CT-738-SSI
Don, 2013

xx(209442.1)

Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional
Unidad Guadalajara

Diseño e implementación de simulador de PCI Express

Tesis que presenta:

José Francisco Lombera Landa

para obtener el grado de:

Maestro en Ciencias

en la especialidad de:

Ingeniería Eléctrica

Director de Tesis

Dr. Mario Angel Siller González Pico

CINVESTAV del IPN Unidad Guadalajara, Guadalajara, Jalisco, Octubre de 2012.

Diseño e implementación de simulador de PCI Express

Tesis de Maestría en Ciencias
Ingeniería Eléctrica

Por:
José Francisco Lombera Landa
Ingeniero en Sistemas Computacionales
Instituto Tecnológico de la Costa Grande 2004-2009

Becario de Consejo Nacional de Ciencia y Tecnología (CONACYT),
expediente no. 234576

Director de Tesis
Dr. Mario Angel Siller González Pico

CINVESTAV del IPN Unidad Guadalajara, Octubre de 2012.

Diseño e implementación de simulador de PCI Express®

José Francisco Lombera Landa

Resumen

El uso de PCI Express® para el diseño de circuitos electrónicos y canales de comunicación demanda de una herramienta con la cual sea posible experimentar en la implementación y utilización de dicha tecnología. Un simulador de PCI Express® representa una solución a dicha herramienta.

Mediante simulación será posible hacer pruebas de diferentes configuraciones en una topología de PCI Express®. Para un investigador será posible el estudio del impacto en el desempeño que supone el uso de PCI Express® en canales de comunicación (inter-sistema e intra-sistema), además de la validación de modelos matemáticos que surjan de dichos estudios. Un fabricante/ensamblador podrá realizar pruebas antes de comenzar la fabricación de un producto.

En esta tesis, se propone el diseño de un simulador de PCI Express®. El simulador es implementado e integrado en el simulador de redes NS-3. Las pruebas realizadas a la implementación del simulador nos revelan el papel que puede jugar una red intra-sistema en el desempeño de redes inter-sistemas.

Abstract

The use of PCI Express® in the designing of electronic circuits and communication channels demands a tool to experiment with the implementation and utilization of such technology. A PCI Express® simulator represents such a tool.

Through the use of simulations it will be possible to test different configurations in a PCI Express® topology. For a researcher, it will be possible to study the impact on the performance of communication channels (intra-system and inter-system), and validate the mathematical models arising from such studies. A manufacturer/vendor may test a product before it is manufactured and released to the market.

In this thesis, the design of a PCI Express® simulator is proposed. The simulator is implemented and integrated into the NS-3 network simulator. Finally, the test of the simulator yields out the role a intra-system network plays in the performance of inter-system networks.

Agradecimientos

A mis padres, por apoyarme incondicionalmente en todo momento y ensañarme lo verdaderamente importante de la vida.

A mis compañeros, por brindarme su amistad y soportarme con todos mis defectos.

A mi asesor, el Dr. Mario Ángel Siller por su apoyo y paciencia.

A CONACYT, por el apoyo económico, sin el cual no habría sido posible la realización de esta tesis.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

PCI Express® se ha convertido en el estándar de facto en materia de conexión intrasistema de dispositivos. Todas las computadoras personales que se fabrican hoy en día se integran usando PCI Express®. Tiene cada vez más presencia en arquitecturas de alto desempeño como clusters, centros de datos y supercomputadoras. Además que es usado por los diseñadores y ensambladores de sistemas empotrados (embedded systems).

PCI Express® cuenta con características avanzadas como: *control de flujo* (FC), *servicios diferenciados* (DiffServ) y *canales virtuales* (VC). Estas características abren nuevas oportunidades a los fabricantes e investigadores para probar nuevas configuraciones y parámetros, incluso en capas superiores ajenas a PCI Express®.

1.1. Motivación

Para un fabricante/ensamblador de dispositivos y para un diseñador de sistemas empotrados, es importante realizar pruebas de las configuraciones que se hará en los dispositivos para verificar que todo funciona correctamente y de acuerdo a lo planeado.
Un investigador, debe validar los modelos, procesos y protocolos que está proponiendo y obtener resultados para respaldar su investigación.

Ambos necesitan una plataforma con la flexibilidad suficiente para que se ajuste a sus necesidades, que en ciertas circunstancias pueden ser diferentes. En la actualidad no se cuenta con dicha plataforma con la suficiente flexibilidad.

1.2. Descripción del problema

El uso de PCI Express® para el diseño de circuitos electrónicos y canales de comunicación por parte de fabricantes/ensambladores e investigadores demanda de una herramienta con la cual sea posible experimentar en la implementación y utilización de dicha tecnología.

Un simulador de PCI Express® representa una solución a dicha herramienta. Mediante el uso de un simulador será posible hacer pruebas de las diferentes configuraciones posibles en una topología de PCI Express®
Para un investigador será posible el estudio del impacto en el desempeño que supone el uso de PCI Express® en canales de comunicación (inter-sistema e intra-sistema), además de la validación de modelos matemáticos que surjan de dicho estudios.

1.3. Objetivos

- Desarrollar un simulador software de la arquitectura PCI Express®.
- Validar que el comportamiento del simulador sea el esperado.

Capítulo 2

Marco teórico

2.1. Simulación

2.1.1. Obtención de datos en la investigación

En la investigación muchas veces se necesitan datos que describan los fenómenos (físicos, biológicos, ópticos, etc.) que se están estudiando. Es importante poder contar con estos datos antes y durante la investigación.
La importancia de tenerlos antes, es porque, a partir de estos se decide el rumbo y/o los objetivos de la investigación, es decir, a partir del análisis de los datos se pueden detectar algunos propiedades de los eventos que están ocurriendo, que muchas veces no son deseables, y de aquí la necesidad de mejorar y/o conocer los procesos que intervienen.
Durante la investigación es importante poder reproducir los fenómenos estudiados, quizá aplicando los resultados y/o hipótesis que se están generando en el proceso de investigación, y así poder comparar los resultados.

Existen básicamente tres métodos para reproducir los fenómenos que se están estudiando y poder generar datos: camas de prueba, modelos matemáticos y simulación. No existe una regla exacta para la elección de que método utilizar.

2.1.1.1. Camas de prueba

Estas son las que nos podrían dar datos más reales y confiables. Consisten básicamente en montar escenarios reales para reproducir los fenómenos que se están estudiando y de alguna manera obtener los datos que se necesiten para su posterior análisis.

Sin embargo existen algunos inconvenientes con este método:

- Deben ser acotados, principalmente en tamaño, debido a que en muchas ocasiones es muy complicado, por razones físicas, desplegar un escenario de tamaño y alcances considerables.
- Dependiendo del fenómeno que se esté estudiando, puede ser muy costoso en tiempo y económicamente montar y realizar cada uno de los escenarios y pruebas requeridos.
- Existen muchos factores que pueden intervenir en un escenario real y que no son el objeto de estudio y producen datos menos objetivos y que es necesario tratar con cuidado en el análisis.

2.1.1.2. Modelos matemáticos

Los modelos matemáticos nos permiten abstraer el comportamiento de los fenómenos estudiados. Son representados usando algún formalismo

matemático. Gracias a que los fenómenos se representan matemáticamente, sus propiedades pueden ser deducidas mediante la matemática. Además de que nos permiten realizar aproximaciones al comportamiento de los fenómenos estudiados al modificar partes de su entorno.

Los datos obtenidos con este método no son tan exactos como los obtenidos con las camas de pruebas, ya que es necesario hacer algunas suposiciones y relajaciones para poder abstraer el fenómeno estudiado. Los datos obtenidos con este método podrían ser considerados como *"datos sintéticos"*

Al igual que las camas de pruebas, los modelos matemáticos tienen algunos inconvenientes:

- Como ya se mencionó, los resultados obtenidos son aproximados.
- No siempre es fácil generar un modelo matemático que represente un fenómeno, siendo muchas veces el descubrimiento de dicho modelo el objetivo de una investigación.
- También se pueden ver limitados en la complejidad y alcance de los fenómenos estudiados, ya que de no hacer relajaciones, los formalismos matemáticos utilizados se pueden hacer muy difíciles de tratar.

2.1.1.3. Simulación

El proceso de simulación consiste en reproducir un fenómeno utilizando tanto hardware como software. Solo es necesario cambiar parámetros de configuración para poder realizar diferentes escenarios de simulación. Por ejemplo, al simular una red de comunicaciones, se puede configurar el "Bit Error Rate" que sufre el canal físico o la tasa de transferencia de datos.

Este método es el más flexible, ya que incluso se puede auxiliar internamente los dos métodos anteriormente mencionados. Sin embargo, también tiene sus inconvenientes:

- En ocasiones es más complicado y costoso el desarrollo del simulador que montar camas de pruebas o la utilización de modelos matemáticos.
- Un simulador "básico" que no se auxilie de modelos matemáticos ni camas de pruebas. probablemente se aproxime menos a los resultados reales que los otros métodos mencionados.
- Es necesario tener cierto conocimiento previo del fenómeno estudiado para poder diseñar e implementar un simulador.
- No siempre es fácil validar el funcionamiento del simulador.

2.1.1.4. Que método elegir

El método a utilizar va a depender de las necesidades y restricciones específicas del fenómeno estudiado.

En general, se deben utilizar camas de pruebas cuando sea viable su implementación y/o se necesite la mayor exactitud en los datos obtenidos; los modelos matemáticos son más convenientes cuando, la complejidad del modelo sea manejable y. se desee abstraer el fenómeno para estudiar ciertas características de forma matemática; la simulación se utilizará cuando ninguno de los métodos anteriores sea viable y/o cubra las necesidades, ya que se puede auxiliar de los anteriores para hacerlo tan complejo y preciso como se desee.

2.2. PCI Express®

PCI Express® es también conocida como la *"Interconexión de E/S de tercera generación"* (3GIO). Su diseño original fue pensando en el desempeño y escalabilidad. A continuación se listan algunas de sus características:

- Retro compatible con PCI/PCI-X.
- Conexión serial punto a punto.
- Ancho de banda escalable y configurable.
- Administración de energía.
- Calidad de servicio (QoS):
  - Clases de tráfico
  - Canales virtuales
  - Control de flujo basado en créditos
  - Control de congestión implícito

2.2.1. Arquitectura de PCI Express®

2.2.1.1. Diseño por capas

El diseño de PCI Express® está hecho por capas: Capa de Transacciones, Capa de Enlace de Datos y Capa Física. Estas se muestran la Figura 2.1. De esta manera es más fácil mantener una compatibilidad entre diferentes versiones de la especificación. Por ejemplo, en nuevas versiones de la especificación se puede aumentar el ancho de banda soportado mejorando

la capa física, sin que los diseños de las capas de transacciones y de enlace de datos se vean afectados.

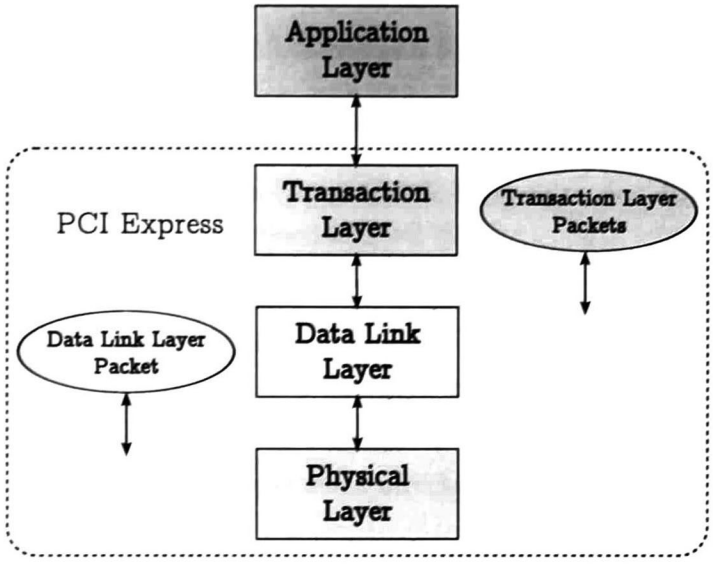

Figura 2.1: Diseño por capas de PCI Express®

Capa de aplicación

Está capa no es propia de PCI Express® y se deja abierta a la implementación. Es la responsable en última instancia de producir y consumir los datos que se envían/reciben en los TLPs (Transaction Layer Packet, véase Sección 2.2.1.2) en comunicaciones fin a fin.

Capa de transacciones

Esta capa es la encargada de la comunicación fin a fin entre dos dispositivos de toda la topología de PCI Express® (véase Sección 2.2.2). Algunas de sus funciones son:

- Genera y procesa TLPs (véase Sección 2.2.1.2).
- Asegura la integridad de los datos de fin a fin.
- Maneja los errores correspondientes a los TLPs.
- Administra el mapeo entre clases de tráfico y canales virtuales.
- Intercambia información de control de flujo entre dos dispositivos de un mismo enlace (véase Sección 2.2.2.1).

Capa de enlace de datos

Esta capa se encuentra entre la capa física y de transacciones. Su labor consiste principalmente en asegurar la comunicación lógica entre dos dispositivos PCI Express® conectados directamente a través de un enlace. Algunas de sus funciones son:

- Genera y procesa DLLPs (Data Link Layer Packet, véase Sección 2.2.1.3).
- Encapsula y envía TLPs y el proceso inverso.
- Detección de errores de transmisión.
- Verificación de la integridad (CRC) de datos al ser enviados entre dos dispositivos.
- Retransmisión de datos en caso de ser necesario.
- Administración de los canales virtuales.

Capa física

Esta capa es la responsable de la conexión física entre dos puertos de PCI Express® (véase Sección 2.2.2.2. Algunas de sus funciones son:

- Codificación/decodificación de datos.
- Serialización/deserialización de datos.
- Inicialización y negociación de ancho de banda.
- Administración de energía.
- Administración y cambio de estados del enlace (Reset, Hot-Plug, status).

2.2.1.2. TLPs

Los "paquetes de la capa de transacciones" (TLPs por sus siglas en inglés) son los responsable de transportar información de fin a fin en una topología de PCI Express®

PCI Express® describe dos formatos de cabeceras de TLPs: 32-bit y 64-bit (extendida). Estas se pueden observar en las Figuras 2.2 y 2.3. Para más detalles sobre cada uno de los campos véase [1].

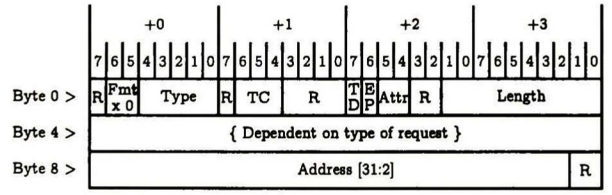

Figura 2.2: Formato de cabecera de 32-bit de TLP.

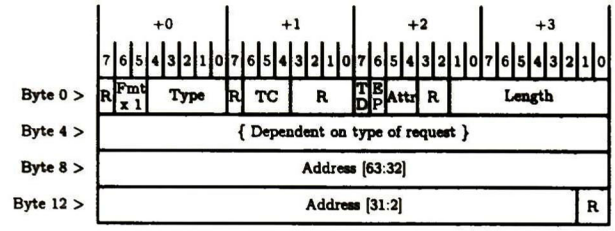

Figura 2.3: Formato de cabecera de 64-bit (extendida) de TLP.

Existen cuatro tipos de TLPs de acuerdo al modo de direccionamiento: mapeo a memoria, mapeo a puertos de E/S, configuración y mensajes. En la Tabla 2.1 se puede observar una descripción de los diferentes tipos.

Tabla 2.1: Tipos de TLPs.

Direccionamiento	Tipo	Descripción
Memoria	Lectura Escritura	Transfiere datos a/desde una dirección mapeada a memoria
E/S	Lectura Escritura	Transfiere datos a/desde una dirección mapeada a un puerto de E/S
Configuración	Lectura Escritura	Configuración de dispositivos
Mensaje	Definidos Proveedor	Desde señalización de eventos hasta mensajes generales

Para ser transmitido a través de un enlace PCI Express® (véase Sección 2.2.2.1), un TLP es encapsulado por la capa de enlace. La capa de enlace añade un número de secuencia y una suma de verificación. El número de secuencia se utiliza para realizar retransmisión de datos en caso de ser necesario. La Figura 2.4 muestra el formato del TLP encapsulado.

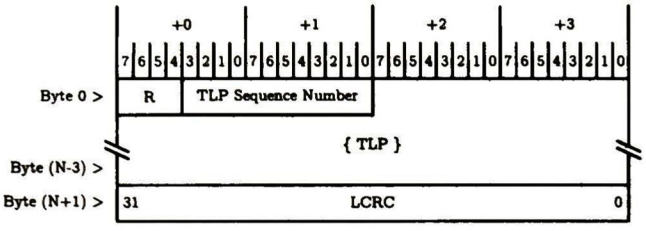

Figura 2.4: Formato de un TLP encapsulado por la capa de enlace de datos.

2.2.1.3. DLLPs

Los "paquetes de capa de enlace de datos" (DLLPs por sus siglas en inglés) son utilizados para transmitir información referente a la capa de enlace. La Figura 2.5 muestra el formato de un DLLP.

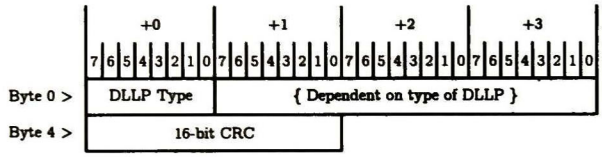

Figura 2.5: Formato de DLLP

La Tabla 2.2 muestra los diferentes tipos de DLLP definidos, que son utilizados para: acuso de recibo de los TLPs recibidos, administración de energía, inicialización y actualización de los créditos de control de flujo y DLLPs específicos del proveedor.

2.2.1.4. Control de flujo

Como ya se mencionó, PCI Express® realiza control de flujo basado en créditos[2]. La manera en como funciona el control de flujo en PCI Express® es la siguiente:

Tabla 2.2: Tipos de DLLPs.

Tipo de DLLP	Descripción
ACK/NACK	Acuso de recibo de los TLPs
PW_Mngt	Administración del consumo de energía
Init_FC	Inicialización de créditos de control de flujo
Update_FC	Actualización de créditos de control de flujo
Proveedor	Específicos del proveedor

- Por cada uno de los VCs, se estable una cantidad de créditos tanto en el emisor como en el receptor. Para el emisor, los créditos representan la cantidad de datos que puede enviar; para el receptor, representa la cantidad de datos que puede recibir.
- Inicialmente, el emisor y receptor tienen la misma cantidad de créditos.
- Cuando el emisor quiere transmitir un TLP:
  1. Calcula cuantos créditos consumirá el TLP.
  2. Si tiene suficientes créditos el TLP es enviado y se resta la cantidad de créditos previamente calculados.
  3. Si no hay suficientes créditos el TLP es retrasado hasta que haya suficientes.
- Cuando el receptor recibe un TLP:
  1. Calcula cuantos créditos consumirá el TLP.
  2. Si tiene suficientes créditos, el TLP es aceptado y almacenado en el buffer correspondiente y se resta la cantidad de créditos consumidos por el TLP.
  3. Si no hay suficientes créditos, el TLP es descartado.

- En el emisor, cuando un TLP que estaba almacenado en los buffers es extraído por capas superiores, los créditos que el TLP había consumido son recuperados en el receptor.

- Si la cantidad de créditos de que disponen el emisor y receptor difieren, el receptor envía créditos al emisor si se cumple una de las siguientes condiciones:

  - Cuando la cantidad de créditos de que disponga el emisor sea menor a la cantidad de créditos necesaria para enviar un TLP de la máxima longitud permitida (1024 créditos).
  - Cualquier otra regla específica de la implementación, como por ejemplo, después de un periodo de tiempo.

  El receptor nunca enviará más créditos de los que dispone.

Como se puede observar en la descripción previa, los créditos, y en general el control flujo, solo es usado para el manejo de TLPs. Los DLLPs son consumidos inmediatamente por el receptor (capa de enlace de datos, ver Figura 2.1), por lo que no consumen créditos.

Como también se observa en la descripción de control de flujo de PCI Express®, el emisor no podrá enviar TLPs al receptor a menos que disponga de los créditos suficientes. Si se respeta la especificación de PCI Express®, el emisor nunca tendrá más créditos que el receptor, asegurando que el receptor siempre pueda recibir los TLPs que el emisor envía. Esto evita la retransmisión de TLPs que se hayan perdido debido a que el receptor no pudiera aceptarlos.

Es importante comentar que un crédito de control de flujo representa 4 bytes, es decir, por cada crédito el transmisor/receptor podrá enviar/recibir 4 bytes de datos.

2.2.1.5. Arbitraje de VCs

Cuando un puerto PCI Express® (véase Sección 2.2.2.2) soporta más de un VC, este debe utilizar un algoritmo de arbitraje para determinar como se comparte el enlace físico entre los diferentes VCs.

PCI Express® define tres métodos de arbitraje:

Strict Priority (SP). En este método, siempre se despacha el VC con mayor prioridad que tenga paquetes en su buffer[1, 3]. La prioridad de los VCs es de forma ascendente, esto es, entre mayor sea el número del VC, mayor es su prioridad.

Round-Robin (RR). Este algoritmo va iterando entre todos los VCs y envía, a lo más un paquete por turno del VC correspondiente a la iteración.[2]

Weighted Round-Robin (WRR). Este algoritmo es parecido al Round-Robin[2], sin embargo, cada VC_i tiene asignado un peso w_i. A cada VC_i se le asigna una fracción f_i del flujo total basado el su peso w_i[2, 1, 3]:

$$W = \sum_{i=0}^{n-1} w_i,$$
$$f_i = \frac{w_i}{W}$$

Low Priority Extended VC Count

PCI Express® define el contador "Low Priority Extended VC" (LPEVC) como la cantidad de VCs de baja prioridad. Usando este contador, se puede dividir el conjunto de canales virtuales en dos grupos: uno de alta prioridad, en el cual se despachan los paquetes de los VCs usando el método de arbitraje

SP; y un grupo de baja prioridad cuyos paquetes se despachan usando RR, WRR o usando algún método específico de la implementación. Los paquetes del grupo de baja prioridad solo serán despachados cuando no haya paquetes en ningún VC del grupo de alta prioridad. La Figura 2.6 muestra un esquema de la división de los VCs para su arbitraje, así como los métodos de arbitraje utilizados.

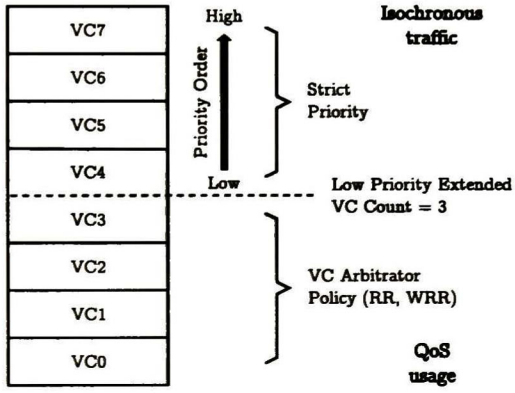

Figura 2.6: Políticas usadas en el arbitraje de VCs.

2.2.2. Topología

La topología de PCI Express® consta de los siguientes componentes principales: Enlace, Puerto, Conmutador, Endpoint, Puente entre PCI Express® y PCI/PCI-X y Root Complex. Esta topología se puede observar en la Figura 2.7. A continuación se detallan cada uno los componentes.

2.2.2.1. Enlace

Es un enlace serial punto a punto entre dos dispositivos PCI Express®. A cada uno de sus extremos se encuentra un puerto PCI Express® (véase

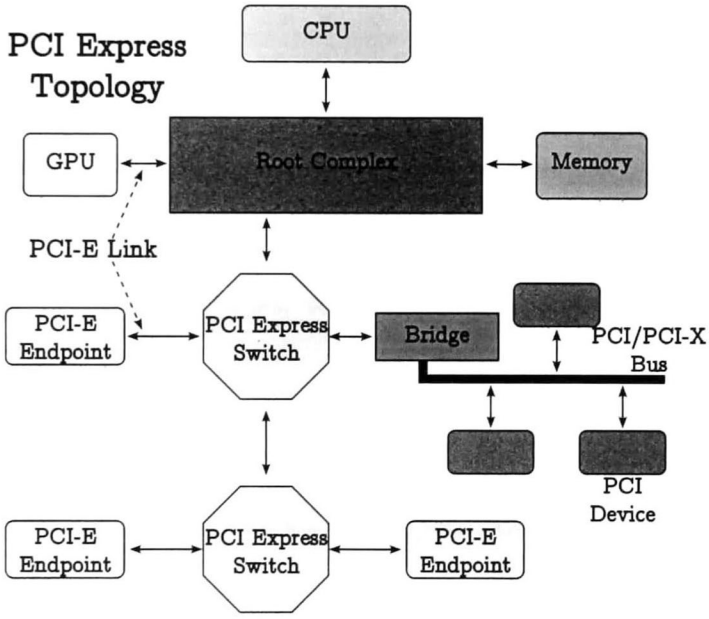

Figura 2.7: Topología de PCI Express®

Sección 2.2.2.2). Se conforma físicamente de varias lineas ("lanes") de transmisión de datos. Al aumentar la cantidad de lineas, se puede aumentar el ancho de banda soportado por el enlace. La cantidad de lineas soportadas está definida en la especificación de PCI Express® como: x1, x2, x4, x8, x12, x16 y x32[1]. El enlace es "full-duplex".

2.2.2.2. Puerto

Como se comento anteriormente, los puertos PCI Express® se encuentran en cada extremo de los enlaces PCI Express®. Este es el encargado de administrar los Canales Virtuales (VCs) y sus respectivos buffers, tanto de entrada como de salida. Realiza el mapeo de Clases de Tráfico (TCs) y VCs. Este componente es el encargado de realizar el arbitraje y multicanalización de los diferentes VCs. Además, se encarga de la retransmisión de TLPs cuando se requiere.
Internamente cuenta con los siguientes componentes:

Buffers Rx/Tx. Contiene buffers separados para recepción/envío (Rx/Tx) de TLPs por cada uno de los VCs soportados.

Árbitro de VCs. Para soportar diferentes VCs para la transmisión sobre el mismo enlace físico, se realiza una multicanalización basada en tiempo. El orden en que se elige un VC para transmitir a través del enlace es decidido por este componente.

Buffer de retransmisión. Este buffer almacena los TLPs enviados para una posterior retransmisión en caso de ser necesario.

Todos los dispositivos PCI Express® deben tener al menos un puerto para poder conectarse y comunicarse con otros dispositivos.

[1]La ultima versión de PCI Express®, la 3.0, solo define hasta x16

Una representación de este componente se puede observar en la Figura 2.8.

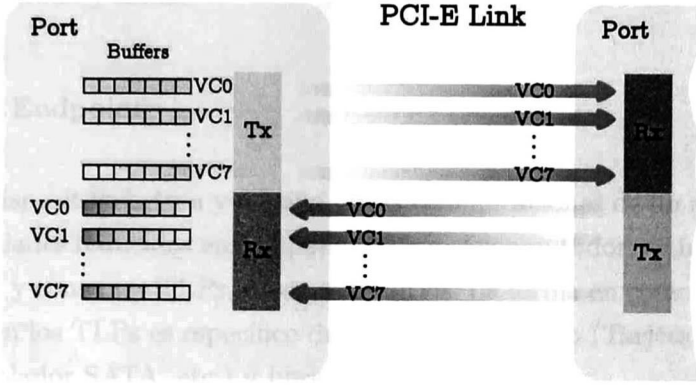

Figura 2.8: Enlace PCI Express®

2.2.2.3. Conmutador

El conmutador (Switch) de PCI Express®, como su nombre lo indica, funciona como punto de interconexión entre varios Endpoints.
Además, realiza el trabajo de encaminador para las comunicaciones de fin a fin en topologías que cuenten con varios conmutadores conectados entre sí; este es el único dispositivo PCI Express® que se "exige" tenga esta funcionalidad.

Internamente se integra de los siguientes componentes:

Puertos. Uno por cada conexión con otro dispositivo.

Registros. Uno por cada puerto. Estos registros almacenan el estado de los puertos y la información necesaria para poder realizar la tarea de conmutación y encaminamiento de los TLPs.

Árbitro de puertos. Se encarga de la tarea de arbitraje entre los puertos de entrada y salida.

2.2.2.4. Endpoint

Es el dispositivo origen y destino de las comunicaciones de fin a fin. Tiene funcionalidades reducidas en comparación con el conmutador; su función es la de generar y consumir TLPs, no de redirigirlos. La forma en como se generan y consumen los TLPs es específico del tipo de dispositivo (Tarjeta gráfica, de red, controlador SATA, etc.) y libre a la implementación.
Como ya se mencionó anteriormente, debe tener un puerto PCI Express®

2.2.2.5. Puente PCI/PCI-X

Es un dispositivo especial que sirve de puente para la comunicación entre una topología PCI Express® y una topología PCI/PCI-X.

2.2.2.6. Root Complex

Solo puede existir un dispositivo de este tipo en una topología de PCI Express®. Su función es la de comunicar componente específicos como la CPU y memoria con la topología PCI Express®. La especificación de PCI Express® solo establece que el punto de conexión con la topología PCI Express® debe seguir la especificación (en la Figura 2.7 se representa en la conexión del Root Complex con el conmutador), la tecnología que se utilice para conectar los componentes se deja libre a la implementación. Así también, es libre a la implementación si el Root Complex tenga capacidades de encaminador.

Capítulo 3

Estado del arte

3.1. PCI Express®

3.1.1. Simuladores de PCI Express®

De acuerdo a nuestra revisión del estado del arte, actualmente no existe ningún simulador de PCI Express® que cubra las necesidades que ya se mencionaron en la Sección 1.1.

Se han encontrado simuladores provistos por fabricantes hardware. Estos simuladores son muy dependientes a los componentes hardware que ellos distribuyen. Además, se enfocan casi únicamente en la verificación de la capa física, no son flexibles y tampoco son de acceso público.

En [4] se implementa un simulador en el cual se monta un escenario muy simple que consta de leer y escribir una palabra (DW) del Root Complex a un Endpoint y viceversa. El simulador que ahí se describe está conformado mayormente por componentes hardware. Es muy poco flexible, ya que no

es posible simular un escenario más complejo que el ya descrito. El único componente del simulador que puede ser modificado es la implementación de la capa de aplicación del Endpoint, ya que para modificar cualquier otro componente del simulador, prácticamente sería necesario diseñar otro simulador.

3.2. Simulador de redes NS-3

NS-3 es el sucesor del simulador NS-2. Su desarrollo es muy activo y cada vez cuenta con un mayor número de modelos para simulación.

El simulador de redes NS-3 tiene las siguientes características:

- Basado en eventos discretos
- Modular
- Software libre
- Desarrollado en C++
- Bindings para Python.

En la Figura 3.1 se puede observar el diseño por capas de NS-3. Este diseño modular y por capas de NS-3 facilita el desarrollo e integración de nuevos módulos y modelos.

Actualmente, NS-3 cuenta con modelos para la simulación de la pila de protocolos TCP/IP (IPv4/IPv6) (y diferentes protocolos de encaminamiento); protocolos de acceso al medio como WiFi[5], WiMAX[6, 7], CSMA; modelos de modulación y propagación de señales; modelos

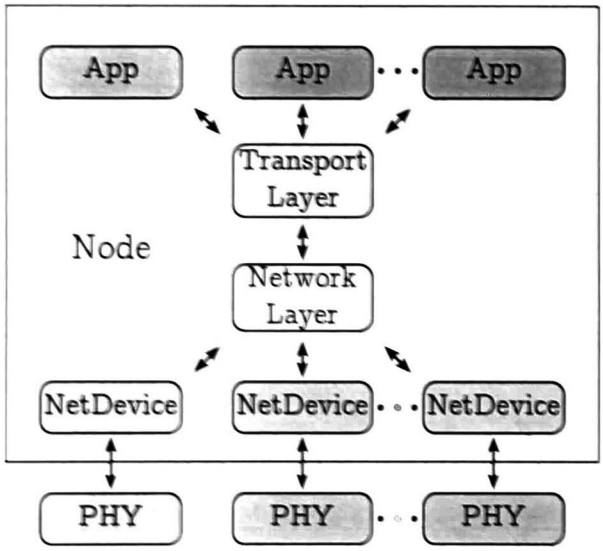

Figura 3.1: Diseño por capas de NS-3.

de consumo de energía; modelos de movilidad de nodos; módulo para conmutadores OpenFlow[8]; módulo para encaminadores Click[9].

Capítulo 4

Propuesta

Dado que no se cuenta con un simulador de PCI Express® con la suficiente flexibilidad, se desarrollará un simulador software de esta tecnología.

Se usará el simulador de redes NS-3 como base para la implementación ya que su diseño modular facilita la integración de nuevos módulos. En específico, se va a desarrollar e integrar un módulo de PCI Express® en NS-3. Esto nos da la ventaja de poder usar todos los modelos y características con que cuenta NS-3 para poder simular escenarios muy amplios y complejos usando PCI Express® como tecnología de interconexión intrasistema.

4.1. Requerimientos

Los requerimientos de un sistema, son los servicios que el sistema provee y sus restricciones operacionales. Estros requerimientos reflejan lo que los usuarios necesitan de un sistema que los ayuda a resolver algunos problemas.[10]

Los requerimientos de un sistema se dividen en: de usuario y de sistema. Los requerimientos de sistema, a su vez se dividen en funcionales y no funcionales.

Requerimientos de usuario. Son sentencias abstractas en lenguaje natural de los servicios que se espera que el sistema provea a los usuarios y las restricciones bajo las cuales debe operar.

Requerimientos de sistema. Establecen las funciones, servicios y restricciones operacionales del sistema en detalle.

Requerimientos funcionales. Establecen los servicios que el sistema debe proveer, como debe reaccionar a datos específicos y como se debe de comportar en situaciones particulares.

Requerimientos no funcionales. Son restricciones en los servicios o funciones ofrecidas por el sistema.

En este documento se van a desplegar los requerimientos de la siguiente manera:

1. Requerimiento de usuario.

   1.1 Requerimiento funcional/no funcional.

   1.2 Requerimiento funcional/no funcional.

   1.3 ...

Se han identificado los siguientes requerimientos para el simulador de PCI Express®:

1. Se debe soportar comunicación fin a fin.

2. Se deben soportar las capacidades de QoS.

   2.1 La transmisión de datos entre dispositivos PCI Express® se debe hacer multicanalizada basada en tiempo usando canales virtuales (VC).

   2.2 A los TLPs se les debe de asignar una clase de tráfico (TC).

   2.3 Cada puerto PCI Express® debe tener un mapeo que asigne a cada TC soportada un y solo un VC.

   2.4 Se debe implementar control de flujo basado en créditos.

3. La implementación del simulador se debe de apegar a la especificación de PCI Express®

   3.1 El formato de los TLPs y DLLPs debe ser compatible a nivel de bit con la especificación (véase Sección 2.2.1.2 y Sección 2.2.1.3).

   3.2 Se deben usar los algoritmos expresados en la especificación para el arbitraje de puertos y de VCs (véase Sección 2.2.1.5).

   3.3 Todos los puertos PCI Express® deben soportar al menos el VC0.

   3.4 La TC0 siempre debe de estar mapeada al VC0.

   3.5 La TC asignada a un TLP en el momento de que es creado no debe cambiar en todo el recorrido que el TLP haga por toda la topología PCI Express®.

4. El usuario debe poder configurar los parámetros del simulador PCI Express®.

   4.1 Individualmente, en cada puerto PCI Express® se deben poder configurar los TCs soportados, los VCs soportados y el mapeo entre estos.

   4.2 Se deben poder configurar los créditos de control de flujo (FC) y el tamaño de los buffers de recepción y envío en los puertos PCI Express®.

4.3 La TC asignada a los TLPs es configurable.

4.4 La configuración de los parámetros antes mencionados es opcional. El simulador debe tener parámetros por defecto.

5. El usuario debe ser capaz de obtener información de simulación.

   5.1 Se deben implementar las *"callbacks"* (`TraceSource` en terminología de NS-3) necesarias con la granularidad más fina posible.

6. El diseño e implementación del simulador debe permitir su fácil modificación y mejora.

   6.1 El diseño e implementación debe ser modular.

7. El módulo se debe de integrar lo más transparente posible a NS-3.

   7.1 Todos los escenarios actuales de simulación de NS-3 deben de poder funcionar sin ninguna modificación.

4.2. Alcance y limitaciones

- Dado que nuestro interés en PCI Express® es desde el punto de vista de redes de datos, no se realizará la simulación de las funciones de la capa física (véase Sección 2.2.1.1), con la excepción del tiempo de transmisión.
- Solo se van a implementar TLPs de tipo de mapeo a memoria (véase Sección 2.2.1.2).
- Solo se van a implementar los siguientes tipos de DLLPs (véase Sección 2.2.1.3): ACK/NACK y Update_FC.

- No se va a implementar el Root Complex.
- No se va a implementar ningún tipo de puente PCI/PCI-X.
- Solo se van a implementar los métodos SP y RR para el arbitraje de VCs.
- Dado que se requiere una cantidad considerable de refactorización en el simulador NS-3, no será posible cumplir con el requerimiento 7.1.

4.3. Diseño

Como ya se mencionó en la Sección 4.2, solo se va a implementar un subconjunto de la topología y arquitectura de PCI Express®

Dado que el simulador se debe de apegar en lo posible al diseño e integración de PCI Express® en la arquitectura de computadoras, y tomando ventaja del diseño por capas de NS-3, el módulo de PCI Express® se va integrar entre las capas de red y de enlace de datos de NS-3. De esta manera se va a simular que la topología de PCI Express® se encuentra entre los dispositivos de red y el software del S.O; justo como sucede en las arquitecturas de computadoras reales.

Tomando en cuentas las Figuras 2.7 y 3.1, en la Figura 4.1 se puede observar a grandes rasgos el diseño por capas de NS-3 una vez que el módulo de PCI Express® sea integrado.

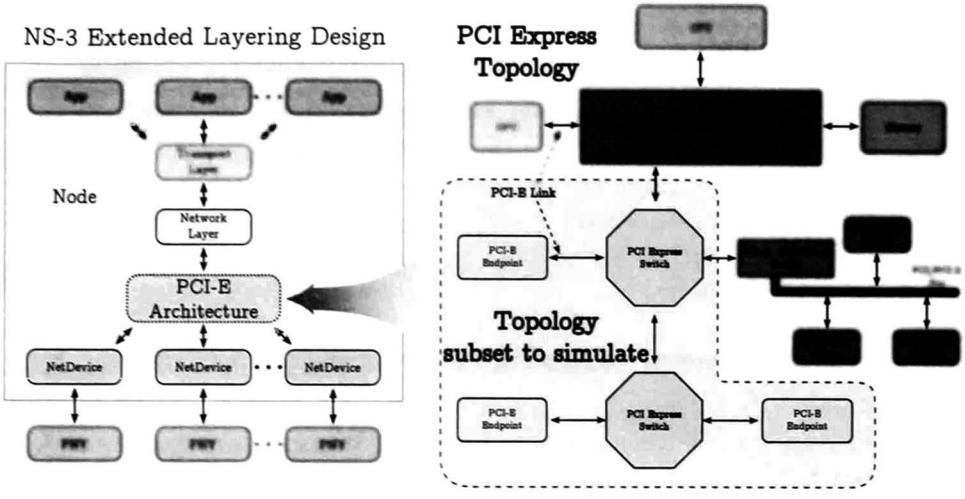

Figura 4.1: Integración de PCI Express® en NS-3.

4.3.1. Componentes del simulador

Vamos a definir dos clases de componentes desde el punto de vista funcional: componentes pasivos y componentes activos.

Componentes pasivos. Son aquellos componentes reactivos que tienen funciones muy simples y secundarias. Normalmente, bajo la misma entrada realizan la misma serie de pasos y devuelven la misma salida. Por lo regular, la vida de estos componentes es temporal y corta en el tiempo de simulación. Ejemplos de estos componentes son estructuras de datos, contenedores y componentes auxiliares.

Componentes activos. Son componentes que realizan funciones complejas. Bajo una misma entrada, pueden no devolver la misma salida y tomar diferentes acciones que de cierta forma los podría definir como pro-activos. Estos componentes son los principales actores de la simulación y su comportamiento y configuración define los

resultados de la simulación. Estos componentes son los encargados de brindar información de la simulación al usuario (requerimiento 5.1). Ejemplos de estos componentes son los dispositivos PCI Express® como Endpoints y Conmutadores.

4.3.1.1. Componentes pasivos

A continuación se describen los principales componentes pasivos del simulador.

TLP y DLLP. Estos componentes representan a los TLPs y DLLPs respectivamente. Su función es la de serializar y deserializar en el formato correcto los datos correspondientes a los TLPs y DLLPs.

VC Map. En este componente se implementa el mapeo entre TCs y VCs.

VC Arbitrator. Es el encargado de realizar el arbitraje de VCs.

VC Resources. Representa y contiene los recursos de un VC como: buffers, FCs.

PCI-E Link. Representa un enlace PCI Express®

4.3.1.2. Componentes activos

A continuación se describen los componentes activos del simulador.

PCI-E Port. Representa un puerto PCI Express® Contiene un VC Map, un VC Arbitrator y un conjunto de VC Resources — uno por cada VC soportado —. Esta conectado a un PCI-E Link.

PCI-E Switch. Representa un conmutador PCI Express®. Contiene un conjunto de PCI-E Ports.

PCI-E Endpoint. Representa un Endpoint PCI Express®. Contiene un PCI-E Port.

En la Figura 4.2 se muestra un diagrama de clases entre los distintos componentes.

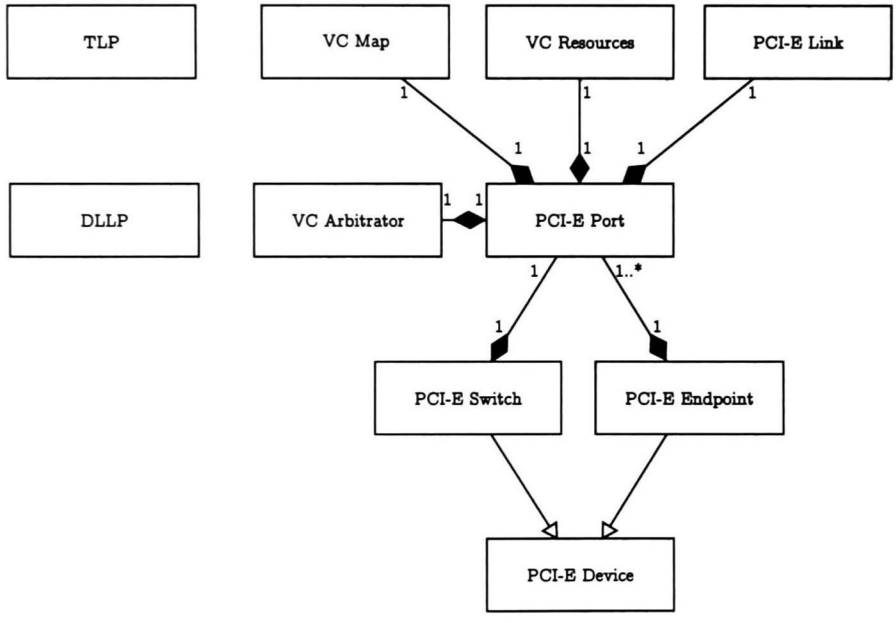

Figura 4.2: Diagrama de clases de componentes del simulador PCI Express®.

4.3.2. Integración con NS-3

La integración del módulo de PCI Express® se debe hacer lo más transparente posible. Como ya se mostró en la Figura 4.1, el lugar adecuado

en el diseño por capas de NS-3 es entre la capa de enlace de datos y la capa de red. Una vista aumentada del módulo de PCI Express® integrado en NS-3 en la Figura 4.1 se puede observar en la Figura 4.3.

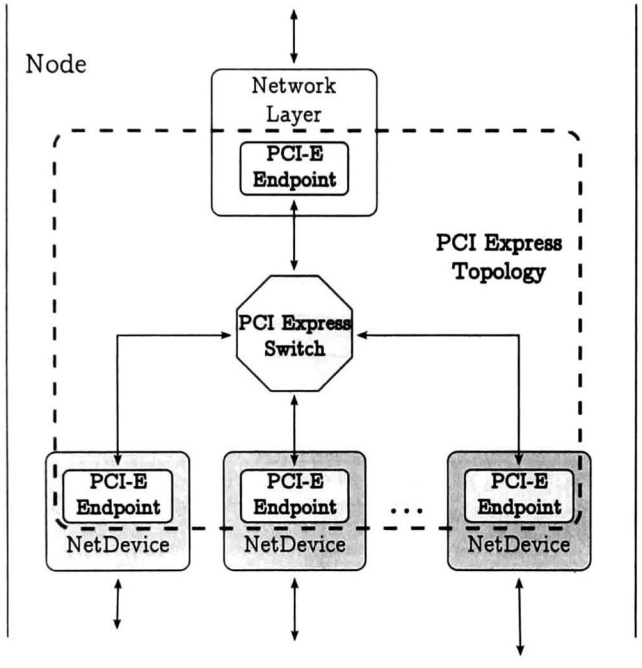

Figura 4.3: Integración de PCI Express® en NS-3 a detalle.

Para lograr esto, es necesario implementar dispositivos Endpoint con un comportamiento especializado para integrarse en los dispositivos de red de NS-3. Esto se puede observar en la Figura 4.4.

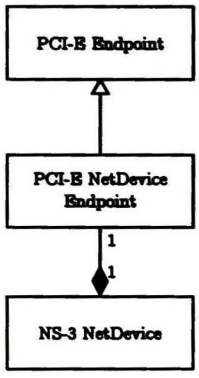

Figura 4.4: Diagrama de clases del "PCI-E NetDevice Endpoint'

Capítulo 5

Análisis y resultados

En esta sección se van a presentar los resultados de la propuesta presentada en el Capítulo 4, así como los resultados del caso de estudio que se describe en la Sección 5.2.

5.1. Implementación de la propuesta

Se logró una implementación funcional del simulador para PCI Express® siguiendo el diseño propuesto en el Capítulo 4.

5.2. Caso de estudio

Para probar el funcionamiento del simulador, se tomará como caso de estudio la transmisión de datos en la topología de red que se observa en la Figura 5.1.

5.2.1. Descripción del caso de estudio

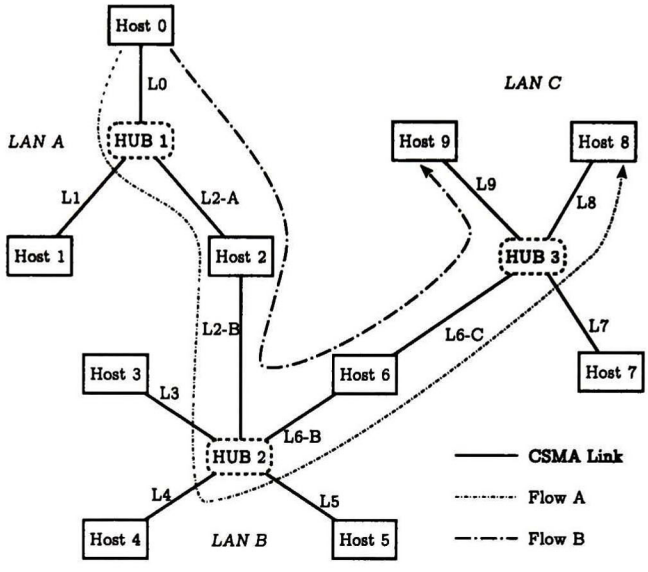

Figura 5.1: Escenario para el caso de estudio.

El escenario consta de una topología de red formada por tres subredes CSMA (LAN A, LAN B y LAN C) interconectadas por nodos comunes en ellas que cumplen la función de encaminadores (Host 2 y Host 6). Se envían dos flujos de datos (Flujo A y Flujo B) desde un nodo de la LAN A (Host 0) hasta dos nodos de la LAN C (Host 8 y Host 9). Estos flujos deben pasar por la LAN B para poder llegar a sus respectivos destinos en la LAN C.

Desde el punto de vista de PCI Express®, los paquetes pasarán por la topología de PCI Express® antes de poder llegar a la red CSMA. En el caso del origen de los flujos, los paquetes saldrán de la pila TCP/IP y serán enviados por la topología de PCI Express® hacia el dispositivo de red. Una vez que los paquetes lleguen al dispositivo de red, este se encargará de enviarlos normalmente por el canal CSMA. Para el caso de los encaminadores (Hosts 3 y 6), los paquetes llegarán a un dispositivo de red, este enviará

los paquetes por la topología de PCI Express® hacia la pila TCP/IP. La pila TCP/IP, puntualmente el protocolo IP, es el encargado de realizar el encaminamiento, por lo que una vez que se haya decidido por que dispositivo deben ser reenviados los paquetes, el protocolo IP enviará los paquetes hacia el dispositivo de red adecuado a través de la topología PCI Express®. Para el caso de los destinos de los flujos, ocurrirá lo mismo que en los encaminadores a la hora de recibir paquetes por un dispositivo de red, sin embargo, dado que ese host es el destinatario de los paquetes, estos no serán encaminados, y en su lugar, serán consumidos por capas superiores.
En la Figura 5.2 se puede observar el camino que recorrerán los paquetes IP dentro de la topología PCI Express®. En la figura solo se representa un flujo, esto es debido a que tanto los paquetes del flujo A como los paquetes del flujo B recorrerán el mismo camino dentro de la topología PCI Express®.

En la Tabla 5.1 se describen las propiedades de los flujos. El significado de los campos en la cabecera de la tabla son los siguientes:

TC. Clase de tráfico a la que está asociado el flujo.

PZ. Tamaño del paquete en bytes ("Packet Size").

PIDT. "Packets Inter-Departure Time" (segundos).

Num. Número de paquetes que se van a transmitir.

Tabla 5.1: Propiedades de los flujos de datos en el caso de estudio.

Flujo	TC	PZ (bytes)	PIDT (s)	Num
Flujo A	TC0	1024	0.001	1000
Flujo B	TC1	1024	0.001	1000

En la Tabla 5.2 se pueden observar las propiedades de los enlaces CSMA.

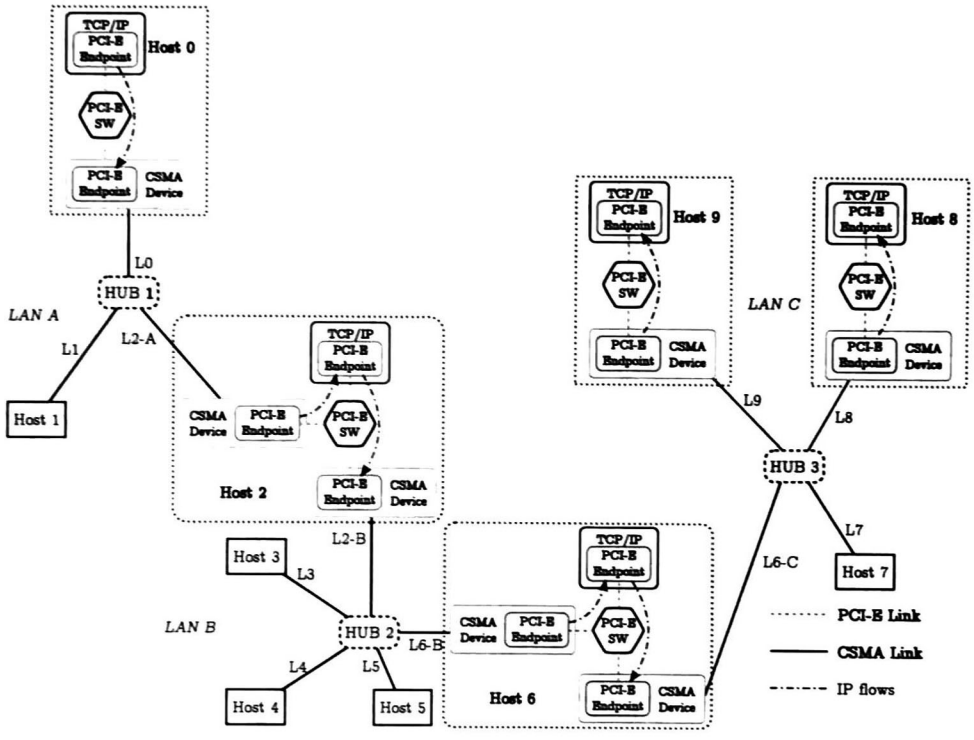

Figura 5.2: Topología PCI Express® en el caso de estudio.

Tabla 5.2: Propiedades de enlaces CSMA en el caso de estudio.

Enlace	Bandwidth (Mbps)
L{*}	10

Para llevar a cabo las pruebas, se van a realizar diferentes configuraciones de mapeos de TCs y VCs en los dispositivos PCI Express® Como ya se observó en la Tabla 5.1, solo se van a utilizar las clases de tráfico TC0 y TC1.

En la Tabla 5.3 se muestran las configuraciones que se van a realizar en las pruebas. El significado de las cabeceras es el siguiente:

Map. Mapeo de TCs a VCs.

FC. "Flow Control credits".

VC Arb. Algoritmo de arbitraje de VCs usado (ver Sección 2.2.1.5)

Tabla 5.3: Configuración de mapeos PCI Express® en las pruebas del caso de estudio.

Prueba	Map.	FC	VC Arb.
Prueba 1	TC0:VC0,TC1:VC0	1024	SP
Prueba 2	TC0:VC0,TC1:VC1	1024	SP
Prueba 3	TC0:VC0,TC1:VC0	4095	SP
Prueba 4	TC0:VC0,TC1:VC1	4095	SP

En la Tabla 5.3, el formato del campo "Map." es de la forma "TCx:VCy,TCw:VCz,...", que significa: la TC x está mapeada al VC y, la TC w está mapeada al VC z, ... El campo "FC" representa la cantidad de

créditos que los puertos PCI Express® van a manejar para cada uno de los VCs (ver Sección 2.2.1.4).

Como se puede observar, se van a realizar cuatro pruebas. Las pruebas 1 y 2 difieren en el mapeo que se hace de las TCs y los VCs. En la prueba 1, tanto la TC 0 como la TC 1 están mapeadas al VC 0, mientras que en la prueba 2, la TC 1 está mapeada al VC 1. De está manera, es posible realizar DiffServ sobre los flujos de tráfico. En ambas pruebas, la cantidad de créditos de control de flujo que se va a utilizar para los VCs es de 1024 créditos. Las pruebas 3 y 4, son idénticas a las pruebas 1 y 2 respectivamente, con la excepción de que en las pruebas 3 y 4, se van a utilizar 4095 créditos de control de flujo para los VCs. Al variar la cantidad de créditos, se traduce directamente a una variación en los tamaños de los buffers, que a su vez afecta al encolamiento.

En las cuatro pruebas, el algoritmo de arbitraje de VCs será "Strict Priority" (SP, véase Sección 2.2.1.5).

5.2.2. Justificación del caso de estudio

De acuerdo a la Tabla 5.1, cada uno de los flujos genera 8192 Kbps (8.192 Mbps). Si sumamos el tráfico que generan ambos flujos, tenemos un tráfico total de 16.384 Mbps. Como se muestra en la Tabla 5.2, los enlaces CSMA soportan un máximo de 10 Mbps; por lo tanto, los flujos generan 6 Mbps de tráfico más del soportado por los enlaces CSMA. Esto convierte a los enlaces CSMA en el cuello de botella de la transmisión de datos.
El objetivo por el cual se establecieron esas configuraciones en los enlaces CSMA y los flujos de tráfico, es para congestionar la red y poder observar características como retardo (de encolamiento) y pérdida de paquetes.

El objetivo de las configuraciones a la topología de PCI Express® en las

diferentes pruebas a realizar descritas en la Tabla 5.3 tiene como objetivo demostrar el funcionamiento de los algoritmos de DiffServ y control de flujo de PCI Express® que se implementaron en el simulador.

En la prueba 1, al estar ambos flujos mapeados al mismo VC, no se le da trato preferencial a ninguno de los flujos. En la prueba 2 el flujo B es mapeado al VC 1, por lo tanto, tendrá mayor prioridad debido a que se está usando el algoritmo de "Strict Priority" para el arbitraje de VCs. De esta manera será posible observar las capacidades de DiffServ de PCI Express® y validar la implementación que se hizo de esos algoritmos.

Las pruebas 3 y 4 tienen como objetivo observar el impacto que tiene en la arquitectura PCI Express® el aumento en el número de créditos de control de flujo, así como la validación de la implementación de control de flujo que se realizó.

Por último, la arquitectura PCI Express® fue integrada en NS-3 tal como se observa en la Figura 4.3 y el flujo de datos dentro de la topología de PCI Express® se puede observar en la Figura 5.2, por lo que el caso de estudio en si nos servirá para validar la integración que se hizo de PCI Express® y NS-3.

5.2.3. Resultados del caso de estudio

El análisis de los resultados del caso de estudio se va a realizar tomando en cuenta los siguientes criterios en la transmisión de flujos de datos:

Retardo (Delay). Está métrica representa el retardo de fin a fin que experimentaron los paquetes. El retardo del paquete i está dado por la ecuación

$$D(i) = t_{rx}(i) - t_{tx}(i) \tag{5.1}$$

donde $t_{tx}(i)$ representa el tiempo en que fue enviado el paquete i en la aplicación origen y $t_{rx}(i)$ representa el tiempo en que fue recibido el mismo paquete i por la aplicación destino.

Pérdida de paquetes. Representa los paquetes perdidos. Se analizará de dos maneras: suma total de paquetes perdidos y tamaño de las ráfagas de paquetes perdidos durante la transmisión.

Jitter. El jitter, también conocido como "IP Packet Delay Variation Metric for IP Performance Metrics"[11], está dado por la ecuación

$$J(i) = D(i) - D(i-1) \tag{5.2}$$

El jitter representa la variación entre el tiempo de inter-partida ("Packet Inter-Departure Time (PIDT)") y el tiempo de inter-arrivo ("Packet Inter-Arrival Time (PIAT)") de fin a fin entre dos paquetes consecutivos ($i-1$ e i). Un jitter diferente de cero representa un encolamiento sufrido por uno (o ambos) de los paquetes a la largo del recorrido de fin a fin.
Algunas aplicaciones requieren que el jitter sea los más cercano a cero posible.

5.2.3.1. Prueba 1

En la Figura 5.3 se puede observar el histograma de las frecuencias de retardo que se presento en la prueba 1. El eje de las x's representa el retardo en milisegundos; el eje de las y's representa la cantidad de paquetes que sufrieron cierto retardo. El histograma se presenta con un "bin" de 10 ms.

Como se puede observar en la figura, el retardo fue prácticamente el mismo para ambos flujos (A y B) en la prueba 1. Esto tiene sentido, ya que

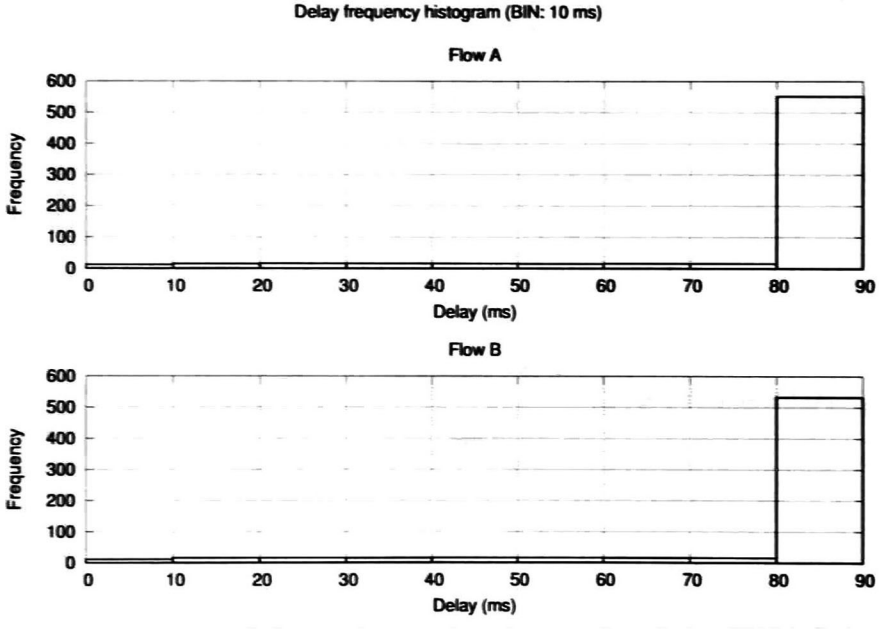

Figura 5.3: Retardo de la prueba 1

no se le da prioridad a ninguno de los flujos, esto es, en la arquitectura PCI Express®, ambos están "mapeados" al mismo VC.

En la Figura 5.4 se presentan los paquetes perdidos. Como se puede observar, cuando hubo pérdida, en mayor medida se presentó en ráfagas de dos paquetes, esto es, se perdían dos paquetes consecutivos. Además, la cantidad de paquetes perdidos fue muy cercana en ambos flujos. El hecho de que el flujo A pierda 20 paquetes más que el flujo B se debe únicamente al orden en que se comienzan a transmitir los flujos.

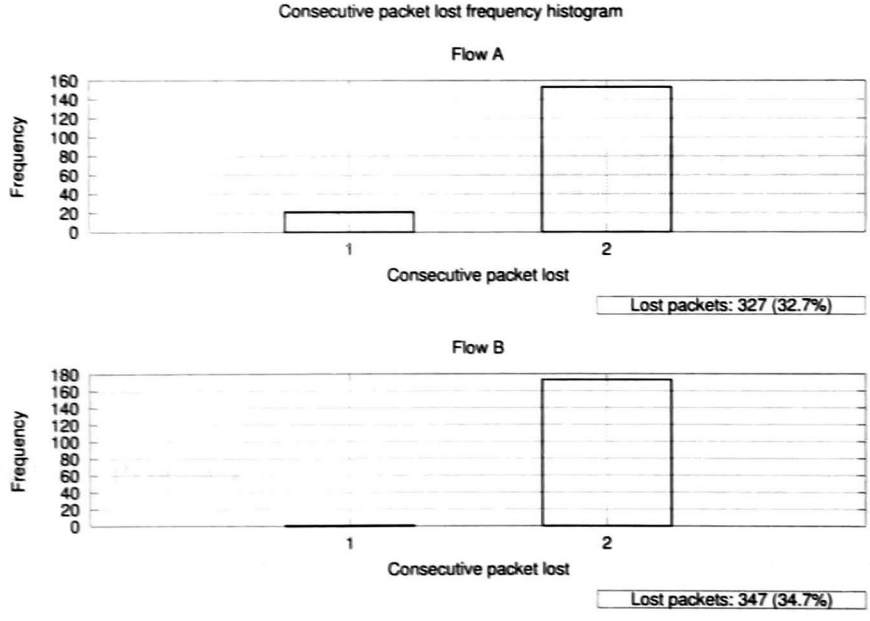

Figura 5.4: Pérdida de paquetes de la prueba 1

En la Figura 5.5 se puede observar el jitter que experimentaron ambos flujos. Al igual que en el caso del retardo, el jitter es muy parecido en ambos flujos. Las pequeñas diferencias están dadas por el orden en que son transmitidos los paquetes de cada flujo (esto es, quien es el primero que

comienza la trasmisión).

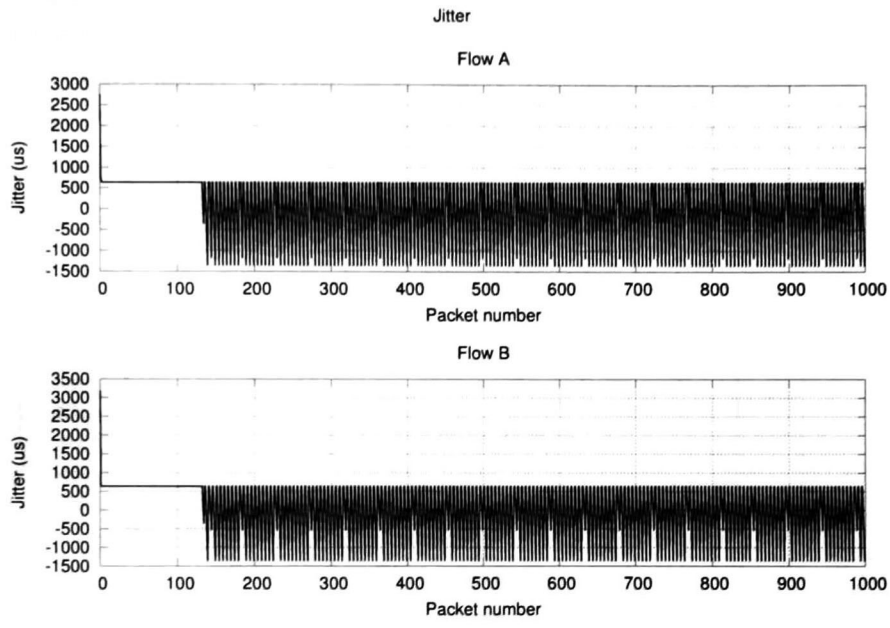

Figura 5.5: Jitter de la prueba 1

5.2.3.2. Prueba 2

Para la prueba dos, como se muestra en la Tabla 5.3, el flujo B fue mapeado al VC 1. Al haber usado el algoritmo "Strict Priority" para el arbitraje de VCs, el flujo B debería tener mayor prioridad en la arquitectura PCI Express®.

En la Figura 5.6 se puede observar que el flujo A, que tiene menor prioridad, sufre mayor retardo que el flujo B. Esto es debido a que, como se explicó en la Sección 5.2.2, el canal CSMA sufre una congestión, que a su vez produce un encolamiento en los dispositivos de red CSMA, que a su

vez produce encolamiento en los "Enpoints" de PCI Express® y en todos los puertos PCI Express® por los que fluyen los flujos A y B. Cuando hay espacio en las colas de los dispositivos de red CSMA (producido al enviar paquetes por el canal CSMA), se reanuda el flujo de paquetes en la arquitectura PCI Express® y a causa del algoritmo de "Strict Priority", van a fluir primero todos los paquetes del flujo B que haya encolados antes de que algún paquete del flujo A pueda ser enviado.

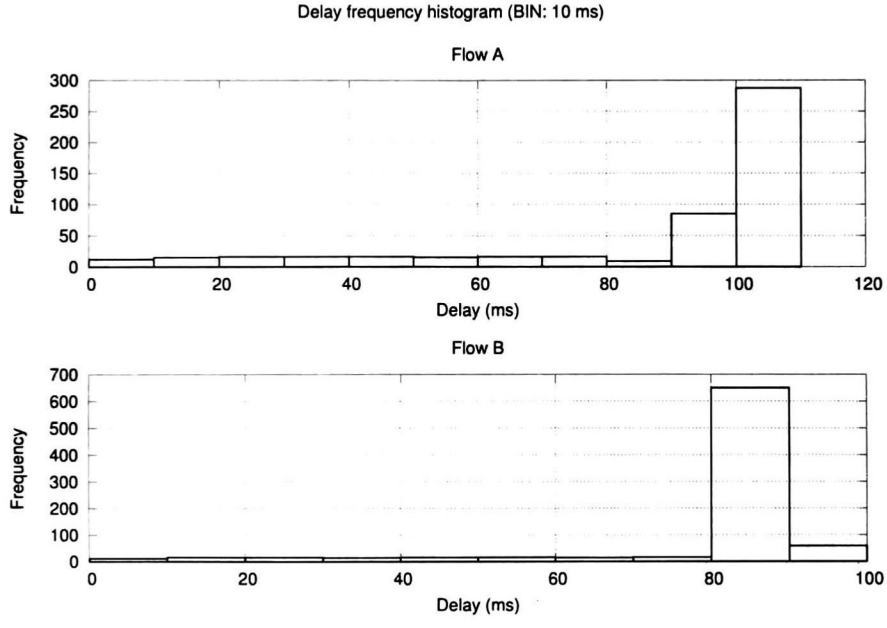

Figura 5.6: Retardo de la prueba 2

En la Figura 5.7 se puede observar menor pérdida de paquetes en el flujo B con respecto a la prueba 1. Para el caso del flujo A, aumenta la cantidad de paquetes perdidos, y además, la mayoría de paquetes que se perdieron fueron en ráfagas de 3.

Para el caso del jitter, en la Figura 5.8 se puede observar una disminución

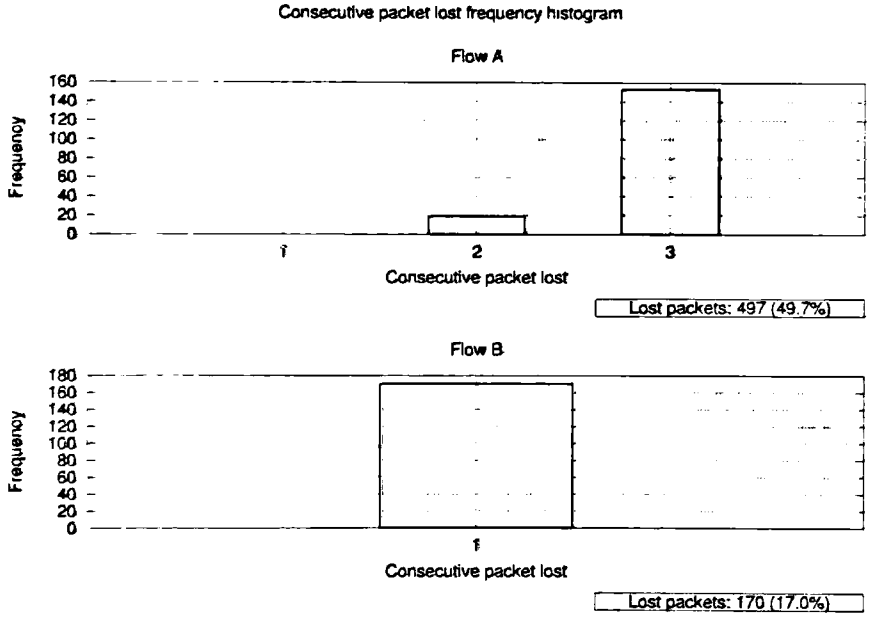

Figura 5.7: Pérdida de paquetes de la prueba 2

considerable del jitter para el flujo B con respecto a la prueba 1, y un aumento considerable del jitter del flujo A. Esto se explica por la congestión y encolamiento que se produce explicado previamente.

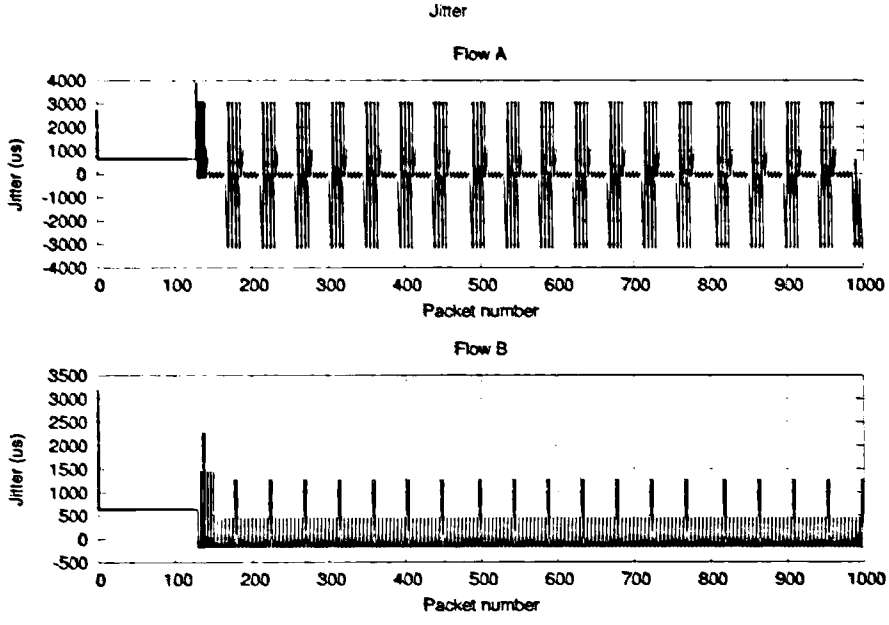

Figura 5.8: Jitter de la prueba 2

5.2.3.3. Prueba 3

La prueba 3 es idéntica a la prueba 1, con la excepción de que se usan cuatro veces la cantidad de créditos de control de flujo de los usados en la prueba 1. Esto es, se usan 4095 créditos PCI Express® (el máximo establecido por la especificación de PCI Express®) en lugar de 1024.

Se puede observar que con el aumento de los tamaños de buffer se disminuye un poco la pérdida de paquetes, sin embargo, se aumenta el retardo (Figura 5.10 y Figura 5.9).

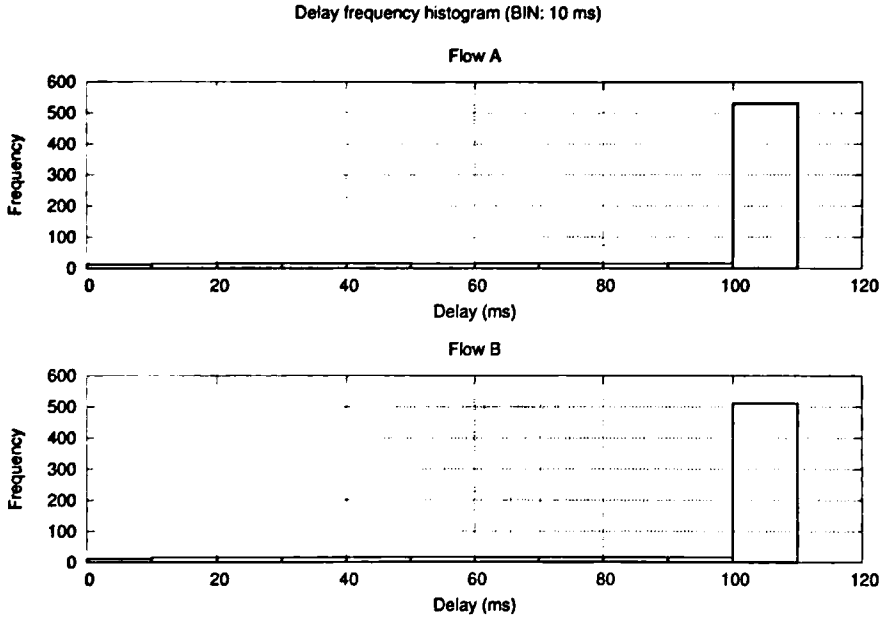

Figura 5.9: Retardo de la prueba 3

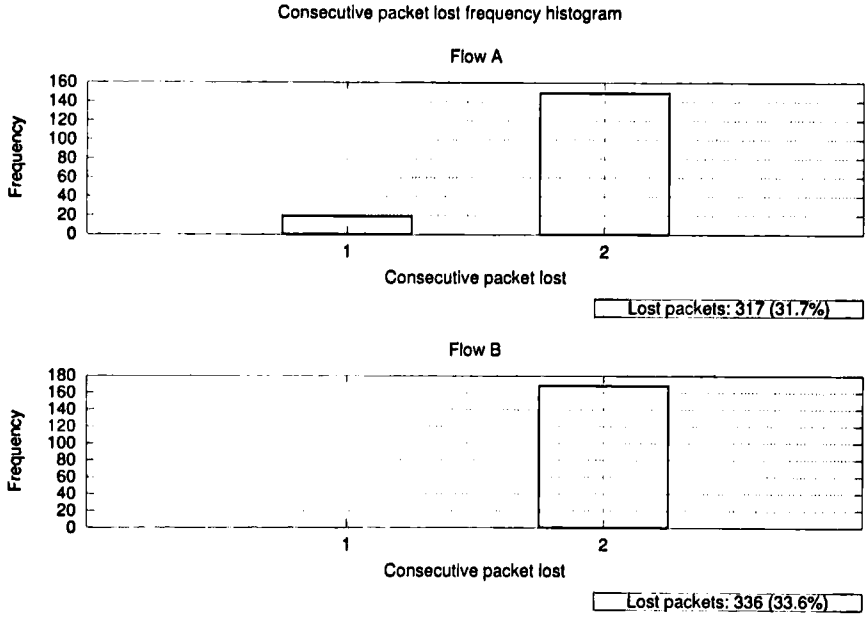

Figura 5.10: Pérdida de paquetes de la prueba 3

El jitter es prácticamente el mismo con respecto a la prueba 1 (Figura 5.11).

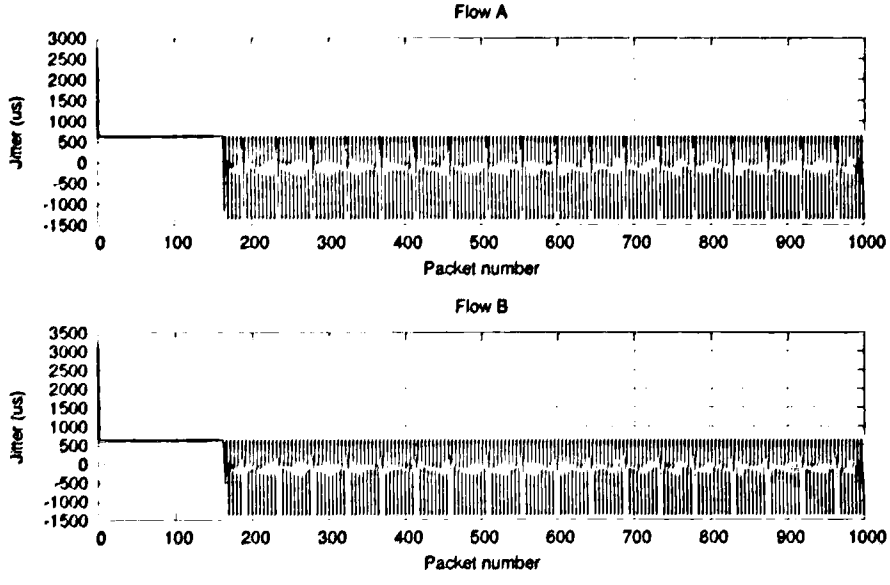

Figura 5.11: Jitter de la prueba 3

5.2.3.4. Prueba 4

La prueba 4 tiene la misma configuración que la prueba 2, sin embargo, al igual que la en prueba 3, se usan 4095 créditos de control de flujo de PCI Express®

En la Figura 5.12 se muestra el retardo de paquetes de la prueba 3. Se puede observar que existe un aumento considerable en el retardo sufrido por los paquetes del flujo A con respecto tanto a la prueba 2 como a la prueba 3. Por otra parte, el flujo B presentó una mejora en el retardo.

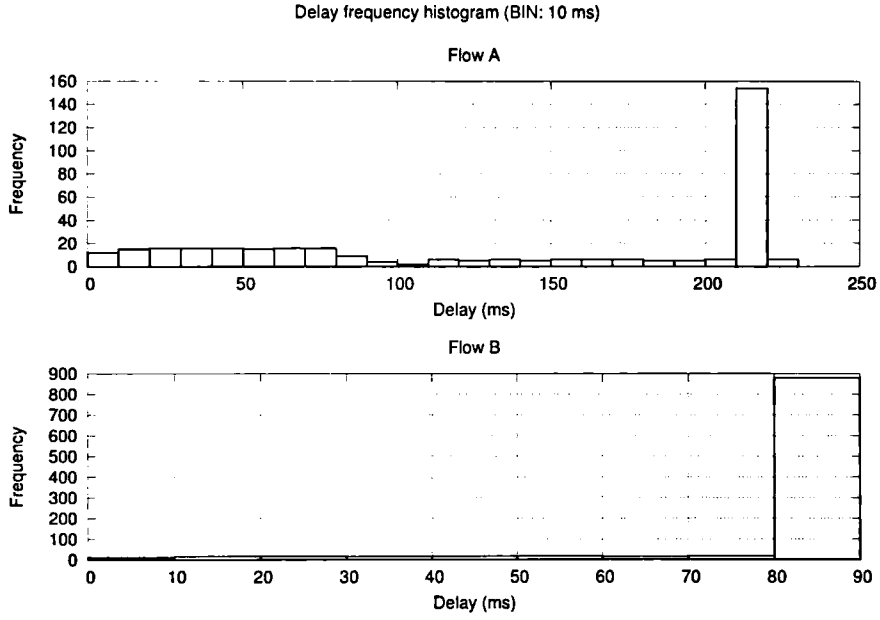

Figura 5.12: Retardo de la prueba 4

En el caso de la pérdida de paquetes mostrada en la Figura 5.13, se puede observar que el flujo B no perdió ningún paquete, mientras que el flujo A presento el peor caso de pérdida de paquetes de las cuatro pruebas. Esto se atribuye al mayor tamaño de buffer (créditos) usado. La diferencia en la pérdida de paquetes entre ambos flujos tiene la misma explicación presentada para la prueba 2, sin embargo, el incremento del tamaño de buffer representa una menor probabilidad de pérdida de paquetes para el flujo B (de aquí que no haya perdido paquetes) pero una espera mayor para la transmisión de paquetes del flujo A debido a que toma más tiempo vaciar el buffer del flujo B para poder enviar paquetes del flujo A.

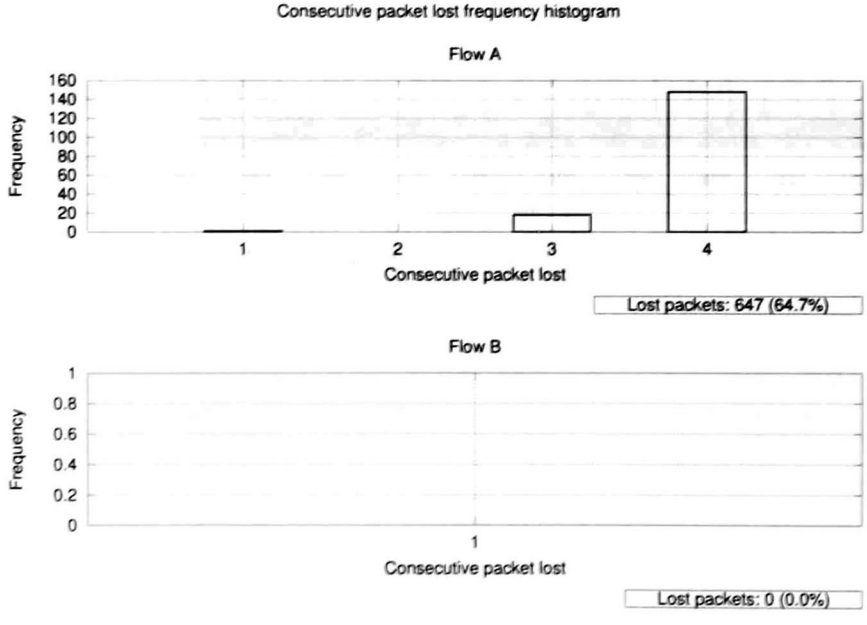

Figura 5.13: Pérdida de paquetes de la prueba 4

En concordancia con el retardo y pérdida de paquetes del flujo A, el jitter de dicho flujo fue también el peor de todas las pruebas. Por otra parte, el jitter del flujo B mejoro con respecto a la prueba 3 y es prácticamente el

mismo que en la prueba 2. Esto se muestra en la Figura 5.14.

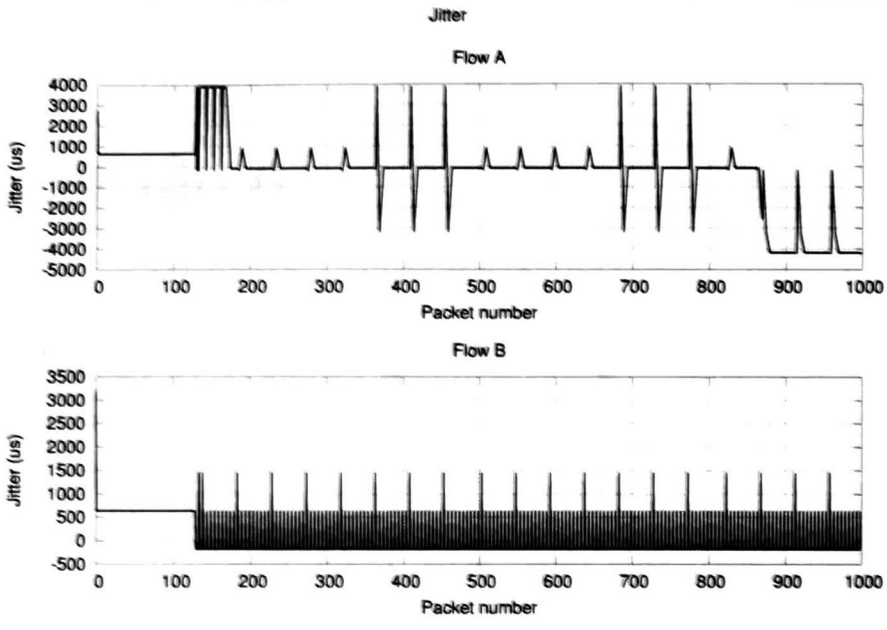

Figura 5.14: Jitter de la prueba 4

Capítulo 6

Conclusiones y trabajo futuro

De acuerdo al planteamiento del problema, el objetivo de esta tesis fue el diseño de un simulador de PCI Express® y su respectiva implementación e integración en el simulador de redes NS-3.

Se realizó el diseño del simulador y se efectuó la correcta implementación e integración en el simulador de redes NS-3.

Para poder comprobar el correcto funcionamiento de la implementación del simulador se uso un caso de estudio que consistió en la transmisión de dos flujos de datos a través de una red CSMA. Para poner a prueba las capacidades del simulador de PCI Express®, los nodos de las red CSMA se configuraron para que internamente integraran la arquitectura PCI Express®. Los resultados del caso de estudio demostraron que la implementación del simulador de PCI Express® se comporto como se esperaba. Además, a partir de los resultados del caso de estudio se puede concluir que diferentes configuraciones en una topología de PCI Express® pueden cambiar de manera significativa el comportamiento de redes de comunicaciones heterogéneas y convencionales, abriendo un área de investigación en la cual el simulador

que dio como resultado esta tesis puede ser usado como herramienta de investigación.

6.1. Trabajo futuro

Se plantean varios puntos como trabajo futuro directamente relacionados con la implementación del simulador de PCI Express®:

- Implementación de diferentes algoritmos de arbitraje de puertos y arbitraje de canales virtuales en el conmutador y puerto de PCI Express® respectivamente.
- Implementación de las reglas de ordenamiento en la transmisión de TLPs.
- Implementar la separación del envío por separado de la cabecera y el cuerpo de los TLPs.
- Implementación de todos los tipos de TLPs y DLLPs.
- Probar el simulador con más escenarios y más variados para comprobar toda la implementación del simulador.
- Validación de la compatibilidad a nivel de bit de los TLPs y DLLPs.
- Refactorización del simulador NS-3 para hacer una integración del módulo de PCI Express® transparente al usuario final de NS-3.
- Integrar el módulo de PCI Express® de manera oficial al simulador NS-3.

Apéndice A

Código fuente de escenario del caso de estudio

A continuación se lista el código fuente del escenario del caso de estudio.

pcie-tesis-scenario.cc

```
1  /* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
2  /*
3      Copyright (c) José Francisco Lombera Landa
4
5    * Author: José Francisco Lombera Landa <jlombera@gdl.cinvestav.mx>
6  */
7
8  #include "ns3/core-module.h"
9  #include "ns3/network-module.h"
10 #include "ns3/csma-module.h"
11 #include "ns3/internet-module.h"
12 #include "ns3/applications-module.h"
13 #include "ns3/tools-module.h"
14 #include "ns3/pcie-module.h"
15
16 #include "multi-bound-callback-impl.h"
17
18 NS_LOG_COMPONENT_DEFINE ("PcieTesisScenario");
19
20 using namespace ns3;
21
22 /**
23     \brief Calculate the UDP payload's size to generate PCI-E TLPs of given size
24     \param size the size of the PCI-E TLP
25     \return the UDP payload's size
26  */
```

```
27  uint16_t
28  calculatePktSize (uint16_t size)
29  {
30    // overhead: UDP + IP + PCI-E NetAddr + PCI-E TLP headers
31    uint16_t overhead = 8 + 20 + 15 + 16;
32
33    return (size    overhead);
34  }
35
36  static void
37  TraceRx (Ptr<OutputStreamWrapper> stream, Ptr<UdpServer> srv,
38           Ptr<const Packet> p)
39  {
40    *stream->GetStream () << Simulator::Now ().GetSeconds () <<
41                          << srv->GetReceived () << std::endl;
42  }
43
44  static void
45  TraceLost (Ptr<OutputStreamWrapper> stream, DelayJitterEstimation *jitt
46             Ptr<const Packet> p)
47  {
48    jitt->RecordRx (p);
49
50    if (jitt->GetNumLost () > 0)
51      {
52        *stream->GetStream () << Simulator::Now ().GetSeconds () << "
53                              << jitt->GetLastSeq () << "
54                              << jitt->GetNumLost () << std::endl;
55      }
56  }
57
58  static void
59  TraceDelay (Ptr<OutputStreamWrapper> stream, Ptr<const Packet> p)
60  {
61    SeqTsHeader seqTs;
62    Ptr<Packet> pkt = p->Copy ();
63
64    pkt->RemoveHeader (seqTs);
65    Time delay = Simulator::Now () - seqTs.GetTs ();
66    *stream->GetStream () << seqTs.GetSeq () <<    " << delay.GetSeconds ()
67                          << std::endl;
68  }
69
70  bool gBreakJitter = false;
71
72  static void
73  TraceJitter (Ptr<OutputStreamWrapper> stream, DelayJitterEstimation *jitt,
74               Ptr<const Packet> p)
75  {
76    SeqTsHeader seqTs;
77    Ptr<Packet> pkt = p->Copy ();
78
79    pkt->RemoveHeader (seqTs);
80    jitt->RecordRx (p);
81
82    uint32_t lost = jitt->GetNumLost ();
83    uint64_t lastJitter = jitt->GetLastJitter ();
84
85    if ((lost > 0) && gBreakJitter)
86      {
87        *stream->GetStream () << std::endl; // a blank line represents packet lost
88      }
```

```
89      else
90        {
91          *stream->GetStream () << Simulator::Now ().GetSeconds () << "
92                                << TimeStep (lastJitter).GetSeconds () << "
93                                << seqTs.GetSeq () << std::endl;
94        }
95    }
96
97    static DelayJitterEstimation jitter[2];
98    static DelayJitterEstimation lost[2];
99
100   static void
101   TraceServers (ApplicationContainer &srvs)
102   {
103     AsciiTraceHelper ascii;
104
105     for (uint32_t i = 0; i < srvs.GetN (); ++i)
106       {
107         char filename[30];
108         Ptr<UdpServer> srv = srvs.Get (i)->GetObject<UdpServer> ();
109
110         snprintf (filename, sizeof(filename), "strmRx%d.txt", i);
111         Ptr<OutputStreamWrapper> stream = ascii.CreateFileStream (filename);
112         srv->TraceConnectWithoutContext ("Rx", MakeMultiBoundCallback (&TraceRx, stream, srv));
113
114         snprintf (filename, sizeof(filename), "strmLost%d.txt", i);
115         stream = ascii.CreateFileStream (filename);
116         srv->TraceConnectWithoutContext ("Rx", MakeMultiBoundCallback (&TraceLost, stream, &lost[i]));
117
118         snprintf (filename, sizeof(filename), "strmDelay%d.txt", i);
119         stream = ascii.CreateFileStream (filename);
120         srv->TraceConnectWithoutContext ("Rx", MakeBoundCallback (&TraceDelay, stream));
121
122         snprintf (filename, sizeof(filename), "strmJitter%d.txt", i);
123         stream = ascii.CreateFileStream (filename);
124         srv->TraceConnectWithoutContext ("Rx", MakeMultiBoundCallback (&TraceJitter  stream,
125                                                                        &jitter[i]));
126       }
127   }
128
129   int
130   main (int argc, char *argv[])
131   {
132     unsigned int numNodesLanA = 3;
133     unsigned int numNodesLanB = 4;
134     unsigned int numNodesLanC = 3;
135
136     // PCI-E attributes
137     unsigned int tcStream0 = 0;
138     unsigned int tcStream1 = 1;
139     unsigned int vcStream0 = 0;
140     unsigned int vcStream1 = 0;
141     unsigned int vcCredits = 1024;
142
143     // CSMA channel attributes
144     uint64_t csmaBps = 10000000; // Data rate: 10 Mbps
145     uint64_t csmaDelay = 0;      // 0 ms
146
147     // UDP applications attributes
148     uint16_t port = 100;
149     uint16_t pktSize = 100;
150     uint16_t maxPackets = 1;
```

```
151     double interval = 100.0;
152
153     double duration = 20.0;
154
155     CommandLine cmd;
156
157     cmd.AddValue ("nodesLanA", "Number of nodes in LAN A", numNodesLanA);
158     cmd.AddValue ("nodesLanB", "Number of nodes in LAN B", numNodesLanB);
159     cmd.AddValue ("nodesLanC", "Number of nodes in LAN C", numNodesLanC);
160     cmd.AddValue ("csmaDataRate", "Data rate of CSMA channels (bps)", csmaBps);
161     cmd.AddValue ("csmaDelay", "Transmission time of CSMA channels (ms)", csmaDelay);
162     cmd.AddValue ("port", "UDP port through which to send packets", port);
163     cmd.AddValue ("pktSize", "Packet size", pktSize);
164     cmd.AddValue ("numPackets", "Number of packets to send", maxPackets);
165     cmd.AddValue ("interval", "Packet Inter-Departure Time (ms)", interval);
166     cmd.AddValue ("duration", "Duration of simulation (sec)", duration);
167     cmd.AddValue ("tcStream0", "PCI-E TC to which stream 0 is mapped ([0-7])", tcStream0);
168     cmd.AddValue ("tcStream1", "PCI-E TC to which stream 1 is mapped ([0-7])", tcStream1);
169     cmd.AddValue ("vcStream0", "PCI-E VC to which 'tcStream0' is mapped ([0-7])", vcStream0)
170     cmd.AddValue ("vcStream1", "PCI-E VC to which 'tcStream1' is mapped ([0-7])", vcStream1)
171     cmd.AddValue ("vcCredits", "Amount of credits per PCI-E VC (1024-4095)", vcCredits);
172     cmd.AddValue ("breakJitter", "Break jitter plot if there is paket lost?", gBreakJitter);
173     cmd.Parse (argc, argv);
174
175     pktSize = calculatePktSize (pktSize);
176
177     NS_LOG_INFO ("Create nodes.");
178
179     NodeContainer nodes;
180     NodeContainer nodesLanA;
181     NodeContainer nodesLanB;
182     NodeContainer nodesLanC;
183     NodeContainer nodesSrv;
184     Ptr<Node> cltNode;
185     NodeContainer gwNodes;
186
187     nodesLanA.Create (numNodesLanA);
188     nodes.Add (nodesLanA);
189
190     nodesLanB.Create (numNodesLanB);
191     nodes.Add (nodesLanB);
192
193     nodesLanC.Create (numNodesLanC);
194     nodes.Add (nodesLanC);
195
196     gwNodes.Add (nodesLanA.Get (numNodesLanA - 1));
197     gwNodes.Add (nodesLanB.Get (numNodesLanB - 1));
198     nodesLanB.Add (gwNodes.Get (0));
199     nodesLanC.Add (gwNodes.Get (1));
200
201     // Server and client nodes
202     cltNode = nodesLanA.Get (0);
203     nodesSrv.Add (nodesLanC.Get (numNodesLanC - 2));
204     nodesSrv.Add (nodesLanC.Get (numNodesLanC - 1));
205
206     NS_LOG_INFO ("Install CSMA devices.");
207
208     CsmaHelper csma;
209     csma.SetChannelAttribute ("DataRate", DataRateValue (DataRate (csmaBps)));
210     csma.SetChannelAttribute ("Delay", TimeValue (MilliSeconds (csmaDelay)));
211
212     NetDeviceContainer devsLanA = csma.Install (nodesLanA);
```

```
213     NetDeviceContainer devsLanB = csma.Install (nodesLanB);
214     NetDeviceContainer devsLanC = csma.Install (nodesLanC);
215
216     NS_LOG_INFO ("Install PCI-E Architecture.");
217
218     // Configure TC to VC mapping
219     PcieVcMap vcMap;
220     vcMap.MapTcToVc (tcStream0, vcStream0);
221     vcMap.MapTcToVc (tcStream1, vcStream1);
222     Config::SetDefault ("ns3::PciePort::VcMap", PcieVcMapValue (vcMap));
223
224     // Configure VC credits
225     Config::SetDefault ("ns3::PciePort::FlwCtrlCredits", UintegerValue (vcCredits));
226
227     PcieTopologyHelper topoHelper;
228     topoHelper.Install (nodes);
229
230     // Attach CSMA devices to PCI-E architecture
231     PcieNetDeviceEndpointHelper epHelper;
232
233     epHelper.InstallNetDevs (devsLanA);
234     epHelper.InstallNetDevs (devsLanB);
235     epHelper.InstallNetDevs (devsLanC);
236
237     NS_LOG_INFO ("Install Internet stack.");
238
239     InternetStackHelper internet;
240     internet.Install (nodes);
241
242     NS_LOG_INFO ("Assign IP addresses.");
243
244     Ipv4AddressHelper addrHelper;
245
246     addrHelper.SetBase ("10.1.1.0", "255.255.255.0");
247     Ipv4InterfaceContainer ifacesA = addrHelper.Assign (devsLanA);
248
249     addrHelper.SetBase ("10.1.2.0"  "255.255.255.0");
250     Ipv4InterfaceContainer ifacesB = addrHelper.Assign (devsLanB);
251
252     addrHelper.SetBase ("10.1.3.0", "255.255.255.0");
253     Ipv4InterfaceContainer ifacesC = addrHelper.Assign (devsLanC);
254
255     Ipv4Address srvAddrs[2]; // Servers addresses
256     srvAddrs[0] = ifacesC.GetAddress (numNodesLanC - 2);
257     srvAddrs[1] = ifacesC.GetAddress (numNodesLanC - 1);
258
259     Ipv4GlobalRoutingHelper::PopulateRoutingTables ();
260
261     NS_LOG_INFO ("Configure mapping between IP Streams and PCI-E TCs.");
262
263     for (uint32_t i = 0; i < nodes.GetN (); ++i)
264       {
265         Ptr<PcieIpStreamTcMap> streamMap = nodes.Get (i)->GetObject<PcieIpStreamTcMap> ();
266         streamMap->AddStream (Ipv4Address::GetAny (), srvAddrs[0], 0, 0, tcStream0);
267         streamMap->AddStream (Ipv4Address::GetAny (), srvAddrs[1], 0, 0, tcStream1);
268       }
269
270     NS_LOG_INFO ("Install UDP servers.");
271
272     UdpServerHelper srvHelper (port);
273     ApplicationContainer srvApps = srvHelper.Install (nodesSrv);
274
```

```
275   NS_LOG_INFO ("Install UDP clients.");
276
277   UdpClientHelper cltHelper (srvAddrs[0], port);
278   cltHelper.SetAttribute ("PacketSize", UintegerValue (pktSize));
279   cltHelper.SetAttribute ("MaxPackets", UintegerValue (maxPackets));
280   cltHelper.SetAttribute ("Interval", TimeValue (MilliSeconds (interval)));
281
282   ApplicationContainer cltApp1 = cltHelper.Install (cltNode);
283   cltHelper.SetAttribute ("RemoteAddress", AddressValue (srvAddrs[1]));
284   ApplicationContainer cltApp2 = cltHelper.Install (cltNode);
285
286   srvApps.Start (Seconds (1.0));
287
288   cltApp1.Start (Seconds (1.0));
289
290   cltApp2.Start (Seconds (1.0 + (((double)interval / 1000) / 2)));
291
292   // Configure trace sources
293   TraceServers (srvApps);
294
295   NS_LOG_INFO ("Run Simulation.");
296
297   Simulator::Stop (Seconds (duration));
298   Simulator::Run ();
299
300   NS_LOG_INFO ("Finised in " << Simulator::Now ().GetSeconds () << " seconds.");
301
302   AsciiTraceHelper ascii;
303   char filename[30];
304
305   for (uint32_t i = 0; i < srvApps.GetN (); ++i)
306     {
307       snprintf (filename, sizeof(filename), "strmTotalLost %d.txt", i);
308       Ptr<OutputStreamWrapper> stream = ascii.CreateFileStream (filename);
309       Ptr<UdpServer> srv = srvApps.Get (i)->GetObject<UdpServer> ();
310       *stream->GetStream () << srv->GetLost () << std::endl;
311
312       NS_LOG_INFO ("Flow " << i << " - Received: "
313                            << srv->GetReceived () << " Lost: "
314                            << srv->GetLost ());
315     }
316
317   Simulator::Destroy ();
318
319   NS_LOG_INFO ("Done.");
320
321   return 0;
322 }
```

multi-bound-callback-impl.h

```
1  #ifndef MULTI_BOUND_CALLBACK_IMPL_H
2  #define MULTI_BOUND_CALLBACK_IMPL_H
3
4  #include "ns3 callback.h"
5
6  namespace ns3 {
7
   template <typename R, typename TX1, typename TX2, typename T1, typename T2, typename T3, typename
9  class MultiBoundFunctorCallbackImpl : public CallbackImpl<R,T1,T2,T3,T4,T5,T6,T7,T8,empty>
10 {
```

```
11  public:
12    template <typename FUNCTOR, typename ARG1, typename ARG2>
13    MultiBoundFunctorCallbackImpl (FUNCTOR functor, ARG1 a1, ARG2 a2)
14        m_functor (functor),
15        m_a1 (a1),
16        m_a2 (a2)
17    {
18    }
19    virtual ~MultiBoundFunctorCallbackImpl ()
20    {
21    }
22    R operator() (void)
23    {
24      return m_functor (m_a1,m_a2);
25    }
26    R operator() (T1 a1)
27    {
28      return m_functor (m_a1,m_a2,a1);
29    }
30    R operator() (T1 a1,T2 a2)
31    {
32      return m_functor (m_a1,m_a2,a1,a2);
33    }
34    R operator() (T1 a1,T2 a2,T3 a3)
35    {
36      return m_functor (m_a1,m_a2,a1,a2,a3);
37    }
38    R operator() (T1 a1,T2 a2,T3 a3,T4 a4)
39    {
40      return m_functor (m_a1,m_a2,a1,a2,a3,a4);
41    }
42    R operator() (T1 a1,T2 a2,T3 a3,T4 a4,T5 a5)
43    {
44      return m_functor (m_a1,m_a2,a1,a2,a3,a4,a5);
45    }
46    R operator() (T1 a1,T2 a2,T3 a3,T4 a4,T5 a5,T6 a6)
47    {
48      return m_functor (m_a1,m_a2,a1,a2,a3,a4,a5,a6);
49    }
50    R operator() (T1 a1,T2 a2,T3 a3,T4 a4,T5 a5,T6 a6,T7 a7)
51    {
52      return m_functor (m_a1,m_a2,a1,a2,a3,a4,a5,a6,a7);
53    }
54    R operator() (T1 a1,T2 a2,T3 a3,T4 a4,T5 a5,T6 a6,T7 a7,T8 a8)
55    {
56      return m_functor (m_a1,m_a2,a1,a2,a3,a4,a5,a6,a7,a8);
57    }
58    virtual bool IsEqual (Ptr<const CallbackImplBase> other) const
59    {
60      MultiBoundFunctorCallbackImpl<T,R,TX1,TX2,T1,T2,T3,T4,T5,T6,T7,T8> const *otherDerived =
61        dynamic_cast<MultiBoundFunctorCallbackImpl<T,R,TX1,TX2,T1,T2,T3,T4,T5,T6,T7,T8> const *> (PeekPointer (o
62      if (otherDerived == 0)
63        {
64          return false;
65        }
66      else if (otherDerived->m_functor != m_functor
67               || otherDerived->m_a1 != m_a1
68               || otherDerived->m_a2 != m_a2 )
69        {
70          return false;
71        }
72      return true;
```

```
73      }
74    private:
75      T m_functor;
76      typename TypeTraits<TX1>::ReferencedType m_a1;
77      typename TypeTraits<TX2>::ReferencedType m_a2;
78    };
79
80    template <typename R, typename TX1, typename TX2,typename ARG1, typename ARG2, typename T1>
81    Callback<R,T1> MakeMultiBoundCallback (R (*fnPtr)(TX1,TX2,T1), ARG1 a1, ARG2 a2)
82    {
83      Ptr<CallbackImpl<R,T1,empty,empty,empty,empty,empty,empty,empty,empty> > impl =
84        Create<MultiBoundFunctorCallbackImpl<R (*)(TX1,TX2,T1),R,TX1,TX2,T1,empty,empty,empty,empty,empty,empty,em
85      return Callback<R,T1> (impl);
86    }
87
88    } // namespace ns1
89
90    #endif // MULTI_BOUND_CALLBACK_IMPL_H
```

Bibliografía

[1] PCI SIG, "PCI Express Base Specifications Revision 1.1," *PCI SIG*, march 2005.

[2] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[3] R. Budruk, D. Anderson, and E. Solari, *PCI Express System Architecture.* Pearson Education, 2003.

[4] J. Kepler, U. Linz, F Nassar, J. Haase, C. Grimm, H. Nachtnebel, and M. Ghameshlu, "Design and Simulation of a PCI Express based Embedded System," 2008.

[5] N. Baldo, M. Requena, J. Nunez, M. Portoles, J. Nin, P. Dini, and J. Mangues, "Validation of the ns-3 IEEE 802.11 model using the EXTREME testbed," in *Proceedings of SIMUTools Conference, 2010*, March 2010.

[6] J. Farooq and T. Turletti, "An IEEE 802.16 WiMAX Module for the NS-3 Simulator," in *2nd International Conference on Simulation Tools and Techniques (SIMUTools09)*, March 2009. [Online]. Available: ftp://ftp-sop.inria.fr/rodeo/turletti/simutools09.pdf

[7] M. A. Ismail, G. Piro, L. A. Grieco, and T. Turletti, "An Improved IEEE 802.16 WiMAX Module for the ns-3 Simulator," in *Proceedings of SIMUTools Conference, 2010*, March 2010.

[8] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, March 2008. [Online]. Available: http://doi.acm.org/10.1145/1355734.1355746

[9] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, pp. 263–297, August 2000. [Online]. Available: http://doi.acm.org/10.1145/354871.354874

[10] I. Sommerville, *Software Engineering: (Update) (8th Edition) (International Computer Science)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[11] C. Demichelis and P. Chimento, "Ip packet delay variation metric for ip performance metrics (ippm)," Internet Engineering Task Force, RFC 3393, nov 2002. [Online]. Available: http://tools.ietf.org/html/rfc3393

CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL I.P.N.
UNIDAD GUADALAJARA

El Jurado designado por la Unidad Guadalajara del Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional aprobó la tesis

Diseño e implementación de simulador de PCI Express

del (la) C.

José Francisco LOMBERA LANDA

el día 11 de Octubre de 2012.

Dr. Félix Francisco Ramos Corchado
Investigador CINVESTAV 3A
CINVESTAV Unidad Guadalajara

Dr. Mario Angel Siller González
Pico
Investigador CINVESTAV 2C
CINVESTAV Unidad Guadalajara

Dra. Susana Ortega Cisneros
Investigador CINVESTAV 2B
CINVESTAV Unidad Guadalajara