)	- 1 (P_0) para la primera variante del escenario - 3 (P_0, P_1 y P_2) para la segunda variante del escenario

Tabla 5. Parámetros empleados en las simulaciones del algoritmo de polling IPACT para una tasa de transmisión de 10Gbps (10G-PON)

4.2.3.1 Evaluación de IPACT considerando 1 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.

Compararemos las gráficas obtenidas con nuestras simulaciones con las obtenidas en el trabajo de Gorka Sainz-Ezquerra [10], en el que se ha realizado el mismo análisis, con los mismos parámetros, pero usando la librería Omnetpy.

En primer lugar, en las Figura 13 y 14, se analiza la evolución del retardo medio que sufren las tramas Ethernet desde que son generadas en cada ONT hasta que son recibidas en el OLT respecto a la carga de la ONT.

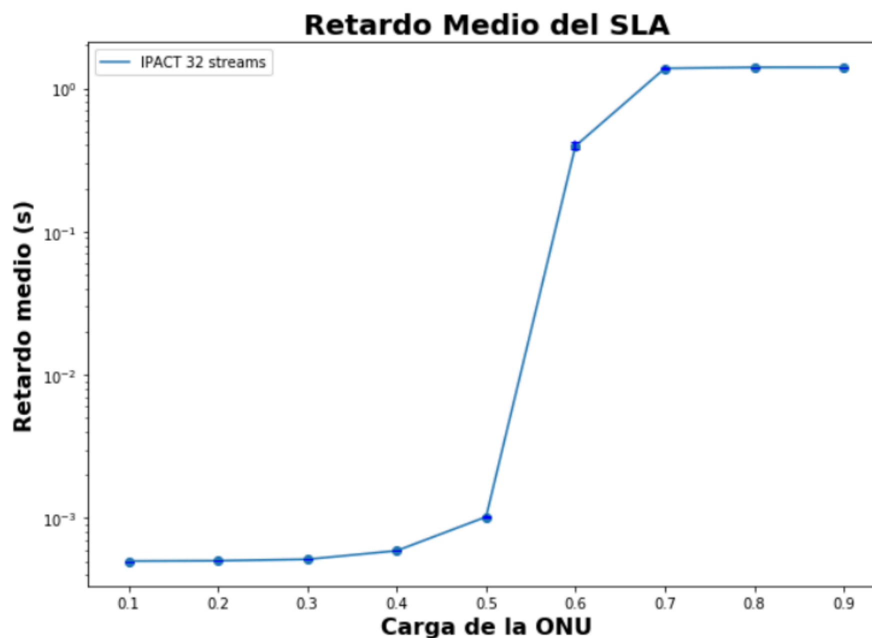

Figura 12. Retardo medio frente a la carga de la ONT para el algoritmo IPACT con un SLA y una sola cola, obtenido en las simulaciones en el trabajo de Gorka Sainz-Ezquerra [10].

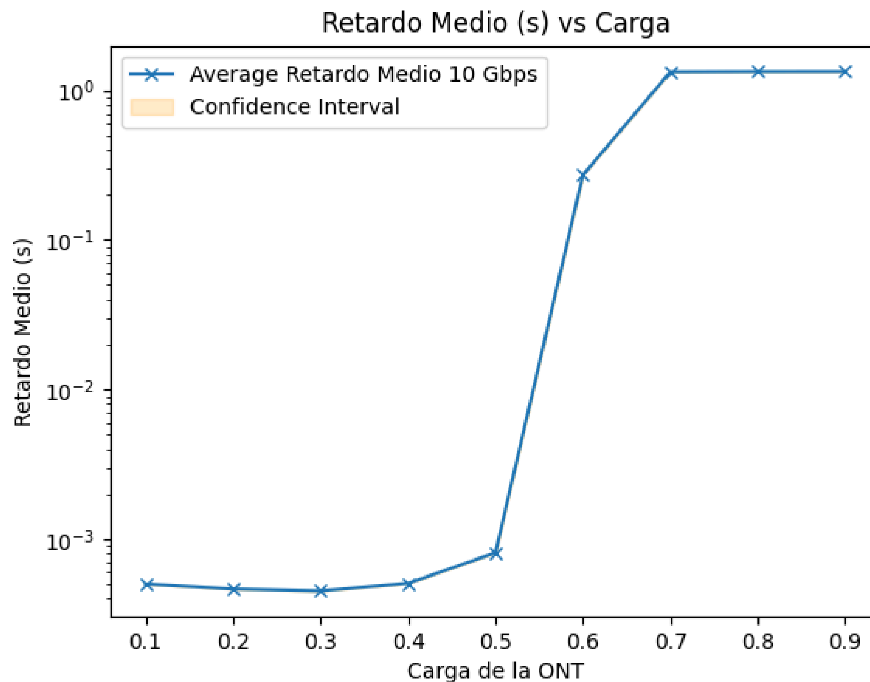

Figura 13. Retardo medio frente a la carga de la ONT para el algoritmo IPACT con un SLA y una sola cola, obtenidos en las simulaciones de este trabajo

Si se compara nuestra gráfica con la obtenida en el trabajo de Gorka Sainz-Ezquerra [10], podemos ver que los resultados son semejantes.

Por otra parte, se presenta también, a continuación (Figuras 14 y 15), la gráfica resultante del análisis de la cantidad media de bytes que quedan en las colas de una ONT tras cada ciclo de 2 ms frente a la carga de la ONT.

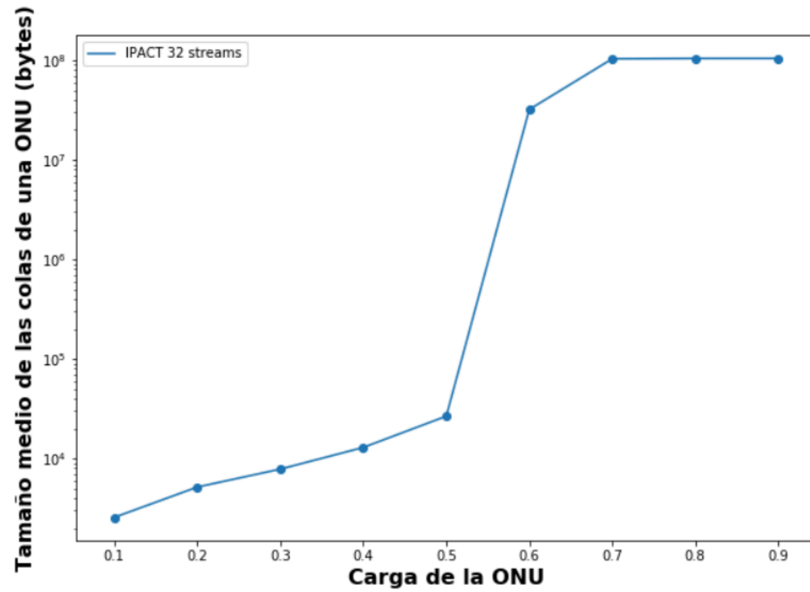

Figura 14. Tamaño medio de la cola en bytes frente a la carga de la ONT para el algoritmo IPACT con un SLA y una sola cola, obtenido con las simulaciones en el trabajo de Gorka Sainz-Ezquerra [10].

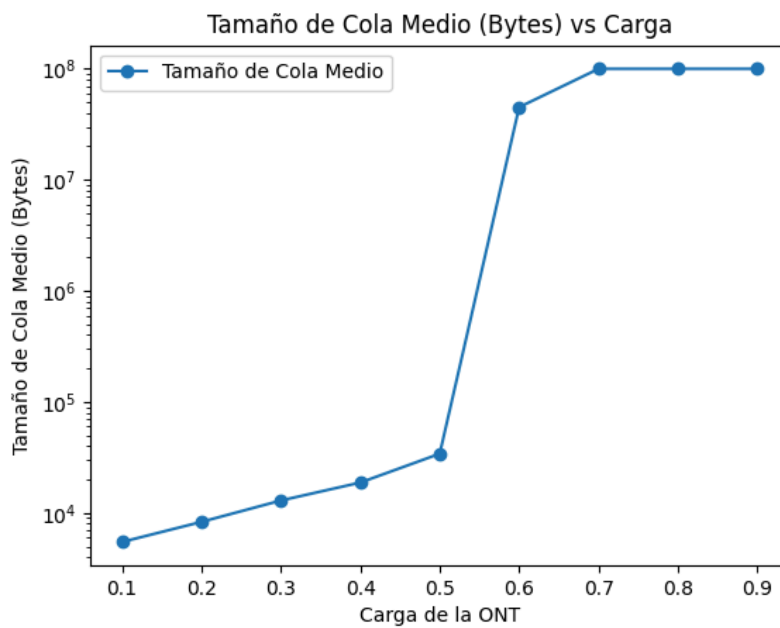

Figura 15. Tamaño medio de la cola en bytes frente a la carga de la ONT para el algoritmo IPACT con un SLA y una sola cola, obtenido con las simulaciones en este trabajo.

Podemos observar que las curvas obtenidas en nuestras simulaciones son semejantes a las obtenidas en el trabajo de Gorka Sainz-Ezquerra [10].

4.2.3.2 Evaluación de IPACT considerando 3 clases 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.

Compararemos las gráficas obtenidas con nuestras simulaciones con las obtenidas en el trabajo de Gorka Sainz-Ezquerra [10].

En primer lugar, en las Figuras 16 y 17, se analiza la evolución del retardo medio que sufren las tramas Ethernet desde que son generadas en cada ONT hasta que son recibidas en el OLT respecto a la carga de la ONT.

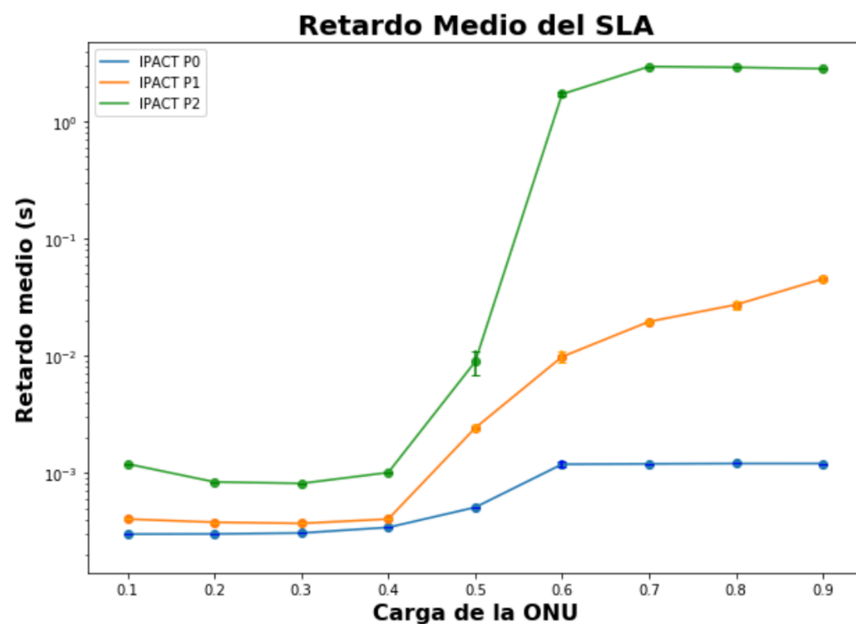

Figura 16. Retardo medio frente a la carga de la ONT para el algoritmo IPACT con un SLA y tres colas, obtenida con las simulaciones en el trabajo de Gorka Sainz-Ezquerra [10].

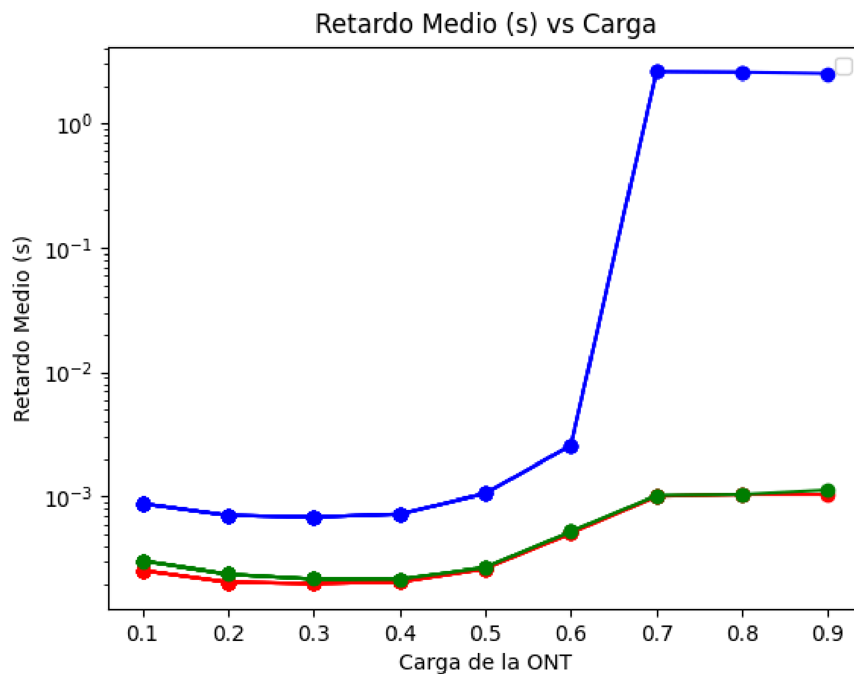

Figura 17. Retardo medio frente a la carga de la ONT para el algoritmo IPACT con un SLA y tres colas, obtenida con las simulaciones en este trabajo.

Si comparamos nuestra gráfica con la obtenida en el trabajo de Gorka Sainz-Ezquerra [10], obtenemos unas curvas para P_0 y P_2 muy parecidas. Sin embargo, la curva P_1 cae inesperadamente y se encuentra a varios órdenes por debajo de lo esperado para cargas superiores a 0.4.

Por otra parte, se presenta también, a continuación (Figura 18), la gráfica resultante del análisis de la cantidad media de bytes que quedan en las colas de una ONT tras cada ciclo de 2ms frente a la carga de la ONT.

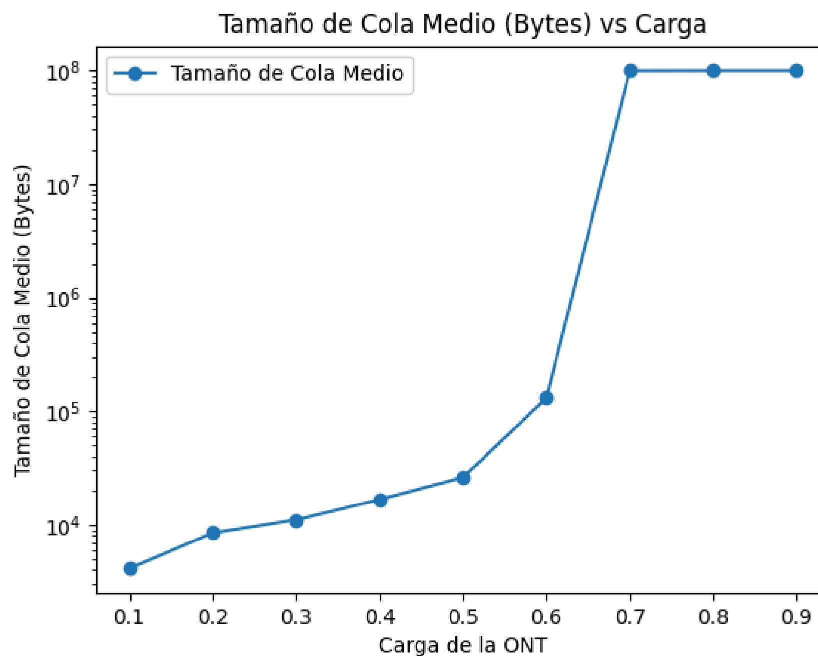

Figura 18. Tamaño medio de la cola en bytes frente a la carga de la ONT para el algoritmo IPACT con un SLA y tres colas, obtenido con las simulaciones en este trabajo.

Puesto que en el trabajo de Gorka Sainz-Ezquerra [10] no se elaboró esta gráfica, no podemos comparar sus resultados con los nuestros. Sin embargo, podemos observar que la gráfica se asemeja a la obtenida con una clase de servicio (Figura 18).

4.2.4 Prueba rápida con diferentes medias

Como se puede observar en los resultados de las simulaciones en las secciones 4.2.2 y 4.2.3, el retardo medio para ciertas cargas superiores a ρ=0.5 obtenemos unos resultados que no se parecen a las gráficas obtenidas en [10] y [15].

En vista de estos resultados diferentes a trabajos anteriores, hemos estado haciendo pequeñas pruebas con diferentes variables para ver de qué forma podemos hacer que las gráficas obtenidas sean las deseadas.

Ha sido de esta manera que hemos observado que si el parámetro `media` de nuestro generador de Pareto pasa de ser 8×10^{-5} a ser 8×10^{-3}, obtenemos un resultado más parecido al deseado.

A continuación se muestra los datos y los resultados de la prueba más relevante. Dicha prueba empleó el mismo escenario de simulación que en la sección 4.2.2 de esta memoria, pero alterando los parámetros mostrados en la Tabla 6.

Parámetro simulación	Valores
Número de servicios de prioridad (número de colas)	3 (P_0, P_1 y P_2)
Tiempo de simulación	100 segundos
Barrido de cargas	{0.7, 0.8}
Parámetro `media` del generador de Pareto	8×10^{-3}
Parámetro `a_on` del generador de Pareto	1.4
Parámetro `a_off` del generador de Pareto	1.2

Tabla 6. Parámetros empleados en la prueba de 100s

Como resultado de la simulación corta obtenemos la gráfica mostrada en la Figura 19. Podemos ver cómo la línea de la P_1 ya adopta unos valores que se asemejan más a los obtenidos en el trabajo de Gorka Sainz-Ezquerra [10] (véase Figura 16).

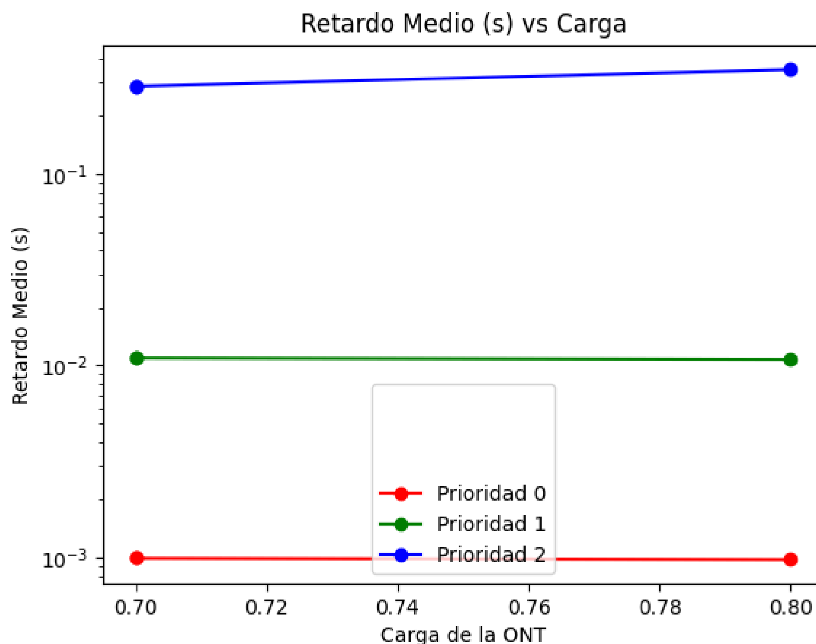

Figura 19. Resultados de la prueba de 100s

Ésto nos hace pensar que la influencia de la media empleada en el generador de Pareto debe ser estudiada con mayor detenimiento para así obtener unos resultados más parecidos a los esperados.

4.3 Conclusiones

En este capítulo de la memoria se ha podido comprobar, a través del simulador de redes PON implementado en Python, el comportamiento que presenta una red de tipo EPON en comparación con otra de tipo 10G-EPON, utilizando un algoritmo IPACT tanto para una clase de servicio como para tres clases de servicio. Hemos realizado un barrido de cargas entre 0.1 y 0.9 en las ONTs de la red y comparado los resultados con los obtenidos en el trabajo de José María Robledo [15] y el trabajo de Gorka Sainz-Ezquerra [10].

En primer lugar, hemos realizado el barrido de cargas para 1 Gbps y una clase de servicio. Los resultados obtenidos son muy similares a los obtenidos en el trabajo de José María Robledo [15]

En segundo lugar, hemos realizado el barrido de cargas para 1Gbps y tres clases de servicio. Los resultados obtenidos son similares a los obtenidos en el trabajo de José María Robledo [15] para las clases de servicio P0 y P2. Sin embargo, para P1, nuestros valores son algo inferiores.

En tercer lugar, hemos realizado el barrido de cargas para 10 Gbps y una clase de servicio. Los resultados obtenidos son análogos a los obtenidos en el trabajo de Gorka Sainz-Ezquerra [10].

En cuarto lugar, hemos realizado el barrido de cargas para 1Gbps y tres clases de servicio. Los resultados obtenidos son similares a los obtenidos en el trabajo de Gorka Sainz-Ezquerra [10] para las clases de servicio P0 y P2. Sin embargo, para P1, nuestros valores son algo inferiores.

Una prueba rápida variando el valor de la media del generador de Pareto nos ha hecho pensar que este parámetro puede tener un efecto significativo sobre las simulaciones.

5

Conclusiones y líneas futuras

5.1 Conclusiones

Inicialmente, nos fijamos dos objetivos principales a conseguir en este trabajo. Ambos han sido conseguidos:

- El primer objetivo, el de llevar a cabo "el diseño e implementación de un simulador de redes EPON en el lenguaje de programación Python" ha sido conseguido con éxito. Hemos podido comprobar que la librería Simpy es válida para modelar este sistema.
- El segundo objetivo, el de "analizar las prestaciones de diferentes algoritmos tanto en infraestructuras EPON (1G) como en infraestructuras 10G-EPON" sobre dicho simulador también ha sido conseguido con éxito. Las simulaciones han durado entre 12 y 18 horas para cada valor de carga para el caso de 1 Gbps, y entre 18 y 36 horas para cada valor de carga para el caso de 10 Gbps.

En cuanto a los objetivos específicos, éstos han sido los resultados:

- El primer objetivo específico era el de "estudiar la viabilidad de Python para desarrollar un simulador de redes PON basado en eventos discretos." El segundo objetivo era el de "diseñar y desarrollar un simulador de redes EPON a diferentes tasas de transmisión: EPON y 10G-EPON." Puesto que hemos conseguido implementar dicho simulador, efectivamente estos dos objetivos han sido conseguidos.
- El tercer objetivo era el de "seleccionar uno o varios algoritmos DBA para su implementación en el simulador de Python." El algoritmo escogido ha sido en este proyecto el de IPACT.

Si examinamos las simulaciones realizadas en este trabajo, podemos comprobar que para una clase de servicio, el simulador nos da unos valores similares a los obtenidos tanto en el trabajo de José María Robledo [15] como a los obtenidos en el trabajo de Gorza Sainz-Ezquerra[10].

En cambio, las simulaciones realizadas en este trabajo para tres clases de servicio nos dan unos resultados que difieren para ciertas cargas de los resultados obtenidos tanto en el trabajo de José María Robledo [15] como a los obtenidos en el trabajo de Gorza Sainz-Ezquerra[10].

5.2 Líneas futuras

Las líneas futuras que se trazan para complementar a trabajo son principalmente dos: el análisis y mejora del generador de fuentes de pareto, y la integración de algoritmos DBA con diferenciación de servicios y SLAs con el fin de dar una mejor calidad de servicio.

En cuanto al generador de fuentes de Pareto, hay que tener en cuenta que en este trabajo se ha intentado recrear de la mejor manera posible el simulador de Kramer. Sin embargo, en vista de que alterando la media de los períodos de ON y OFF conseguimos diferentes gráficas carga-retardo, podemos pensar que el simulador no es lo suficientemente veraz. Por lo tanto, puede ser útil estudiar el generador de fuentes de Pareto para que éste modele el tráfico rafagoso de una red EPON/GEPON de una forma más fiel a la realidad.

En cuanto a la integración de algoritmo DBA con diferenciación de servicios y SLAs, ésto permitirá explorar cómo los diferentes niveles de calidad de servicio pueden ser gestionados eficientemente en Redes Ópticas Pasivas (PONs) mediante la asignación dinámica de ancho de banda.

6

Bibliografía

[1] Omnetpy Repository, [En línea]. Disponible: https://github.com/mmodenesi/omnetpy [Último acceso: febrero 2024]

[2] SimPy 4.1.1 documentation. [En línea] Disponible: https://simpy.readthedocs.io/en/latest/ [Último acceso: febrero 2024]

[3] Reitz, K., & Schlusser, T. (2016). The Hitchhiker's Guide to Python: Best Practices for Development. O'Reilly Media.

[4] SimPy 4.1.1 documentation. Resources [En línea]. https://simpy.readthedocs.io/en/latest/topical_guides/resources.html [Último acceso: febrero 2024]

[5] Kramer, Glen. On generating self-similar traffic using pseudo-Pareto distribution. Disponible en: http://wwwcsif.cs.ucdavis.edu/ [Último acceso: febrero 2024]

[6] Numpy.random.pareto — NumPY v1.26 Manual. [En línea] https://numpy.org/doc/stable/reference/random/generated/numpy.random.pareto.html [Último acceso: febrero 2024]

[7] IEEE Standard for Ethernet, en IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015) , vol., no., pp.1-5600, 31 Aug. 2018, doi: 10.1109/IEEESTD.2018.8457469. Disponible: https://ieeexplore.ieee.org/document/8457469 [Último acceso: febrero 2024]

[8] IEEE Standard for Ethernet, en IEEE Std 802.3-2018 (Revision of IEEE Std 802.3ah-2004) , vol., no., pp 464-465, 2004: Disponible: https://www.ieee802.org/21/doctree/2006_Meeting_Docs/2006-11_meeting_docs/802.3ah-2004.pdf [Último acceso: febrero 2024]

[9] IEEE Standard for Ethernet, en IEEE Std 802.3-2018 (Revision of IEEE Std 802.3ah-2004) , vol., no., pp 466-467, 2004. Disponible: https://www.ieee802.org/21/doctree/2006_Meeting_Docs/2006-11_meeting_docs/802.3ah-2004.pdf [Último acceso: febrero 2024]

[10] Gorka Sainz-Ezquerra Calvo (2021), "Integración en OMNeT++ de módulos desarrollados en Python: Aplicación a un simulador de redes ópticas pasivas", Trabajo Fin de Grado del Grado en Ingeniería de Tecnologías de la Telecomunicación, E.T.S.I. de Telecomunicación, Universidad de Valladolid.

[11] Welcome to Python.org. (2024, March 13). Python.org. https://www.python.org/about/ [Último acceso: febrero 2024]

[12] Project Jupyter Documentation — Jupyter Documentation 4.1.1 alpha documentation. (n.d.). Disponible: https://docs.jupyter.org/en/latest/ [Último acceso: febrero 2024]

[13] Documentation for Visual Studio code. (2021, November 3) Disponible: https://code.visualstudio.com/docs [Último acceso: febrero 2024]

[14] 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.

[15] 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.

[16] Ten Little Algorithms, Part 3: Welford’s Method (and Friends) - Jason Sachs. (n.d.). Disponible: https://www.embeddedrelated.com/showarticle/785.php [Último acceso: febrero 2024]