Universidad Zaragoza

Trabajo Fin de Grado

Optimización de Tráfico de Red mediante Compresión, Multiplexión y Tunelado: Análisis y Caracterización.

Autor

José Ignacio Forcén Cabrejas

Director:

José Mª Saldaña Medina

Ponente:

Julián Fernández Navajas

Escuela de Ingeniería y Arquitectura

2014/2015

Repositorio de la Universidad de Zaragoza – Zaguan http://zaguan.unizar.es

Agradecimientos

En primer lugar, quisiera dar las gracias a José Mª y Julián, director y ponente del trabajo, por permitirme estudiar y conocer este campo de investigación tan interesante. Agradecerles los consejos, la dedicación y la disponibilidad que siempre han tenido conmigo a lo largo de la realización de este trabajo fin de grado.

Desde aquí aprovecho para agradecer a mi familia y amigos el apoyo incondicional durante este tiempo. En especial a mis padres, por todo el esfuerzo que han hecho para darme la oportunidad de cursar estos estudios.

Resumen

“Optimización de tráfico de red mediante compresión, multiplexión y tunelado: análisis y caracterización.”

Hoy en día, en Internet, existen multitud de servicios interactivos y aplicaciones en tiempo real como pueden ser VoIP (Voz sobre IP), servicios de videoconferencia, telemedicina, vídeo-vigilancia, juegos online, etc.

Este tipo de servicios generan un perfil de tráfico caracterizado por altas tasas de paquetes pequeños, que necesitan ser enviados con alta frecuencia y bajo retardo entre los dos extremos de la comunicación. Para todas estas aplicaciones, podemos concluir que se produce un elevado “*overhead*”, que eleva el consumo del ancho de banda de la red.

Para paliar esta ineficiencia en la red, aparecieron los conceptos de la multiplexión de paquetes y la compresión de cabeceras, con el objetivo de disminuir en la medida de lo posible el envío de información no útil a través de Internet.

En este Trabajo de Fin de Grado se ha realizado el estudio y mejora de un protocolo de optimización de tráfico, Simplemux, creado dentro del grupo de investigación CeNITEQ[1], para su estandarización a nivel global por parte del IETF *(Internet Engineering Task Force)*.

Para llevar a cabo el análisis y la puesta en funcionamiento de la herramienta, se han diseñado varios escenarios de red en los que se ha lanzado una batería de pruebas para obtener unas medidas fiables y consistentes.

Por último, mediante el análisis de los resultados obtenidos, se han extraído conclusiones de aquellas condiciones o entornos de red en los que es más eficiente el uso de esta herramienta.

1 CeNITEQ: *Communication Networks and Information Technologies for e-Health and Quality of Experience Group.* Grupo de investigación perteneciente al Instituto Universitario de Investigación en Ingeniería de Aragón (I3A).

Índice general

1. Introducción 1
1.1 Problemática 1
1.2 Objetivos 2
1.3 Motivación 3
1.4 Organización de la memoria 4
2. Estado del arte 6
2.1 TCRTP 6
2.2 Eficiencia en el Protocolo Wi-Fi (IEEE 802.11) 7
3. Optimización de tráfico de paquetes pequeños 13
3.1 Selección del tipo de tráfico a optimizar 13
3.1.1 VoIP 13
3.1.2 Juegos Online 15
3.1.3 Traza general 15
3.2 Optimización mediante Compresión, Multiplexión y Tunelado 16
3.2.1 Algoritmo de compresión 17
3.2.2 Multiplexado 19
3.2.3 Tunelado 24
4. Escenario de pruebas 26
4.1 Escenarios 26
4.1.1 Escenario Ethernet 26
4.1.2 Escenario Wi-Fi (ad-hoc). 28
4.2 Herramientas utilizadas en las pruebas 29
4.2.1 D-ITG 29
4.2.2 Linux Traffic Control 30
4.2.3 Capturas de tráfico 31
4.3 Equipos 31
5. Análisis de resultados 32
5.1 Pruebas UDP 32
5.1.1 Juegos Online 32
5.1.2 VoIP 34
5.1.3 Enlace congestionado 38
5.2 Pruebas con TCP 39

5.3 Traza general ... 44
5.4 Coste de procesado ... 49
6. Conclusiones y líneas futuras ... 50
6.1 Conclusiones ... 50
6.2 Líneas futuras ... 51
6.3 Planificación del trabajo ... 52
Bibliografía ... 53
Anexos ... 55
A. Acrónimos ... 55
B. Compilación de Simplemux en un equipo Linux. ... 57
C. Compilación cruzada de Simplemux. ... 59
D. Manual de Simplemux. ... 62
E. Manual de D-ITG. ... 68
F. Scripts utilizados durante el trabajo. ... 73
G. Generación de distribuciones de tráfico para D-ITG a partir de una captura .pcap .. 80

Índice de figuras

Figura 1.1. *Overhead* en paquetes Ethernet (usados en redes cableadas)...........................2
Figura 2.1. Pila de protocolos TCRTP...7
Figura 2.2. Formato paquetes multiplexados..7
Figura 2.3. Algoritmo de transmisión de una trama MAC 802.11 con el mecanismo CSMA/CA...9
Figura 2.4. Terminal oculto y terminal expuesto CSMA/CA... 10
Figura 2.5. Envío de una trama 802.11 con mecanismo RTS/CTS 11
Figura 3.1. Ejemplo de un paquete Simplemux multiplexando paquetes pertenecientes a distintos protocolos. 16
Figura 3.2. Esquema de optimización. 17
Figura 3.3. Compresor/descompresor ROHC. 18
Figura 3.4. ROHC: modo unidireccional.. 18
Figura 3.5. ROHC: modo bidireccional. 19
Figura 3.6. Separadores Simplemux.. 20
Figura 3.7. Formato del primer separador Simplemux. 20
Figura 3.8. Formato del resto de separadores Simplemux, caso SPB = ‘1’. 21
Figura 3.9. Formato del resto de separadores Simplemux, caso SPB = ‘0’. 21
Figura 3.10. Multiplexión controlada por número de paquetes. 22
Figura 3.11. Multiplexado controlado por tamaño.. 23
Figura 3.12. Multiplexión controlada por *Timeout*... 23
Figura 3.13. Multiplexado controlado por periodo.. 24
Figura 3.14. Paquete Simplemux en modo red. 24
Figura 3.15. Paquete Simplemux en modo transporte. 25
Figura 4.1. Escenario Ethernet. 26
Figura 4.2. Esquema de funcionamiento de una empresa con TCM................................... 27
Figura 4.3. Escenario Ad-hoc. 28
Figura 4.4. Esquema de implantación celular móvil a través de enlace punto a punto Wi-Fi.. 29
Figura 4.5. Arquitectura D-ITG... 30
Figura 5.1. *Quake III*: Comparativa ancho de banda. 33
Figura 5.2. *Quake III*: Comparativa paquetes por segundo... 33
Figura 5.3. VoIP: Ahorro de ancho de banda con GSM optimizado. 34
Figura 5.4. VoIP: Ahorro en paquetes por segundo con GSM optimizado. 36
Figura 5.5. VoIP: Tamaño paquetes GSM optimizados. 36
Figura 5.6. VoIP: Comparativa de BW con distintas codificaciones optimizadas. 37
Figura 5.7. VoIP: Comparativa pps para distintas codificaciones optimizadas. 38
Figura 5.8. Pérdidas Wi-Fi en enlace congestionado. 39
Figura 5.9. Esquema pruebas TCP. 40
Figura 5.10. Ahorro de ancho de banda optimizando paquetes ACK. 41
Figura 5.11. Tiempo de transmisión fichero en función del periodo de multiplexión. 42
Figura 5.12. Esquema pruebas FTP. 42
Figura 5.13. Comparativa del ancho de banda obtenido por dos transmisiones de ficheros con TCP. 43
Figura 5.14. Histograma de tráfico real entre dos nodos centrales. 44

Figura 5.15. Esquema de clasificación de paquetes. 45
Figura 5.16. Distribución de paquetes según su tamaño. 48
Figura 5.17. Distribución de paquetes después de la optimización 48
Figura 5.18. Tráfico traza general. 49
Figura 6.1. Diagrama de Gantt 52

Índice de Tablas

Tabla 1. Versiones Wi-Fi 8
Tabla 2. Rendimiento Wi-Fi. Fuente: Asignatura Redes de Acceso. Unidad II. 11
Tabla 3. Rendimiento Wi-Fi en función del tamaño del paquete sin RTS/CTS. 12
Tabla 4. Rendimiento Wi-Fi en función del tamaño del paquete con RTS/CTS. 12
Tabla 5. Ahorro de ancho de banda para *Quake III*. 34
Tabla 6. Factor MOS. 35
Tabla 7. Coste de procesado 49
Tabla 8. Clasificación de los paquetes en el fichero de log. 63

Capítulo 1

Introducción

El presente documento tiene como objeto recoger toda la información relacionada con la realización del Trabajo Fin de Grado (TFG) titulado *Optimización de tráfico de red mediante compresión, multiplexión y tunelado: análisis y caracterización.*

Este trabajo pertenece a la titulación de Grado de Ingeniería de Tecnologías y Servicios de Telecomunicación de la Universidad de Zaragoza. Ha sido realizado por el alumno José Ignacio Forcén Cabrejas y dirigido por José Mª Saldaña Medina, actuando como ponente Julián Fernández Navajas.

En este primer capítulo se incluyen tres partes: la problemática que se ha tratado de solucionar mediante la herramienta de optimización de tráfico "Simplemux"; los objetivos y motivaciones que han llevado a su realización; y la estructura según la cual se organiza este documento.

1.1 Problemática

Internet, en sus inicios, no se diseñó para aplicaciones o servicios con requerimientos de tiempo real (algunos de los primeros servicios eran el e-mail, la transferencia de ficheros, el acceso a web…). Sin embargo, con el tiempo se han desarrollado nuevos servicios, y ha surgido la necesidad de dar soporte a aplicaciones en tiempo real como puede ser VoIP (Voz sobre IP), aplicaciones de vídeo-conferencia, telemedicina, etc.

Esta tendencia ha producido que se lleven a cabo estudios para hacer más eficientes las comunicaciones a través de Internet con el objetivo de no desperdiciar los recursos de red que tenemos actualmente.

En la Figura 1.1 se puede observar la relación entre el *payload* (bytes útiles de información de un paquete) y el tamaño completo del paquete para diversos tipos de tráfico. Por ejemplo, para un paquete TCP que alcanza el MTU[2] (*Maximum Transfer Unit*) vemos que la eficiencia a nivel IP es muy buena (97%). Sin embargo, para paquetes RTP[3] (*Real Time Protocol*), típicamente usados para tráfico de voz, la eficiencia no es superior al 33%.

[2] MTU: Unidad Máxima de Transferencia. Hace referencia al tamaño máximo de un paquete, depende del protocolo y de la tecnología subyacente.

3 RTP: protocolo de nivel de sesión utilizado para la transmisión de información en tiempo real. Diseñado por el IETF y publicado por primera vez en la RFC 1889, y actualizado en la RFC 3550.

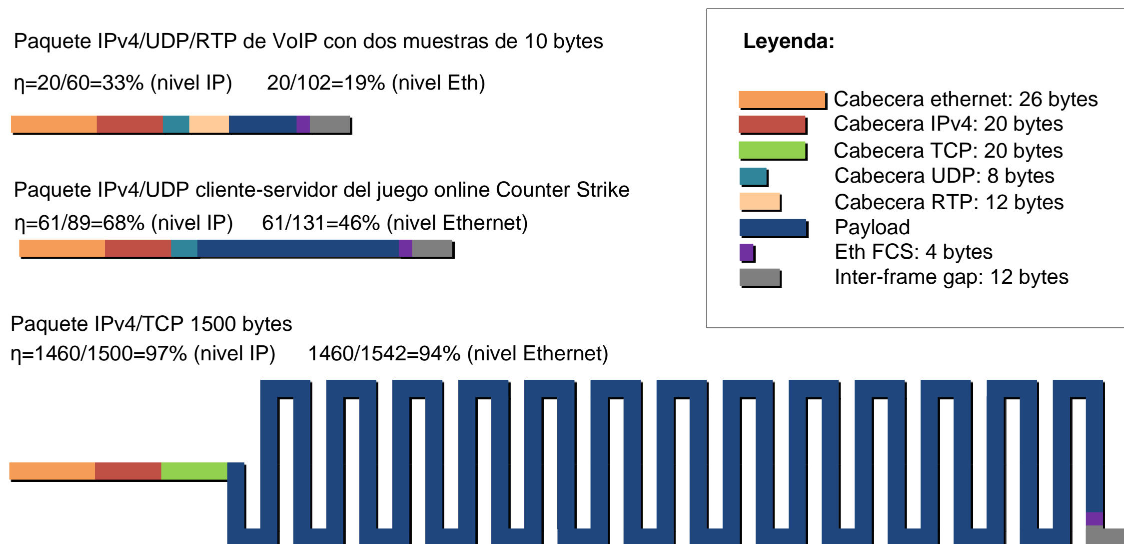

Figura 1.1. *Overhead* en paquetes Ethernet (usados en redes cableadas).

Entre las medidas adoptadas por el IETF (*Internet Engineering Task Force*) para dar soporte a este tipo de aplicaciones poco eficientes, destaca el estándar TCRTP (*Tunneling Multiplexed Compressed RTP*) definido en la RFC 4170 en 2005 [1], con la categoría de "*Best Current Practice*", lo que significa que el estándar no define ningún nuevo protocolo, sino que establece una manera para combinar varios ya existentes[4].

TCRTP combina protocolos de compresión, multiplexión y tunelado, como se explicará en detalle más adelante. Sin embargo, presenta algunas limitaciones: *a)* sólo se puede utilizar para tráfico RTP, y *b)* utiliza cuatro protocolos diferentes, lo que se traduce en un excesivo *overhead*.

Por todas estas limitaciones, surgió la idea de crear un nuevo protocolo que fuera capaz de proporcionar todas las prestaciones de TCRTP, añadiendo nuevas funcionalidades y compactando varios protocolos en uno común, denominado "Simplemux", permitiendo encapsular en una trama cualquier tipo de paquete, ya sea UDP o TCP. Desde el grupo de investigación CeNITEQ de la Universidad de Zaragoza se está trabajando en el desarrollo y estandarización de dicho protocolo.

1.2 Objetivos

Para contribuir a la estandarización del protocolo "Simplemux" a nivel global, es necesario realizar las pruebas oportunas para verificar su correcto funcionamiento y detectar aquellas situaciones en las que su uso resulta beneficioso para la red. El presente Trabajo Fin de Grado se enmarca dentro de este esfuerzo, y esperamos que suponga una ayuda para dar pasos hacia la aceptación de Simplemux dentro del IETF. De hecho, algunas de las pruebas presentadas en este

[4] Ver Apartado 6 de *RFCs and Internet-Drafts, de The Tao of IETF: A Novice's Guide to the Internet Engineering Task Force*, disponible en https://www.ietf.org/tao.html

Trabajo se han utilizado en la presentación de Simplemux realizada en la reunión IETF93, celebrada en Praga en julio de 2015[5].

Por tanto, podemos decir que el objetivo de este trabajo es el estudio de aquellas aplicaciones caracterizadas por la transmisión de paquetes muy pequeños con alta tasa de envío, y la posible mejora de su ineficiencia mediante técnicas de multiplexión, compresión y tunelado, respetando los límites de retardo que garantizan una buena calidad al usuario.

Las fases del trabajo realizado han sido las siguientes:

- Estudio teórico de la eficiencia y ahorro de ancho de banda a conseguir utilizando técnicas de multiplexión, compresión y tunelado en aplicaciones de tiempo real.

- Estudio y manejo de la herramienta de multiplexión (Simplemux) y de compresión de cabeceras ROHC[6] (*RObust Header Compression*) [2] para su posterior uso en las pruebas.

- Configuración y puesta en marcha de los diferentes escenarios de pruebas. En este proyecto se han diseñado escenarios con redes cableadas (Ethernet) e inalámbricas (Wi-Fi).

- Realización de pruebas modificando parámetros de red como pueden ser pérdidas, retardo, *jitter*, etc. También se han realizado pruebas modificando el periodo de multiplexión de paquetes o el número de paquetes que se introducen en una trama "Simplemux", con el objetivo de observar qué casos son los más idóneos en función del tráfico que circula por la red.

- Comparación de las medidas prácticas con las teóricas y análisis de resultados.

1.3 Motivación

En los últimos años, la facilidad de acceso a Internet mediante la aparición y el éxito de nuevos dispositivos como los *Smartphone*, *Tablet*, etc., así como la multitud de aplicaciones emergentes, como redes sociales, servicios interactivos, *Internet of Things* (IoT), etc. han producido un incremento del tráfico de datos a través de la red. Todas estas circunstancias están produciendo un consumo cada vez más elevado de los recursos disponibles. La movilidad de los usuarios puede producir que en ciertos lugares o momentos aparezcan picos de tráfico (fenómeno conocido como *flashcrowd),* haciendo que la cantidad de tráfico sea más imprevisible.

Esta tendencia está provocando que desde la comunidad científica comience la búsqueda de mecanismos que puedan dar soporte al crecimiento de la utilización de la red, añadiendo

---

[5] Simplemux se presentó en la reunión del grupo GAIA *(Global Access to the Internet for All),* celebrada en Praga el 22 de julio de 2015. Se puede ver la agenda de la sesión en https://datatracker.ietf.org/meeting/93/agenda/gaia/

[6] La implementación de ROHC que hemos utilizado se encuentra en https://rohc-lib.org/.

también más flexibilidad y capacidad de adaptación del tráfico de la red a los recursos del sistema. Simplemux se engloba dentro de este esfuerzo.

Además, la idea de que este protocolo pueda llegar a ser un estándar global de comunicaciones usado para la distribución del tráfico entre los nodos de Internet es algo que resulta especialmente atractivo y motivador desde el punto de vista académico. El poder conocer y aprender las técnicas de optimización de tráfico que se están desarrollando ahora para su implantación en un futuro es una rama interesante de la Telemática.

1.4 Organización de la memoria

El presente documento consta de dos partes diferenciadas. La primera está dedicada a la memoria del trabajo realizado; la segunda contiene anexos que añaden y complementan la información de la memoria.

La memoria está dividida en los siguientes capítulos:

- En el Capítulo 1 se introduce brevemente el trabajo, detallando la problemática existente en las comunicaciones actuales, y los principales objetivos que se persiguen.

- En el Capítulo 2 se ofrece una visión general sobre la eficiencia en comunicaciones IP. Para conocer el estado actual de los estándares de optimización de tráfico, se realiza una breve introducción a TCRTP. Seguidamente, para entender mejor la eficiencia de las comunicaciones inalámbricas, se describe de forma general la tecnología Wi-Fi, principalmente en términos de eficiencia.

- En el Capítulo 3, se pasa a realizar un análisis del método de optimización de tráfico, incluyendo las técnicas que utiliza para comprimir, multiplexar (con Simplemux) y tunelar los paquetes. También se detallan en este capítulo las aplicaciones cuyo tráfico se ha seleccionado para la realización de las pruebas, y las razones de esta elección.

- En el Capítulo 4 se detalla la arquitectura del sistema y la plataforma de pruebas utilizada, describiendo de forma general los escenarios de red. También se indican las herramientas usadas para las pruebas con tráfico real, configuración de parámetros de red, etc. Finalmente, se especifican los equipos utilizados durante la realización de este trabajo.

- En el Capítulo 5 se presenta el análisis de los resultados obtenidos en las distintas baterías de pruebas, caracterizándolas según hayan sido realizadas con tráfico UDP, con tráfico TCP, o con una traza general (obtenida en Internet), en la que aparece todo tipo de tráfico.

- En el Capítulo 6 se reflexiona sobre las conclusiones obtenidas del trabajo realizado, y se indican las posibles líneas futuras. También se describe la planificación del trabajo desarrollado.

- Finalmente, los anexos recogen información complementaria sobre el estudio realizado, así como manuales de uso de aquellas herramientas que se han utilizado en el trabajo, útiles para la puesta en marcha de diferentes configuraciones en un escenario de red.

Capítulo 2

Estado del arte

En este capítulo se verá cómo el envío de paquetes pequeños puede llevar a una gran ineficiencia, que puede acentuarse dependiendo de las tecnologías empleadas. En particular, veremos que Wi-Fi presenta una eficiencia muy baja, especialmente para paquetes pequeños. También hablaremos de TCRTP, un método de optimización propuesto por el IETF para mejorar la eficiencia en estos tráficos, mediante compresión de cabeceras, multiplexión y tunelado.

2.1 TCRTP

El Protocolo de Transporte en tiempo Real (RTP) [3] fue definido por el IETF para la transmisión de datos de sesiones multimedia. Este protocolo, ampliamente utilizado, está definido en la RFC (*Request for Comments*) 3550 y funciona sobre el protocolo de transporte UDP (*User Datagram Protocol*).

Funciona conjuntamente con el protocolo RTCP (*Real-time Transport Control Protocol*) que realiza el control del flujo multimedia enviado por RTP. Asimismo aporta información sobre los participantes de la sesión y realiza una monitorización de la calidad de servicio. Este protocolo está definido en la RFC 3605 [4].

TCRTP (*Tunneling Multiplexed Compressed RTP*), definido por el IETF bajo la categoría "*Best Current Practice*", es un método para optimizar la utilización de ancho de banda de múltiples flujos RTP que comparten un mismo camino de red. Para lograr este objetivo, combina varios protocolos que proporcionan compresión, multiplexado y tunelado sobre la capa de transporte.

Su pila de protocolos se puede observar en la Figura 2.1. En primer lugar, se usa ECRTP (*Enhanced Compressed RTP*) [5] para comprimir conjuntamente las cabeceras RTP, UDP e IP. Posteriormente, se usa PPPMux (*Point-to-Point Protocol Multiplexing*) [6] para incluir varios paquetes en uno, que finalmente se envía mediante PPP (*Point-to-point Protocol*) [7] y un túnel L2TP (*Layer 2 Tunneling Protocol*) [8], necesario para encapsular el protocolo PPP sobre IP. El uso de tunelado permite utilizar ECRTP extremo a extremo, evitando así la necesidad de aplicarlo en cada nodo del camino.

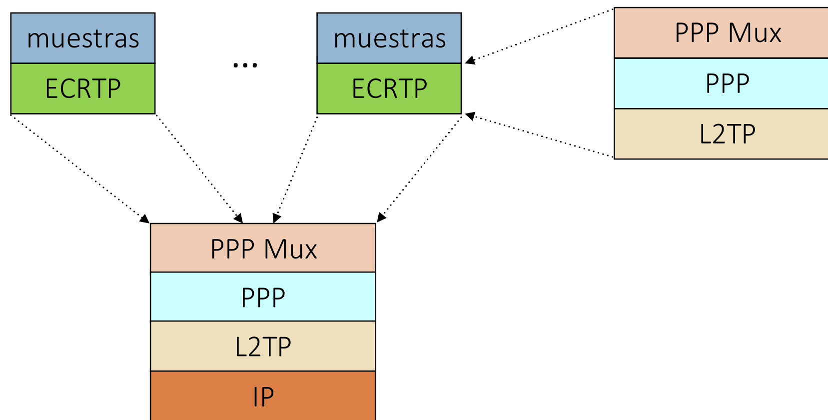

Figura 2.1 Pila de protocolos TCRTP.

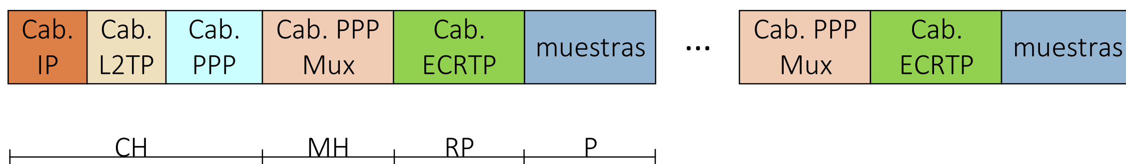

Figura 2.2. Formato paquetes multiplexados.

Como podemos ver en la Figura 2.2, un paquete multiplexado se puede descomponer en las siguientes partes:

- CH: Cabecera Común (*Common Header*): Corresponde a las cabeceras IP, L2TP y PPP.
- MH: Cabecera PPPMux (*Multiplexed Header*): Se incluye al principio de cada paquete comprimido.
- RH: Cabecera reducida (*Reduced Header*): Corresponde a la cabecera comprimida IP/UDP/RTP de cada paquete nativo.
- P: Contenido (*Payload*): Contenido de los paquetes nativos generados por la aplicación, es decir, en el caso de VoIP son las muestras de voz.

2.2 Eficiencia en el Protocolo Wi-Fi (IEEE 802.11)

Wi-Fi es un nombre comercial que recibe la especificación 802.11 del IEEE (*Institute of Electrical and Electronics Engineers*). La función principal de este tipo de redes es proporcionar conectividad y acceso a las tradicionales redes cableadas (*Ethernet, Token Ring*...), como si de una

extensión de éstas se tratara, pero con la flexibilidad y movilidad que ofrecen las comunicaciones inalámbricas.

La tecnología Wi-Fi está siendo promovida por un conjunto de empresas que se agrupan en la *Wi-Fi Alliance*, creada con el objetivo de promover esta tecnología, y certificar que los dispositivos son compatibles con lo establecido en el estándar 802.11. De esta forma, existe un mecanismo de conexión inalámbrica compatible entre distintos dispositivos a nivel global.

A lo largo de los años han aparecido diferentes versiones de 802.11. Las más importantes se pueden resumir en la siguiente tabla:

Tecnología Wi-Fi	Banda frecuencial	Velocidad de transmisión máxima	Año
IEEE 802.11a	5 GHz	54 Mbps	1999
IEEE 802.11b	2,4 GHz	11 Mbps	1999
IEEE 802.11g	2,4 GHz	54 Mbps	2003
IEEE 802.11n	2,4 y 5 GHz	600 Mbps	2009
IEEE 802.11ac	2,4 y 5 GHz	1.3 Gbps	2014

Tabla 1. Versiones Wi-Fi.

El protocolo MAC de 802.11 contempla dos modos de operación o funciones de coordinación:

- Distributed Coordination Function (DCF):

- Está orientado a servicios asíncronos sin requisitos de calidad de servicio.

- Se basa en el mecanismo *Carrier Sense Multiple Access– Collision Avoidance* (CSMA-CA) [9].

- Point Coordination Function (PCF):

- Está orientado a servicios síncronos con requisitos de calidad de servicio en cuanto a tiempo real.

- En este caso, un punto de acceso Wi-Fi (*Access Point*, *AP*) actúa como coordinador (*Point Coordinator*) y regula las transmisiones de las estaciones mediante un método de encuesta (*polling*).

- Solamente puede usarse en modo infraestructura.

La realización de este trabajo ha estado centrada en el primer modo de operación, que es usado mayoritariamente en la actualidad. Como esta tecnología utiliza un medio físico (aire) compartido por múltiples usuarios, puede suceder que en cierto momento varios nodos transmitan tráfico a la vez, produciéndose una colisión. Para evitar colisiones y controlar el acceso al medio, es necesario sondear el canal, es decir, escuchar el medio para ver si está ocupado.

Esta técnica de escucha del canal y acceso al medio se denomina *método de contienda*. El algoritmo más utilizado actualmente como método de contienda es CSMA/CA. La Figura 2.3 presenta un esquema de su funcionamiento, que se podría resumir en los siguientes eventos:

- Cuando una estación quiere enviar una trama, realiza un proceso de sondeo en el canal para detectar si existe una transmisión en curso. La función de sondeo del canal se denomina *Clear Channel Assessment* (CCA)[7].
- Cuando el canal está libre, la estación no transmite inmediatamente, sino que debe seguir escuchando el canal para asegurarse de que sigue libre durante un período de tiempo DIFS (*DCF Inter-Frame Space*).
- Antes de enviar una trama, el emisor genera un tiempo aleatorio denominado "tiempo de *backoff*".
- Si transcurrido este tiempo, el medio sigue libre, envía la trama.
- Para asegurar un acceso equilibrado al canal, si una estación acaba de transmitir una trama y tiene otra preparada para ser transmitida, deberá esperar obligatoriamente un periodo de tiempo aleatorio.
- Cuando una estación recibe correctamente una trama de datos, espera un intervalo de tiempo SIFS (*Short Inter-Frame Space*) y manda después la confirmación pertinente (ACK). SIFS es más pequeño que DIFS para proporcionar prioridad a los envíos de los ACK respecto a otras estaciones que estuviesen a la espera de que el canal quede libre para transmitir sus tramas de datos.
- Si una estación transmite una trama y no recibe confirmación en un tiempo determinado, dará la trama por perdida, procediendo a su retransmisión. Para ello inicia de nuevo el proceso de sondeo del canal.

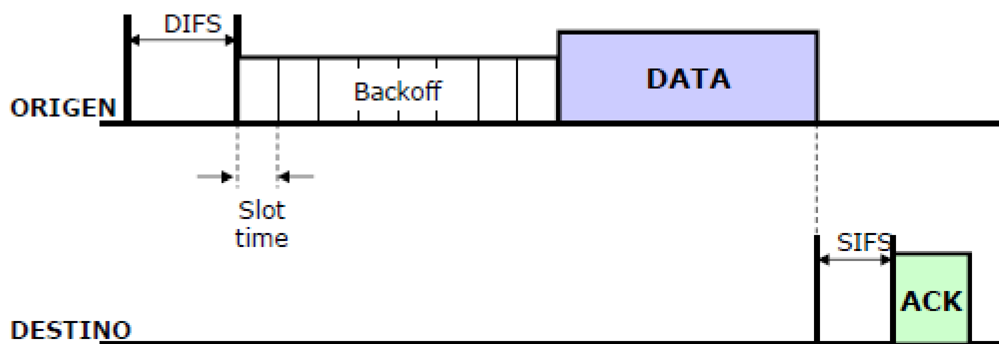

Figura 2.3. Algoritmo de transmisión de una trama MAC 802.11 con el mecanismo CSMA/CA. (Fuente: asignatura Redes de Acceso, Unidad II).

Además, este tipo de entornos generan dos problemas añadidos (ver Figura 2.4):

- **terminal oculto**: situación en la que una estación cree que el canal está libre, pero en realidad está ocupado por un nodo al que no escucha pero que está dentro del área de cobertura del receptor. Fijándonos en la figura, supongamos que C quiere transmitir una trama a A; C escucha el medio y como lo ve libre, transmite.

[7] *Clear Channel Assessment:* función lógica utilizada a nivel físico para determinar el estado actual de uso de un medio inalámbrico. Ver sección 17.3.10.5 de IEEE 802.11.

Mientras C está transmitiendo, B tiene una trama para transmitir a A, escucha el canal, y al detectar el medio libre, transmite y se produce una colisión en A.

- **terminal expuesto**: situación en la que una estación cree que el canal está ocupado, pero en realidad está libre pues el nodo al que oye no le interferiría. Atendiendo a al esquema de la figura, supongamos que A quiere transmitir a C y al mismo tiempo B quiere transmitir a D. C sondea el medio y lo encuentra ocupado por A, por lo que no transmite, aun pudiendo hacerlo, pues C está fuera de su cobertura y no le interferiría. Concluimos que B está expuesto a la transmisión de A.

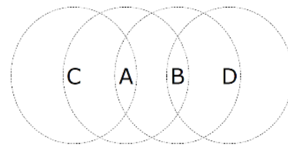

Figura 2.4. Terminal oculto y terminal expuesto CSMA/CA.

Para solucionar estos problemas de terminal oculto y terminal expuesto, se utiliza el mecanismo RTS/CTS (*Request To Send / Clear to Send*). Gracias a él se consigue reducir el número de colisiones, a costa de aumentar la latencia (cada envío ha de ir precedido del intercambio de dos mensajes).

El funcionamiento es el siguiente (ver Figura 2.5):

- Cuando un terminal desea transmitir datos, envía una trama RTS al receptor de destino. La trama RTS contiene la información de la cantidad de datos que desea transmitir. Cualquier estación diferente a la destinataria de la trama RTS que la reciba, retrasa sus transmisiones durante un tiempo igual al que tarde la recepción de la correspondiente trama CTS.

- El destinatario devuelve un paquete CTS (*Clear-to-Send*) si está disponible para recibir datos. La trama CTS contiene la indicación de la cantidad de datos que el transmisor original desea transmitir para que cualquier estación que lo reciba pueda estimar el tiempo en el que el canal va a estar ocupado.

- El terminal que había enviado la trama RTS envía los datos.

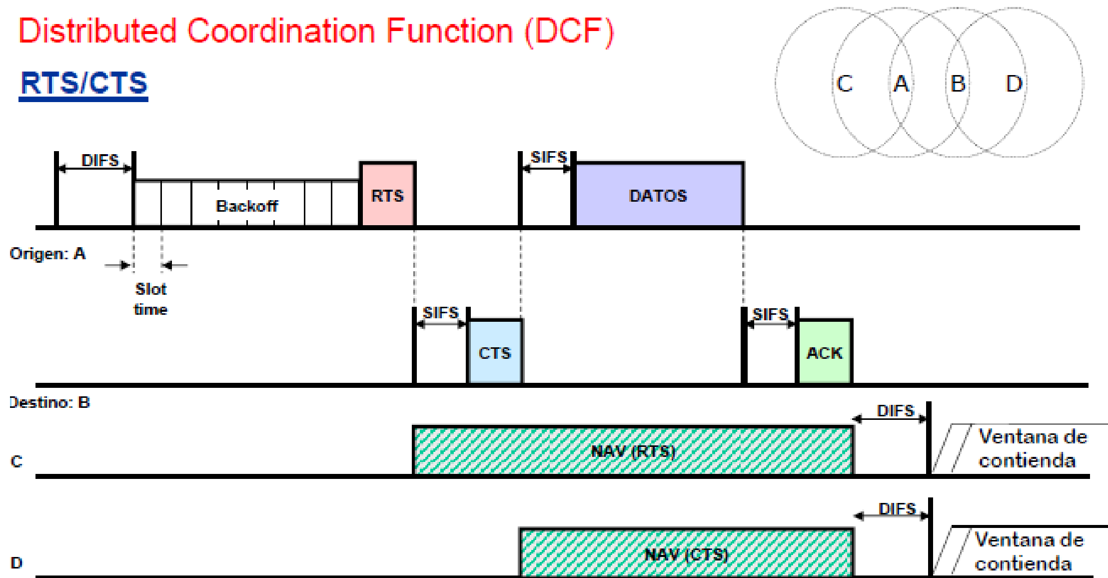

Figura 2.5. Envío de una trama 802.11 con mecanismo RTS/CTS. (Fuente: Asignatura *Redes de Acceso*, Unidad II).

Como se puede ver en la Figura 2.5, el tiempo útil de envío de datos es muy inferior al tiempo en el que el canal está ocupado. El tiempo NAV *(Network Allocation Vector)* es un temporizador que actúa como mecanismo de sondeo virtual del medio. Con él, una estación controla el tiempo que resta para que el canal quede libre.

En las tablas[8] siguientes se puede ver el rendimiento para diferentes velocidades y en función del tamaño del paquete IP.

	Rendimiento Nominal			
	Sin RTS/CTS		Con RTS/CTS	
Rb (Mbps)	% eficiencia	Mbps	% Eficiencia	Mbps
11	62,8	6,9	54	5,94
5,5	76,4	4,2	65,8	3,62
2	88,73	1,77	77,5	1,55

Tabla 2. Rendimiento Wi-Fi. (Fuente: Asignatura *Redes de Acceso*. Unidad II).

[8] Datos obtenidos de la asignatura Redes de Acceso, Unidad II.

Sin RTS/CTS	1500 (IP/UDP)		500 (IP/UDP)		300 (IP/UDP)	
Rb (Mbps)	% Eficiencia	Mbps	% Eficiencia	Mbps	% Eficiencia	Mbps
11	62,8	6,9	36	3,96	25,3	2,79
5,5	76,4	4,2	52	2,86	39,4	2,16
2	88,73	1,77	72,4	1,44	61,1	1,22

Tabla 3. Rendimiento Wi-Fi en función del tamaño del paquete sin RTS/CTS.

Con RTS/CTS	1500 (IP/UDP)		500 (IP/UDP)		300 (IP/UDP)	
Rb (Mbps)	% Eficiencia	Mbps	% Eficiencia	Mbps	% Eficiencia	Mbps
11	54	5,94	28,11	3,09	19	2,09
5,5	65,8	3,62	43	2,36	31	1,7
2	77,5	1,55	63,8	1,27	51,4	1,02

Tabla 4. Rendimiento Wi-Fi en función del tamaño del paquete con RTS/CTS.

Capítulo 3

Optimización de tráfico de paquetes pequeños

Este capítulo se divide en dos secciones: en primer lugar se expondrán los distintos tipos de tráfico que se van a optimizar, explicando las razones que nos han llevado a seleccionarlos. En la segunda parte se explicará en detalle el método de optimización.

3.1 Selección del tipo de tráfico a optimizar

La optimización de tráfico será especialmente útil en aquellos casos en los que las comunicaciones resultan más ineficientes. Los flujos generados por servicios interactivos (por ejemplo, Voz sobre IP o juegos online) requieren un envío periódico de información, por lo que tienden a generar paquetes pequeños. Por este motivo, resultan especialmente interesantes para este estudio, dada su baja eficiencia. Además se va a trabajar con una traza de tráfico genérica (capturada en Internet), de tal forma que también podamos obtener resultados para tráficos más genéricos. En este apartado explicaremos las características de cada uno de los flujos que vamos a optimizar.

3.1.1 VoIP

El crecimiento y fuerte implantación de las redes IP así como el desarrollo de técnicas avanzadas de digitalización de voz, protocolos de transmisión en tiempo real, y los nuevos estándares que permiten la calidad de servicio en redes IP, han creado un entorno donde es cada vez más frecuente transmitir voz sobre IP. Esta tecnología supone en muchos casos un ahorro económico importante para las empresas, pues reduce sus costes de telefonía.

La tecnología VoIP [10] supuso un reto tecnológico que consistió en realizar la convergencia entre las redes de voz (que empleaban conmutación de circuitos) y las redes de datos (que emplean conmutación de paquetes). Esta convergencia se pudo llevar a cabo gracias a la implementación de estándares creados por diferentes entidades entre las que se encuentran ANSI (*American National Standards Institute*), IEEE (Ins*titute of Electrical and Electronics Engineers*), ISO (*International Organization for Standardization*), UIT (Unión Internacional de Telecomunicaciones) e IETF (*Internet Engineering Task Force*).

Gracias a estándares como H.323, H.245, H.225 es posible integrar en la red convencional llamadas sobre IP. Es decir, convertir la señal de voz en paquetes de datos comprimidos para ser transportados por la red. Esta digitalización de la señal de voz se realiza a través de un códec.

La tecnología VoIP se caracteriza por la transmisión de una gran cantidad de paquetes de poco tamaño. Como veremos posteriormente, el tamaño del *payload* (unas pocas decenas de bytes) hace muy poco eficiente la transmisión de este tipo de comunicaciones.

La tecnología VoIP presenta diferentes formatos en función del códec utilizado para la digitalización de la voz. Para la realización de las pruebas se han elegido algunos de los códec más utilizados hoy en día:

- **GSM** (***Global System for Mobile communications***): también conocido como RPE-LTP (*Regular Pulse Excitation Long-Term Prediction*), aunque se conoce más por GSM ya que es el códec que soporta la tecnología GSM. Consigue un flujo de datos en una conexión *Full-Rate* de 13kbps.
  Cuando este códec se transporta sobre RTP, el paquete formado contiene un tamaño de información útil *(payload)* de 45 bytes. La tasa de envío de este códec es de 50 pps (paquetes por segundo).
  A nivel IP, el códec GSM muestra el siguiente porcentaje de información útil:

$$eficiencia\ GSM = \frac{payload}{cab.IP + cab.UDP + cab\ RTP + payload} =$$
$$= \frac{45\ bytes}{20 + 8 + 12 + 45\ \text{bytes}} = 52{,}9\%$$

  iLBC (***Internet Low Bitrate Codec***): definido en la RFC 3951 [11], es un codificador de voz en un inicio de software libre, posteriormente adquirido por Google y utilizado en múltiples aplicaciones entre las que destaca *"Google Talk"*. El flujo de datos es de tan solo 13,3 kbit/s, equivalente a GSM, pero con mejor calidad. Además, resulta más robusto frente a pérdidas que otros codificadores como pueden ser el G.729A o G.723.1. Tiene dos modalidades de tasa de envío: cada 20 o cada 30 ms. Cuando este códec se transporta sobre RTP, el paquete formado contiene un tamaño de información útil (*payload*) de 50 bytes si se envía con una tasa de 50 pps (20 ms). Para la modalidad de envío de paquetes cada 30 ms, su *payload* alcanza los 62 bytes con una tasa de envío de 33,3 pps.
  Por tanto, a nivel IP, iLBC muestra el siguiente porcentaje de información útil:

$$eficiencia\ iLBC_{20ms} = \frac{payload}{cab.IP + cab.UDP + cab\ RTP + payload} =$$
$$= \frac{50\ \text{bytes}}{20 + 8 + 12 + 50\ \text{bytes}} = 55{,}5\%$$

$$eficiencia\ iLBC_{30ms} = \frac{payload}{cab.IP + cab.UDP + cab\ RTP + payload} =$$
$$= \frac{62\ \text{bytes}}{20 + 8 + 12 + 62\ \text{bytes}} = 60{,}8\%$$

- **G.711. (PCM-A, PCM-U: *Pulse Code Modulation ley A, ley µ*)**: definido en la RFC 5391 [12], es un codificador de voz con dos versiones de compresión logarítmica: *µ-law*, usado en Estados Unidos y Japón, y *A-law* (usado en Europa y el resto del mundo). Cuando este códec se transporta sobre RTP, el paquete formado contiene un tamaño de información útil *(payload)* de 172 bytes y una tasa de envío de paquetes de 33,3 pps.

$$eficiencia\,G.711 = \frac{payload}{cab.IP + cab.UDP + cab\,RTP + payload} =$$
$$= \frac{172\,bytes}{20 + 8 + 12 + 172\text{ bytes}} = 81{,}1\%$$

Vemos por tanto que algunos de los códec presentan una eficiencia muy baja: el tamaño del *payload* útil está en el mismo orden de magnitud que el tamaño de las cabeceras.

3.1.2 Juegos Online

Los juegos *online* son un servicio que crece día a día en Internet. Algunos títulos tienen millones de usuarios, y por eso las empresas desarrolladoras se enfrentan a un difícil problema cada vez que lanzan un nuevo juego: necesitan recursos *hardware* y de red para evitar que su infraestructura se sature. Se ha elegido este tipo de tráfico porque presenta unos requerimientos de red muy estrictos, similares a los de VoIP, siendo también muy sensible al retardo, y se caracteriza por el envío de paquetes UDP pequeños. Utilizando este tipo de tráfico, se podrá observar si nuestro método de optimización de tráfico es capaz de funcionar en este tipo de aplicaciones sin que baje su calidad de servicio.

En concreto, en la realización de pruebas, se ha elegido tráfico del juego *"Quake III"*. Este juego *online* se puede incluir en la categoría FPS (*First-Person Shooter*). Dado su interés para otros estudios, el generador de tráfico D-ITG [13] incluye la posibilidad de generar flujos de tráfico con las estadísticas correspondientes a este juego.

El tráfico de este tipo de videojuegos está caracterizado por:

- Tráfico UDP.
- Paquetes de tamaño variable, inferior a 100 bytes.
- Alta tasa de paquetes por segundo (de 25 a 80 pps).

Si analizamos en concreto el juego *Quake III* [14] podemos ver que el tráfico de cliente a servidor es independiente del número de jugadores y que envía paquetes de tamaño entre 50 y 70 bytes, con un tamaño medio de 64,15 bytes. El tiempo entre paquetes oscila dependiendo de la tarjeta gráfica y el mapa en el que se esté jugando, obteniéndose de media 10,75 ms, que se traduce en una tasa de 93 pps. Con estos datos podemos concluir que *Quake III* presenta una eficiencia de:

$$eficiencia\,Quake\,III = \frac{payload}{cab.IP + cab.UDP + payload} = \frac{64{,}15\,bytes}{20 + 8 + 64{,}15} = 69{,}61\%$$

3.1.3 Traza general

Se ha escogido también una traza de tráfico general para estudiar su posible optimización, porque se analizó una traza de tráfico obtenida en un enlace de Internet, y disponible en un repositorio como *open data* [15], y se apreció que casi la mitad de los paquetes eran menores de 200 bytes. Vemos, por tanto, que la optimización de tráfico en este tipo de enlaces podría resultar útil. También es cierto que casi la otra mitad de los paquetes tienen un tamaño cercano al MTU,

por lo tanto, no se consideran ineficientes en términos de *payload*. Como veremos más adelante, podemos multiplexar sólo los paquetes pequeños, de manera que los grandes se verán inalterados. Gracias a esto se podrán conseguir ahorros considerables, dada la cantidad de paquetes pequeños que circulan por este tipo de enlaces.

En el capítulo 5.3 se podrá ver información adicional de este tipo de tráfico y las mejoras producidas al utilizar un proceso de optimización.

3.2 Optimización mediante Compresión, Multiplexión y Tunelado

Una posible mejora de la utilización de ancho de banda en un enlace reside en el empleo conjunto de tres técnicas conocidas como Compresión, Multiplexión y Tunelado. La primera de ellas busca reducir el tamaño de las cabeceras de los paquetes. Por otro lado, la Multiplexión consiste en agrupar varios paquetes en uno solo, y el Tunelado permite a ese paquete ser enviado por la red como un paquete IP normal. Existen diferentes protocolos para implementar cada una de estas tres técnicas.

El presente trabajo se centra en Simplemux, un protocolo de multiplexión definido en el *draft "Simplemux. A generic multiplexing protocol",* y presentado al IETF como draft-saldana-tsvwg-simplemux-03 [16]. Dicho protocolo, en combinación con otros de compresión de cabeceras y tunelado, permite enviar un número de paquetes pertenecientes a distintos protocolos, en un único paquete. Se busca que el tamaño de las cabeceras de multiplexión sea lo más pequeño posible, para mantener el objetivo de reducir el *overhead*. En cada cabecera de multiplexión se incluirá el campo "*Protocol Number*" que servirá para determinar el protocolo de la siguiente cabecera encapsulada. Podemos ver un ejemplo en la Figura 3.1.

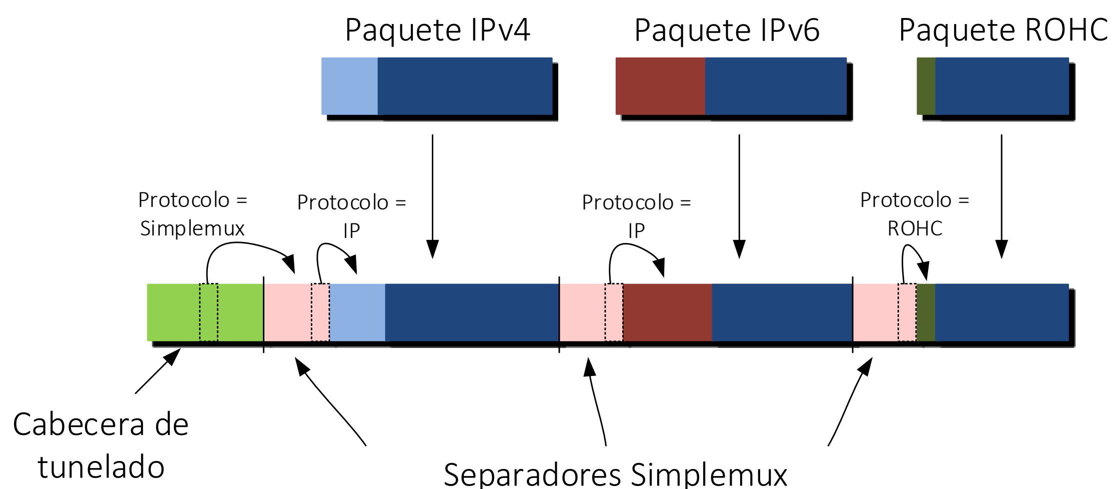

Figura 3.1. Ejemplo de un paquete Simplemux multiplexando paquetes pertenecientes a distintos protocolos.

Simplemux está diseñado para optimizar un número de flujos que comparten el mismo camino o segmento de red. La multiplexión es realizada entre los dos nodos extremos de dicho segmento. El primer nodo, denominado nodo de entada *(ingress)*, multiplexa los paquetes que

llegan de diferentes flujos. El segundo nodo, nodo de salida *(egress)*, será el encargado del demultiplexado de los paquetes antes de que lleguen al equipo final, ya que estas operaciones deben ser invisibles para los equipos terminales. La estructura de funcionamiento se muestra en la Figura 3.2.

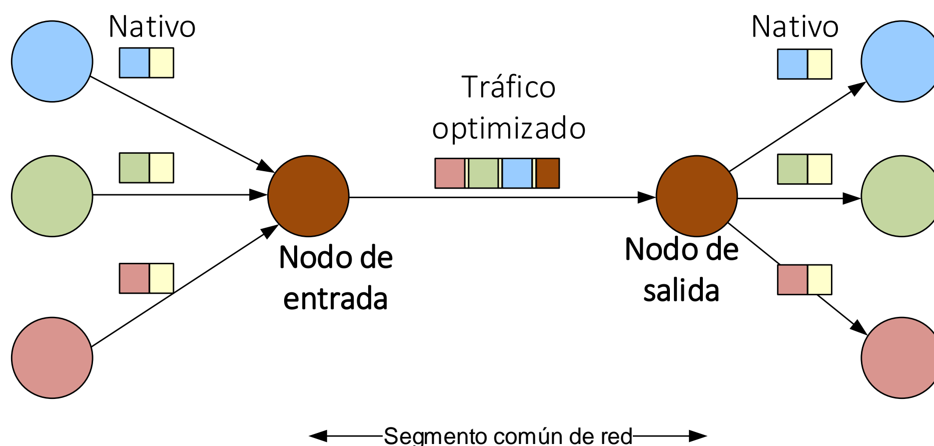

Figura 3.2. Esquema de optimización.

Para facilitar las pruebas con Simplemux, dentro del grupo CeNITEQ de la Universidad de Zaragoza, se dispone de una implementación de dicho protocolo, que se encuentra disponible en el repositorio de código Github[9]. Esta implementación utiliza IP o IP/UDP como protocolos de tunelado, y es capaz de multiplexar paquetes IP nativos, o también paquetes con cabecera comprimida mediante ROHC *(RObust Header Compression),* como se explicará más adelante.

3.2.1 Algoritmo de compresión

Para la compresión de cabeceras, que se puede usar en combinación con la multiplexión, se puede utilizar el protocolo ROHC[10]. Para su funcionamiento, se emplean dos nodos: un compresor, y un descompresor que se encargarán de implementar el protocolo (Figura 3.3).

En un flujo de paquetes entre un emisor y un receptor, hay muchos campos de la cabecera que se repiten (dirección IP origen, dirección IP destino, puertos, etc.). ROHC aprovecha esta redundancia en las cabeceras de los paquetes para evitar el envío de estos campos en cada paquete.

Para el correcto funcionamiento de compresión y descompresión, ROHC utiliza un "contexto", es decir, un conjunto de valores de los campos, que deben estar sincronizados entre el emisor y el receptor. Posteriormente se añade un campo (identificador de flujo) a cada paquete, que ROHC utilizará para marcar los distintos paquetes y así determinar a qué flujo pertenecen.

Su funcionamiento es el siguiente: el primer paquete del flujo se manda íntegro, sin comprimir. Esto lo realiza para que el receptor conozca la cabecera original. A partir de ahí envía

[9] La implementación de Simplemux está disponible en https://github.com/TCM-TF/simplemux
[10] La implementación de ROHC que hemos utilizado se encuentra en https://rohc-lib.org/

únicamente la información variable de la cabecera, es decir, los campos que cambian para cada paquete (p.ej. números de secuencia, tamaños de ventana TCP, etc.). Cuando es posible, este tipo de campo además, es enviado con un menor número de bytes que los que se emplearían para este campo en una transmisión normal. Esto ocurre por ejemplo con números de secuencia, para los que sólo es necesario enviar el incremento *(delta)* respecto al valor del campo en el paquete anterior.

Cada cierto intervalo de tiempo, se vuelve a mandar un paquete entero (sin comprimir la cabecera) para evitar problemas de sincronización. Cuando entre dos nodos circulan paquetes ROHC pertenecientes a distintos flujos, los paquetes se marcan para poder luego diferenciarlos correctamente. De esta manera, una vez que el paquete es identificado y marcado (campo llamado *stream context*), se comprime atendiendo a uno de los siguientes perfiles de compresión disponibles en ROHC:

- Sin comprimir.
- Sólo IP.
- IP/UDP.
- IP/UDP-Lite.
- IP/ESP.
- IP/UDP/RTP.
- IP/UDP-Lite/RTP.
- IP/TCP.

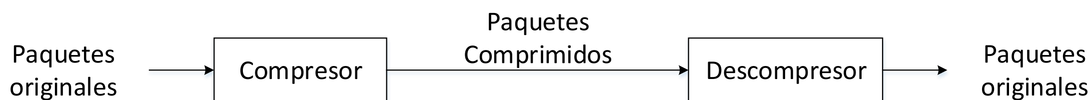

Figura 3.3. Compresor/descompresor ROHC.

ROHC tiene tres modos de funcionamiento, aunque de momento sólo dos de ellos están implementados en la librería que utilizaremos:

- **ROHC unidireccional**: Los paquetes son transmitidos únicamente en una dirección, desde el compresor al descompresor. Este sistema resulta útil para enlaces en los que no es deseable un camino de vuelta. Con el fin de gestionar posibles errores de transmisión, el compresor enviará actualizaciones periódicas del contexto de secuencia sin comprimir para evitar problemas de sincronización.

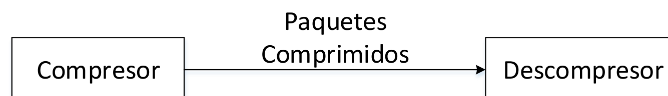

Figura 3.4. ROHC: modo unidireccional.

- **ROHC bidireccional optimista**: Este modo añade al anterior una realimentación que sirve para que el descompresor le comunique al compresor posibles errores en

recepción solicitando de esta manera un reenvío para poder descomprimir correctamente.

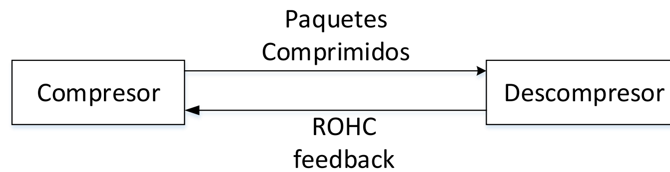

Figura 3.5. ROHC: modo bidireccional.

- **ROHC bidireccional fiable**: añade más mecanismos para evitar la desincronización del contexto. Estará disponible en futuras versiones de nuestra implementación de ROHC.

Estados del compresor/descompresor:

Independiente del modo de operación, el compresor/descompresor va a trabajar en uno de los siguientes tres estados:

- **Estado de inicialización y actualización:** Aquí se define un contexto para un flujo nuevo de paquetes que llegan al compresor y se envía el paquete íntegro, sin comprimir, para que el descompresor sepa cuál debe ser la cabecera completa de los paquetes que lleguen con ese mismo campo "contexto".

- **Estado de primer orden:** Cuando el compresor recibe un paquete que pertenece a un contexto ya asignado, envía la cabecera de los paquetes comprimida.

- **Estado de segundo orden:** Cuando el compresor recibe una realimentación del descompresor informando de algún tipo de error.

3.2.2 Multiplexado

Para realizar el multiplexado de los paquetes, Simplemux utiliza un "separador" entre los diferentes paquetes que encapsula. Estos separadores pueden ser de dos tipos. Uno para la primera cabecera, y otro para el resto de cabeceras.

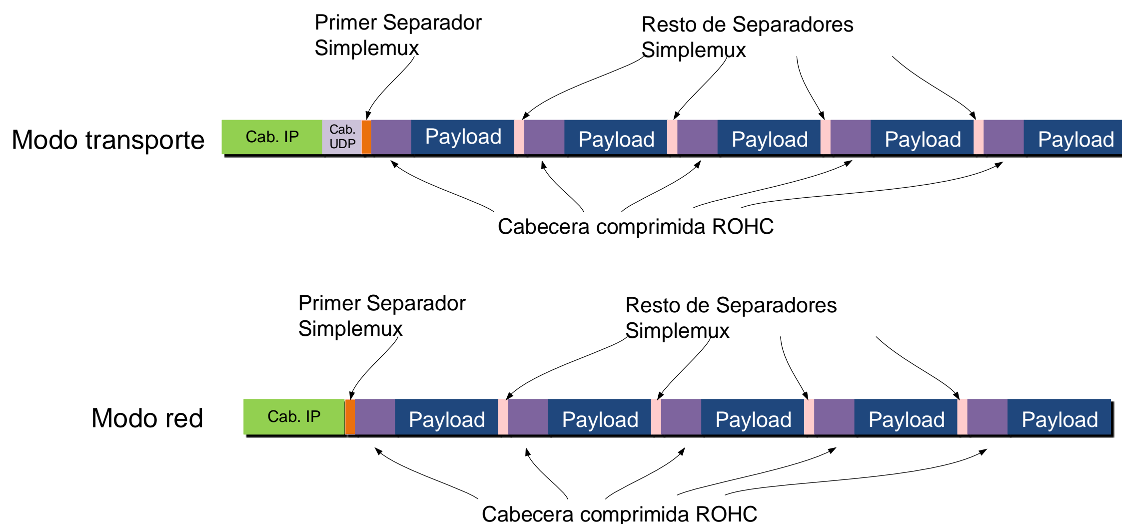

Figura 3.6. Separadores Simplemux.

Formato del primer separador Simplemux

En función del tamaño del paquete que multiplexa, el separador tendrá dos formatos distintos (Figura 3.7):

- Tamaño del paquete < 64 bytes. Se emplearán 6 bits para el campo longitud.
- Tamaño del paquete ≥ 64 bytes. Se emplearán 14 bytes para el campo longitud.

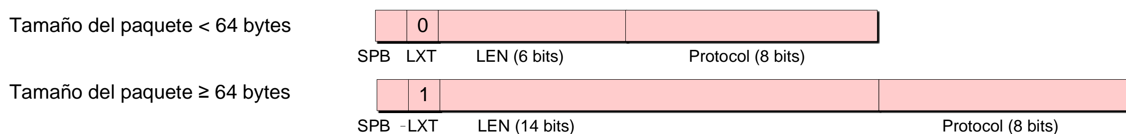

Figura 3.7. Formato del primer separador Simplemux.

Veamos qué campos tienen estos separadores:

- ***Single Protocol Bit*** **(SPB, 1 bit):** Sólo aparece en la primera cabecera Simplemux. Se pone a '1', si todos los paquetes multiplexados pertenecen al mismo protocolo. En caso afirmativo, sólo se escribiría el campo Protocolo en la primera cabecera, pudiéndonos ahorrar este campo en el resto de separadores.

- ***Length Extension*** **(LXT, 1 bit):** Campo utilizado para saber si el tamaño del paquete que multiplexa es mayor o igual que 64 bytes. Si es 0, el paquete será menor, y por tanto el campo de longitud se expresará con 6 bits. En caso contrario, el paquete será mayor y el campo de longitud se deberá expresar con 14 bits.

- ***Length*** **(LEN, 6 o 14 bits):** Campo que indica la longitud del paquete multiplexado, en bytes. Dicho campo tendrá una longitud de 6 bits para paquetes menores a 64 bytes, y 14 bits en caso contrario. El tamaño máximo del paquete que se podrá multiplexar

es de 16.383 bytes. Normalmente, este tamaño tan grande no se alcanzará nunca, sino que se ajustará al MTU de la red o vendrá condicionado por otros parámetros como el retardo, que veremos más adelante. Los paquetes mayores deberán ser enviados en su forma original, sin multiplexar.

- ***Protocol* (8 bits):** Es el campo Protocolo "*Assigned Internet Protocol Numbers*" definido por IANA. En él se indica el protocolo al que pertenecen los paquetes multiplexados.

Podemos concluir que el tamaño de este primer separador no será mayor de 3 bytes.

Formato del resto de separadores Simplemux

Este separador admite dos formatos, en función de lo que indique el campo SPB del primer separador.

Si el SPB tiene el valor '1', indica que todos los paquetes multiplexados pertenecen al mismo protocolo, de manera que dicho campo no necesitamos incluirlo en estos separadores. Así que el formato queda de la siguiente manera (Figura 3.8).

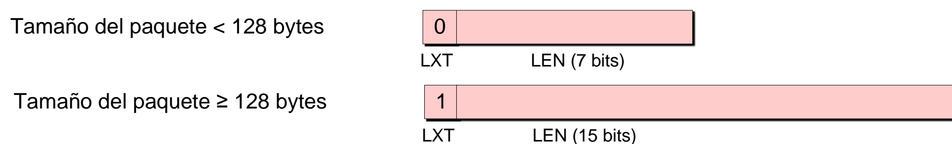

Figura 3.8. Formato del resto de separadores Simplemux, caso SPB = '1'.

En el caso de que SPB valga 0, significa que los paquetes que multiplexaremos en una trama Simplemux, van a pertenecer a diferentes protocolos, por esta razón, no debemos omitir el campo *Protocol* (Figura 3.9).

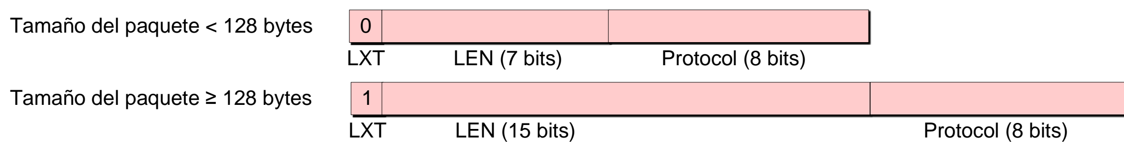

Figura 3.9. Formato del resto de separadores Simplemux, caso SPB = '0'.

De la misma manera que para el caso de los primeros separadores, el resto de separadores coinciden en los campos LXT, LEN y *Protocol*. Podemos concluir, que el tamaño del resto de separadores no será mayor de 2 bytes, en el caso de enviar paquetes de un solo protocolo. En caso contrario, el tamaño máximo será de 3 bytes.

Política de multiplexado

La cantidad de paquetes encapsulados en un paquete Simplemux vendrá determinada por cuatro condiciones configurables. En el momento en el que se cumpla alguna de ellas, el multiplexor creará el paquete Simplemux y lo enviará. Las condiciones para el envío son:

- **Número de paquetes:** Cuando un número determinado de paquetes llega al multiplexor, este los agrupa y envía la trama (Figura 3.10).

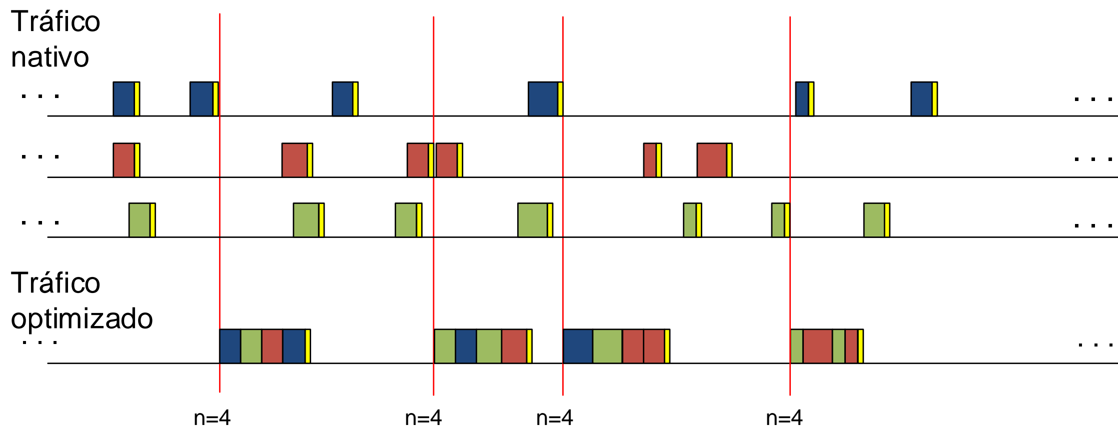

Figura 3.10. Multiplexión controlada por número de paquetes.

- **Tamaño:** Aquí aparecerán dos condiciones posibles (Figura 3.11):
    - El tamaño del paquete multiplexado ha excedido el umbral especificado por el usuario, pero no ha alcanzado el MTU (*Maximum Transfer Unit*). En este caso el paquete se envía y se reinicia el *buffer* de entrada de paquetes.
    - El tamaño del paquete multiplexado excede el MTU. En este caso, debemos mandar la trama Simplemux excluyendo el último paquete que había llegado, y almacenándolo para la siguiente trama.

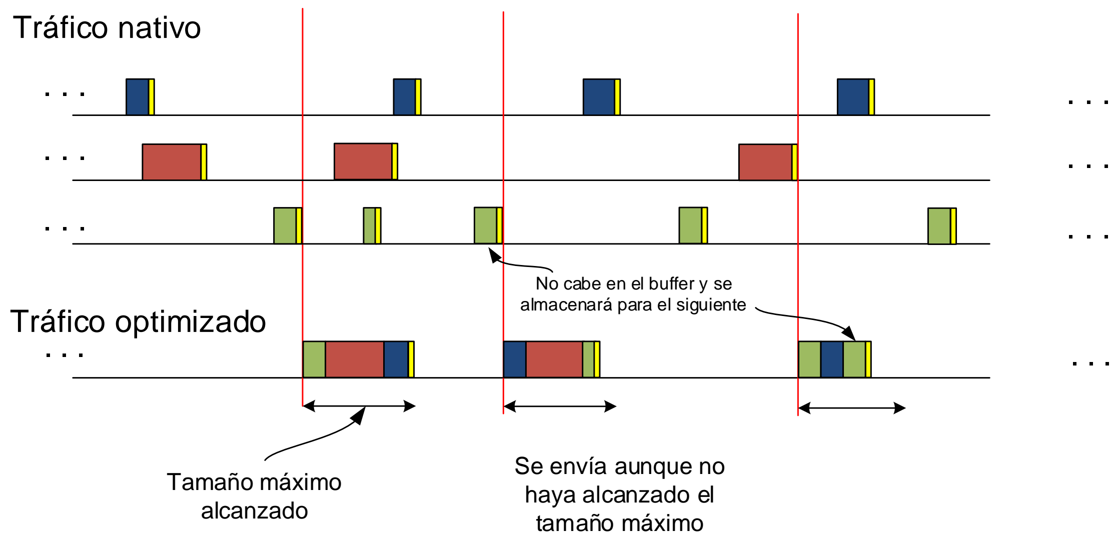

Figura 3.11. Multiplexado controlado por tamaño.

- **Timeout:** Cuando un paquete llega al multiplexor, este inicia un *Timeout*. Cuando dicho tiempo expire, se esperará a recibir otro paquete. En ese momento se enviarán todos los paquetes que hayan llegado y se reiniciará el *Timeout*.
  Esta opción sólo es recomendable en procesadores muy limitados computacionalmente, ya que se evita tener un proceso ejecutando constantemente un algoritmo de espera activa. De esta manera, si no se reciben paquetes, el programa no está malgastando recursos del procesador. Podemos observar el funcionamiento de este modo en la Figura 3.12.

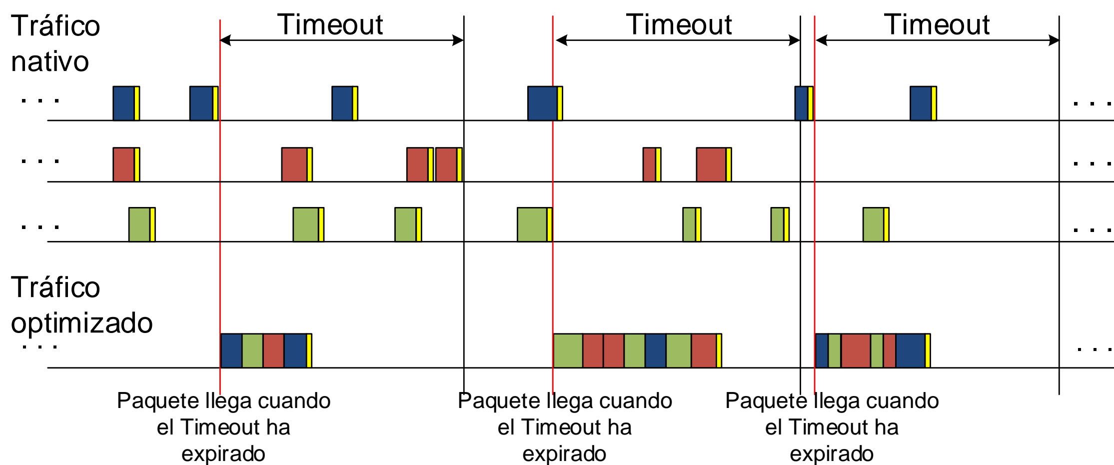

Figura 3.12. Multiplexión controlada por *Timeout*.

- **Periodo:** Una espera activa se pone en marcha y, cuando se cumpla el periodo, se enviarán los paquetes que hayan llegado en ese intervalo de tiempo (Figura 3.13).

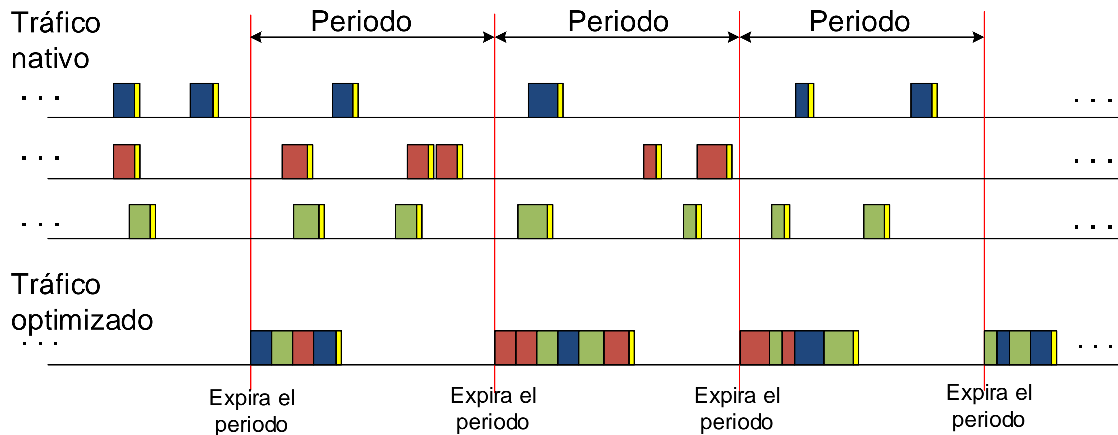

Figura 3.13. Multiplexado controlado por periodo.

La forma de utilizar todas estas políticas se indica en el Anexo D de esta memoria.

Hay que tener en cuenta que se pueden activar varias políticas de multiplexado al mismo tiempo. En caso de que haya varias activadas, el paquete Simplemux partirá en el momento en el que se cumpla alguna de ellas.

Además, debemos saber que la condición del periodo es la única que nos garantiza un retardo máximo para los paquetes.

3.2.3 Tunelado

Simplemux requiere un protocolo de tunelado para poder mandar los paquetes extremos a extremo. En nuestra implementación, Simplemux puede trabajar en dos modos:

- **Modo red:** el paquete multiplexado es enviado en un datagrama IP, utilizando el número 253 en el campo "*Protocol*" de la cabecera IP. Este código está definido por la IANA para la realización de pruebas [17] *(Use for experimentation and testing)*. Este es el método más eficiente en cuanto a ahorro de ancho de banda se refiere, pues sólo introduce el *overhead* de una cabecera IP (20 bytes para IPv4), pero puede ocasionar problemas con los *firewall,* si no se han permitido previamente conexiones con este valor en el campo *Protocol* de la cabecera IP.

Figura 3.14. Paquete Simplemux en modo red.

- **Modo transporte:** el paquete multiplexado es enviado en un datagrama UDP. En este caso, el valor del campo *Protocol* de la cabecera IP será el 17 (correspondiente a UDP), y ambos nodos deberán ponerse de acuerdo para trabajar en el mismo puerto UDP. De esta manera, resultará más fácil atravesar *firewall*, al utilizar paquetes UDP y no un protocolo desconocido para ellos. Este modo de transmisión es menos eficiente que el modo de red, ya que debemos incluir la cabecera UDP (8 bytes) en cada paquete multiplexado.

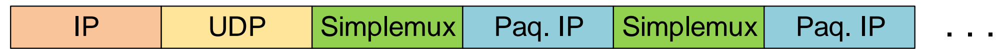

Figura 3.15. Paquete Simplemux en modo transporte.

Capítulo 4

Escenario de pruebas

En este capítulo se presentarán los elementos que han intervenido en la realización de las pruebas. En primer lugar se expondrán los escenarios de red utilizados tanto para pruebas cableadas (*Ethernet*) como para las inalámbricas (*Wi-Fi*). Seguidamente se introducirán aquellos programas utilizados para la generación de tráfico y análisis estadístico. Finalmente se especificarán los equipos utilizados durante el trabajo.

4.1 Escenarios

Se han diseñado varios escenarios de red con el objetivo de realizar una batería de pruebas para observar el comportamiento de la herramienta de multiplexión Simplemux ante diferentes entornos de red y con distinto tráfico.

4.1.1 Escenario Ethernet

Para entornos cableados, se diseñó y se puso en marcha el esquema de red que podemos ver en la Figura 4.1.

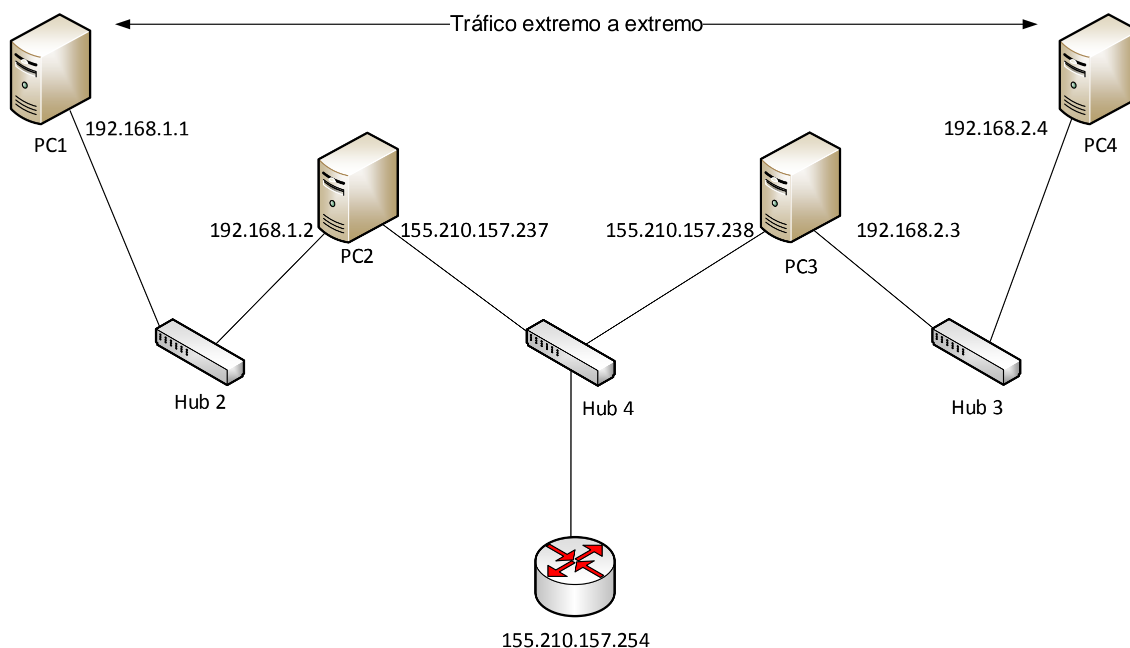

Figura 4.1. Escenario Ethernet.

En este diseño, podemos ver tres subredes: La red 192.168.1.0, donde se encuentran los terminales PC1 y PC2; la red central, con direcciones IP públicas, formada por el PC2 y PC3; finalmente tenemos la subred 192.168.2.0, formada por el PC3 y PC4.

Los equipos PC1 y PC4 funcionan como equipos terminales en la comunicación. Cada uno de ellos pertenece a una red distinta. El PC2 y el PC3 ejercen de *Gateway* en cada red y permiten la comunicación entre los equipos terminales.

Simplemux está activado tanto en el PC2 como en el PC3, de manera que el tráfico entre ellos podrá estar optimizado. Cuando el tráfico salga de la red central, los equipos PC2 y PC3 traducirán y reconstruirán los paquetes a su forma original, para que la optimización resulte transparente para los equipos terminales.

Este escenario se correspondería, por ejemplo, con el caso real de una empresa que quiere optimizar el tráfico entre oficinas remotas, y establece túneles Simplemux entre ellas. El esquema podría ser el que aparece en la Figura 4.2.

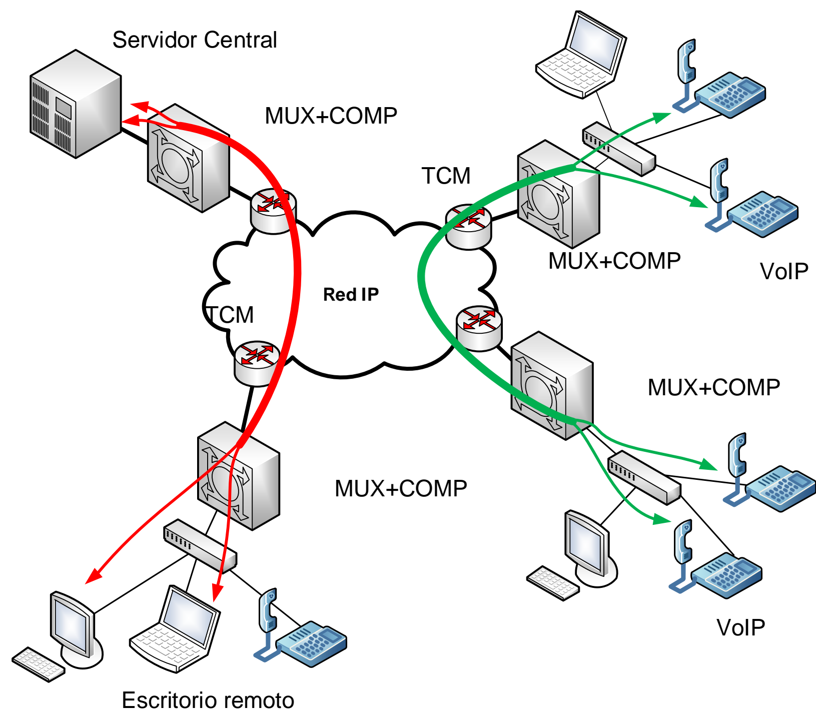

Figura 4.2. Esquema de optimización de tráfico entre oficinas de una empresa con TCM.

4.1.2 Escenario Wi-Fi (ad-hoc).

En este escenario (Figura 4.3) se añade una conexión inalámbrica entre los equipos PC2 y PC3. Se trata de una conexión 802.11 en modo ad-hoc, usando unas tarjetas de red que trabajan a 5,56 GHz. Están configuradas en esta banda para evitar posibles interferencias con otros dispositivos en la banda de 2,4 GHz. Gracias a la configuración de *iptables,* establecemos el encaminamiento de manera que la comunicación entre el PC1 y el PC4 se realice a través de la conexión inalámbrica. Para salir al exterior usarán el *router* 155.210.157.254, como en el caso del escenario Ethernet.

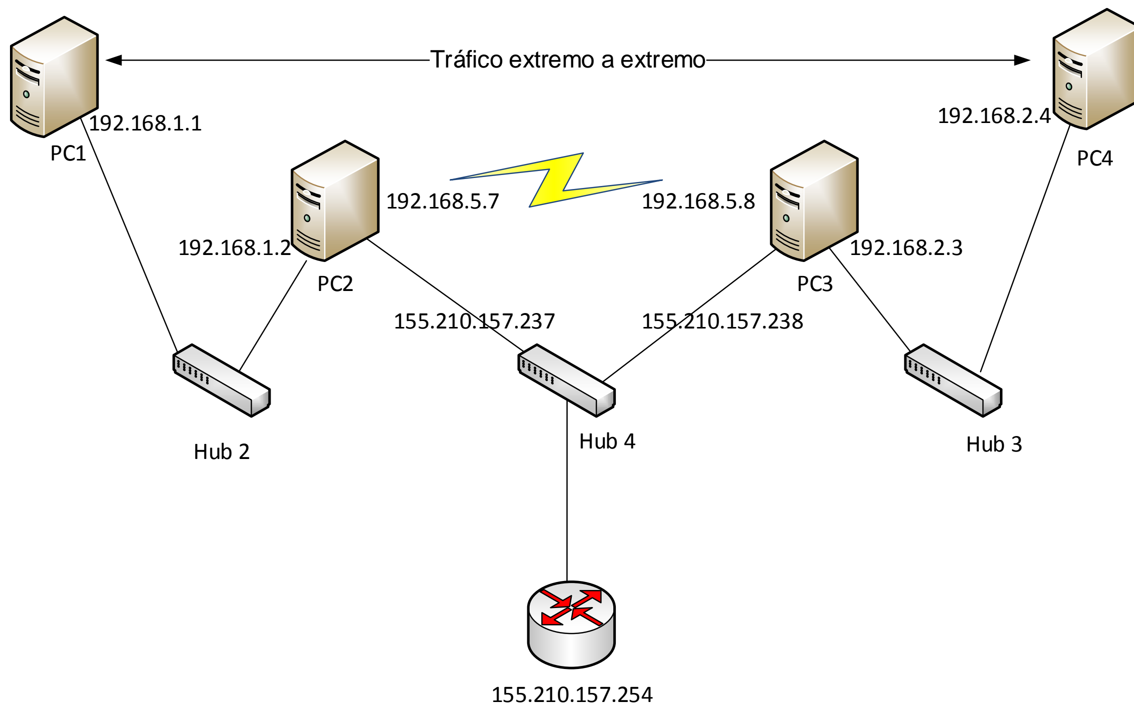

Figura 4.3. Escenario Ad-hoc.

Este escenario se corresponde con lo que sucede, por ejemplo, en las denominadas *Community Networks* son redes basadas en enlaces inalámbricos [18], usadas en entornos rurales, donde la optimización de tráfico puede resultar interesante. Normalmente, en áreas metropolitanas, el proveedor de servicios de Internet es el dueño de la infraestructura de red, o bien es una empresa que alquila la infraestructura de red para dar servicio a sus clientes. El problema radica en aquellas zonas en las que la densidad de población es muy baja y no resulta rentable económicamente para ninguna compañía realizar una inversión en infraestructura para dar servicio a esa área. Es en estas zonas donde las *Community Networks* se están empezando a desarrollar.

Otro escenario de interés sería el siguiente: una comunidad aislada (p.ej. un pueblo en una zona remota) posee una infraestructura propia para conexión a Internet. En este caso, puede suscribirse un acuerdo entre la comunidad y una compañía de telefonía que quiera dar servicio en esa zona. La comunidad puede ceder al operador parte del ancho de banda que no utiliza, y así el operador puede colocar una femtocelda 3G o 4G, y de esta manera dar servicio al área rural con un coste mucho menor al de si tuviera que desplegar su propia red cableada (Figura 4.4).

Con la femtocelda se consigue cobertura celular móvil gracias a una comunicación punto a punto Wi-Fi. Dicha implementación está siendo investigada e implementada, por ejemplo, en el proyecto europeo TUCAN3G[11] [19].

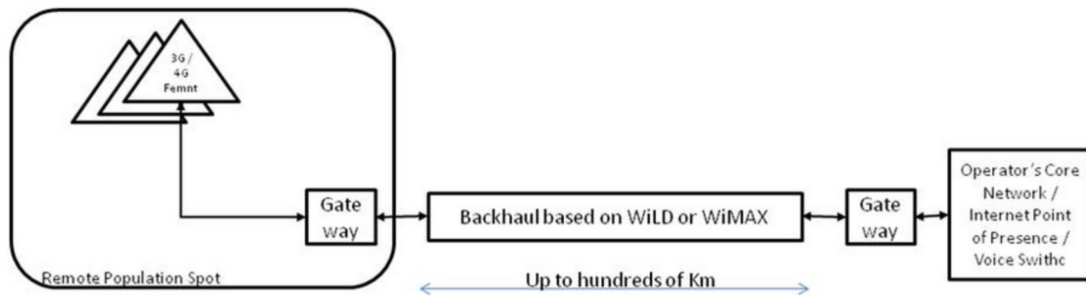

Figura 4.4. Esquema de implantación celular a través de enlace punto a punto Wi-Fi. Fuente: Web del Proyecto TUCAN3G.

4.2 Herramientas utilizadas en las pruebas

Para la realización de las pruebas hemos utilizado una serie de herramientas que han permitido modificar parámetros de red como el *jitter* o las pérdidas, así como un generador de tráfico.

4.2.1 D-ITG

D-ITG es un generador de tráfico capaz de enviar paquetes tanto IPv4 como IPv6 con diferentes características. Soporta tanto tráfico UDP como TCP, puede simular tráfico de juegos online (*Quake III*) o tráfico multimedia RTP. Además, esta plataforma está dotada de un sistema de análisis estadístico en el que se puede obtener el retardo de los paquetes, el *jitter*, el *throughput*, etc.

[11] Tucan 3G, Wireless technologies for isolated rural communities in developing countries based on cellular 3G femtocell deployments, http://www.ict-tucan3g.eu/

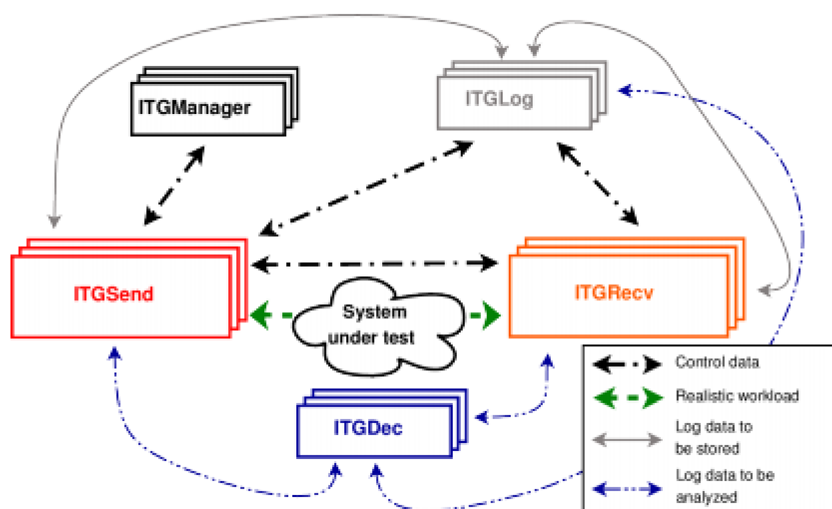

Figura 4.5. Arquitectura D-ITG.
Fuente: Página web de D-ITG, http://traffic.comics.unina.it/software/ITG/manual/

El generador se compone de un programa emisor *ITGSend* y un programa receptor *ITGRecv.* D-ITG utiliza dos canales de transmisión: uno para la señalización y otro para los datos. Ambos programas pueden generar ficheros de *log* en los que se almacenan resultados estadísticos de la comunicación. Dichos ficheros de *log* necesitarán decodificarse para su posterior lectura a través del sub-programa *ITGDec*.

El funcionamiento y configuración del simulador de tráfico D-ITG se puede encontrar en el Anexo B de esta memoria.

4.2.2 Linux Traffic Control

La herramienta *Traffic Control* (*TC*) [20], incluida dentro de prácticamente todas las distribuciones de Linux, permite ajustar el tráfico en una interfaz de red. El proceso es controlado a través de tres tipos de objetos: colas, clases y filtros.

TC soporta diferentes disciplinas de cola (denominadas *qdisc*), que permiten gestionar la forma en que se envían y reciben los datos. Su funcionamiento se basa en reordenar, retrasar o descartar los paquetes. A su vez, las colas pueden tener clases, lo que permite aplicar criterios diferentes a cada tipo de tráfico. Para encolar los paquetes en las distintas colas se pueden utilizar filtros.

Para la realización de las pruebas no ha sido necesaria la configuración de clases ya que sólo se quería establecer un determinado ancho de banda, aplicar pérdidas o modificar el tiempo entre paquetes.

4.2.3 Capturas de tráfico

Para capturar tráfico se ha utilizado la herramienta de GNU/Linux *tcpdump*, que permite capturar el tráfico en una interfaz de red utilizando la librería de captura de paquetes *libpcap*, es decir, nos permite monitorizar los paquetes que vienen dirigidos a dicha interfaz.

A su vez, se puede definir una serie parámetros o filtros para seleccionar el tráfico que queremos capturar o monitorizar y guardarlos para su posterior análisis en un fichero *.pcap*.

4.3 Equipos

A continuación se enumeran brevemente las características de los equipos que conforman los escenarios que aparecen en el Apartado 4.1 de esta memoria.

- PC1, PC2, PC3 y PC4 (Sistema operativo Debian GNU/Linux 6.0.9, procesador Intel Core i3 CPU M370 2,4Ghz, tarjeta de red Realtek PCIe GBE Family Controller).
- Hub1, Hub2 y Hub3: concentrador 3Com de 100Mbps (modelo 3C16611 SuperStack® II Dual Speed Hub 500 24-Port TP).
- Switch: Conmutador 3Com de 100Mbps (modelo 3CR17561-91 SuperStack®3 Switch 4500 26-Port).
- Adaptador Inalámbrico USB de doble Banda Archer T2U (AC 600Mbps, USB2.0).
- Tarjetas de red externas USB D-Link (modelo DUB E100 USB 2.0 Fast Ethernet adapter).
- Punto de acceso TP-LINK TL-WR1043ND de doble banda.

Capítulo 5

Análisis de resultados

5.1 Pruebas UDP

5.1.1 Juegos Online

Como se comentó en el Apartado 3.1.2, los juegos online son un objetivo directo de la optimización de tráfico, por su excesivo *overhead*. Por ello, se han realizado pruebas con un juego online típico como es *Quake III*. Se ha elegido este juego porque su perfil de tráfico se adecúa al de la mayoría de juegos online *First Person Shooter*, caracterizado por la transmisión de paquetes pequeños a una alta tasa de paquetes por segundo.

Las pruebas con este tipo de tráfico se han realizado en el escenario Ethernet (Apartado 4.1.1 de esta memoria).

En la Figura 5.1 se puede observar la comparativa entre el consumo de ancho de banda en una conexión con 20 flujos simultáneos con el tráfico nativo del juego, y con el tráfico optimizado. La optimización del enlace se ha realizado con un periodo de multiplexión de 10 ms. De esta manera, el ahorro se consigue a costa de introducir un pequeño retardo en la comunicación, que para algunos paquetes (los que lleguen al inicio del periodo) será como máximo de 10 ms, y de 5 ms en media. Desde el punto de vista del usuario, la usabilidad prácticamente no se verá afectada ya que se considera que para juegos online, incrementar el *ping*[12] en unos 5 ms resultará inapreciable para el usuario [21].

[12] Ping: término que se utiliza para referirse al *lag* o latencia de la conexión de juegos online. Se mide como el tiempo que le cuesta a un paquete ir desde el equipo del usuario hasta el servidor, y volver.

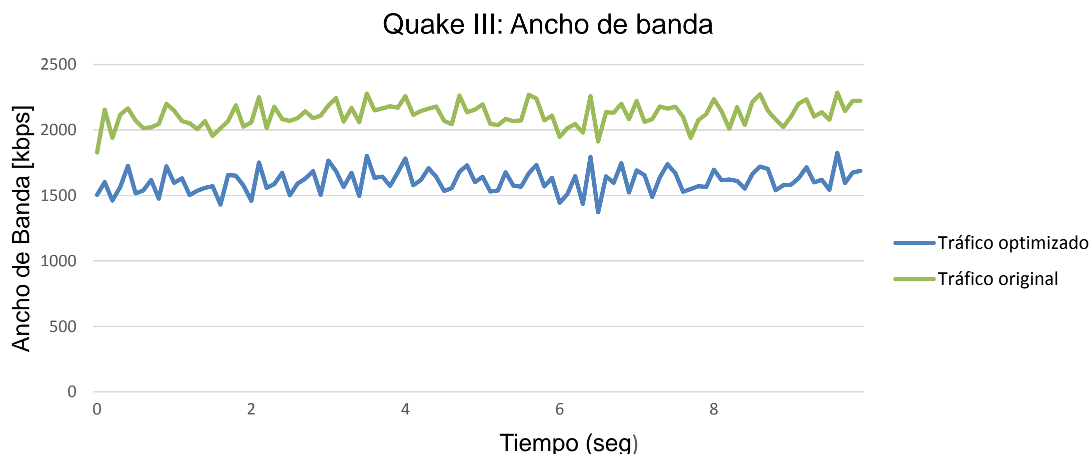

Figura 5.1. *Quake III*: Comparativa ancho de banda.

Analizando los paquetes por segundo (Figura 5.2), resulta evidente que se consigue disminuir drásticamente la tasa de envío de paquetes gracias a la multiplexión de estos paquetes pequeños en otros más grandes.

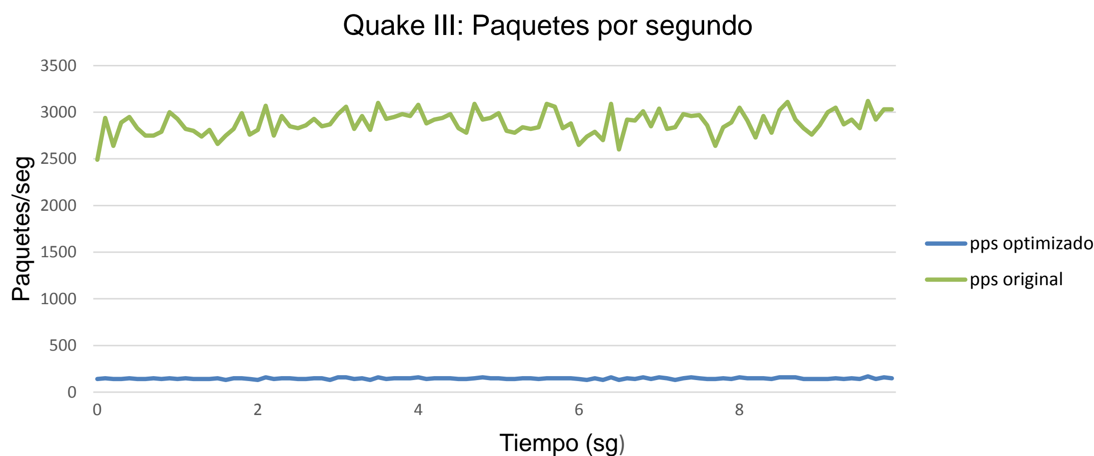

Figura 5.2. *Quake III*: Comparativa paquetes por segundo.

Gracias a la multiplexión, se puede enviar la misma información útil que de forma nativa, pero con muchos menos paquetes circulando por Internet, aunque de mayor tamaño. A su vez, esto disminuirá la carga de procesado de los *router* intermedios de la red, al tener que gestionar un número menor de paquetes.

Los resultados de las pruebas para 20 flujos de comunicación con un periodo de multiplexión de 10 ms se pueden resumir en la siguiente tabla.

	BW (kbps)	Paquetes/sg
Tráfico original	2115	2885
Tráfico optimizado	1610	146
Ahorro	23,8 %	95 %

Tabla 5. Ahorro de ancho de banda para *Quake III*.

5.1.2 VoIP

Otro tipo de tráfico donde el uso de Simplemux va a resultar interesante, es el tráfico de VoIP. Como se vio en el Apartado 3.1.1, este tráfico contiene un alto *overhead*, dado que el *payload* de este tipo de aplicaciones no llega en ocasiones al 60% del tamaño total del paquete IP.

Dado que este tráfico funciona sobre RTP, ROHC podrá no sólo comprimir las cabeceras IP y UDP de cada paquete, sino también la cabecera RTP, consiguiéndose mejores resultados de ahorro que en el caso de juegos online.

Las pruebas se han realizado sobre el escenario Ethernet para distintos códec de VoIP. En las primeras pruebas utilizaremos el códec GSM. En la Figura 5.3 se puede ver que se logra alcanzar el 47,19 % de ahorro de ancho de banda, si lo comparamos con la comunicación GSM nativa. Este porcentaje se consigue cuando el paquete enviado alcanza el MTU y ya no se pueden multiplexar más paquetes.

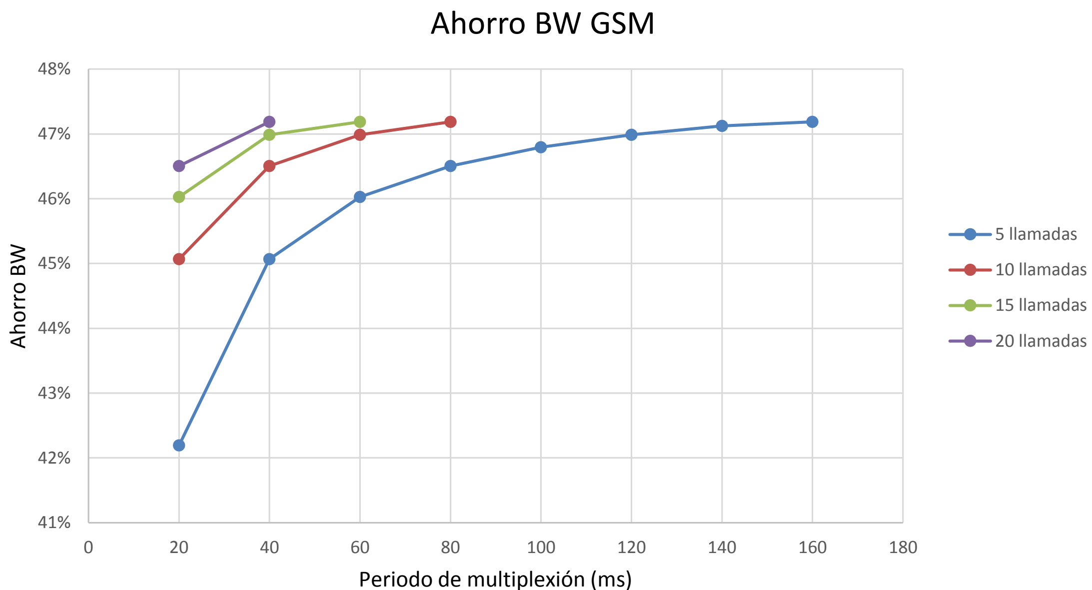

Figura 5.3. VoIP: Ahorro de ancho de banda con GSM optimizado.

Se puede apreciar que, cuantas más llamadas simultáneas tengamos, más fácilmente se podrá llegar a ese límite de ahorro, es decir, necesitaremos un periodo de multiplexión menor. Como inconveniente, si se tienen pocas llamadas simultáneas y queremos conseguir el máximo ahorro, deberemos sacrificar algo de calidad de la llamada al incrementar el retardo. De manera que nos encontramos con un compromiso entre calidad y ahorro.

Para medir la calidad en una llamada IP, suele utilizarse el MOS *(Mean Opinion Score),* factor que disminuye con el retardo. Como se ve en la Tabla 6, los valores del retardo adicional que estamos manejando sólo supondrían un pequeño decremento en el MOS.

	Retardo (ms)	Factor MOS
Excelente	t < 50	MOS > 4,3
Bueno	50 < t < 150	MOS > 4
Aceptable	150 < t < 300	MOS > 3,6
Inaceptable	t > 300	MOS < 3,6

Tabla 6. Factor MOS para distintos valores del retardo. Fuente: PacketGuide™, Blue Coat's Product Information Source, https://bto.bluecoat.com/packetguide/9.2/info/voip-rfactor.htm

Por lo tanto, a la hora de configurar un enlace para su optimización, deberemos guardar un equilibrio entre ahorro y retardo adicional. Para ello se deberá analizar el volumen de llamadas que va a tener que soportar el enlace, y en función de eso, tomar una decisión a la hora de elegir el periodo de multiplexión.

Como sucedía con el tráfico de los juegos *online*, al multiplexar estos paquetes pequeños en uno (que alcance prácticamente el MTU), se consigue un ahorro significativo en la tasa de paquetes por segundo. Como se aprecia en la Figura 5.4, en este caso el ahorro puede llegar hasta un 97 %. Gracias a este hecho, la carga de procesado de los *router* intermedios se puede reducir drásticamente, de manera que la utilización de este método de optimización podría servir para paliar problemas de congestión de la red.

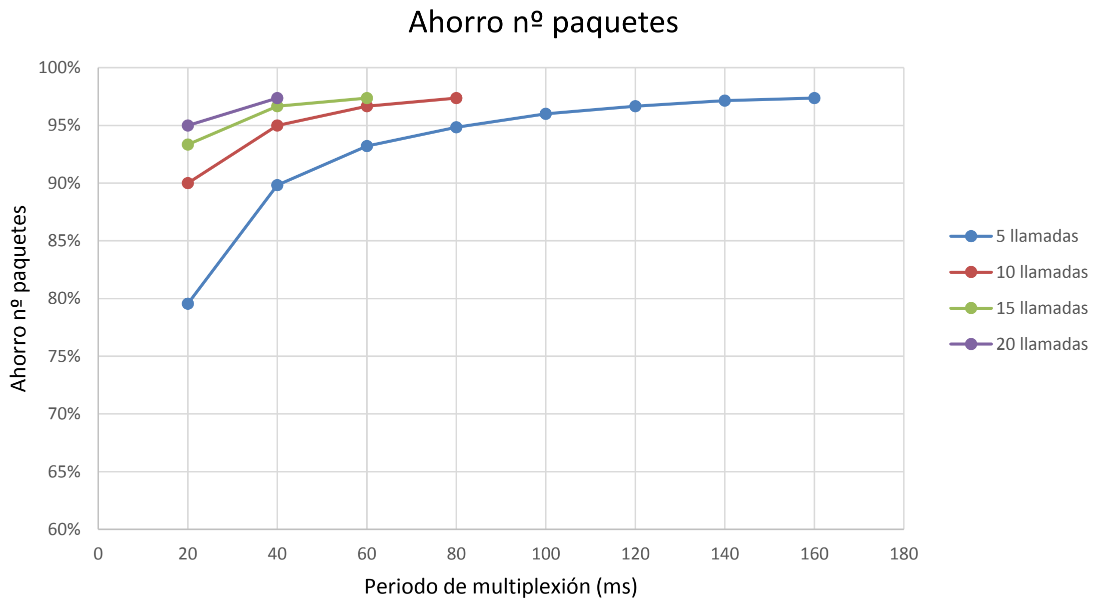

Figura 5.4. VoIP: Ahorro en paquetes por segundo con GSM optimizado.

En la Figura 5.5 se muestra la diferencia entre el tamaño de los paquetes nativos GSM y el tamaño de los paquetes de tráfico optimizado. En estas pruebas se han conseguido encapsular hasta 38 paquetes nativos (tamaño original paquete GSM: 73 bytes a nivel IP) en un solo paquete de 1.465 bytes, alcanzando casi el MTU de la red.

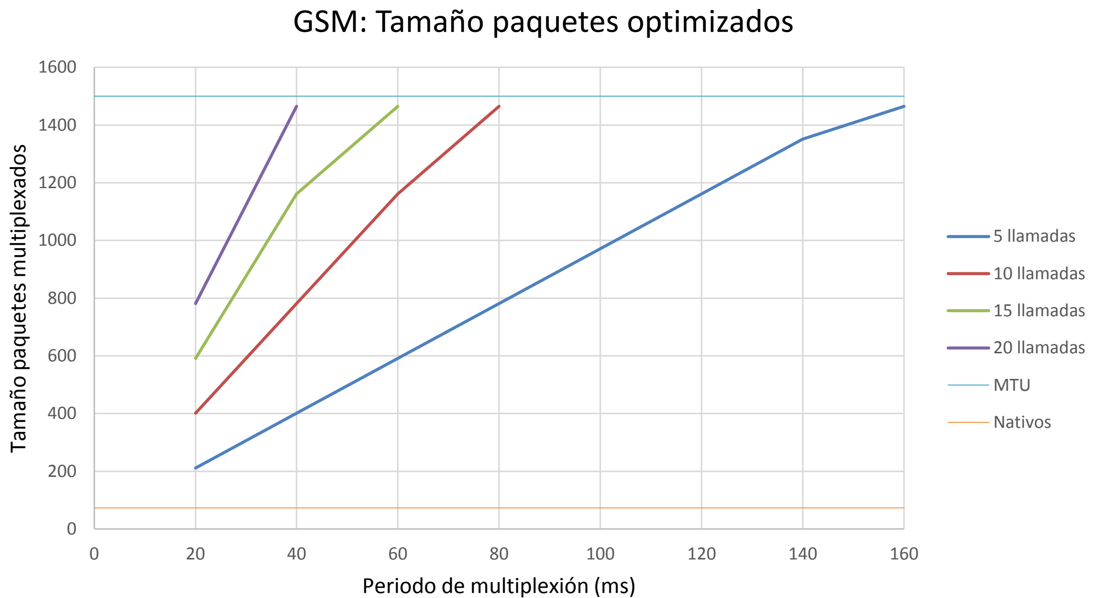

Figura 5.5. VoIP: Tamaño paquetes GSM optimizados.

Una vez analizado el tráfico de VoIP con el códec GSM, se considera de interés compararlo con otras codificaciones y su comportamiento al aplicarle la optimización mediante Simplemux. En la Figura 5.6 se realiza una comparativa del ahorro que se produce al realizar la optimización de tráfico en distintas codificaciones. Como era de esperar, aquella codificación menos eficiente (GSM) será aquella en la que más ahorro conseguiremos.

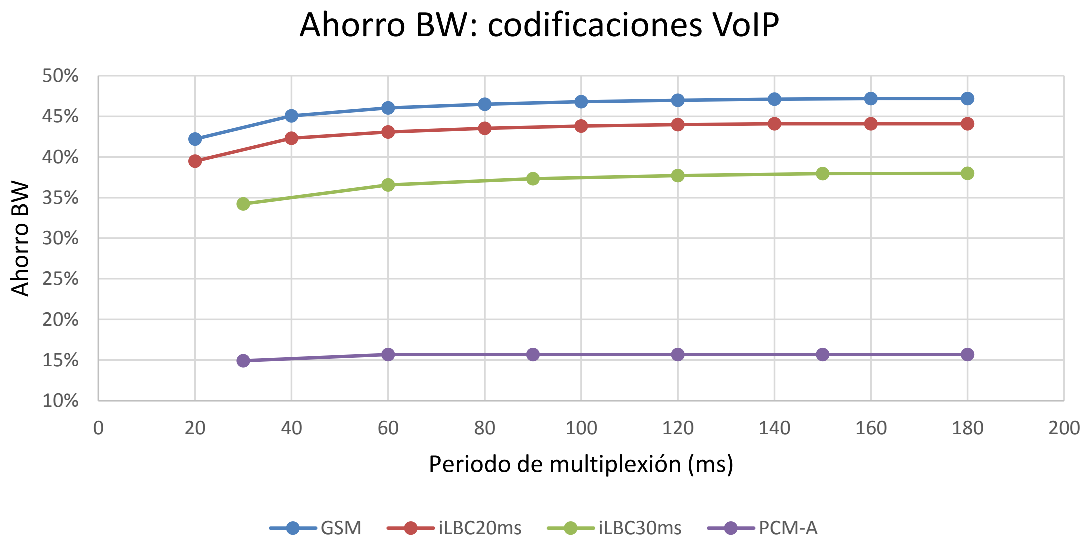

Figura 5.6. VoIP: Comparativa de BW con distintas codificaciones optimizadas.

Como vimos en el Apartado 3.1.1, el códec G.711 (codificación PCM ley A o ley µ) transmite un paquete cada 33 ms, con un tamaño de 172 bytes a nivel de aplicación (212 bytes a nivel IP). De manera que los *payload* no son tan pequeños como por ejemplo GSM (45 bytes). Por lo tanto, en un paquete multiplexado caben menos paquetes nativos de G.711 y el ahorro que conseguimos con este códec es solamente del 15,69%, bastante menor del 47,19% que conseguíamos con GSM.

Donde sí notaremos ahorro, con cualquier codificación, es en la tasa de envío de paquetes. En la Figura 5.7 podemos ver tasas de paquetes por segundo que suponen de un 85 a un 97 % menos que de forma nativa. Esto es debido a que en el caso de GSM se consigue multiplexar hasta 38 paquetes GSM en un paquete IP, y en el caso del códec G.711 (codificación PCM-A) se consigue encapsular hasta 7 paquetes en uno IP. Como sucedía en el caso de juegos online, este ahorro produce como contrapartida un aumento en el retardo de la comunicación, al añadir un buffer de multiplexión.

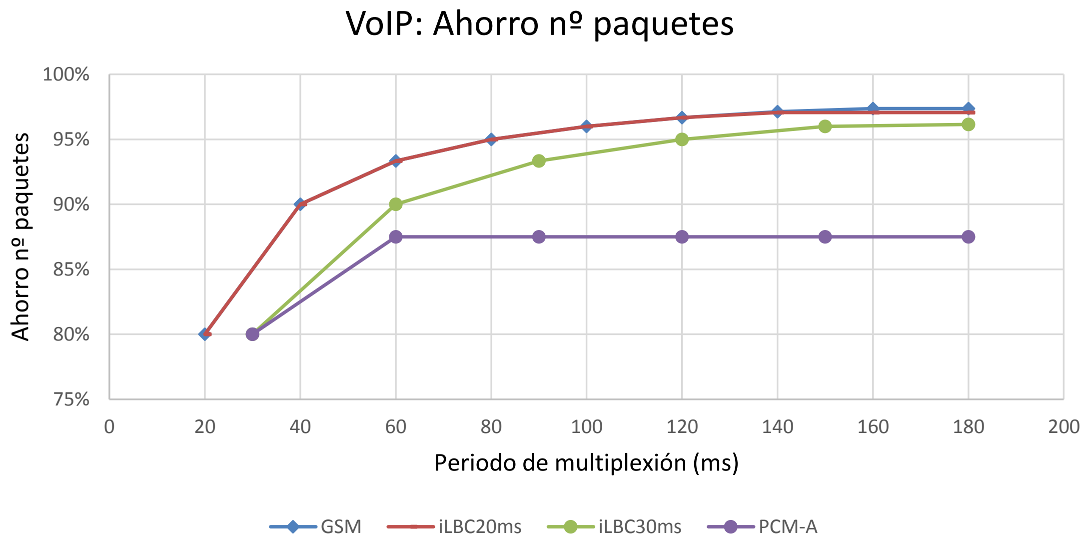

Figura 5.7. VoIP: Comparativa pps para distintas codificaciones optimizadas.

5.1.3 Enlace congestionado

Hay situaciones en las que la red de un proveedor de VoIP o de videojuegos puede estar congestionada. Hasta ahora, para prevenir la congestión, los gestores del servidor solían optar por sobredimensionar el enlace en el que se produce el cuello de botella. La capacidad de un nodo de comunicaciones de la red viene determinada por dos parámetros, su ancho de banda (del orden de Gbps en la actualidad) y su capacidad de procesado de paquetes por segundo. Por lo tanto, el estado de congestión se producirá cuando se sobrepase alguno de estos dos parámetros.

Para evitar este tipo de situaciones, se propone utilizar la herramienta Simplemux, por el ahorro que produce tanto en el ancho de banda como en la tasa de paquetes por segundo. De este modo, evitamos la congestión a través de software en vez de hardware. Como se ha comentado anteriormente, Simplemux también puede proporcionar flexibilidad: puede permanecer desactivado en situaciones normales, y activarse cuando se detecta un incremento significativo del tráfico.

La siguiente prueba (Figura 5.8) se realizó en un enlace Wi-Fi en modo Ad-hoc entre el PC2 y el PC3 (escenario del Apartado 4.1.2). Se configuró el enlace a 9 Mbps, y se realizó una transmisión de paquetes de 60 bytes a 7,2 Mbps a nivel IP. El enlace se satura porque, al ser una transmisión Wi-Fi, el rendimiento es mucho menor de 9 Mbps. Como se puede ver en la figura (columna “nativo”), las pérdidas están por encima del 60%, por lo que la tasa real es mucho menor de 9 Mbps. Sin embargo, si multiplexamos paquetes, las pérdidas se van reduciendo. Si además se aplica ROHC, el ahorro es aún mayor y las pérdidas se reducen aún más.

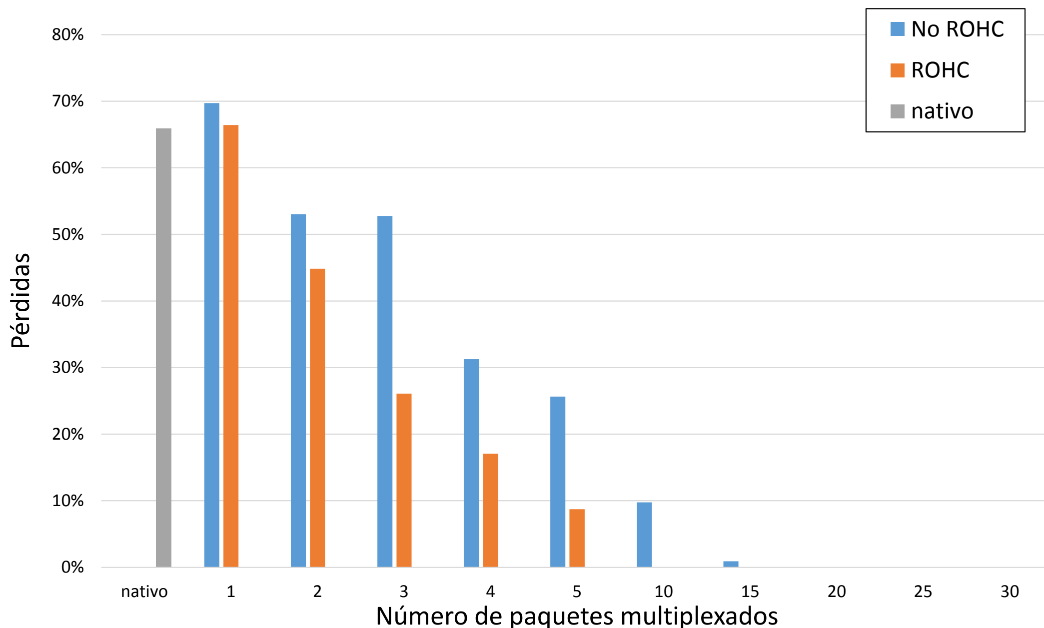

Figura 5.8. Pérdidas Wi-Fi en enlace congestionado.

La explicación de que conforme vamos multiplexando paquetes, conseguimos disminuir las pérdidas (hasta eliminarlas), podemos buscarla en la Figura 2.5. Es decir, conforme más grande sea el campo de datos del paquete, menos *overhead* se producirá y el algoritmo de *back off* de Wi-Fi resultará menos significativo que para paquetes pequeños. Gracias a la multiplexión, conseguimos pasar de un enlace en estado de congestión en el que se pierde el 65 % de los paquetes a un enlace sin pérdidas.

5.2 Pruebas con TCP

A lo largo de la memoria se ha ido hablando del *overhead* de las comunicaciones IP. Pero en el caso de TCP hay un tipo de paquete que se podría considerar por antonomasia puramente *overhead,* al no llevar *payload*: se trata de los *TCP Acknowledgement* (ACK). Este tipo de paquete se utiliza como mecanismo de control, y es enviado por el receptor para informar al emisor que ha recibido correctamente uno o varios paquetes. Estos ACK viajan por la red como paquetes normales. Así que dada su ineficiencia, podría resultar interesante multiplexar los ACK para optimizar este tipo de tráfico. En comunicaciones bidireccionales TCP, se suelen enviar los ACK incluidos en un paquete TCP con datos, técnica que se conoce como *piggybacking*. Este no será nuestro caso a optimizar, ya que ahí el ACK no está produciendo *overhead* en el enlace. Nos centraremos en los ACK sin *payload*, que son la mayoría, como veremos.

En los casos en los que sí se mandan ACK sin *payload* por la red es donde utilizaremos Simplemux para ver qué mejoras puede aportar a la comunicación. El funcionamiento de TCP consiste en buscar la congestión para después controlarla y así transmitir a la máxima velocidad

posible que le permite el enlace. Este sistema lo realiza a través del crecimiento de una ventana de congestión que depende de los ACK recibidos. Por tanto, hay que tener especial cuidado con la acumulación de paquetes ACK para multiplexar, porque el retardo añadido podría influir en el crecimiento en la velocidad de transmisión de TCP.

Para evaluar esta influencia se han realizado varias pruebas usando el mismo escenario Ethernet del Apartado 4.1.1 y se ha optimizando tráfico de ACKs de TCP, mientras se descargaba un fichero con FTP. El esquema de las pruebas realizadas se puede ver en la Figura 5.9:

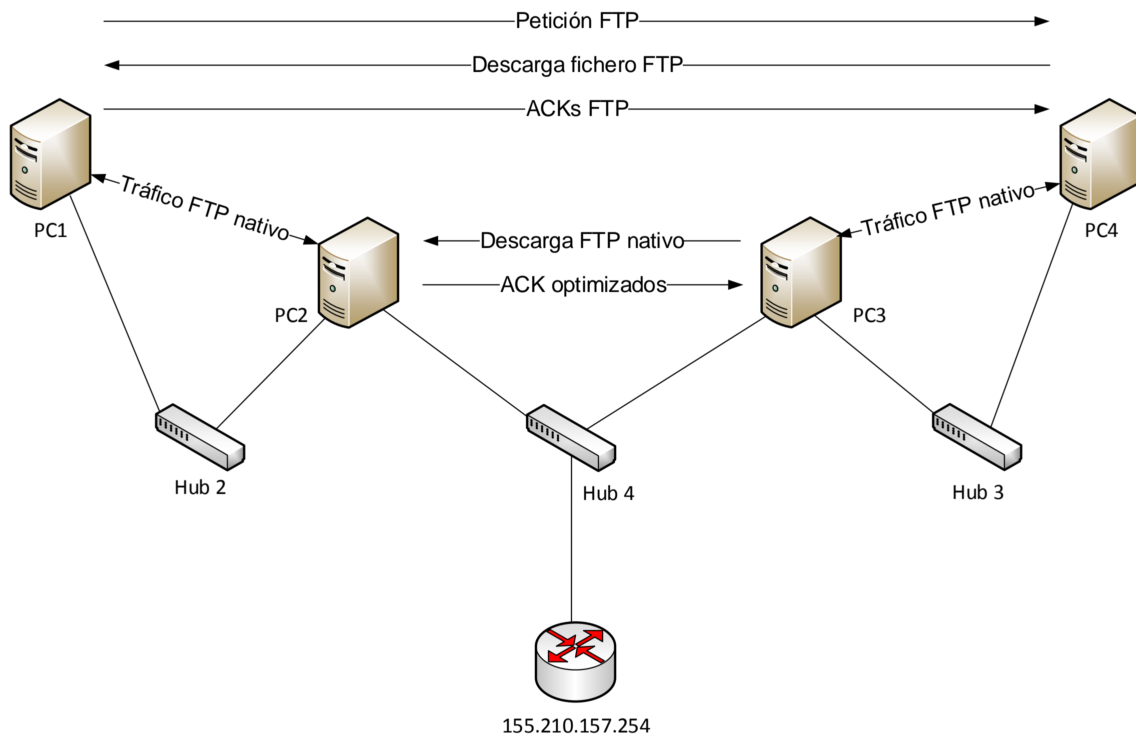

Figura 5.9. Esquema pruebas TCP.

Como se observa en la Figura 5.10, si optimizamos los paquetes ACK, obtendremos un ahorro que podría llegar al 55 %. Este ahorro se logra gracias al protocolo ROHC, que consigue comprimir las cabeceras de los paquetes (recordemos que estos paquetes sólo constan de cabecera).

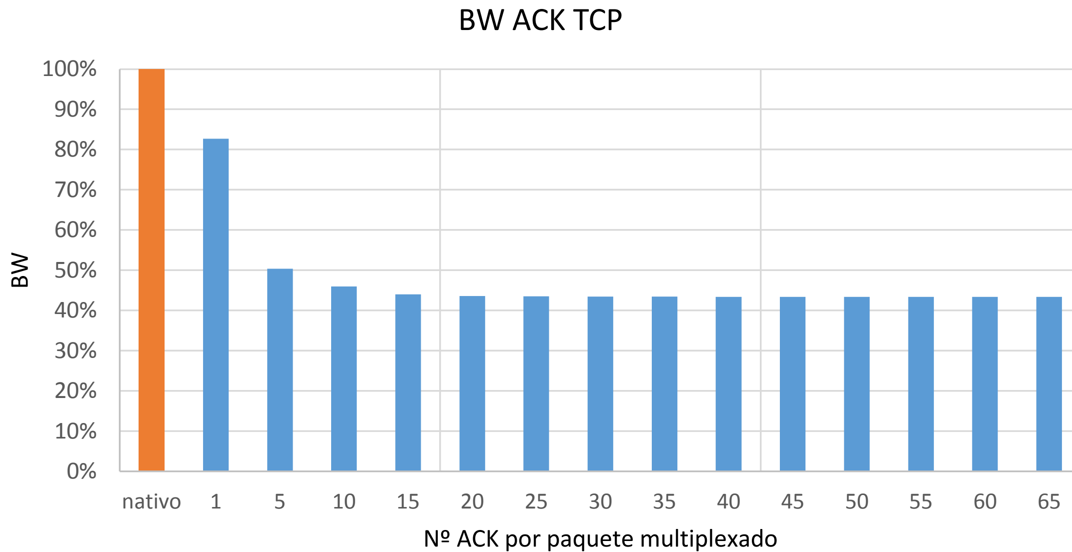

Figura 5.10. Ahorro de ancho de banda optimizando paquetes ACK.

Como se ha comentado anteriormente, es muy importante que con tráfico TCP, los paquetes ACK no se almacenen durante mucho tiempo, es decir, el periodo de multiplexión debe ser lo más bajo posible, para no interferir en el crecimiento de la ventana TCP, y así evitar que la comunicación empeore.

Para observar este efecto, se han realizado las pruebas mostradas en la Figura 5.11. Se han multiplexado los ACK de una sola comunicación TCP, utilizando diferentes periodos. Aunque se ahorra ancho de banda en los ACK, el tiempo de transmisión del fichero ha aumentado con el periodo de multiplexión (línea roja), cuando éste ha sido excesivo. Para periodos bajos (1 ms, 2 ms), la transmisión ha durado prácticamente lo mismo que de forma original (próximo al 100%). Sin embargo, podemos apreciar que conforme aumentamos el periodo de multiplexión, el tiempo de transmisión del fichero aumenta considerablemente. De hecho, para un periodo de 30 ms, tarda más del triple del tiempo de lo que lo haría con tráfico sin optimizar. Comprobamos por tanto que la multiplexión de los ACK puede tener una influencia negativa en el crecimiento del ancho de banda de una transmisión TCP.

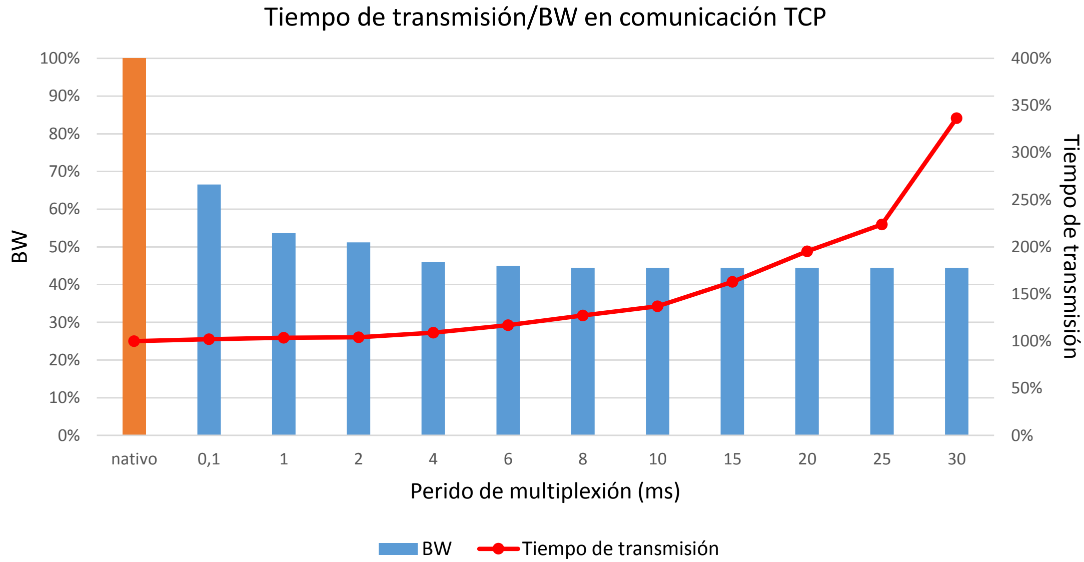

Figura 5.11. Ancho de banda y Tiempo de transmisión de un fichero en función del periodo de multiplexión.

Dado que TCP está diseñado para compartir la capacidad del enlace con el resto de comunicaciones, parecía interesante realizar pruebas con dos transmisiones simultáneas compartiendo el mismo enlace (Figura 5.12): en una de ellas se multiplexan los ACK, y la otra no sufre ninguna modificación en sus paquetes. En los test que se llevaron a cabo se configuró un máximo de n=10 paquetes multiplexados por trama. También se ha establecido un periodo de multiplexión máximo de 10 ms, para que la fase de establecimiento de TCP funcione con normalidad.

En algunas de las pruebas se han añadido artificialmente pérdidas y *jitter* en el enlace central (entre el PC2 y PC3) utilizando la herramienta *TC*. Cada prueba se ha repetido 100 veces y se ha hecho la media de los resultados obtenidos. El esquema de funcionamiento ha sido el siguiente:

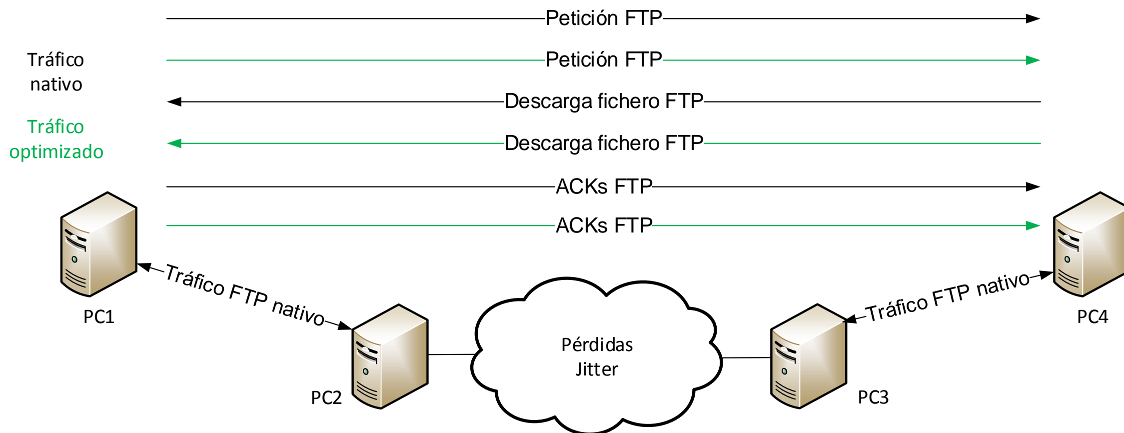

Figura 5.12. Esquema pruebas FTP.

En la Figura 5.13 se representan los resultados de estas pruebas. Las barras azules representan el ancho de banda obtenido por el tráfico no optimizado, mientras que las barras naranjas representan el tráfico optimizado con Simplemux. El ancho de banda ha sido calculado a partir del tiempo que tarda en descargarse el fichero (tamaño 22 Mbytes).

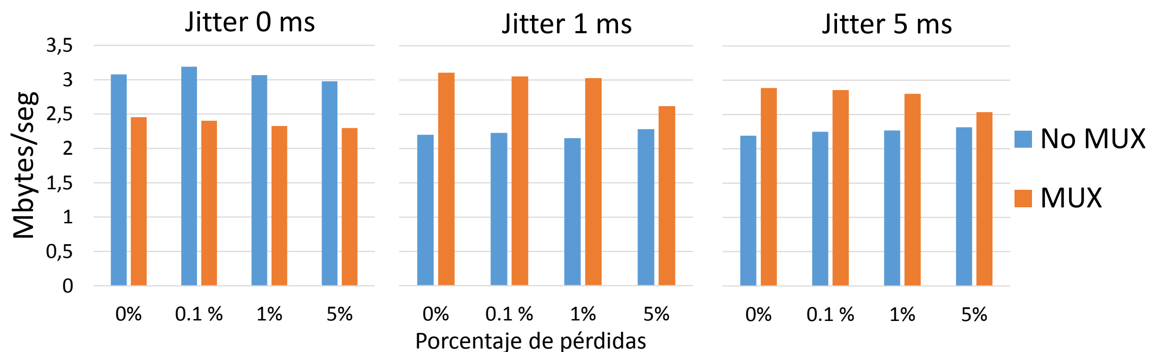

Figura 5.13. Comparativa del ancho de banda obtenido por dos transmisiones de ficheros con TCP.

En la figura podemos observar que en situaciones ideales (sin pérdidas ni *jitter,* primer par de columnas de la izquierda) la comunicación no multiplexada consigue transmitir más rápido que su competidor en el enlace: en concreto, logra un 11,24 % más de ancho de banda.

Observamos que esto cambia en el momento en el que aparece *jitter* en la red (1 ms o 5 ms): se produce un efecto no esperado a priori: los flujos multiplexados consiguen más ancho de banda. Una posible explicación es que Simplemux está almacenando ACK en un buffer, y los manda todos juntos, de manera que el emisor, en el momento en que los recibe, puede hacer crecer su ventana de transmisión rápidamente. En la comunicación original, al producirse *jitter*, los ACK no llegan con una periodicidad normal, y esto produce que la ventana de congestión no crezca de manera natural.

Analizando qué ocurre en el enlace cuando se producen pérdidas, podemos concluir que estas afectan más al tráfico optimizado: mientras las columnas azules varían muy poco, las columnas naranjas (derecha) muestran una mayor variabilidad con las pérdidas. El motivo es que cuando se produce una pérdida de un paquete Simplemux, no sólo se pierde un ACK, sino un conjunto de ellos (varios ACK encapsulados en un paquete Simplemux). A priori esto no debería ser un problema, siempre que se reciba seguidamente un ACK con número de secuencia superior. Pero el problema radica en que en estas pruebas se ha utilizado ROHC y al perderse varios ACK el descompresor se ha desincronizado y no es capaz de descomprimir los siguientes paquetes correctamente. Por tanto deberá descartar los paquetes que no sepa descomprimir y pedirá al compresor que le envíen los paquetes originales con su cabecera completa para poder sincronizarse de nuevo. Todo esto provocará que la transmisión se ralentice.

Podemos concluir que en casos de *jitter* el tráfico multiplexado obtiene ventaja frente al nativo, y en casos de pérdidas, la ventaja la obtiene el tráfico nativo. Si no hubiéramos realizado compresión de cabeceras mediante ROHC no se habría obtenido esa desventaja debida a las

pérdidas. Por otro lado, cuando combinamos pérdidas con *jitter*, se produce un mayor equilibrio entre los dos tráficos, aunque viendo los resultados prácticos parece más influyente el *jitter* que las pérdidas a la hora de competir por el ancho de banda de un enlace.

5.3 Traza general

Hasta ahora se ha utilizado Simplemux para optimizar flujos pertenecientes a un tipo de tráfico, como VoIP, juegos o una descarga de un fichero usando TCP. En esta sección se estudiará el ahorro conseguido para una traza general de tráfico. Por esta razón, se utilizará una captura de tráfico cuya distribución estadística corresponda al tráfico en Internet. Se ha utilizado una traza disponible en CAIDA[13], un repositorio público en el que se ofrecen trazas de tráfico. Se ha seleccionado una traza obtenida en un enlace de 10 Gbps entre dos nodos situados en Chicago y Seattle respectivamente. La traza se analizó con el objetivo de obtener la distribución de paquetes según sus tamaños, como se muestra en la Figura 5.14.

La distribución muestra que el 46 % de los paquetes son menores de 200 bytes, el 41 % de los paquetes son de (alrededor de) 1500 bytes. El tamaño medio es de 753 bytes.

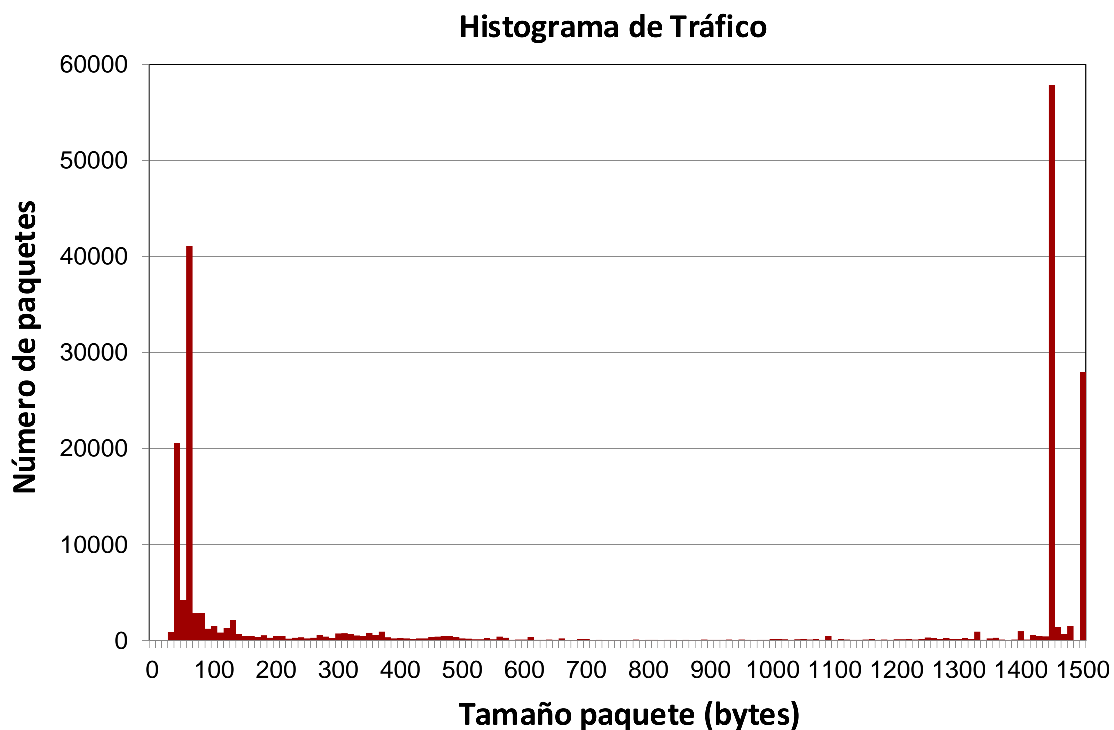

Figura 5.14. Histograma de tráfico real entre dos nodos centrales.

Como se puede observar en el histograma, casi la mitad de los paquetes pueden ser considerados "pequeños" y es en estos donde será eficiente la utilización de Simplemux. Por lo tanto, con una correcta configuración de *iptables* se podría conseguir que se optimicen sólo los

[13] Center for Applied Internet Data Analysis, http://www.caida.org/

paquetes cuyo tamaño sea inferior a un umbral, y que los paquetes mayores atraviesen el enlace sin modificación alguna, como se muestra en la Figura 5.15.

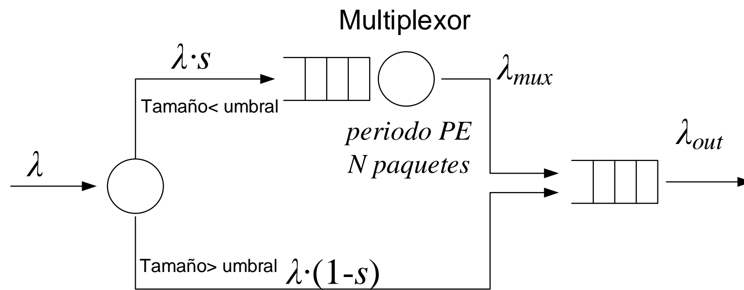

Figura 5.15. Esquema de clasificación de paquetes.

Para tener una idea del ahorro que se puede esperar (en términos de ancho de banda y de paquetes por segundo), deberemos realizar un análisis, con el objetivo de obtener la tasa de salida de paquetes λ_{out}, así como el ancho de banda de salida.

Siguiendo este esquema, tenemos una tasa de llegada de paquetes λ, y siendo *s* la probabilidad de que un paquete sea menor que el umbral:

$$s = P\ [tamaño\ paq. \leq\ limite]$$
$$1 - s = P\ [tamaño\ paq. >\ limite]$$

La configuración del multiplexor vendrá dada por un periodo, *PE*, y un número máximo de paquetes multiplexados, *N*. Como se explicó en el Apartado 3.2.2, el paquete multiplexado saldrá del nodo en el momento en que se alcance el periodo *PE* o el número de paquetes *N*.

Si llamamos *k* al número de paquetes que llegan al multiplexor en un periodo, tendremos tres casos posibles: no llega ningún paquete; se cumple el periodo *PE;* se alcanza el número de paquetes *N*.

Por tanto, la tasa de salida del multiplexor será:

$$\lambda_{mux} = \lambda_{mux|k=0} \cdot Pr\ (k = 0) + \lambda_{mux|0<k<N} \cdot Pr\ (0 < k < N) +$$
$$+ \lambda_{mux|k=N} \cdot Pr\ (k = N)$$

Desarrollando esta expresión:

- Si no llegan paquetes en un periodo:

$$\lambda_{mux} = \lambda_{mux|k=0} = 0$$

- Si en un periodo llegan *k* paquetes y *k* es menor a *N*:

$$\lambda_{mux|0<k<N} = \frac{1}{PE}$$

- Finalmente, si llegan N paquetes antes de que finalice un periodo:

$$\lambda_{mux|k=N} = \frac{\lambda \cdot s}{N}$$

Así que simplificando la expresión nos queda:

$$\lambda_{mux} = \frac{1}{PE} Pr\,(0 < k < N) + \frac{\lambda \cdot s}{N} \cdot Pr\,(k = N)$$

En nodos con un nivel de tráfico elevado, podemos asumir que la cantidad de paquetes por segundo que llegan al nodo es alta, entonces siempre se alcanzará la cota de *N*, antes de que se llegue a un periodo PE. Por tanto, podríamos aproximar λ_{mux} por la siguiente expresión:

$$\lambda_{mux} \approx \frac{\lambda \cdot s}{N} \qquad si \qquad \Pr(k = N) \approx 1$$

$$\lambda_{out} \approx \lambda_{mux} + \lambda_{No\,mux} = \frac{\lambda \cdot s}{N} + \lambda \cdot (1 - s) = \lambda \left[\frac{s}{N} + (1 - s) \right]$$

Como conclusión, podemos ver que si el tráfico que optimizamos es de paquetes de pequeño tamaño (por ejemplo, se trata de tráfico de VoIP o juegos *online*), entonces $\mathrm{s} \approx 1$ y el factor de ahorro de paquetes por segundo será *N*. Si por el contrario, tenemos una traza real de Internet entre dos nodos centrales como la que muestra la Figura 5.14 y decidimos multiplexar los paquetes menores a un umbral, podríamos obtener unos ahorros de este orden (usando un umbral de 200 bytes para definir un paquete como pequeño):

$$s = 0{,}46$$

$$N = \frac{MTU}{Tamaño\ paq\ a\ multiplexar} = \frac{1500}{200} = 7{,}5$$

Tendríamos *N* = 7,5 en el peor de los casos, ya que los paquetes pequeños podrían ser menores a 200 bytes.

Con estos datos tendríamos una tasa de salida λ_{out}:

$$\lambda_{out} \approx \lambda_{mux} + \lambda_{no\,mux} = \frac{\lambda \cdot 0{,}46}{7{,}5} + \lambda \cdot (1 - 0{,}46) = \lambda \cdot 0{,}601$$

Vemos por tanto que con una traza general, con una distribución de tráfico realista, conseguimos un ahorro de paquetes por segundo en torno a un 40%.

Una vez que tenemos el ahorro en la tasa de paquetes por segundo, vamos a pasar a analizar el ancho de banda que tendremos a la salida del sistema:

$$BW_{out} = \lambda_{mux} \cdot E[tamaño\,paq.multiplexados] + \lambda \cdot (1 - s) \cdot E[tamaño\,paq.\ grandes]$$

El ahorro en ancho de banda se producirá gracias al uso de ROHC, y dependerá del tipo de paquete a multiplexar, de su tamaño, y de la distribución de este tráfico.

Para este tipo de tráfico general, en el que hay tanto paquetes pequeños como grandes, el tamaño de los paquetes grandes estará casi un orden de magnitud por encima del de los pequeños. Por tanto, el ahorro en ancho de banda sólo será significativo cuando $\lambda_{mux} >> \lambda \cdot (1 - s)$. Si no se da esta situación, tendremos $BW_{in} \approx BW_{out}$.

Pongamos un ejemplo para justificar estas conclusiones. Supongamos que tenemos un tráfico de 10.000 paquetes/seg. Seguiremos considerando 200 bytes como el umbral para considerar *pequeño* un paquete.

Supongamos que conseguimos comprimir un 25% los paquetes pequeños (no conocemos el tipo de tráfico, por lo que *a priori* no se puede conocer el nivel de compresión que se puede alcanzar). El ancho de banda nativo sería:

$$BW_{nativo} : n^{o}paq/seg \cdot (tamaño\ paq\ pequeño \cdot s + tamaño\ paq\ grande \cdot (1 - s))$$

Con estos valores, tenemos 56,75 Mbps de tráfico nativo. Si queremos obtener el ancho de banda (tasa) una vez aplicada la optimización, tendríamos:

$$BW_{optimizado} : n^{o}paq/seg \cdot (tamaño\ paq\ optimizado \cdot s + tamaño\ paq\ grande \cdot (1 - s))$$

Sustituyendo los valores, obtenemos 56,04 Mbps. El ahorro en términos de ancho de banda es pequeño, como se puede observar. Esto se debe a que la reducción del 25 % de la cabecera sólo se aplica al 41 % de los paquetes. Aunque son muchos paquetes, su tamaño está muy por debajo del de los paquetes grandes (1500 bytes), por lo que apenas se aprecia reducción de ancho de banda (resulta un 1,25%).

Una vez realizado el análisis teórico, pasamos a realizar varias pruebas con el tráfico obtenido de la traza general. En las pruebas se utilizaron los 200.000 primeros paquetes de dicha traza. En la Figura 5.16 se muestra la distribución de paquetes a lo largo del tiempo en función de su tamaño. En la Figura 5.17 se observa la nueva distribución de tráfico si se optimizan los paquetes pequeños (los paquetes optimizados se muestran con una “x”). Con la optimización de tráfico, los paquetes menores a 200 bytes han desaparecido, y se han convertido en paquetes de tamaño variable, situados en la zona central de la gráfica, en torno a 800 bytes. Gracias a esta optimización se consigue disminuir en gran medida el *overhead* de los paquetes pequeños y la tasa de envío de paquetes por segundo en el enlace.

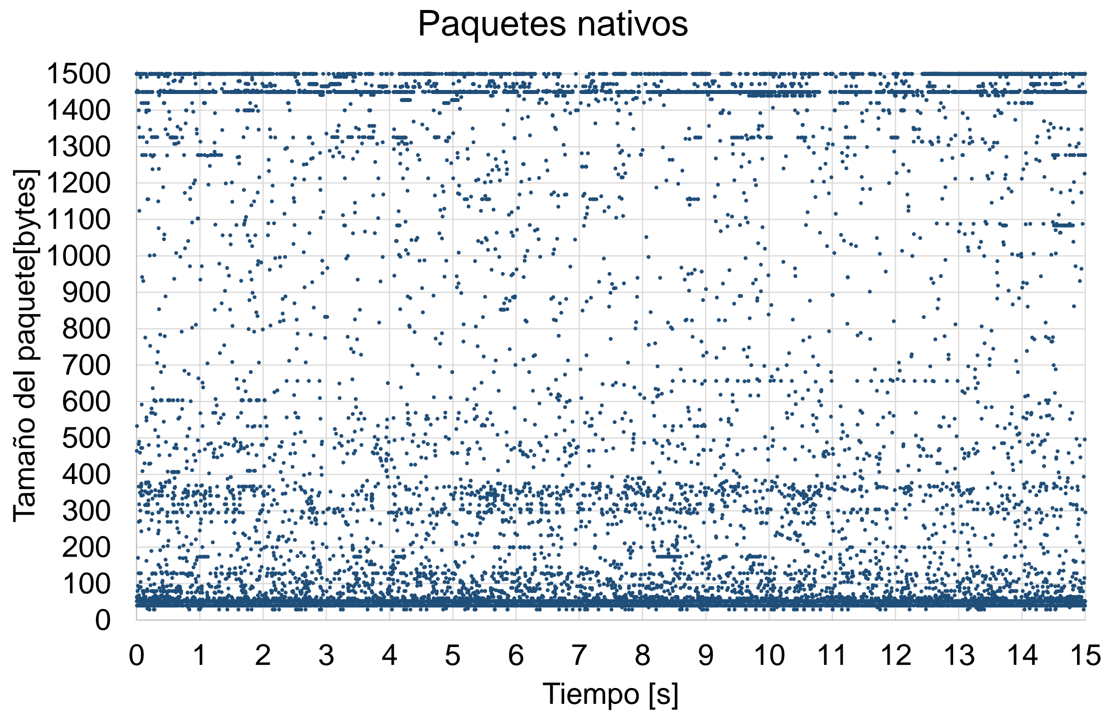

Figura 5.16. Distribución de paquetes según su tamaño.

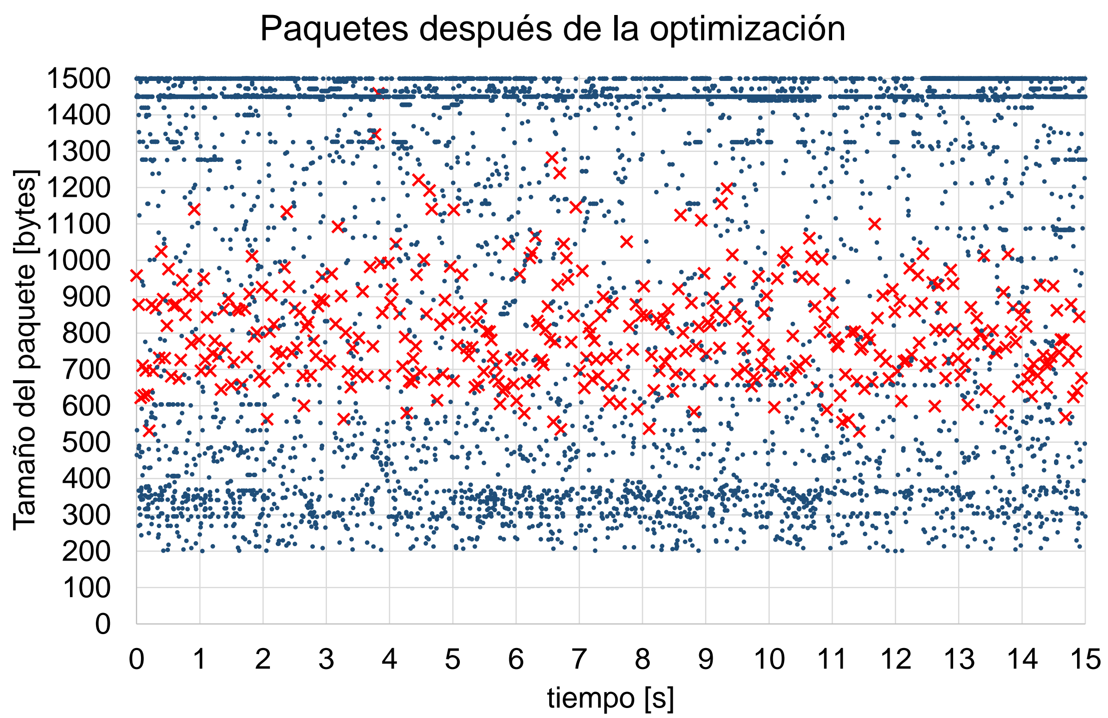

Figura 5.17. Distribución de paquetes después de la optimización (los paquetes optimizados se muestran con una “x”).

Llevando a la práctica este caso de estudio, optimizando la traza de la Figura 5.14 se consigue un ahorro en la tasa de paquetes del 36%.

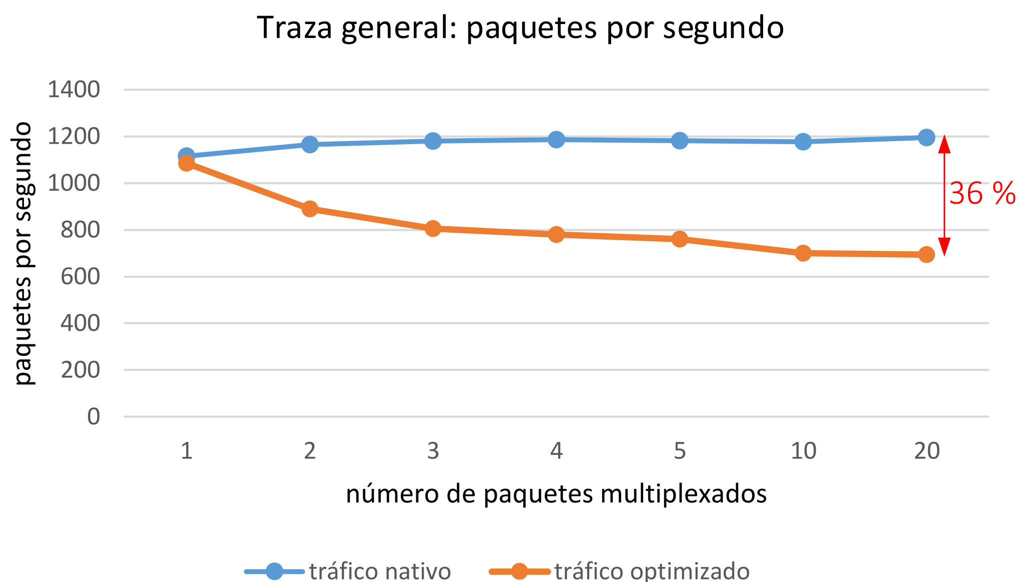

Figura 5.18. Tráfico traza general.

5.4 Coste de procesado

Se quería probar que la implementación de Simplemux es ligera y puede correr en equipos con poca capacidad de procesado. Por eso, aparte de implementarse en los equipos Linux instalados en el PC2 y PC3, se pensó su instalación en un *router* TP-Link WR1043ND. Dado que este *router* no tiene capacidad para compilar (OpenWRT no incluye por defecto el compilador), se ha recurrido a la compilación cruzada (Anexo D), para obtener un ejecutable que se instaló en el *router*. Se observaron los siguientes resultados:

	MiniPC2	Router TP-Link
Coste de procesado	0,25 ms	3,5 ms

Tabla 7. Coste de procesado.

Capítulo 6

Conclusiones y líneas futuras

6.1 Conclusiones

Después de exponer el trabajo realizado, estamos ahora en disposición de presentar un resumen de las conclusiones y de los resultados obtenidos.

En este trabajo, se ha realizado un análisis de las técnicas de optimización de tráfico a través de la combinación de protocolos a tres niveles: compresión (a través de ROHC), tunelado y multiplexado. La combinación de estas técnicas se conoce como TCM. Simplemux es una nueva propuesta presentada al IETF para cubrir la capa de multiplexión.

En esta memoria se han presentado pruebas centradas en Simplemux, combinado con compresión de cabeceras y túneles IP y UDP, para mejorar la eficiencia de flujos de altas tasas de paquetes pequeños. Este tráfico tiene un *overhead* alto y puede provocar situaciones de congestión en la red, que con el uso de este protocolo podrán ser evitadas o reducidas.

Se han diseñado varios escenarios de red, y se han puesto en marcha en el laboratorio, permitiendo la realización de múltiples pruebas con diferentes parámetros de red. La realización de estas pruebas ha permitido conocer y mejorar el protocolo Simplemux, ver en qué situaciones es recomendable su uso y dejar constancia de su eficiencia en casos de congestión.

Se espera que estas pruebas contribuyan al avance en la estandarización de Simplemux en el IETF, para que pueda ser adoptado como un nuevo protocolo de comunicaciones a nivel global.

Se ha observado que para VoIP este protocolo supone una mejora muy importante en la eficiencia. Se consigue reducir el *overhead* y pasamos a ahorrar hasta un 47% de ancho de banda, y entre un 80 y 97 % en la tasa de paquetes por segundo, dependiendo de la codificación.

Para juegos *online*, se ha reducido el *overhead* y gracias a la compresión de ROHC se ha conseguido un ahorro de un 23,8 % para el ancho de banda, y gracias a la multiplexión de Simplemux se ha logrado un ahorro del 95% de los paquetes.

Cuando se multiplexan paquetes ACK de TCP logramos liberar a los nodos de procesar gran cantidad de paquetes. De hecho, se consigue ahorrar un 98,5 % el número de paquetes por segundo gracias a que con Simplemux, se consiguen incluir hasta 65 paquetes ACK en un paquete IP. En términos de ahorro de ancho de banda también conseguimos una mejora del 55 % gracias al protocolo ROHC.

Finalmente, se ha multiplexado el tráfico de una traza general en el que sólo se optimizaba una parte de todo el tráfico que circulaba por el enlace, y se obtuvo una mejora en la tasa de paquetes por segundo del 36 %.

Por otra parte, la realización de esta batería de pruebas ha permitido probar extensivamente la implementación de Simplemux, permitiendo identificar algunos fallos de

programación y proponiendo algunas mejoras, por ejemplo, la identificación del MTU de la red y el modo de presentar los ficheros de *log.*

6.2 Líneas futuras

Una línea futura de trabajo reside en la mejora de la clasificación de tráfico para decidir en tiempo real qué paquetes optimizar y cuáles enviar de forma nativa. Hasta ahora, esta decisión se ha realizado a través de la configuración de *iptables*, pero en un futuro, se espera que se pueda utilizar un algoritmo de detección de tráfico, desarrollado por la universidad de Melbourne [22]. Dicha implementación está desarrollada bajo el nombre de *Diffuse*, y ha sido probada en diferentes puntos de acceso Wi-Fi.

Diffuse es un algoritmo basado en *Machine Learning*[14] que permite conocer o caracterizar un tráfico sólo observando el tamaño y el tiempo entre paquetes. De esta manera, no necesita leer el contenido del paquete, evitando así problemas de privacidad y ahorrando tiempo de procesado. El objetivo futuro sería integrar *Diffuse* con Simplemux, combinando ambas herramientas y observando las mejoras que se obtienen.

En este trabajo se han utilizado varios flujos TCP simultáneos con periodos de multiplexión muy pequeños para intentar interferir lo menos posible en el crecimiento de la ventana TCP. Pero de los resultados obtenidos con las pruebas TCP, se observó que Simplemux conseguía mejorar las comunicaciones que presentaban *jitter*, pero no funcionaba tan bien en situaciones sin *jitter*. Por esta razón, una línea futura de trabajo sería estudiar el comportamiento de la ventana de congestión de TCP y ver cómo se modifica en función del periodo de multiplexión, obteniendo conclusiones sobre cuál es la configuración más eficaz de Simplemux para optimizar tráfico TCP.

---

[14] *Machine learning*: el aprendizaje automático es una rama de la inteligencia artificial cuyo objetivo es desarrollar técnicas que permitan a las computadoras aprender.

6.3 Planificación del trabajo

La Figura 6.1 muestra el diagrama de Gantt, con las fases en las que se ha dividido el proyecto.

Id.	Nombre de tarea	Comienzo	Fin	Duración	2014			2015								
					oct	nov	dic	ene	feb	mar	abr	may	jun	jul	ago	sep
1	Estudio previo y documentación	01/10/2014	20/01/2015	80d												
2	Elección de equipos	02/02/2015	09/02/2015	6d												
3	Diseño y configuración de los escenarios de red	12/02/2015	16/03/2015	23d												
4	Simulación de tráfico para las pruebas	13/03/2015	03/04/2015	16d												
5	Pruebas en entornos ethernet	20/03/2015	28/04/2015	28d												
6	Pruebas en entornos Wi-Fi	01/05/2015	19/06/2015	36d												
7	Análisis de resultados	30/03/2015	30/06/2015	67d												
8	Desarrollo y mejora de Simplemux	06/04/2015	30/06/2015	62d												
9	Redacción de la memoria del proyecto	28/05/2015	31/08/2015	68d												

Figura 6.1. Diagrama de Gantt.

Bibliografía

[1] Thompson, B., Koren, T., Wing, D., RFC4170, Tunneling Multiplexed Compressed RTP (TCRTP), Nov. 2005.

[2] Sandlund, K., Pelletier, G., and L-E. Jonsson, RFC 5795, The RObust Header Compression (ROHC) Framework, 2010

[3] Schulzrinne, H., Casner, S., Frederick, R., & Jacobson, V. (2003). RTP: A Transport Protocol for Real-Time Applications IETF RFC 3550. United States.

[4] Huitema, C. (2003). Real time control protocol (RTCP) attribute in session description protocol (SDP).

[5] Koren, T., Casner, S., Geevarghese, J., Thompson, B., & Ruddy, P. (2003). Enhanced Compressed RTP (CRTP) for Links with High Delay. Packet Loss and Reordering, RFC3545.

[6] Ali, I., Pazhyannur, R., & Fox, C. (2001). PPP Multiplexing. RFC 3153.

[7] Simpson, W. (1992). The point-to-point protocol (PPP) for the transmission of multi-protocol datagrams over point-to-point links.

[8] Lau, J., & Goyret, I. (2005). Layer two tunneling protocol-version 3 (L2TPv3).

[9] Ho, T. S., & Chen, K. C. (1996, October). Performance analysis of IEEE 802.11 CSMA/CA medium access control protocol. In proc. PIMRC (Vol. 96, pp. 407-411).

[10] Black, U. (1999). Voice over IP. Prentice-Hall, Inc.

[11] Andersen, S., Duric, A., Astrom, H., Hagen, R., Kleijn, W., & Linden, J. (2004). RFC3951: Internet low bit rate codec (iLBC). Network Working Group.

[12] Sollaud, A. (2008). RTP Payload Format for ITU-T Recommendation G. 711.1.

[13] Botta, A., Dainotti, A., Pescapè, A., "A tool for the generation of realistic network workload for emerging networking scenarios", Computer Networks (Elsevier), 2012, Volume 56, Issue 15, pp 3531-3547.

[14] Lang, T., Branch, P., & Armitage, G. (September 2004). A synthetic traffic model for *Quake III*. In Proceedings of the 2004 ACM SIGCHI International Conference on Advances in computer entertainment technology (pp. 233-238). ACM.

[15] The CAIDA UCSD equinix-chicago- 20150219-130000, https://data. caida.org/datasets/passive-2015/equinix-chicago/20150219-130000. UTC/equinix-chicago.dirA.20150219-125911.UTC.anon.pcap.gz. [Accedido septiembre 2015].

[16] Saldana, J. "Simplemux. A generic multiplexing protocol," draft-saldana-tsvwg-simplemux-03, Jul. 2015, available at http://datatracker. ietf. org/ doc/ draft-saldana-tsvwg-simplemux/

[17] Internet Assigned Numbers Authority (IANA) Assigned Internet Protocol Numbers, available at http://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml. [Accedido septiembre 2015].

[18] Cerda-Alabern, L., “On the topology characterization of Guifi.net,” Proceedings Wireless and Mobile Computing, Networking and Communications, IEEE 8th International Conf, pp. 389-396, 2012.

[19] Rey-Moreno, C., Bebea-Gonzalez, I., Foche-Perez, I., Quispe-Taca, R., Linan-Benitez, L. and Simo-Reigadas, J. “A telemedicine WiFi network optimized for long distances in the Amazonian jungle of Peru,” Proceedings of the 3rd Extreme Conference on Communication: The Amazon Expedition, ExtremeCom '11 ACM, 2011.

[20] The Guide to IP Layer Network Administration with Linux, disponible en http://linux-ip.net/. [Accedido septiembre 2015].

[21] Claypool, M. and Claypool, K. "Latency and player actions in online games", Communications of the ACM 49, 2006.

[22] Nguyen, T. T. T.; Armitage, G.; Branch, P.; Zander, S., "Timely and Continuous Machine-Learning-Based Classification for Interactive IP Traffic," in Networking, IEEE/ACM Transactions on , vol.20, no.6, pp.1880-1894, Dec. 2012

Anexos

A. Acrónimos

- 2G: Segunda Generación de telefonía móvil.
- ACK: Acknowledgement.
- AP: Access Point.
- CeNITEQ: Communication Networks and Information Technologies for e-Health and Quality of Experience Group.
- DCF: Distributed Coordination Function.
- DIFS: Interframe Space duration.
- ECRTP: Enhanced Compressed RTP.
- FPS: First Person Shooter.
- GSM: Global System for Mobile communications.
- I3A: Instituto universitario de Investigación en Ingeniería de Aragón.
- IANA: Internet Assigned Numbers Authority.
- IETF: Internet Engineering Task Force.
- iLBC: Internet Low Bitrate Codec.
- IoT: Internet of Things.
- IPv4: Internet Protocol version 4.
- IPv6: Internet Protocol version 6.
- L2TP: Layer 2 Tunneling Protocol.
- LEN: Length.
- LXT: Length Extension.
- MAC: Media Access Control.
- MTU: Maximun Transfer Unit.
- PCF: Point Coordination Function.
- PCM: Pulse Code Modulation.
- PPP: Point-to-point Protocol.
- PPPMux: Point-to-Point Protocol Multiplexing.
- PSTN: Public Switched Telephone Network.
- RFC: Request for Comments.
- ROHC: RObust Header Compression.
- RPE-LTP: Regular Pulse Excitation Long-Term Prediction.
- RTCP: Real-time Transport Control Protocol.
- RTP: Real Time Protocol.
- RTS/CTS: Ready To Send/Clear to send.
- SIFS: Short Interframe Space.
- SPB: Single Protocol Bit.
- TCP: Transmission Control Protocol.
- TCRTP: Tunneling Multiplexed Compressed Real Time Protocol.
- UDP: User Datagram Protocol.

- USB: Universal Serial Bus.
- VoIP: Voice over IP.

B. Compilación de Simplemux en un equipo Linux.

En este anexo se va a explicar la compilación de Simplemux para sistemas operativos Linux.

La implementación de Simplemux, se encuentra desarrollada en lenguaje C, y está disponible en GitHub[15]. Ahí encontraremos los siguientes ficheros:

- *simplemux.c*: en este fichero se encuentra el programa principal de Simplemux.

- *simplemux_multiplexing_delay.pl*: fichero en Perl utilizado para el análisis estadístico del tiempo entre paquetes en el interfaz optimizado.

- *simplemux_throughput_pps.pl*: fichero en Perl utilizado para el análisis estadístico del ancho de banda en el interfaz optimizado.

- *driveGnuPlotStreams.pl*: fichero en Perl que se usa para observar el ancho de banda instantáneo en un interfaz optimizado por Simplemux.

Para la compresión de cabeceras, y así, conseguir ahorros en el ancho de banda, Simplemux utiliza la librería de software libre ROHC[16].

Instalación de ROHC para sistema operativo Linux:

- Lo primero que debemos hacer es descargar la versión 1.7.0 de ROHC desde el enlace https://rohc-lib.org/support/download/.
- Una vez descargada, desde el terminal vamos al directorio donde lo hemos guardado y ejecutamos los siguientes comandos:

```
# ./configure –prefix=/usr
# make all
# make check
# make install
```

Una vez hecho esto, ya podemos compilar Simplemux.c para obtener un programa ejecutable. Para ello, ejecutamos la siguiente instrucción:

```
# gcc -o simplemux -g -Wall $(pkg-config rohc --cflags)
simplemux_vX.Y.c $(pkg-config rohc --libs )
```

[15] Simplemux en Github: https://github.com/TCM-TF/simplemux
[16] ROHC: https://rohc-lib.org/

Con esta instrucción ya tendremos un archivo ejecutable en el directorio. El uso del programa se encuentra en el Anexo D.

C. Compilación cruzada de Simplemux.

La compilación cruzada se utiliza para crear un programa ejecutable en una plataforma distinta de aquella en la que se está ejecutando el compilador.

Este anexo puede ser de utilidad para ejecutar Simplemux en un punto de acceso Wi-Fi. La compilación se realizará desde un equipo Linux. El sistema operativo que utilizaremos en el punto de acceso Wi-Fi será OpenWRT[17].

A continuación se detallan los pasos para realizar la compilación cruzada en un punto de acceso TP-Link-WR1043ND.

Lo primero que debemos hacer es instalar la versión de OpenWRT correspondiente a nuestro modelo. Podremos descargarla desde el siguiente enlace:

http://downloads.openwrt.org/barrier_breaker/14.07/ar71xx/generic/

En este enlace, buscamos el modelo correspondiente y formateamos el equipo con la versión de OpenWRT descargada.

Una vez hecho esto, debemos bajar el fichero *toolchain* para compilación cruzada para MIPS:

http://downloads.openwrt.org/latest/ar71xx/generic/OpenWrt-Toolchain-ar71xx-for-mips_34kc-gcc-4.8-linaro_uClibc-0.9.33.2.tar.bz2

Descomprimimos el paquete en el directorio `/home/proyecto` (opcional). Ahora podremos ir al directorio:

```
/home/proyecto/OpenWrt-Toolchain-ar71xx-for-mips_34kc-gcc-4.8-
linaro_uClibc-0.9.33.2/toolchain-mips_34kc_gcc-4.8-linaro_uClibc-
0.9.33.2/bin/
```

Ahora, debemos incluir las librerías ROHC en el directorio de *toolchain*:

- Lo primero que debemos hacer es descargarnos la versión 1.7.0 de ROHC desde el enlace https://rohc-lib.org/support/download/.
- Descomprimimos el fichero descargado en el directorio de toolchain creado antes.

```
/home/proyecto/OpenWrt-Toolchain-ar71xx-for-mips_34kc-gcc-4.8-
linaro_uClibc-0.9.33.2/toolchain-mips_34kc_gcc-4.8-linaro_uClibc-
0.9.33.2/
```

Una vez en este directorio, modificamos la variable CC del sistema para que cualquier *"configure"* que se ejecute, utilice el *gcc* de las *toolchain*, y no el que viene por defecto.

```
#export CC=/home/proyecto/OpenWrt-Toolchain-ar71xx-for-mips_34kc-gcc-
4.8-linaro_uClibc-0.9.33.2/toolchain-mips_34kc_gcc-4.8-linaro_uClibc-
0.9.33.2/bin/mips-openwrt-linux-gcc
```

---

17 OpenWRT: distribución de Linux basada en firmware, utilizada para *router* o puntos de acceso Wi-Fi comerciales.

Con el comando `#set` se puede comprobar si la variable `CC` ha cambiado correctamente. Para volver a tener el *gcc* por defecto podemos hacer:

```
#export CC=Null
```

A continuación instalamos las librerías ROHC en el *toolchain*, para ello, vamos al directorio donde habíamos descargado ROHC:

```
/home/proyecto/OpenWrt-Toolchain-ar71xx-for-mips_34kc-gcc-4.8-
linaro_uClibc-0.9.33.2/toolchain-mips_34kc_gcc-4.8-linaro_uClibc-
0.9.33.2/rohc-1.7.0/
```

Desde ahí, ejecutamos las siguientes instrucciones:

```
#./configure --disable-app-fuzzer --disable-app-performance --disable-
app-sniffer --enable-app-tunnel --disable-app-stats --disable-linux-
kernel-module --disable-doc --disable-doc-man --host=mips --
prefix=/home/proyecto/OpenWrt-Toolchain-ar71xx-for-mips_34kc-gcc-4.8-
linaro_uClibc-0.9.33.2/toolchain-mips_34kc_gcc-4.8-linaro_uClibc-
0.9.33.2/

#make all

#make check

#make install
```

Con estos pasos habremos creado en la carpeta `lib` del *toolchain* los ficheros `.a` necesarios para la compilación con librerías estáticas (`librohc-common.a librohc-comp.a` y `librohc-decomp.a`) [18]

Finalmente, sólo nos queda compilar el archivo ejecutable de Simplemux:

```
#/home/proyecto/OpenWrt-Toolchain-ar71xx-for-mips_34kc-gcc-4.8-
linaro_uClibc-0.9.33.2/toolchain-mips_34kc_gcc-4.8-linaro_uClibc-
0.9.33.2/bin/mips-openwrt-linux-gcc -o /home/proyecto/simplemux-mips -
g -Wall /home/proyecto/simplemux/simplemux.c -I ./include/ -L ./lib/ -
lrohc_comp -lrohc -lrohc_common -lrohc_decomp –static
```

Este ejecutable tendrá todas las librerías que necesita incluidas dentro. Por esta razón, es posible que el tamaño del fichero sea superior a la memoria de la que disponemos en el punto de acceso. En caso de que no se pueda copiar el programa a la memoria interna, podrá hacerse a una memoria USB conectada al punto de acceso.

Una vez creado el ejecutable, copiamos el archivo al punto de acceso:

```
#ssh root@<IP del AP>
```

[18] Las librerías dinámicas son `.so`

Montamos el dispositivo:

```
AP# mount /deb/sdX /mnt/usb
```

Por último, copiamos el fichero desde nuestro ordenador:

```
# scp ./simplemux-mips root@a<IP-AP>:/mnt/usb
```

D. Manual de Simplemux

En este anexo se explica el uso de Simplemux. Para ello, deberemos tener previamente el programa ejecutable, que habremos obtenido mediante la compilación del programa (ver Anexo B).

Una vez compilado y generado el ejecutable Simplemux, desde el terminal y en el directorio donde esté el programa, usaremos la siguiente instrucción para ejecutar Simplemux en el interfaz deseado:

```
# ./simplemux -i <ifacename> -e <ifacename> -c <peerIP> -M <N or T> [-
p <port>] [-d <debug_level>] [-r <ROHC_option>] [-n <num_mux_tun>] [-m
<MTU>] [-b <num_bytes_threshold>] [-t <timeout (microsec)>] [-P
<period (microsec)>] [-l <log file name>] [-L]
```

Ahora explicaremos cada parámetro de entrada que se puede incluir en la llamada al programa.

`-i <ifacename>`: nombre del interfaz de entrada al enlace optimizado. A este interfaz deberá llegar el tráfico que queremos optimizar. Por aquí se mandarán los paquetes multiplexados.

`-e <ifacename>`: nombre del interfaz del otro extremo del enlace optimizado. En este interfaz se recibirán los paquetes multiplexados.

`-c <peerIP>`: parámetro para especificar la dirección IP del receptor del enlace optimizado. El interfaz e, deberá pertenecer a esta dirección IP.

`-M <mode>`: este parámetro admite dos opciones:

- `Transport (T)`: en este modo el paquete irá encapsulado en un paquete UDP.
- `Network (N)`: aquí el paquete Simplemux irá directamente sobre IP.

`-p <port>`: puerto de escucha. Por defecto, estará escuchando en el puerto 55555.

`-d`: parámetro utilizado para sacar información por pantalla. Actúa como *debug,* dándo información sobre lo que está sucediendo. Tiene 4 opciones:

`0`: sin debug; 1, 2 y 3. Estos 3 parámetros son graduales: de menos a más información por pantalla.

`-r`: parámetro que hace referencia al uso de la librería ROHC: Tiene 3 opciones:

`0`: Simplemux no usará ROHC. Por tanto, no comprimirá cabeceras.

`1`: Simplemux usará ROHC unidireccional. En este caso el descompresor no enviará información al compresor sobre posibles pérdidas, recuperación del contexto, etc.

`2`: Simplemux utilizará ROHC bidireccional. Esta función permite que en el enlace haya un canal de control entre el compresor y el descompresor para el envío de información de estado desde el descompresor al compresor.

`-n`: número de paquetes que se almacenarán en el buffer y que se multiplexarán en una trama Simplemux. Por defecto tiene valor 1, como máximo se podrán multiplexar 100 paquetes.

`-m`: este parámetro hace referencia al MTU de la red. Por defecto escoge el que haya en el interfaz de red en el que este activado Simplemux.

`-b`: Tamaño límite paquete multiplexado, por defecto coge el máximo posible, es decir, MTU-20 cuando trabaja en modo red y MTU-28 cuando trabaja en modo transporte.

`-t`: *Timeout* en microsegundos. Tiempo de espera en el buffer de un paquete. Cuando el *Timeout* expire se vaciará el buffer enviándose los paquetes que había en una trama Simplemux.

`-P`: Periodo en microsegundos. Cuando se cumpla el periodo se enviarán los paquetes almacenados en el *buffer*.

`-l <logfile>`: parámetro para indicar un fichero en el que se escribirá información sobre los eventos que se han producido durante la ejecución del programa.

`-L`: variante de `-l` utilizada para que el fichero de log tenga en el nombre detalles del momento de ejecución. El formato del nombre de log es: *Año-mes-día_Horas_minutos_segundos*.

`-h`: muestra una ayuda por pantalla con la explicación de los posibles comandos a introducir.

El fichero de log que genera Simplemux, sigue la clasificación de paquetes que aparece en la Tabla 6.

timestamp	event	type	size	sequence number	from/to	IP	port	number of packets	triggering event(s)
%"PRIu64"	text	text	%i	%lu	text	%s	%d	%i	text
microsec.	rec	native	packet size in bytes	sequence number	-	-		-	
		muxed			from	ingress IP address	port		
		ROHC feedback							
	sent	muxed			to	egress IP address	port	n umber	numpacket_limit size_limit timeout period MTU
		demuxed			-	-		-	
	forward	native			from	i ngress IP address	ort	-	
	error	bad_separator			-	-		-	
		demux_bad_length			-	-		-	
		decomp_failed			-	-		-	
		comp_failed			-	-		-	

Tabla 8. Clasificación de los paquetes en el fichero de log.

- **timestamp:** medido en microsegundos, obtenido mediante la función GetTimeStamp().

- **event y type:** evento producido y tipo de paquete:
  - **rec:** paquete recibido.
    - **native:** cuando llega al nodo de entrada del enlace optimizado un paquete nativo (sin modificación alguna).

    - **muxed:** cuando llega un paquete al nodo de salida.

    - **ROHC_feedback:** cuando el nodo de entrada recibe un paquete que contiene información de control de ROHC, enviado por el nodo de salida.

  - **sent:** paquete enviado. Puede ser de dos tipos:
    - **muxed:** el nodo de entrada envía un paquete multiplexado.

    - **demuxed:** el nodo de salida envía al receptor el paquete desmultiplexado.

  - **forward:** cuando un paquete llega al nodo de entrada con un puerto diferente en el que está activada la optimización. Simplemente el paquete es enviado por la red en su forma nativa.

  - **error:** pueden aparecer varios tipos de error:
    - **bad_separator:** la cabecera Simplemux no está bien estructurada.

    - **demux_bad_length:** la longitud de la cabecera Simplemux es excesiva.

    - **decomp_failed:** la descompresión de ROHC ha fallado.

    - **comp_failed:** la compresión de ROHC ha fallado.

  - **drop:** paquete descartado.
    - **no_ROHC_mode:** se ha recibido un paquete con compresión ROHC y no está activada la descompresión en el nodo de salida.

- **size:** tamaño del paquete en bytes incluyendo cabecera IP.

- **sequence number:** número de secuencia generado por Simplemux. Hay dos diferentes: uno para los paquetes recibidos y otro para los enviados.

- **IP:** dirección IP del otro extremo de optimización del enlace.

- **port:** puerto de destino del paquete.

- **Number of packets:** número de paquetes incluidos en un paquete multiplexado.

- **Triggering event(s):** evento producido para el envío del paquete multiplexado. Pueden aparecer los siguientes eventos:
    - **Numpacket_limit:** cuando se alcanza el número de paquetes a multiplexar.
    - **size_limit:** se ha alcanzado el tamaño máximo del paquete multiplexado.
    - **timeout:** paquete enviado al expirar el timeout.
    - **period:** paquete enviado al expirar el periodo.
    - **MTU:** el MTU ha sido alcanzado.

Un posible ejemplo de lo que veríamos en un fichero de log generado por Simplemux sería el siguiente:

- En el nodo de entrada.

```
1417693720928101 rec native 63 1505
1417693720931540 rec native 65 1506
1417693720931643 rec native 52 1507
1417693720936101 rec native 48 1508
1417693720936210 rec native 53 1509
1417693720936286 rec native 67 1510
1417693720937162 rec native 57 1511
1417693720938081 sent muxed 237 1511 to 192.168.137.4 55555 7 period
```

- En el nodo de salida:

```
1417693720922848 rec muxed 237 210 from 192.168.0.5 55555
1417693720922983 sent demuxed 63 210
1417693720923108 sent demuxed 65 210
1417693720923186 sent demuxed 52 210
1417693720923254 sent demuxed 48 210
1417693720923330 sent demuxed 53 210
1417693720923425 sent demuxed 67 210
1417693720923545 sent demuxed 57 210
```

Los programas que siguen a continuación son utilizados para obtener análisis estadísticos del tráfico que ha circulado por el enlace.

Uso de *simplemux_throughput_pps.pl*.

Programa escrito en Perl que obtiene el *throughput*[19] en bits por segundo y la tasa de paquetes por segundo para cada intervalo de tiempo configurado como parámetro de entrada.

Uso:

```
# perl simplemux_throughput_pps.pl <trace file> <tick(us)> <event>
<type> <peer IP> <port>
```

Parámetros:

`<trace file>`: fichero de log donde se ha guardado la información a procesar.

`<tick(us)>`: intervalo de tiempo en microsegundos en el que se obtendrá el throughput y los paquetes transmitidos en ese intervalo.

`<event>`: Evento del paquete. El evento de los paquetes se puede consultar en la tabla 6.

`<type>`: tipo de paquete. El tipo de los paquetes se puede consultar en la tabla 6.

`<peerIP>`: IP origen de los paquetes.

`<port>`: Puerto de destino de los paquetes.

Ejemplo de uso:

```
# perl simplemux_throughput_pps.pl tracefile.txt 1000000 rec native
all all
```

En este ejemplo estaríamos obteniendo el *troughput* cada segundo de los paquetes recibidos nativos, de cualquier IP y con cualquier puerto. Un posible resultado de lo que imprimiría por pantalla sería:

```
tick_end_time(us) throughput(bps) packets_per_second
1000000     488144      763
2000000     490504      759
3000000     475576      749
4000000     483672      760
5000000     481784      758
6000000     487112      762
7000000     486824      760
8000000     488792      765
9000000     483528      761
10000000    486360      760
```

Uso de *simplemux_multiplexing_delay.pl.*

Programa escrito en Perl que obtiene el tiempo entre paquetes, y un pequeño resumen con la media del retardo y la desviación típica.

[19] Throughput: término que hace referencia a la cantidad de información que fluye a través de un enlace.

Ejemplo de uso:

```
# perl simplemux_multiplexing_delay.pl <trace file> <output file>
```

Parámetros:

<trace file>: fichero de log de entrada.

<output file>: fichero en el que almacenaría los resultados.

Un posible resultado de lo que imprimiría por pantalla sería:

```
packet_id multiplexing_delay(us)
1     5279
2     1693
3     1202
4     507
5     10036
6     8471
7     6974
8     5588
9     1143
10    10435
11    8935
12    7522
13    5981
14    4520
15    3011
...
total native packets: 6661
Average multiplexing delay: 5222.47680528449 us
stdev of the multiplexing delay: 3425.575192789 us
```

E. Manual de D-ITG.

Instalación:

D-ITG, *Distributed Internet Traffic Generator*, es un generador de tráfico de software libre. Se puede descargar desde la página:

http://traffic.comics.unina.it/software/ITG/download.php

- Una vez descargada la última versión, la descomprimimos en el directorio deseado.

- Desde el terminal, dentro del directorio D-ITG-2.8.1-r1023/src, ejecutamos:

  ```
  # make
  ```

- Con este comando se habrán copiado a la carpeta /bin los ejecutables necesarios para poner en funcionamiento el programa. Deberemos hacer lo mismo tanto en el equipo emisor como en el receptor.

Manual de uso:

Inicio rápido:

Para poner en funcionamiento el sistema, debemos activar primero la escucha de tráfico en el equipo receptor, y posteriormente enviar tráfico desde el emisor. Para ello, desde el directorio D-ITG-2.8.1-r1023/bin ejecutamos el siguiente programa:

Equipo receptor:

```
# ./ITGRecv
```

Equipo transmisor:

```
# ./ITGSend -T UDP -a 192.168.1.2 -c 100 -C 10 -t 15000 -l sender.log -x receiver.log
```

En este ejemplo, ITGSend habrá generado un tráfico UDP con paquetes de 100 bytes transmitiéndolos a una tasa de 10 paquetes por segundo, durante 15 segundos. Además, habrá guardado las estadísticas en el fichero receiver.log generado en el mismo directorio donde se ejecutó el programa: D-ITG-2.8.1-r1023/bin.

El fichero *receive.log* mostrará lo siguiente:

```
----------------------------------------------------------
Flow number: 1
From 127.0.0.1:44225
To 127.0.0.1:8999
----------------------------------------------------------
Total time = 14.944263 s
Total packets = 150
```

```
Minimum delay = 0.000000 s
Maximum delay = 0.000000 s
Average delay = 0.000000 s
Average jitter = 0.000000 s
Delay standard deviation = 0.000000 s
Bytes received = 15000
Average bitrate = 8.029837 Kbit/s
Average packet rate = 10.037297 pkt/s
Packets dropped = 0 (0.00 %)
Average loss-burst size = 0.000000 pkt
----------------------------------------------------------
__________________________________________________________
**************** TOTAL RESULTS ******************
__________________________________________________________
Number of flows = 1
Total time = 14.944263 s
Total packets = 150
Minimum delay = 0.000000 s
Maximum delay = 0.000000 s
Average delay = 0.000000 s
Average jitter = 0.000000 s
Delay standard deviation = 0.000000 s
Bytes received = 15000
Average bitrate = 8.029837 Kbit/s
Average packet rate = 10.037297 pkt/s
Packets dropped = 0 (0.00 %)
Average loss-burst size = 0 pkt
Error lines = 0
```

Ahora pasaremos a explicar cada comando con sus parámetros:

ITGSend:

ITGSend permite enviar un único flujo de datos o varios. Si queremos mandar varios flujos simultáneamente deberemos crear un script con varias llamadas al programa. Se puede ver en el apartado de ejemplo.

Uso:

```
# ./ITGSend [options]
```

Opciones:

- l [logfile]: genera fichero de log.

- t <duration>: duración de la transmisión, en milisegundos.

- z <#_of_packets>: establece el número de paquetes a generar.

- k <#_of_KBytes>: establece el número de kbytes a generar.

- d <delay>: empieza la transmisión de paquetes después de un tiempo d en milisegundos.

-a <dest_address>: establece la dirección de destino.

-sa <src_address>: establece la dirección de origen.

-rp <dest_port>: establece el puerto de destino.

-rp <src_port>: establece el puerto de origen.

-p <payload_metadata>: especifica los datos que se van a incluir en el payload de los paquetes. Por defecto tiene el valor 0.

- 0: contiene información estándar utilizada para el análisis estadístico de la transmisión.
- 1: sólo envía el número de secuencia en el payload.
- 2: no envía información en el payload.

-T <protocol>: por defecto UDP. Admite la posibilidad de usar los siguientes protocolos:

- UDP: User Datagram Protocol.
- TCP: Transport Control Protocol.
- ICMP: Internet Control Messaging Protocol.

Opciones de señalización: comunicación TCP de control entre emisor y receptor.

-Sda <signaling_dest_addr>: establece la dirección destino para el canal de control. Por defecto tendrá la misma que la indicada en el parámetro –a.

-Sdp <signaling_dest_port>: establece el Puerto de destino para el canal de control. Por defecto 9000.

-Ssa <signaling_src_addr>: establece la dirección origen para el canal de control. Por defecto es asignada por el sistema operativo.

-Ssp <signaling_src_port>: establece el puerto original del canal de señalización. Por defecto es asignada por el sistema operativo.

-Si <signaling_interface>: establece el interfaz de señalización para el canal de control.

Tiempo entre paquetes:

-C <rate>: establece Tiempo entre paquetes constante.

-U <min_rate> <max_rate>: distribución uniforme.

-E <mean_rate>: distribución exponencial.

-N <mean> <std_dev>: distribución normal.

-O <mean>: distribución de Poisson.

-V <shape> <scale>: distribución de Pareto.

-Y <shape> <scale>: distribución de Cauchy.

-G <shape> <scale>: distribución Gamma.

-W <shape> <scale>: distribución Weibull.

Tamaño de paquetes:

-c <pkt_size>: tamaño constante.

-u <min_pkt_size> <max_ pkt_size >: distribución uniforme.

-e <average_ pkt_size>: distribución exponencial.

-n <mean> <std_dev>: distribución normal.

-o <mean>: distribución de Poisson.

-v <shape> <scale>: distribución de Pareto.

-y <shape> <scale>: distribución de Cauchy.

-g <shape> <scale>: distribución Gamma.

-w <shape> <scale>: distribución Weibull.

En el caso que queramos transmitir un tráfico con una distribución que no se encuentra entre las anteriores, D-ITG permite que le pasemos hasta tres ficheros en el que cada uno indicará: tiempo, tamaño y *payload* de cada paquete. De esta manera transmitirá cualquier flujo personalizado. Esta opción puede ser de utilidad por ejemplo para transmitir tráfico RTP. La generación de los ficheros a partir de una captura .pcap se puede encontrar en el Anexo G.

Además, para que el *payload* sea exactamente el de nuestro fichero deberemos activar la opción –p 2 como parámetro de ITGSend para que no incluya metadatos en el campo de payload.

Los parámetros a utilizar para este tráfico personalizado son:

-Ft <filename>: fichero con tiempos entre paquetes.
-Fs <filename>: fichero con el tamaño de los paquetes.
-Fp <filename>: fichero con el payload de los paquetes a enviar.

Opciones a nivel de aplicación:

- Telnet.
- DNS.
- *Quake III.*
- CS (Counter Strike).
- VoIP. Admitiendo los siguientes códec.
  - G.711.
  - G.723.
  - G.729.

ITGRecv:

ITGRecv será lanzado en el receptor. Permite establecer la conexión y decodificar información estadística.

Uso:

```
# ./ITGRecv [options]
```

Opciones:

-a <bind_address>: establece la dirección origen del flujo de tráfico a recibir. Por defecto lo lee del canal de control entre emisor y receptor.

-i <interface>: establece el interfaz de escucha de tráfico. Si no se especifica ningún interfaz, ITGRecv escuchará en la que se haya configurado en el emisor ITGSend en su parámetro –a. Esta información la recibirá a través del canal de control que mantienen el emisor y el receptor.

-Sp <port>: Puerto de señalización. Por defecto 9000.

-l [logfile]: genera un fichero de log. Nombre por defecto: ITGRecv.log.

ITGDec:

Para decodificar los ficheros de log, necesitamos el programa ITGDec. Su funcionamiento es el siguiente:

```
# ./ITGDec <filename> [options]
```

Por defecto imprimirá por pantalla el resumen del fichero de log.

Opciones:

-o <outfile>: Guarda el fichero de log en un archivo compatible con Octave/Matlab.

-d <DT> [filename]: Imprime el retardo medio cada intervalo de DT milisegundos.

-j <JT> [filename]: imprime la media del retardo cada JT milisegundos.

-b <BT> [filename]: imprime la media de la tasa de bits cada BT milisegundos.

-p <PT> [filename]: imprime la media de paquetes perdidos cada PT milisegundos.

-P: imprime por la salida estándar el tamaño de cada paquete capturado.

-I: imprime por la salida estándar el tiempo entre paquetes.

-h: imprime una ayuda con información sobre los parámetros a utilizar.

F. Ficheros de *script* utilizados durante el trabajo

adapt_ipt_for_a_throughput.pl: En función del *throughput* que se desee, este programa generará el fichero de tiempo entre paquetes que necesita D-ITG para enviar tráfico. Como parámetro de entrada necesitará el fichero con el tamaño de los paquetes y la duración del envío de tráfico.

```
#!/usr/bin/perl
# this PERL script reads two arguments:
#
# $perl adapt_ipt_for_a_throughput.pl desired_throughput duration
packet_size_file.txt ipt_file.txt
#
# The script generates a new inter-packet time file with the duration
and throughput desired when using the
#packet sizes in the input file.
# It expands or shortens the inter-packet time for achieving the
desired throughput
#these are the arguments:
# - desired_throughput: throughput obtained when using the packet size
file and the output ipt file
# - duration: duration of the desired trace in seconds
# - packet size file: text file where packet lengths are (in bytes).
One packet size per line
# - ipt file: file where inter-packet time (in seconds) is stored. One
ipt per line
# THE OUTPUT IPTs are in MILLISECONDS
# usage example:
# $perl calculate_ipt_for_a_throughput.pl 1000 120 sizes.txt ipts.txt
# this means:
#  1 kbps
#  120 seconds
# this subroutine returns a line from the file. If the file is ended,
it returns 0
sub read_file_line {
  my $fh = shift;
  if ($fh and my $line = <$fh>) {
    chomp $line;
    return [ split(/\t/, $line) ];
  }
  return 0;
}
# get the parameters
$desired_throughput = $ARGV[0];
$desired_duration = $ARGV[1];
$size_file = $ARGV[2];
$ipt_file = $ARGV[3];
# variables
my $total_desired_bytes = 0;
my $acum_bytes = 0;
my $end_size_file = 0;
my $end_ipt_file = 0;
my $original_throughput = 0;
my $ipt_relationship;               #relationship between the inter-
packet times
my $adapted_ipt = 0;
#1) calculate de number of bytes required for achieving the desired
```

```
throughput during the desired duration
$total_desired_bytes = $desired_throughput * $desired_duration / 8;
#print STDOUT "total desired bytes: $total_desired_bytes\n";
#2) go through the packet size input file until the number of desired
bytes is achieved.
#   go through the ipt input file calculating the total time.
#   Obtain the original throughput
open(my $size_file_, $size_file);
open(my $ipt_file_, $ipt_file);
my $size_line_ = read_file_line($size_file_);
my $ipt_line_ = read_file_line($ipt_file_);
# I read the size file
while (($end_size_file == 0) & ($end_ipt_file == 0) & ($acum_bytes <
$total_desired_bytes)) {
      # I accumulate the packet size
      if ( $size_line_->[0] != -1) {
            $acum_bytes = $acum_bytes + $size_line_->[0];
            $acum_packets = $acum_packets + 1;
            $acum_time = $acum_time + $ipt_line_->[0];
      }
      $size_line_ = read_file_line($size_file_);
      $ipt_line_ = read_file_line($ipt_file_);
      if (not $size_line_ ) {
            $end_size_file = 1;
      }
      if (not $ipt_line_ ) {
            $end_ipt_file = 1;
      }
}
#print STDOUT "total desired packets: $acum_packets\n";
#print STDOUT "total time original trace: $acum_time\n";
close($size_file_);
close($ipt_file_);
# the file has ended before getting the desired number of bytes
if ($end_size_file == 1) {
      print STDOUT "Not enough packets in the size file for the
desired througput and duration\n";
}
# the file has ended before getting the desired number of bytes
else {
      if ($end_ipt_file == 1) {
            print STDOUT "Not enough inter-packet times in the ipt
file\n";
      }
      # the number of bytes has been achieved
      else {
            #3) calculate the throughput of the original files
            $original_throughput = $acum_bytes * 8 / $acum_time ;
            #print STDOUT "original throughput:
$original_throughput\n";
            # calculate the relationship between the original and the
desired ipt
            $ipt_relationship = $original_throughput /
$desired_throughput ;
            # add a factor of 1000 to get the ipt in milliseconds
            my $ipt_relationship_ms = $ipt_relationship * 1000;
            close($size_file_);
            #4) go through inter-packet time file and recalculate the
inter-packet times
            #   write an output ip file as output_ipt = original_ipt *
original_throughput / desired_throughput
```

```
			open($ipt_file_, $ipt_file);
			$end_ipt_file = 0;
			$ipt_line_ = read_file_line($ipt_file_);
			for (my $i=0; $i < $acum_packets; $i++) {
				# I calculate the new inter-packet time and write it
				if ( $ipt_line_->[0] != -1) {
					$adapted_ipt = $ipt_line_->[0] *
$ipt_relationship_ms ;
					#ipt cannot be null
					if ($adapted_ipt == 0.0 ) {
						$adapted_ipt = 0.00000000001;
					}
					#print STDOUT "$acum_packets\t$i\t$ipt_line_-
>[0]\t$adapted_ipt\n";
					print STDOUT "$adapted_ipt\n";
				}
				$ipt_line_ = read_file_line($ipt_file_);
				if (not $ipt_line_ ) {
					$end_ipt_file = 1;
				}
			}
			if ($end_ipt_file == 1) {
				print STDOUT "Not enough packets in the ipt file";
			}
			close($ipt_file_);
		}
}
#print STDOUT "ipt relationship: $ipt_relationship\n";
```

Pruebas TCP:

Nocturno.sh: Fichero en Shell utilizado para lanzar diversas pruebas de TCP y modificar de manera remota parámetros de red. A su vez es el encargado de ejecutar el programa principal.c.

```
ssh root@192.168.1.2 tc qdisc add dev eth7 root netem delay 10ms 0ms
ssh root@192.168.2.3 tc qdisc add dev eth8 root netem delay 10ms 0ms
sleep 5
./principal 100 20 22 eth10_D10x10_J0x0_n10 10000
ssh root@192.168.1.2 tc qdisc del dev eth7 root netem delay 10ms 0ms
ssh root@192.168.2.3 tc qdisc del dev eth8 root netem delay 10ms 0ms
sleep 5
ssh root@192.168.1.2 tc qdisc add dev eth7 root netem delay 20ms 0ms
ssh root@192.168.2.3 tc qdisc add dev eth8 root netem delay 20ms 0ms
sleep 5
./principal 100 20 22 eth10_D20x20_J0x0_n10 10000
ssh root@192.168.1.2 tc qdisc del dev eth7 root netem delay 20ms 0ms
ssh root@192.168.2.3 tc qdisc del dev eth8 root netem delay 20ms 0ms
sleep 5
ssh root@192.168.1.2 tc qdisc add dev eth7 root netem delay 10ms 1ms
distribution normal
ssh root@192.168.2.3 tc qdisc add dev eth8 root netem delay 10ms 1ms
distribution normal
sleep 5
./principal 100 20 22 eth10_D10x10_J1x1_n10 10000
ssh root@192.168.1.2 tc qdisc del dev eth7 root netem delay 10ms 1ms
distribution normal
ssh root@192.168.2.3 tc qdisc del dev eth8 root netem delay 10ms 1ms
distribution normal
sleep 5
ssh root@192.168.1.2 tc qdisc add dev eth7 root netem delay 10ms 5ms
distribution normal
ssh root@192.168.2.3 tc qdisc add dev eth8 root netem delay 10ms 5ms
distribution normal
sleep 5
./principal 100 20 22 eth10_D10x10_J5x5_n10 10000
ssh root@192.168.1.2 tc qdisc del dev eth7 root netem delay 10ms 5ms
distribution normal
ssh root@192.168.2.3 tc qdisc del dev eth8 root netem delay 10ms 5ms
distribution normal
sleep 5
ssh root@192.168.1.2 tc qdisc add dev eth7 root netem delay 20ms 1ms
distribution normal
ssh root@192.168.2.3 tc qdisc add dev eth8 root netem delay 20ms 1ms
distribution normal
sleep 5
./principal 100 20 22 eth10_D20x20_J1x1_n10 10000
ssh root@192.168.1.2 tc qdisc del dev eth7 root netem delay 20ms 1ms
distribution normal
ssh root@192.168.2.3 tc qdisc del dev eth8 root netem delay 20ms 1ms
distribution normal
sleep 5
ssh root@192.168.1.2 tc qdisc add dev eth7 root netem delay 20ms 5ms
distribution normal
ssh root@192.168.2.3 tc qdisc add dev eth8 root netem delay 20ms 5ms
distribution normal
sleep 5
./principal 100 20 22 eth10_D20x20_J5x5_n10 10000
ssh root@192.168.1.2 tc qdisc del dev eth7 root netem delay 20ms 5ms
distribution normal
```

```
ssh root@192.168.2.3 tc qdisc del dev eth8 root netem delay 20ms 5ms
distribution normal
sleep 5
```

Principal.sh: Programa que evalúa el tiempo de descarga de un fichero. Se especifica el número de veces (con fines estadísticos) que se descargará el fichero remoto.

```
#!/bin/bash
# 5 parametros:
#numero de repeticiones
#IP1 de descarga, solo el ultimo byte
#IP2 de descarga, solo el ultimo byte
#parametro n del tunel para edicion de resultados
#parametro P del tunel para edicion de resultados
for (( c=1; c<=`expr $1`; c++ ))
do
      echo prueba `expr $c`
      #echo prueba `expr $c`>>resultadosA_$2_$3_$4_$5
      #echo prueba `expr $c`>>resultadosB_$2_$3_$4_$5
        if test $3 -eq 0
      then
      ./llamada_descarga1 $1 $2 $3 $4 $5 $c
      sleep 2
      else
      ./llamada_descarga2 $1 $2 $3 $4 $5 $c
      sleep 30
      fi
done
```

llamada_descarga.sh: programa que lanza de manera paralela la descarga de un fichero desde dos direcciones IP distintas.

```
rm debian.iso.$2
rm debian.iso.$3
./descarga_$2 $4 $5 $6>> resultadosA_$2_$3_$4_$5 &
./descarga_$3 $4 $5 $6>> resultadosB_$2_$3_$4_$5
```

Descarga.sh: programa que realiza la descarga del fichero FTP mediante el comando scp.

```
echo -e "prueba_$3\tMUX20_$1_$2\t\tstart\t$(date +%s%N)"
scp
proyecto@192.168.2.20:/home/proyecto/pruebas_transferencia_ficheros_mu
ltiplexando/debian.iso.20
/home/proyecto/pruebas_transferencia_ficheros_multiplexando/debian.iso
.20
echo -e "prueba_$3\tMUX20_$1_$2\t\tend\t$(date +%s%N)"
```

Configuración de escenarios:

Ethernet:

MiniPC2: equipo que actuará de *router* para PC1 y tendrá comunicación directa con el PC1 y el PC3.

```
ifconfig wlan0 down
ifdown eth0
ifdown eth7
ifup eth0
ifup eth7
route -nn add -net 192.168.2.0 netmask 255.255.255.0 gw
155.210.157.238
```

```
ifconfig eth0
ifconfig eth7
route -nn
```

MiniPC3: equipo que actuará de *router* para PC4 y tendrá comunicación directa con el PC2 y el PC4.

```
ifconfig wlan0 down
ifdown eth0
ifdown eth8
ifup eth0
ifup eth8
route -nn add -net 192.168.1.0 netmask 255.255.255.0 gw
155.210.157.237
ifconfig eth0
ifconfig eth8
route -nn
```

Escenario Wi-Fi:

Modo Ad-Hoc: utilizado para la comunicación entre el PC2 y el PC3.

MiniPC2: equipo que actuará de *router* para PC1. Se comunicará con el PC3 mediante un enlace inalámbrico en la banda de 5 GHz.

```
ifconfig ra0 down
ifconfig ra0 192.168.5.7 netmask 255.255.255.0 up
iwconfig ra0 mode ad-hoc rate 54M channel 112 essid AH5GWi5
sleep 10
#ifup ra0
route -nn add -net 192.168.2.0 netmask 255.255.255.0 gw 192.168.5.8
ifconfig ra0
#route -nn
iwconfig ra0
```

MiniPC3: equipo que actuará de *router* para PC4. Se comunicará con el PC2 mediante un enlace inalámbrico en la banda de 5 GHz.

```
ifconfig ra0 down
ifconfig ra0 192.168.5.8 netmask 255.255.255.0 up
iwconfig ra0 mode ad-hoc rate 54M channel 112 essid AH5GWi5
sleep 10
#ifup ra0
route -nn add -net 192.168.1.0 netmask 255.255.255.0 gw 192.168.5.7
ifconfig ra0
#route -nn
iwconfig ra0
```

Modo infraestructura: se utilizarán en este escenario los puntos de acceso Wi-Fi de Tp-Link.

MiniPC2: equipo que actuará de *router* para PC4 y se comunicará con el PC3 a través del punto de acceso Wi-Fi.

```
ifconfig ra0 down
ifconfig ra0 192.168.5.7 netmask 255.255.255.0
iwconfig ra0 mode managed channel 153 essid AP5GWi5
sleep 2
ifup ra0
route -nn add -net 192.168.2.0 netmask 255.255.255.0 gw 192.168.5.8
```

```
ifconfig ra0
route –nn
```

MiniPC3: equipo que actuará de *router* para PC4 y se comunicará con el PC3 a través del punto de acceso Wi-Fi.

```
ifconfig ra0 down
ifconfig ra0 192.168.5.8 netmask 255.255.255.0
iwconfig ra0 mode managed channel 153 essid AP5GWi5
sleep 2
ifup ra0
route -nn add -net 192.168.1.0 netmask 255.255.255.0 gw 192.168.5.7
ifconfig ra0
route -nn
```

Configuración del Túnel: script necesario para crear un interfaz de red al que enviar el tráfico a optimizar.

```
openvpn --mktun --dev tun0 --user root
ip link set tun0 up
ip rule add fwmark 2 table 3
ip route add default dev tun0 table 3
ip route flush cache
ip route show table 3
```

Iptables: script utilizado para especificar aquellas direcciones IP y puertos en los que se quiera que el tráfico dirigido a ellos se marquen con un 2 en la tabla *mangle* y de esta manera sean enviados al interfaz túnel.

```
# iptables -t mangle -A  PREROUTING -m length --length 20:200 -p udp -
-dport 8999 -j MARK --set-mark 2
iptables -F
iptables -X
iptables -t mangle -F
iptables -t mangle -X
iptables -t mangle -A PREROUTING -p udp --dport 9000 -d 192.168.2.20 -
j MARK --set-mark 2
iptables -t mangle -A PREROUTING -p udp --dport 9000 -d 192.168.2.21 -
j MARK --set-mark 2
iptables -t mangle -A PREROUTING -p udp --dport 9000 -d 192.168.2.22 -
j MARK --set-mark 2
iptables -t mangle -A PREROUTING -p udp --dport 9000 -d 192.168.2.23 -
j MARK --set-mark 2
iptables -t mangle -A PREROUTING -p udp --dport 9000 -d 192.168.2.24 -
j MARK --set-mark 2
iptables -t mangle -A PREROUTING -p udp --dport 9000 -d 192.168.2.25 -
j MARK --set-mark 2
iptables -t mangle -A PREROUTING -p udp --dport 9000 -d 192.168.2.26 -
j MARK --set-mark 2
iptables -t mangle -A PREROUTING -p udp --dport 9000 -d 192.168.2.27 -
j MARK --set-mark 2
iptables -t mangle -A PREROUTING -p udp --dport 9000 -d 192.168.2.28 -
j MARK --set-mark 2
iptables -t mangle -A PREROUTING -p udp --dport 9000 -d 192.168.2.29 -
j MARK --set-mark 2
```

G. Generación de distribuciones de tráfico para D-ITG a partir de una captura .pcap

El objetivo de este anexo es generar dos ficheros para D-ITG: uno con el tamaño de los paquetes y otro con el tiempo entre paquetes a partir de una captura real .pcap.

Los pasos que deberemos realizar son:

- Con Wireshark, abrimos la captura del tráfico que queremos simular con D-ITG.
- Hacemos clic derecho sobre un paquete del flujo deseado y elegimos “Follow UDP Stream”
- Guardamos la captura como sólo texto. Para ello, desde la pestaña file, elegimos la opción “Print”. En esta ventana marcamos la pestaña “plain text” y “output to file” para guardarla en un directorio. En “Packet Format” sólo deberemos dejar marcada la opción “Packet summary line”.
- Una vez que tenemos la captura en formato .txt, la abrimos con un editor de tablas de cálculo, por ejemplo Excel. Desde ahí, separamos el texto en celdas y guardamos la columna ip.len en el fichero “lengths_ip.len.txt”.
- Guardamos también la columna inter-packet time. (No coger la de tiempo absoluto, sino la diferencia). Esta columna la podemos guardar en el fichero ipt.txt.
- Una vez hecho esto, debemos ajustar la longitud para quitar las cabeceras IP y UDP. Para ello, desde el terminal (en equipo Linux) restamos 28 a cada longitud del fichero “lengths_ip.len.txt” mediante el siguiente comando:

```
# perl -nle 'print $_-28' lengths_ip.len.txt > lengths-
28.txt
```

- También ajustamos el tamaño de los paquetes para evitar que haya alguno de longitud 0 o mayor de 1500. Utilizaremos el programa “adjust_packet_sizes.perl”

```
# perl adjust_packet_sizes.perl lengths-28.txt 1 1472 >
lengths-adjusted.txt
```

Si queremos modificar la tasa de bits enviados podemos utilizar el programa adapt_ipt_for_a_throughput.pl que editará el fichero de tiempo entre paquetes para ajustarlo a la velocidad deseada.

```
# perl adapt_ipt_for_a_throughput_.pl 11000000 30 lengths-adjusted.txt
ipt.txt > ipt_11M.txt
```

En este ejemplo, obtendríamos un *throughput* de 11 Mbps durante 30 segundos de duración.

Ahora ya tendríamos nuestros dos ficheros para D-ITG. Un posible ejemplo de uso sería el siguiente:

```
#./ITGSend -a 192.168.2.4 -Fs lengths-adjusted.txt -Ft ipt.txt -t
```

```
30000 -l log_file
```

En este ejemplo estamos usando el fichero de tiempo entre paquetes y el de tamaños. Al no haber indicado un fichero con el *payload*, ITGSend enviará información estadística y *padding* en el campo de *payload*.

Si queremos incluir el *payload* de una captura de tráfico real deberemos obtener un fichero binario de la captura .pcap. Para ello realizaremos los siguientes pasos:

- Abrimos la captura .pcap con Wireshark, hacemos clic derecho en un paquete del flujo deseado y elegimos "Follow UDP Stream". Se abrirá una ventana nueva y con la opción elegida "Raw", hacemos clic en "Save as…". De esta manera ya tendremos el archivo binario necesario para D-ITG.

Un ejemplo de uso con todo lo anterior explicado podría ser:

```
#./ITGSend -a 192.168.2.4 -Fs lengths-adjusted.txt -Ft ipt.txt -t
30000 –Fp captura.bin –p 2 -l log_file
```

Recordemos que la opción –p 2 hace que no se vea alterado el *payload* con medidas estadísticas de D-ITG.