Estimación de ancho de banda disponible por ajuste en los tiempos de trasmisión y recepción de paquetes de prueba a través de NetFPGA

TESIS
Presentada para obtener el título
Ingeniero de Sistemas
Universidad Autónoma de Bucaramanga

Manuel Fernando Jaimes Mejía
Junio 2016

Para mis viejos, para mi hermano y para Vanessa ♡. ¡Muchas gracias!

Empty your memory,

with a `free()`*...*

like a pointer...

If u cast a pointer to a integer,

it becomes the integer,

If u cast a pointer to a struct,

it becomes the struct...

The pointer can crash...

and can Overflow...

Be a pointer my friend...

Resumen

Las herramientas de estimación de ancho de banda disponible requieren del envío de paquetes de prueba a tiempos exactos. El sistema operativo no puede garantizar que el envío de estos paquetes sean a los tiempos exactos debido al intercambio y prioridades de los procesos en el sistema lo cual adiciona tiempos innecesarios entre paquetes y genera errores en la estimación.

El objetivo de este proyecto es modificar un estimador de ancho de banda disponible mediante mecanismos de variación en los tiempos de transmisión y recepción para que interactúe con la plataforma NetFPGA, la cual garantizará el envío de los paquetes a los tiempos determinados por la herramienta.

En los resultados se plantean la revisión de la literatura referente a mecanismos de modificación de tiempos de transmisión y recepción de paquetes, se justifican y detallan las variaciones hechas en la herramienta de estimación de ancho de banda que garantizarán el envío de los paquetes a tiempos exactos, posteriormente se implementa el generador de paquetes y el estimador sobre un testbed de red diseñado especialmente para el proyecto, y finalmente se realiza se realiza una comparación entre las dos versiones de la herramienta.

Palabras clave: Estimación de ancho de banda disponible, traceband, NetFPGA, libpcap, generador de paquetes.

Índice general

Índice de tablas

Índice de figuras

Índice de código fuente

Capítulo 1

Introdución

La estimación del ancho de banda disponible (en adelante ab_disp) entre dos nodos finales a través de Internet es un área que ha llamado, en los últimos veinte años, la atención de investigadores alrededor del mundo debido a la utilidad que tiene para diversas aplicaciones de red. Por ejemplo, herramientas de administración pueden controlar con precisión la utilización de un enlace; proveedores de servicio de Internet pueden monitorear y verificar niveles de calidad de servicio; protocolos de transporte pueden determinar la mejor tasa de transmisión según sea la cantidad de ancho de banda disponible en la red; sistemas de detección de intrusos pueden generar alertas basadas en un aumento inesperado en la utilización de la red. Estas y otras aplicaciones requieren una estimación del ancho de banda disponible de extremo a extremo dado que no se tiene control sobre los nodos intermedios a través de los cuales se establece el canal de comunicación.

Para estimar el ab_disp de extremo a extremo de una red, es necesario inferir por muestreo, el ab_disp entre los nodos intermedios o enrutadores. Estas muestras se obtienen enviando trenes de pares de paquetes de prueba a tiempos determinados por la aplicación desde un extremo de la red y observando en el extremo opuesto las variaciones en separación o retardos que sufren dichos paquetes de prueba al

interactuar con el tráfico de la red en los nodos intermedios Guerrero and Labrador (2010a).

Las muestras u observaciones del ab_disp usadas por las herramientas de estimación existentes, están sujetas a varias fuentes de error que no pueden ser controladas por dichas herramientas: incorrectos registros de tiempos de envío y recepción, recepción de paquetes fuera de orden, replicación y corrupción de paquetes de prueba, son algunos ejemplos Zhou et al. (2006).

Este proyecto parte de la hipótesis de que enviando paquetes de prueba desde el *hardware* de la NetFPGA posibilita la reducción de errores de estimación que tienen origen en la imprecisión del tiempo de envío de paquetes a través del *software*. El proyecto se encuentra organizado de la siguiente manera: El capítulo 2 presenta los objetivos planteados, el capítulo 3 el marco teórico dentro del cual se enmarca la presente investigación, el capítulo 4 contempla el estado del arte, a continuación, el capítulo 5 describe el proceso investigativo, el capítulo 6 muestra los resultados y finalmente en los capítulos 7 y 8 se presentan las recomendaciones y conclusiones.

Capítulo 2

Objetivos de la investigación

2.1. Objetivo general

Desarrollar un sistema NetFPGA para estimación de ancho de banda disponible por ajuste en los tiempos de trasmisión y recepción de paquetes de prueba.

2.2. Objetivos específicos

- Analizar literatura existente sobre mecanismos de modificación de tiempos de transmisión de paquetes tanto en hardware como en software.
- Modificar un estimador de ancho de banda disponible para que interactúe con el hardware del diseño de referencia de la interfaz de red de la NetFPGA.
- Implementar un módulo en NetFPGA que permita enviar y recibir paquetes de prueba a tiempos específicos determinados por un estimador de ancho de banda disponible.
- Construir una red de prueba que permita evaluar la efectividad del estimador desarrollado.

- Comparar el error, cantidad de tráfico extra y tiempo del estimador modificado para NetFPGA con la versión estándar del mismo.

Capítulo 3

Marco teórico

El contexto teórico del proyecto se ubica en tres conceptos en las áreas de redes de computadoras: Ancho de banda disponible, Libpcap y NetFPGA.

3.1. Ancho de banda

3.1.1. Definición

El ancho de banda es una medida de velocidad que se define como la cantidad de bits transmitidos en un canal de comunicaciones por unidad de tiempo. Hay dos métricas asociadas con el ancho de banda. Una es la capacidad y la otra es el ancho de banda disponible. La capacidad de un enlace es el ancho de banda máximo o la máxima cantidad de bits que se pueden transmitir al enlace por unidad de tiempo Morillo (2011).

3.1.2. Ancho de banda disponible

El ancho de banda disponible (AB) es la capacidad no utilizada en el enlace Guerrero and Labrador (2010b). Ver Figura 3.1.

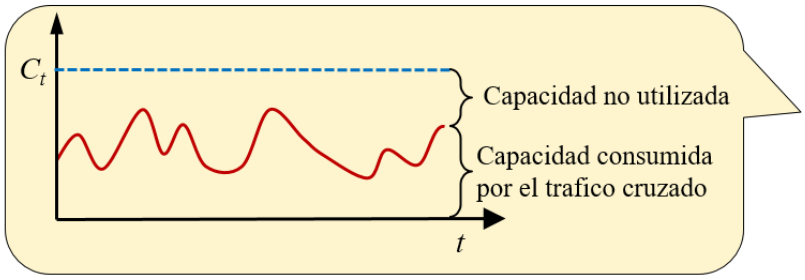


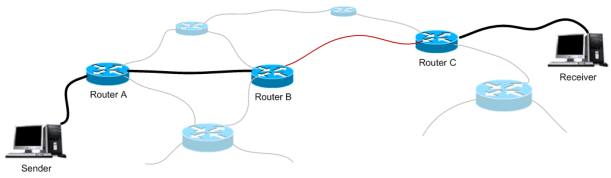
Figura 3.1. Ancho de banda disponible

El AB está dado por la Ecuación 3.1

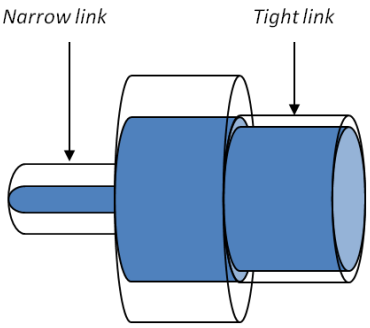
$$AB_i = C_i - (Trafico\ cruzado)_i \tag{3.1}$$

3.1.3. Ancho de banda disponible de extremo a extremo

El ancho de banda disponible (ab_disp) de extremo a extremo (end-to-end) es la mínima capacidad no utilizada de todos los enlaces en una ruta de comunicación entre dos nodos extremos. En la Figura 3.2a, los nodos extremos (*end-hosts*) son los equipos *sender* y *receiver* y la ruta de comunicación está formada por cuatro enlaces entre enrutadores y *end-hosts*.



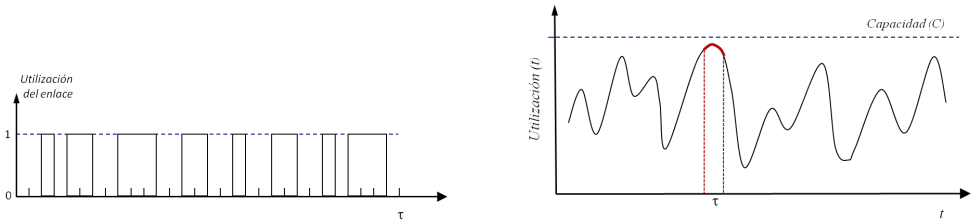
(a) Comunicación de extremo a extremo.



(b) Narrow link y Tight link.

Figura 3.2. Comunicación de extremo a extremo y enlaces que determinan el ancho de banda: *Narrow link* y *Tight link*

El enlace que tiene la mínima capacidad de todos los enlaces de la ruta se denomina el *narrow link* y su valor es la capacidad de toda la ruta. Por otra parte, el *tight link* es el enlace que tiene el valor mínimo de capacidad disponible determinando así el valor del ancho de banda disponible de la ruta. Obsérvese en la Figura 3.2b, que el enlace que tiene la mínima capacidad *(narrow link)* no es necesariamente el enlace que determina el ancho de banda disponible *(tight link)*.



(a) Utilización de un enlace durante un intervalo de tiempo muy pequeño τ.

(b) Consumo del ancho de banda disponible a lo largo τ del tiempo.

Figura 3.3. Utilización de un enlace durante un intervalo de tiempo muy pequeño τ y consumo del ancho de banda disponible a lo largo del tiempo.

De manera formal, la utilización de un enlace toma valores discretos de *0* (canal sin utilizar) y *1* (se utiliza el canal) en un instante de tiempo dado. Esto hace que la utilización de un enlace se hable en términos de promedios. En el ejemplo de la Figura 3.3a, el enlace es usado en *15* de *30* intervalos de tiempo lo cual da una utilización de *0.5* en promedio.

La utilización promedio de un enlace i durante un periodo de tiempo $(t,\ t+\tau)$, está dada por la Ecuación 3.2 Jain and Dovrolis (2002a).

$$u_i\left(t, t+\tau\right) = \frac{1}{\tau}\int_{t}^{t+\tau} u_i\left(t\right) dt \tag{3.2}$$

Conociendo la utilización, el ancho de banda disponible en el enlace i estaría dado por la Ecuación 3.3:

$$ab\_disp_i\,(t, t+\tau) = ab\_disp_i^{\tau}\,(t) = C_i\,[1 - u_i\,(t, t+\tau)] \tag{3.3}$$

Donde C_i es la capacidad del enlace i. De ésta manera, el ancho de banda disponible de extremo a extremo en una ruta compuesta por H enlaces, está definido por la Ecuación 3.4

$$ab\_disp\,(t, t+\tau) = ab\_disp^{\tau}\,(t) = \min_{i=1\ldots H}\{ab\_disp_i\,(t, t+\tau)\} \tag{3.4}$$

Para estimar el ab_disp en una red es necesario enviar paquetes de prueba y analizar los retardos que sufren al interactuar con el tráfico existente en la red (tráfico cruzado). Existe sin embargo una propuesta basada en el uso de paquetes de datos cuidadosamente seleccionados y retardados para que sirvan como paquetes de prueba sin insertar tráfico adicional en la red . En términos generales, el comportamiento de un par de paquetes cuando salen del *tight link* se muestra en la Figura 3.4.

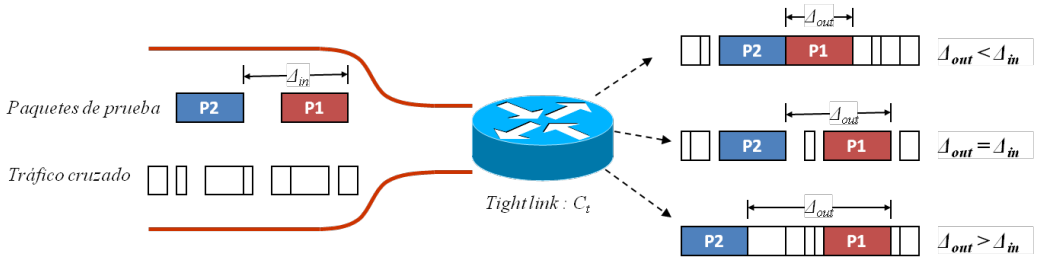


Figura 3.4. Interacción de un par de paquetes de prueba en el tight link.

Si dos paquetes consecutivos son enviados hacia un enlace, ellos llegan al nodo con cierta separación de tiempo entre ellos (Δ_{in}). Después de interactuar en la cola del *tight link* con el tráfico cruzado que viene de diferentes fuentes, el par de paquetes de prueba saldrá del enrutador con una nueva separación de tiempo (Δ_{out}). La diferencia entre ellos $\Delta_{in} - \Delta_{out}$ es la dispersión del par de paquetes.

La dispersión del par de paquetes puede ser negativa, positiva o igual a cero. Tal como se ve en la Figura 3.4, un valor negativo ($\Delta_{out} < \Delta_{in}$) ocurre cuando el primer paquete encuentra paquetes de tráfico cruzado en la cola seguido por el

segundo paquete. Un valor positivo ($\Delta_{out} > \Delta_{in}$) ocurre cuando se insertan paquetes de tráfico cruzado entre el par de paquetes de prueba en la cola. Finalmente, un valor de cero ($\Delta_{out} = \Delta_{in}$) ocurre cuando el enlace no tiene suficiente tráfico cruzado para afectar la separación inicial de los paquetes.

3.1.4. Técnicas de estimación de ancho de banda disponible

Con base en el modelo de la Figura 3.4, existen dos enfoques para estimar el ancho de banda disponible en una ruta de extremo a extremo Prasad et al. (2003): el *Probe Gap Model* (en adelante PGM) y el *Probe Rate Model* (en adelante PRM). El PGM observa la dispersión de cada par de paquetes de prueba cuando interactúa con el tráfico existente en la red (tráfico cruzado). El PRM observa los retardos en un sentido de los paquetes de prueba buscando el punto en el que los retardos muestran una tendencia incremental. La velocidad de envío de paquetes de prueba que genera ese retardo incremental corresponde al ancho de banda disponible de la ruta.

Método *Probe Rate Model*

Basado en los conceptos de congestión auto inducida; PRM se basa en la siguiente consideración, si se envía tráfico de prueba a una razón más baja que el ancho de banda disponible a través de un enlace, entonces, la razón de llegada del tráfico de prueba debe coincidir con la razón a la cual fue enviado. Por el contrario, si la razón a la que se envía el tráfico de prueba es mayor que el ancho de banda disponible, se crea una cola en la red y el tráfico de prueba debe presentar retardo. Como resultado, la razón de los paquetes de prueba en el receptor, debe ser menor que la de envío. De esta manera, se puede medir el ancho de banda disponible, localizando el punto de retorno, en el cual los paquetes de prueba enviados y los recibidos, comienzan a coincidir Carrasquilla et al. (2006).

Las herramientas que hacen uso de este método como *Pathload*, *PTR*, *Pathchirp* y *TOPP* se basan en las siguientes observaciones Lakshminarayanan et al. (2004)

- Un tren de paquete de prueba enviados a una velocidad menor que el ancho de banda disponible debería ser recibido a la misma velocidad enviada (en promedio).

- Sin embargo si la velocidad a la cual fue enviado excede el ancho de banda disponible la velocidad de recepción es menor que la de trasmisión y los paquetes de prueba tienden a encolarse provocando retardo en ese sentido.

- Por lo tanto el ancho de banda disponible puede ser estimado mediante la observación de la velocidad de trasmisión en la cual sucede una transición entre estos dos comportamientos.

Método *Probe Gap Model*

Este modelo hace uso de la información del espacio de tiempo entre la llegada sucesiva de dos paquetes de prueba al receptor. Un par de paquetes de prueba es enviado con un espacio determinado entre ellos $\Delta_{entrada}$ y llega al receptor con una distancia Δ_{salida}. Asumiendo un único cuello de botella y que la cola no se encuentra vacía entre el envío del primer y el segundo paquete de prueba, entonces Δ_{salida} es el tiempo que le toma al cuello de botella transmitir el segundo paquete de prueba en el par y el tráfico cruzado que llega durante $\Delta_{entrada}$, como se muestra en la Figura 3.5 Carrasquilla et al. (2006).

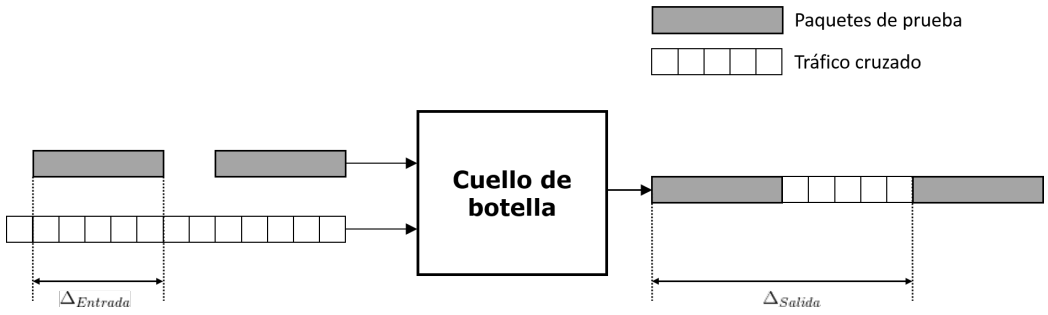


Figura 3.5. Método *Probe Gap Model*

De esa forma, el tiempo para transmitir el tráfico cruzado es $\Delta_{salida} - \Delta_{entrada}$, y la razón de llegada del tráfico cruzado es $\frac{\Delta_{salida}-\Delta_{entrada}}{\Delta_{entrada}} * C$, C es la capacidad del cuello de botella.

El ancho de banda disponible es dado por la Ecuación 3.5

$$A = C * \left(1 - \frac{\Delta_{salida} - \Delta_{entrada}}{\Delta_{entrada}} * C\right) \tag{3.5}$$

Adicionalmente, las herramientas basadas en este método como *Spruce*, *Delphi* e *IGI* envían pares de paquetes de prueba de igual tamaño separados acorde al tiempo de transmisión de las pruebas sobre el cuello de botella del enlace. Si no se inserta tráfico cruzado entre los paquetes de prueba el espacio entre los paquetes de prueba se mantiene hasta el destino. Por otro lado, el incremento del espacio entre paquetes de prueba se utiliza para estimar el volumen del tráfico cruzado el cual luego es restado de la capacidad estimada para obtener el valor del ancho de banda disponible estimado.

Ambos modelos usan trenes de paquetes para generar una medición y toman como ciertos los siguientes supuestos:

- La cola en todos los enrutadores es atendida siguiendo el modelo FIFO.
- El tráfico cruzado sigue un modelo fluido.
- Las tazas de transmisión del tráfico cruzado, cambian lentamente y de forma constante para cada medición.

Adicionalmente, el modelo PGM asume un único cuello de botella que puede ser el *narrow link* o el *tight link* para cada enlace punto a punto. Estas consideraciones son necesarias para el análisis del modelo, pero las herramientas pueden trabajar, aún, si algunas de ellas no se cumplen Carrasquilla et al. (2006).

3.1.5. Estimadores de ancho de banda disponible

Existen en la actualidad varias herramientas de estimación de ancho de banda disponible, entre las más conocidas podemos citar las siguientes: *Abing*, *Cprobe*, *IGI*, *Pathchirp*, *Pipechar*, *pathload*, *Spruce*, *Pathload* y *Traceband*.

Existen dos herramientas para estimar el ancho de banda disponible de extremo a extremo, que han sido consideradas como las más representativas, por sus rendimientos. La primera es conocida cómo *Spruce* y utiliza el método PGM, la segunda llamada *Pathload*, ésta utiliza el método PRM Guerrero and Labrador (2010b).

Otro estimado de ancho de banda importante es *Traceband*, que es una herramienta cliente-servidor escrita en ANSI C que utiliza cadenas de Markov, en la dinámica de ancho de banda disponible para proporcionar estimaciones de ancho de banda, rápidas, continua y precisas. El cliente *Traceband* se ejecuta por ciclos de diez estimaciones. En la primera estimación la herramienta envía 50 pares de paquetes UDP de 1498 *bytes*. Las restantes nueve estimaciones se realizan con 30 pares de paquetes cada una. Esta reducción es posible gracias a la introducción de HMM, que son capaces de aprender la dinámica AB con una muestra inicial y mantener el modelo actualizado con muestras de tamaño reducido Santos (2015).

El método de estimación utilizado por *Traceband* es el *Probe Gap Model* y basado en éste, utiliza valores diferentes para las instancias del *intra-gap* e *inter-gap* de los pares de paquetes. El *intra-gap* hace referencia al tiempo entre los dos paquetes de cada par de paquetes. El *intra-gap* o $\Delta_{entrada}$ se específica el tiempo en el emisor y es igual al tiempo de transmisión de un paquete único en el enlace apretado *(Tight link)*. De esta manera, el par de paquetes será capaz de capturar tráfico cruzado en la cola, si hubiese. El *inter-gap* o Δ_{salida}, se refiere al tiempo entre los pares de paquetes de sondeo. Es decir, el tiempo entre el segundo paquete de sondeo par $(i-1)$ y el primer paquete de sondeo par i. Estos tiempos se obtienen utilizando la función

`gettimeofday`, y sus valores son enviados al receptor en la carga útil del paquete Santos (2015).

3.1.6. Dificultades en las estimaciones

Las dificultades para estimar, con base en los modelos descritos, el ab_disp entre dos *end-hosts* obedecen principalmente a dos razones. Una es la naturaleza de variaciones abruptas del tráfico cruzado y la otra está relacionada con errores generados por los *end-hosts* y los enrutadores a lo largo de la ruta de un extremo a otro. Debido a la naturaleza de variaciones abruptas del tráfico cruzado, un solo par de paquetes de prueba no puede capturar la carga de tráfico promedio en el modelo descrito en la Figura 3.4. Para manejar este problema, las herramientas de medición basadas tanto en el enfoque PGM como el PRM utilizan un tren de paquetes de prueba para generar una sola medición promediada.

Los errores asociados a los *end-hosts* generando paquetes de prueba son complejos de controlar y en algunos casos están fuera del alcance de cualquier estimador, dado que son errores asociados con planificación de procesos donde las herramientas de estimación, a nivel de aplicación, no pueden determinar la prioridad de los mismos Michaut and Lepage (2005); Paxson (1997); Zhou et al. (2006). Los registros incorrectos de tiempos de salida o llegada de paquetes, imposibilidad para usar la capacidad real de transmisión de la tarjeta de red, llegada de paquetes fuera de orden, replicación de paquetes, corrupción de paquetes y comportamientos cambiantes en las colas de los enrutadores, afectan la precisión de la medición llevada a cabo por un solo par de paquetes de prueba. La mayoría de estos errores se pueden manejar en un ambiente de prueba controlado pero no en un escenario real donde los usuarios pueden correr la aplicación de medición pero no tienen conocimiento sobre como configurar sus máquinas convenientemente.

3.2. Plataforma NetFPGA

3.2.1. Definición

NetFPGA es una plataforma de *hardware* y *software* abierto que permite procesar y transmitir paquetes a la velocidad del enlace sin perder ningún paquete.

3.2.2. Historia

La idea de tener una plataforma abierta de dispositivos de red con propósitos de enseñanza, surge al comienzo de la década en la Universidad de Stanford. El objetivo de esta iniciativa era el crear una plataforma sobre tarjetas *FPGA (Field-Programmable Gate Array)* orientadas a servicios de red que al ser usadas en un *VNS* (*Virtual Network System*) pudiesen integrarse dinámicamente a diferentes topologías de red y ser accedidas desde Internet Casado et al. (2005).

El resultado fue la creación de una tarjeta NetFPGA la cual es un sistema completo de enseñanza para el diseño de hardware de redes Ethernet. Los fines académicos de la tarjeta marcaron su diseño y funcionalidad en cuanto a flexibilidad, bajo costo, robustez y facilidad de uso.

La iniciativa fue rápidamente adoptada por diferentes universidades alrededor del mundo y se desarrollaron tanto cursos de nivel de posgrado basados en la tecnología como proyectos de investigación altamente innovadores Beheshti et al. (2007); Covington et al. (2009a,b); Gibb et al. (2008); Naous et al. (2008).

Este rápido desarrollo llamó la atención de fabricantes y empresas de tecnología que hoy soportan el desarrollo de tecnología alrededor de NetFPGA: Cisco, Google, Xilinx, Agilent, Juniper, entre otras [1].

Existe una comunidad académica que interactúa permanentemente a través de talleres, conferencias, fórums en línea y otros medios ampliamente usados por otras comunidades de software libre. Esta comunidad de "open-source hardware" se basa

[1] http://netfpga.org

en compartir diseños de hardware que potencian rápidos desarrollos de dispositivos mejorados de red Covington et al. (2009b).

Actualmente existen cuatro tipos de tarjeta NetFPGA. Una que opera a 1Gbps (Figura 3.6), otra que opera con cuatro puertos de 10Gbps (Figura 3.10), una mejora de la versión de 1Gbps y compatible con los diseños de 10Gbps (Figura 3.11 y finalmente, la SUME (Figura 3.12) que opera a velocidades de 10 y 100 *Gbps*. El cuadro 3.1 resume las características de las tarjetas.

Tabla. 3.1

Comparación entre las tarjetas NetFPGA.

Item	**NetFPGA 1G**	**NetFPGA 10G**	**NetFPGA CML**	**NetFPGA SUME**
Puertos	4 x 1Gbps.	4 x 10Gbps.	4 x 1Gbps	4 x 10Gbps
RAM	4.5 MB ZBT SRAM	27 MB QDRII-SRAM	4.5 MB QDRII+	3x72Mbit QDRII+ SRAMs
	64 MB DDR2 SDRAM	288 MB RLDRAM-II	512 MB DDR3	2x4GB DDR3 SODIMMs
Interfaz	PCI	PCI Expresss x8	PCI Expresss x4	PCI Expresss x8
Procesador	Virtex II-Pro 50	Virtex 5 TX240T	Kintex-7 XC7K325T	Virtex-7 690T

3.2.3. NetFPGA 1G

La tarjeta NetFPGA 1G tiene cuatro puertos *1 Gigabit Ethernet*. Se conecta al computador a través de una interfaz PCI estándar. Posee un chip *FPGA Xilinx Virtex-II Pro 50*. Esta tarjeta cuenta con *4.5 MBytes* de memoria *SRAM* adecuada para almacenar tablas de enrutamiento y *64 MBytes* de memoria *DDR2 DRAM* con un rendimiento de *25.6 Gbps* para almacenamiento de paquetes de datos.

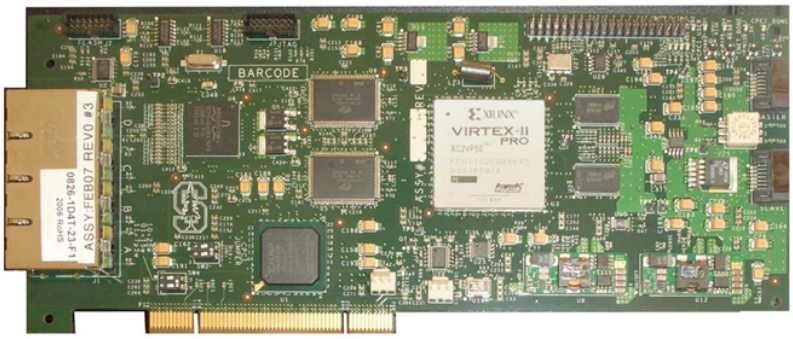


Figura 3.6. Tarjeta NetFPGA 1G.

Mayores especificaciones de la tarjeta son las siguientes:

1. Field Programmable Gate Array (FPGA) Logic
   - Xilinx Virtex-II Pro 50 [2]
   - 53.136 celdas lógicas
   - 4.176 Kbit de bloque RAM
   - 2 x PowerPC cores
   - Totalmente programable por el usuario
2. Puertos de red Gigabit Ethernet
   - Caja de derivación a la izquierda de las interfaces de PCB *(Printed Circuit Board)* a 4 conectores RJ-45 externos.
   - Interfaces con cables de red de cobre con estándar Cat5E o Cat6 utilizando *Broadcom Physical Layer Transceiver of the OSI Model - PHY* [3]
   - Procesamiento a la velocidad del cable en todos los puertos en todo momento utilizando lógica FPGA: 1 Gbits * 2 (bi-direccional) * 4 (puertos) = 8 Gbps
3. *Static Random Access Memory (SRAM)* [4]

[2] http://www.xilinx.com/support.html

[3] Los transceptores de capa física son los que permiten enviar y recibir paquetes desde una conexión de cable de red hacia la NetFPGA

[4] Memoria estática de acceso aleatorio donde los datos se mantienen constante mientras se suministra energía eléctrica al chip de memoria.

- Adecuado para el almacenamiento de datos de la tabla de reenvío.
- Retorno de bus cero (*Zero-bus turnaround - ZBT*), sincronizado con la lógica.
- Dos bancos paralelos de 18 Mbit (2.25 MByte) memoria ZBT.
- Capacidad total: 4.5 MBytes
- Cypress: CY7C1370D-167AXC [5]

4. *Double-Date Rate Random Access Memory (DDR2 DRAM [6])*

   - 400 MHz Reloj Asíncrono.
   - Adecuado para el almacenamiento de paquetes.
   - 25.6 Gbps de rendimiento de memoria.
   - Capacidad Total: 64 MBytes.
   - Micron: MT47H16M16BG-5E

5. Multi-gigabit I/O

   - Dos conectores SATA Multi-Gigabit I/O en el lado derecho del PCB.
   - Permite la conexión de múltiples tarjetas NetFPGA en una CPU estandar.
   - Tarjeta PCI estándar.
   - Puede ser usado en una ranura PCI-X.
   - Habilita la rápida configuración de la FPGA sobre el bus PCI sin usar el cable *Joint Test Action Group (JTAG).*
   - Proporciona acceso a los registros de memoria asignados y la memoria en el hardware de la NetFPGA.

6. Puertos de depuración del Hardware

   - Conector del cable JTAG se puede utilizar para ejecutar el *Xilinx ChipScope Pro.*

[5] http://www.chipcatalog.com/Cypress/CY7C1370D-167AXC.htm

[6] Memoria dinámica de acceso aleatorio, que tiene una manera eficiente de almacenar datos en la memoria, ya que requiere menos espacio físico para almacenar la misma cantidad de datos que si se almacena estáticamente.

Se tratara a detalle la arquitectura y estructura organizacional *software* y *hardware* de esta tarjeta por ser la plataforma sobre la cual serán implementadas las modificaciones al estimador de ancho de banda seleccionado.

Arquitectura

A nivel de bloques, se puede considerar la implementación de la NetFPGA según se observa en la Figura 3.7. En esta arquitectura se empiezan a diferenciar dos componentes en la implementación. El primero hace referencia a los recursos de *hardware* y *software* del computador que contiene la tarjeta y el otro a la tarjeta en si misma.

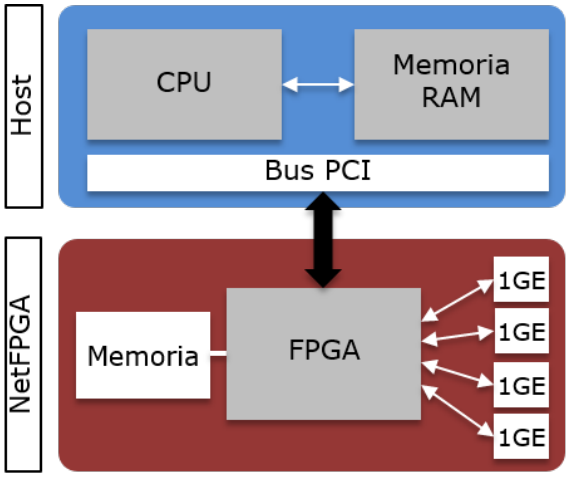


Figura 3.7. Arquitectura de alto nivel de la NetFPGA.

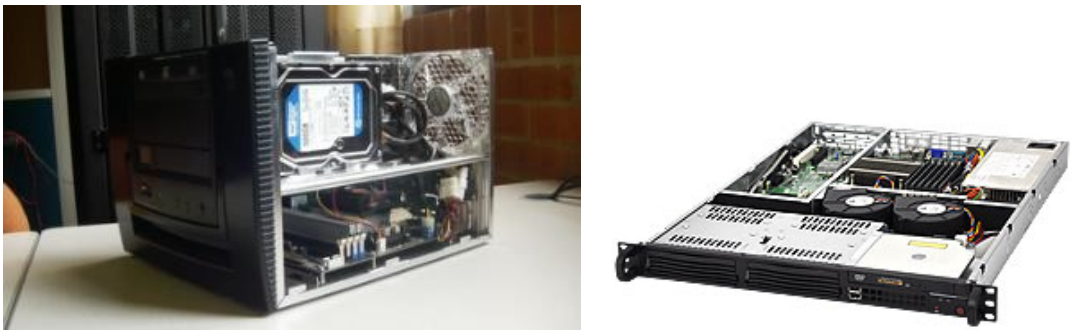
En el caso de un enrutador [7], los componentes que están en el host corresponden al *Control Plane* y los que están en la tarjeta NetFPGA corresponden al *Data Plane*. Lo que se tiene por una parte es una computadora de propósito general que ejecuta tareas de alta demanda en términos de procesamiento pero no de velocidad de ejecución *(control plane)*. Por otra parte, se tiene un *hardware* de alta velocidad de transmisión de paquetes a través de la administración de puertos de red por parte de una FPGA *(data plane)*.

[7] Modelo base mayormente utilizado para la implentación de nuevos proyectos

La computadora que contiene la tarjeta puede ser un PC de escritorio o un servidor de red. Anteriormente, se ofrecían a través de la página del proyecto, dos tipos de implementación verificados y aprobados por la Universidad de Stanford:

Cube Computadora de propósito general con procesador AMD Dual Core o Quad Core (Figura 3.8a)

Servidor Un servidor 1U para montaje en rack con procesador Intel Dual Core o Quad Core (Figura 3.8b)



(a) Cube **(b)** 1U

Figura 3.8. Implementaciones de NetFPGA sobre un computador de escritorio o servidor

Estructura del Software

La organización de directorios se rige a la organización de los proyectos de referencia. A continuación se presenta la estructura de la biblioteca base de la NetFPGA y se describe la función que cumple cada carpeta dentro del sistema:

La biblioteca base de la NetFPGA contiene las siguientes carpetas:

```
netfpga/
├── bin/ ..................Contiene scripts para simulaciones,
│                           síntesis, registros y configuraciones
│                           de entorno.
├── bitfiles/ .............Contiene los bitfiles (hardware
│                           compilado) de todos lo proyectos que
│                           han sido sintetizados.
└── lib/ ..................Librerías y herramientas de software.
```

```
    ├── C/ ..................Software común y código para los diseños
    │                        de referencia.
    ├── java/ ...............Librerías y software para las interfaces
    │                        gráficas de usuario - GUI.
    ├── makefiles/ ..........Makefiles para la simulación y síntesis.
    ├── Perl5/ ..............Bibliotecas para interactuar con los
    │                        diseños de referencia y archivos de ayuda
    │                        para el proceso de simulación y XXXtests
    │                        de regresión.
    ├── Python/ .............Librerías comunes de ayuda en los test de
    │                        regresión.
    ├── Scripts/ ............Scripts útiles - Menos usados que los del
    │                        directorio bin/.
    └── Verilog/ ............Módulos Verilog que pueden ser
                             reutilizados en los diseños.
        ├── core/ ...........Módulos usados por los diseños de
        │                    referencia, desarrollados como parte
        │                    del proceso del diseño de la NetFPGA.
        └── contrib/ ........Módulos contribuidos por los comunidad
                             NetFPGA.
└── projects/ ..............Carpeta de los proyectos.
    └── <project name>/ .....Nombre del proyecto.
        ├── doc/ ............Documentación sobre el proyecto.
        ├── include/ ........Archivos que definen macros y otros
        │                    que especifican qué módulos Verilog de
        │                    la biblioteca base son incluidos en el
        │                    proyecto.
        ├── lib/ ............Cabeceras Perl/Python y C.
        ├── regress/ ........Test de regresión.
        ├── src/ ............Código verilog específico del proyecto.
        ├── sw/ .............Contiene todas las piezas de software que
        │                    acompañan al proyecto.
        ├── synth/ ..........Archivos .XCO específicos del proyecto
        │                    para generar el Xilinx core y un Makefile
        │                    para implementar el diseño.
        └── verif/ ..........Contiene archivos para los test de
                             simulación.
```

La documentación de cada proyecto está localizada en el directorio `doc/` de la carpeta del proyecto. El directorio `include/` contiene un archivo *XML (project.xml)* el cual nombra y describe el proyecto, lista los módulos *verilog* usados, y define la localización de todos los registros en los módulos dentro del proyecto. También contiene todos los archivos *XML* necesarios para los módulos específicos del proyecto. Todos los módulos específicos de un proyecto (por ejemplo, módulos nuevos o modificados de la librería base) están localizados en el directorio `src/`. Los Makefiles de la plataforma NetFPGA aseguran que módulos dentro del directorio `src/` de un

proyecto sobreescriban cualquier modulo con el mismo nombre dentro de la librería base.

El directorio de librerías `lib/` es usado por los *scripts* de la NetFPGA para almacenar registro de los lenguajes C como Perl. Este directorio y los archivos de registro son automáticamente generados cuando se corren simulaciones o síntesis. Los archivos de registro permiten al software llamar a los registros mediante su nombre en lugar de direcciones. Esto brinda la posibilidad de modificar direcciones de registros sin tener que preocuparse de cambiar los valores de las direcciones codificadas.

El directorio de síntesis `synth/` es donde todos los archivos con extensión *.xco* de Xilinx están localizados. Estos archivos son usados para crear los cores necesarios para el *Xilinx Core Generator*. A su vez es donde el proyecto es sintetizado. Después de ejecutar la síntesis y de haberse generado el *bitfile* del proyecto los test de regresión pueden ser ejecutados sobre el *bitfile* recientemente generado.

El directorio de verificación `verif/` contiene *tests* de simulación que permiten comprobar el correcto funcionamiento del *gateware*. Cada test es alojado en una carpeta diferente dentro del directorio `verif/`. El esquema de nombres del directorio test es <TEST>_<MAJOR>_<MINOR>, donde las etiquetas <MAJOR> y <MINOR> pueden ser nombradas como el diseñador del proyecto desee. Los *script* de simulación utilizan el <MAJOR> y <MINOR> para poder determinar cual test ejecutar. Si el *script* de simulación es ejecutado sólo con la opción<MAJOR>, entonces todos los *tests* que tengan un <MAJOR> igual al especificado serán ejecutados secuencialmente. En caso de que el <MAJOR> y el <MINOR> sean especificados, sólo será ejecutado el test que corresponda a estos dos parámetros. Cada *test* contiene un archivo llamado *makepackets.pl*, el cual usa librerías Perl incluidas en el *NFP* [8], para crear paquetes y para la lectura/escritura de registros en la simulación. Finalmente, el directorio `sw/` incluye todo el *software* (funcional y de diagnostico) para el proyecto específico.

[8]Por sus siglas en inglés, *NetFPGA Package*

Estructura del Hardware

La división del hardware se ha hecho mediante módulos. La comprensión de estos módulos es esencial para el entendimiento de la mayoría de los diseños de referencia. Todos los proyectos distribuidos en el *NFP* siguen esta misma estructura modular. Este diseño es un *pipeline* donde cada etapa es un módulo separado. Un diagrama del *pipeline* es mostrado en la Figura 3.9.

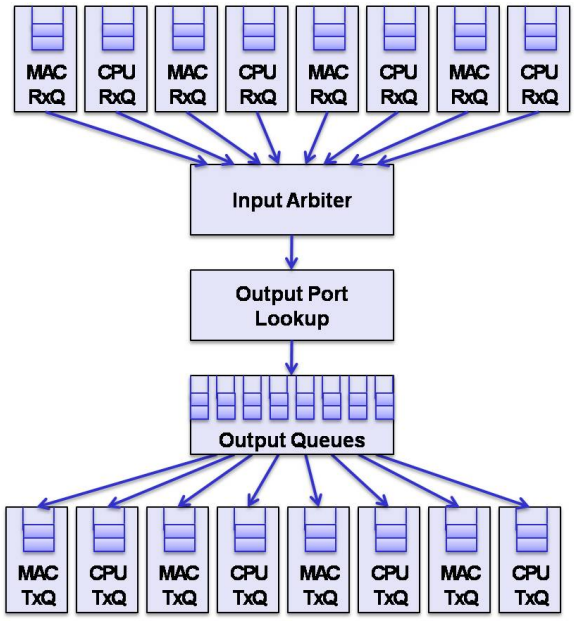


Figura 3.9. Pipeline del referencia

.

Pipeline de referencia

La primera etapa, consta de varias colas que se denominan colas Rx *(Rx Queues)*. Estas colas reciben paquetes desde los puertos de E/S (tales como los puertos *Ethernet* y *PCI*) y proporcionan una interfaz unificada para el resto del sistema. Estos puertos están conectados al *User Data Path*, el cual contiene las etapas de procesamiento. El enrutador, que es el diseño de referencia, consta de 4 colas *Rx Ethernet* y 4 colas *CPU DMA*. Las colas *Ethernet* y *DMA* se encuentra intercaladas de manera que para el *User Data Path*, la cola *Rx* 0 es el puerto *Ethernet* 0, la cola *Rx* 1 es el puerto *DMA*

0, la cola *Rx* 2 es el puerto *Ethernet* 1, y así sucesivamente. Los paquetes que llegan a la cola *CPU DMA Rx* `X` son paquetes que han sido enviados por software a través de la interfaz *nf2cX*.

Dentro del *User Data Path*, el primer módulo que un paquete atraviesa es el *Input Arbiter*. Este módulo decide a cual cola *Rx* atiende, saca el paquete de esa cola y se lo entrega al siguiente módulo en el *pipeline*: el módulo *Output Port Lookup*. El *Output Port Lookup* se encarga de decidir por cuál puerto un paquete debe salir. Después de que se ha hecho esta decisión, el paquete es entregado al siguiente módulo: el modulo *Output Queues*, el cual almacena el paquete en las colas correspondientes al puerto de salida, hasta que la cola *Tx* acepte el paquete para su transmisión.

Las colas *Tx* son análogas a las colas *Rx*, sólo que envían paquetes desde los puertos de entrada/salida en lugar de recibirlos. Las colas *Tx* también se encuentran intercaladas, de modo que los paquetes enviados fuera del *User Data Path* por el puerto *0* son enviados a la cola *Ethernet Tx 0* , y los paquetes enviados fuera del *User Data Path* por el puerto *1* son enviados a la cola *CPU DMA Tx 0*, y así sucesivamente. Los paquetes que se transmiten a la cola *DMA Tx X* salen por la interfaz *nf2cX*.

Para cada uno de estos módulos, hay un conjunto de registros que proveen información del estado, acceso y las señales de control de ajuste [9].

3.2.4. NetFPGA 10G

Desde marzo de 2012 se encuentra disponible la plataforma NetFPGA de 10GB. La tarjeta tiene cuatro puertos de *10 Gigabit Ethernet* con interfaces *SFP+* que le permiten conectar tanto cobre como fibra óptica. Es conectada al computador a través de una interfaz *PCI Express Gen2 x8*. Posee un chip *FPGA Xilinx Virtex-5 TX240T* con un número mucho mayor de compuertas lógicas con respecto a la NetFGPA de 1G. Esta tarjeta cuenta con *27 MBytes* de memoria *QDRII SRAM* y *288 MBytes* de memoria *RLDRAM-II*. Ver Figura 3.10.

[9] https://github.com/NetFPGA/netfpga/wiki/DevelopersGuide#Register_system

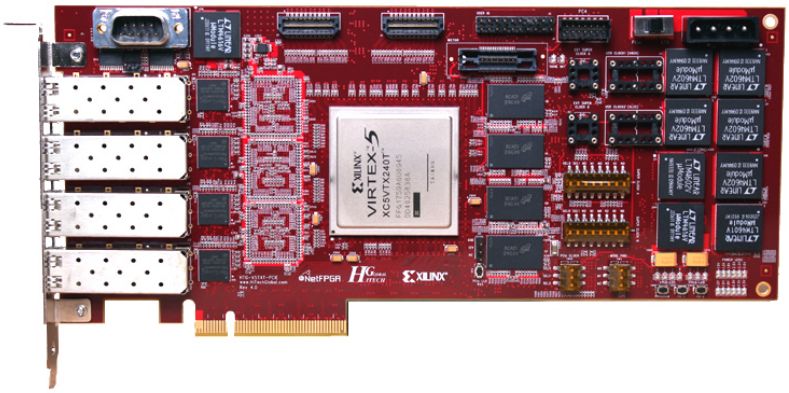


Figura 3.10. Tarjeta NetFPGA 10G.

Especificaciones mas detalladas tomadas del fabricante [10] son las siguientes:

- FPGA Xilinx Virtex-5 XC5VTX240T-2FFG1759C
- Cuatro interfaces SFP+ (usando 16 RocketIO GTX transceivers y 4 puertos Broadcom EDC) que soportan modos de operación de 10Gbps o de 1Gbps
- Conector PCI Express X8 Gen 2
- Veinte Transceivers GTX seriales configurables
- Tres x36 Cypress QDR II (CY7C1515JV18)
- Cuatro x36 Micron RLDRAM II (MT49H16M36HT-25)
- Dos Xilinx Platform XL Flash (128mb each)
- Un Xilinx XC2C256 CPLD
- Un puerto DB9 (RS232)
- Dimensiones: 9.5"x 4.25"

3.2.5. NetFPGA CML

Al igual que versiones anteriores la NetFPGA CML es una plataforma de desarrollo de *hardware* de red versátil y de bajo costo. Cuenta con una *FPGA Xilinx Kintex-7 XC7K325T-1FFG676* e incluye cuatro interfaces *Ethernet* capaces

[10] Disponible en la Web: http://www.hitechglobal.com/boards/PCIExpress_SFP+.htm

de operar a conexiones de hasta *1 Gbps*. *512MB DDR3* a *800 MHz* soportan almacenamiento de paquetes de alto rendimiento, mientras que *4.5 MB* de *QDRII+* mantienen acceso de baja latencia a los datos de demanda alta, como tablas de enrutamiento. La configuración de arranque rápido es apoyada por una *Flash BPI de 128MB*, que también está disponible para aplicaciones de almacenamiento no volátil. El conector *FMC* proporciona una interfaz de expansión, conveniente para ampliar las funcionalidades de la tarjeta a través *Select I/O* e interfaces seriales *GTX*. El conector *FMC* puede soportar velocidades de datos *SATA-II* para el almacenamiento de aplicaciones de red y también utilizarse para extender la funcionalidad a través de una amplia variedad de otras tarjetas diseñadas para la comunicación, medición y control.

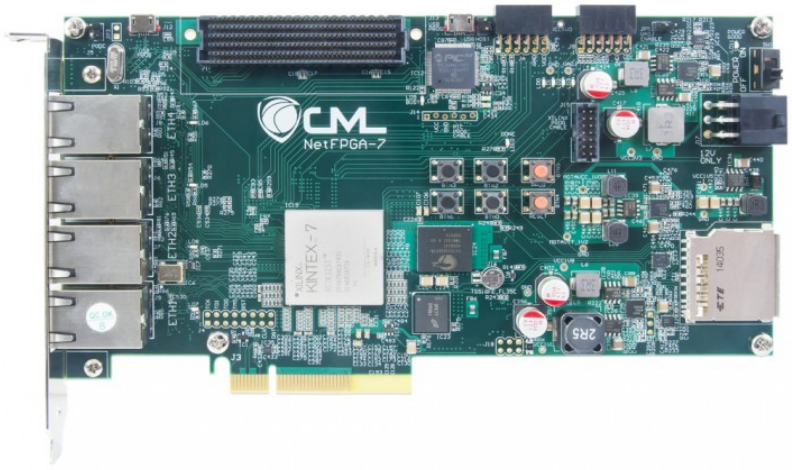


Figura 3.11. Tarjeta NetFPGA CML.

Mayores especificaciones de la tarjeta son las siguientes:

- FPGA *Xilinx Kintex-7 XC7K325T-1FFG676*
- Conector *PCI Express X4 Gen 2*
- Flash BPI de 1Gbit
- Slot para tarjeta SD
- Reloj de tiempo real

- Chip de Cryptoautenticación
- Puede ser conectada dentro o fuera de un PC a través de una fuente de alimentación *ATX* estándar con conector *PCIe* de 6 pines o de 6 pines +2.
- RAM estática *X16 4.5 MB QDRII+ (450 MHz)*
- RAM dinámica *X8 512 MB DDR3 (800 MHz)*
- Oscilador *Low-jitter* a *200 MHz*
- Cuatro conectores 10/100/1000 *Ethernet* con *RGMII*
- Micro-controlador *PIC* de *32-bit*
- Micro-controlador *USB*
- Cuatro LEDs *on-board* y cuatro botones de propósito general *on-board*
- Dos puertos *Pmod*

3.2.6. NetFPGA SUME

La NetFPGA SUME es una tarjeta *PCI Express* basada en FPGA con capacidades de E/S para operar a velocidades de 10 y 100 *Gbps*. Contiene un adaptador *PCIe x8 Gen3* que incorpora una *FPGA Virtex-7 690T* de Xilinx. Puede ser usada como una tarjeta de interfaz de red (NIC), switch multipuerto, *firewall*, entorno de prueba/medición y muchas más aplicaciones.

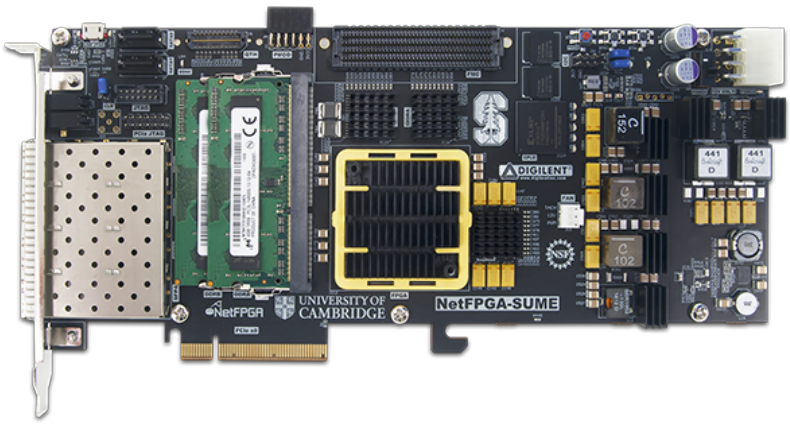


Figura 3.12. Tarjeta NetFPGA SUME.

Mayores especificaciones de la tarjeta son las siguientes:

1. Field Programmable Gate Array (FPGA) lógica
    - Xilinx Virtex-7 690T
    - 693.120 células de lógica
    - 52.920 Kbit bloque RAM
    - hasta Kbit 10.888 distribuido RAM
    - 30 emisoras GTH (hasta 13.1Gbps)
    - Totalmente programable por el usuario
2. Puerto de red 10-gigabit Ethernet
    - Bloque conector de PCB izquierda interfaces para 4 puertos externos SFP+
    - Conectado directamente a la FPGA.
    - Procesamiento de velocidad de cable en todos los puertos en todo momento mediante lógica FPGA.
3. Memoria de acceso aleatorio estática de tasa datos cuádruple (QDRII + SRAM)
    - Conveniente para el almacenamiento y reenvío de datos de la tabla
    - Tarifa de datos 500MHz Quad (2 Giga transacciones cada segundo), sincrónico con la lógica
    - Tres bancos paralelos de 72 QDRII MBit + recuerdos
    - Capacidad total: 27 MBytes
    - Ciprés: CY7C25652KV18-500BZC
4. Memoria de acceso aleatorio de una cita doble tasa (DDR3 DRAM)
    - Conveniente para los búferes de paquete
    - Dos módulos SoDIMM DDR3 reemplazables
    - Reloj de 933MHz (1866MT/s)
    - 238.8 rendimiento de memoria de pico Gbps
    - Capacidad total: 8 GB (soporta hasta 32 GB)
    - Micrón: MT8KTF51264HZ-1G9E5

5. PCI Express gen 3
    - Tercera generación de la interfaz de PCI Express, 8Gbps/carril
    - 8 carriles (x 8)
    - IP duro
    - Proporciona acceso de CPU a registros mapeados en memoria y memoria en el hardware de NetFPGA
6. Interfaces de expansión
    - Totalmente compatible con conector de VITA-57 FMC HPC, incluyendo 10 enlaces serie de alta velocidad
    - Conector SAMTEC QTH-DP, conectado a 8 enlaces serie de alta velocidad
    - Permite para conectar adicional 18 transceptores GTH
    - Conector de expansión Digilent PMOD
7. Almacenamiento de información
    - 2 conectores SATA
    - Ranura para micro SD
    - 2 dispositivos FLASH, cada 512Mbit (1Gbit total)
8. Características adicionales
    - Circuito de recuperación del reloj
    - Sensores de voltaje
    - Sensores de corriente
    - LEDs de usuario & los botones
9. Factor de forma estándar PCIe
    - Tarjeta PCIe estándar
    - Longitud total, altura completa
10. Flexible, código de fuente abierta

3.3. Libpcap

Libpcap es una librería *open source* escrita en C que provee una interfaz de alto nivel al sistema de captura de paquetes de red. Fue creada en 1994 por McCanne, Leres y Jacobson (investigadores del Laboratorio Nacional Lawrence Berkeley de la Universidad de California) como como parte de un proyecto de investigación para mejorar el rendimiento de *TCP* e *Internet gateways*.

El principal objetivo de los autores de Libpcap era crear una API independiente de la plataforma y de está manera eliminar la dependencia de los módulos de análisis, captura y construcción de paquetes de red de la aplicación del sistema operativo, púes cada uno implementaba sus propios mecanismos Jacobson et al. (1994).

La API libpcap es diseñada para ser usada desde C y C++. Sin embargo hay varios *ports* que permiten su uso desde lenguajes como Perl, Python, Java, C# o Ruby. Libpcap corre sobre la mayoría de sistemas operativos tipo UNIX (Linux, Solaris, BSD, HP-UX, entre otros). También existe una versión para Windows llamada Winpcap. En la actualidad la API libpcap es mantenida y desarrollada por el Tcpdump Group. El código fuente y la documentación completa está disponible en la Web oficial de tcpdump [11].

3.3.1. ¿Por qué usar Libpcap?

Frecuentemente las aplicaciones de red acceden al medio a través de llamadas al sistema operativo conocidas *sockets* [12]. Es fácil acceder a los datos en la red con este enfoque debido a que el sistema operativo se encarga de los detalles de bajo nivel (Manejo de protocolos, re-ensamblaje de paquetes, entre otros) y proporciona una interfaz familiar, similar a la utilizada para leer y escribir archivos.

[11] *http://http://www.tcpdump.org/* para Windows *(http://www.winpcap.org)*

[12] El término socket es también usado como el nombre de una interfaz de programación de aplicaciones (API) para la familia de protocolos de Internet TCP/IP, provista usualmente por el sistema operativo.

A veces, sin embargo, el camino fácil no es la tarea, ya que algunas aplicaciones requieren del acceso directo a los paquetes. Es decir, necesitan acceso a los datos "crudos"de la red sin la interposición del procesamiento de protocolos hechos por el sistema operativo.

El propósito de Libpcap es dar este tipo de acceso a las aplicaciones y ofrece facilidades para:

- Capturar paquetes crudos, paquetes destinados a la máquina donde se ejecuta y los que intercambia con otros hosts.(en medios compartidos).
- Transmitir paquetes crudos a la red.
- Filtrar paquetes según las reglas especificadas por el usuario antes de enviarlos a la aplicación *(BSD Packet Filter)*.
- Reunir información estadística sobre el tráfico de la red.

Este conjunto de características se obtiene mediante un controlador de dispositivo, que es instalado en la porción de red dispuesta dentro del *kernel* y son exportadas a través de una interfaz de programación bien definida, fácilmente explotable por las aplicaciones y disponible para diferentes sistemas operativos.

3.3.2. Tipo de programas de hacen uso Libpcap

La interfaz de programación de Libpcap puede ser utilizada por muchos tipos de herramientas de red para el análisis, seguridad, monitoreo y solución de problemas. En particular, son herramientas clásicas que se basan en Libpcap:

- Analizadores de red y protocolos
- Monitores de red
- Registradores de tráfico
- Generadores de tráfico
- Bridges y enrutadores de nivel de usuario
- Sistemas de detección de intrusiones de red (NIDS)

- Escáneres en red
- Herramientas de seguridad

El Cuadro 3.2 muestra algunos de los programas que hacen uso de esta API Morillo (2011).

3.3.3. Esquematización de un programa

Todo programa que hace uso de Libpcap, sin importar su grado de complejidad, seguirá el esquema mostrado en la Figura 3.13 Garcia (2008)

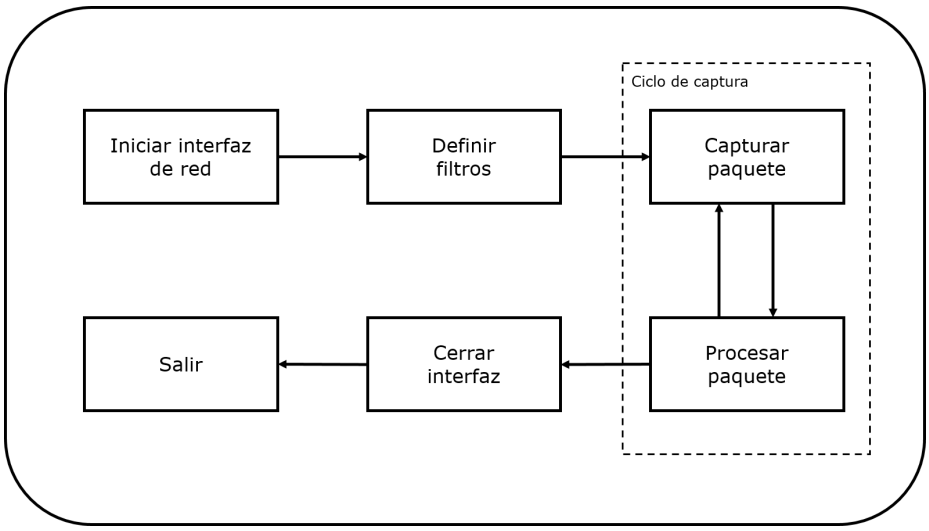


Figura 3.13. Esquema general de un programa que hace uso de Lipcap.

Iniciar interfaz de red

Para obtener información del sistema, la aplicación debe iniciar las interfaces y ejecutar sobre estas funciones capaces de recopilar características propias de la máquina. Una vez la aplicación conoce las interfaces de red instaladas y las configuraciones de estas interfaces (mascara de red, dirección de red, . . .) el programa puede pasar a la siguiente etapa.

Tabla. 3.2

Lista de programas que utilizan Libpcap

Herramienta	**Descripción**
tcpdump	Herramienta de captura y dumping de paquetes para análisis posteriores.
ngrep	También conocido como "network grep", aísla strings en los paquetes, muestra los datos del paquete en formato legible.
Wireshark	Herramienta gráfica de captura de paquetes y análisis de protocolos.
Snort	Sistema de detección de intrusos de red.
Nmap	Utilidad de red de escaneo de puertos y fingerprinting.
Bro IDS	Plataforma de monitoreo de red.
URL Snooper	Utilidad para localizar las direcciones URL de archivos de audio y video para posteriormente grabarlas.
L0phtCrack	Aplicación de auditoría y recuperación de contraseñas.
iftop	Herramienta para la visualización de uso de ancho de banda.
EtherApe	Herramienta gráfica para el control de uso de tráfico y ancho de banda en tiempo real.
Bit-Twist	Generador de paquetes Ethernet basado en libpcap y editor para Windows, Linux y BSD.
NetSim	Software de simulación de red para redes R&D.
XLink Kai	Software que permite a varios juegos de consola LAN jugar online.
Firesheep	Extensión para el navegador Firefox que captura paquetes y realiza Hijacking.
Suricata	Plataforma de análisis y prevención de intrusiones en red.
WhatPulse	Aplicación de medición estadística. (entrada, red, tiempo de actividad).
Xplico	Herramienta de análisis forense de red (NFAT).
Scapy	Herramienta para la manipulación de paquetes de red escrita en Python.
GNS3	Software de simulación de red utilizando imágenes reales de Cisco IOS.

Definir filtros

Libpcap es una librería de funciones y los procesos que ejecutan estas funciones, lo hacen a nivel de usuario *(user space)*. Sin embargo la captura real de datos tiene lugar en las capas inferiores del sistema operativo, en la llamada zona del *kernel (Kernel area)*, por lo tanto debe existir un mecanismo capaz de traspasar esta frontera de un modo eficiente y seguro, ya que cualquier fallo en capas tan profundas degradaría el rendimiento de todo el sistema.

Supongamos que nos interesa programar una aplicación capaz de monitorear la red en tiempo real en busca de paquetes TCP cuyo puerto destino sea el *8080*. Si no existiese ningún mecanismo de filtrado, el *kernel* no sabría cuáles son los paquetes en los que está interesada la aplicación, por lo que tendría que traspasar la frontera *kernel-user space* por cada paquete que transita la red. Para evitar esto, la aplicación establece filtros en la zona *kernel* que solo dejan pasar los paquetes cuyo puerto TCP destino sea el *8080*. Esta es la principal labor de un filtro o *Packet/Socket Filter*. Actualmente no existe un sistema único de filtrado, por el contrario prácticamente cada sistema operativo reescribe su propia solución: NIT para SunOS, Ultrix Packet Filter para DEC Ultrix, BPF para sistemas BSD y LSF para Linux. Se profundizará únicamente en BPF por ser el más extendido y la arquitectura base para la creación de los demás.

El funcionamiento de BPF se basa en dos grandes componentes: El *Network tap* y el *Packet Filter*. El primero es el encargado de recopilar los paquetes desde el driver del dispositivo de red y entregárselos a aquellas aplicaciones a las que vaya destinado. El segundo es el encargado de decidir sí el paquete debe ser aceptado y en caso afirmativo, cuanto de ese paquete debe ser entregado a la aplicación McCanne and Jacobson (1993).

En la Figura 3.14 puede verse el esquema general de funcionamiento de BPF dentro del sistema. Cada vez que llega un paquete a la interfaz de red, el *driver* de red lo envía hacia la pila de protocolos *(protocol stack)* siempre y cuando no esté

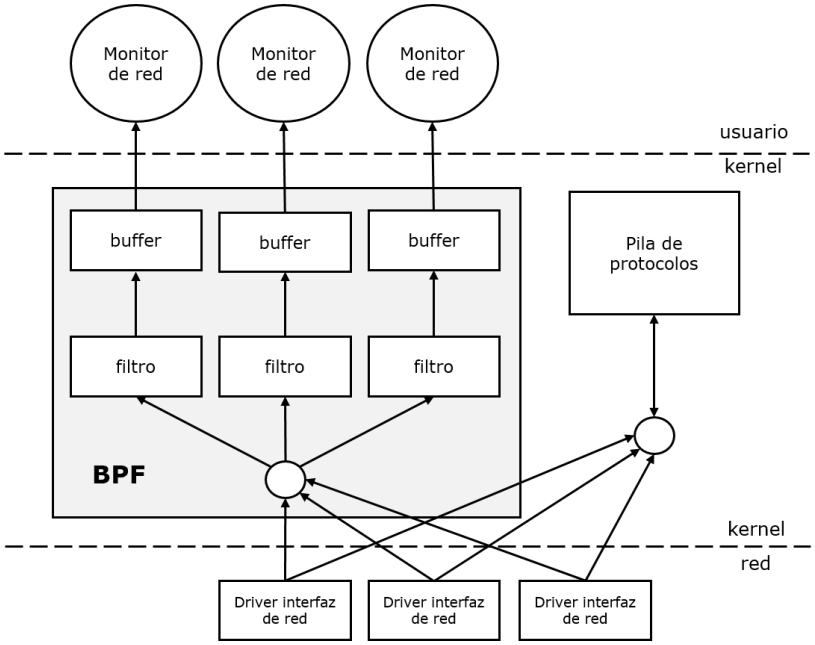


Figura 3.14. Diagrama del funcionamiento de BPF.

activo BPF en cuyo caso antes de enviarlo a la pila, el paquete debe ser procesado por él.

BPF será el encargado de comparar el paquete con cada uno de los filtros establecidos, pasando una copia de dicho paquete a cada uno de los *buffers* de las aplicaciones cuyo filtro se ajusta a los contenidos del paquete. En caso de que no exista ninguna coincidencia, el paquete será reenviado al *driver*, el cual actuará normalmente, es decir, si el paquete está dirigido a la propia maquina lo pasará a la pila de protocolos y en caso contrario lo descartará sin que el paquete haya salido en ningún momento de la zona *Kernel*.

Los filtros BPF están escritos en un lenguaje especial similar al *assembly* [13]. Sin embargo, libpcap y tcpdump implementan un lenguaje de alto nivel que permite

[13]Lenguaje de programación de bajo nivel para los computadores, microprocesadores, microcontroladores y otros circuitos integrados programables. Implementa una representación simbólica de los códigos de máquina binarios y otras constantes necesarias para programar una arquitectura dada de CPU y constituye la representación más directa del código máquina específico para cada arquitectura legible por un programador.

definir filtros de una manera mucho más fácil. La sintaxis especifica de este lenguaje está fuera del alcance de este documento.

Una vez definido, traducido a lenguaje BPF e insertado a el *kernel* el filtro es pueda iniciar a cumplir sus funciones.

Capturar paquete

Una vez la aplicación conoce listado de las interfaces del sistema, sus configuraciones, y se han definido los filtros deseados la librería está preparada para comenzar con la captura de paquetes. Existen varias funciones de capturar que se diferencian principalmente por:

- Número de paquetes que se desea capturar.
- Modo de captura (normal o promiscuo)
- Manera en que se definen sus funciones de llamada o *callbacks* (Función invocada cada vez que se captura un paquete).

Procesar paquete

Cuando un paquete es capturado lo único que la aplicación ha obtenido es un grupo de *bytes*. Usualmente el *driver* de la tarjeta de red y la pila de protocolos son los encargados de procesar estos datos pero la captura de paquetes desde la aplicación se da a partir los niveles mas bajos, por lo tanto esta debe ser la encargada de hacer los datos inteligible, es decir extraer las cabeceras añadidas por el remitente e interpretar los campos propios del paquete.

Cerrar interfaz

Terminado el ciclo de captura y procesado de paquetes es necesario inhabilitar los procesos adicionales que fueron cargados al *kernel* y de esta manera dejar al *driver* de la tarjeta de red funcionar normalmente.

Salir

Antes de salir de la aplicación, Libpcap ofrece la posibilidad de volcar los datos a un fichero, para esto dispone de diversas funciones que se adaptan a las necesidades de procesado requerido del usuario, algunas de estas funciones son:

- `pcap_dumper_t *pcap_dump_open(pcap_t *p, char *fname)`
- `pcap_open_offline(filename, *errbuf)`
- `void pcap_dump(u_char *user, struct pcap_pkthdr *h, u_char *sp)`
- `void pcap_dump close(pcap_dumper_t *p)`

Capítulo 4

Estado de arte

La estimación del ancho de banda disponible de extremo a extremo ha sido estudiada por varios investigadores alrededor del mundo. Existen trabajos en el área que resumen las técnicas y conceptos relacionados con estimaciones en Internet de medidas como capacidad, ancho de banda y otras relacionadas Jain and Dovrolis (2002a); Michaut and Lepage (2005).

Las principales herramientas de estimación que han sido desarrolladas hasta el momento se basan en las dos técnicas de estimación descritas anteriormente. Bajo el *Probe Rate Model*, las herramientas más representativas son *Pathload* Jain and Dovrolis (2002b), *Pathchirp* Ribeiro et al. (2003), *Bart* Hartikainen et al. (2005), *Yaz* Sommers et al. (2007) y *Train of Packet Pair (TOPP)* Melander et al. (2000, 2002). Bajo el *Probe Gap Model*, las herramientas más representativas son *Traceband* Guerrero and Labrador (2008, 2010b), *Spruce* Strauss et al. (2003a), *Abing* Navratil and Cottrell (2003) y *IGI* Hu and Steenkiste (2003).

Existen igualmente estudios comparativos que revelan aspectos diferenciadores de cada una de las herramientas cuando se pone a prueba en diferentes escenarios de red y bajo diferentes tipos de tráfico cruzado Guerrero and Labrador (2006); Shriram et al. (2005). La complejidad del problema ha llevado a varios autores a hacer

consideraciones erradas que se ven reflejadas en el comportamiento de sus respectivas herramientas de estimación Jain and Dovrolis (2004).

Dadas los tiempos de respuesta, errores altos de estimación y cantidad de paquetes de prueba que se introducen en la red, la aplicabilidad de las herramientas de estimación de ab_disp es y es aún tema de investigación Guerrero and Labrador (2010a).

En la vía de reducir errores de estimación, se han utilizado, de manera relativamente exitosa, técnicas no tradicionales en redes de computadoras. Tal es el caso de las técnicas de aprendizaje de máquina muy comunes en problemas de reconocimiento de patrones y de inteligencia artificial en general. Ejemplo de ello es el uso de modelos escondidos de Markov Guerrero and Labrador (2008) y de filtros de Kalman Hartikainen and Ekelin (2006). Estas técnicas intentan generar un modelo que "aprenda" o sea alimentado por cada una de las observaciones tomadas del ancho de banda disponible. Otra técnica de aprendizaje de máquina no explorada en estimación de ancho de banda disponible, objeto de un proyecto del investigador principal, es el uso de algoritmos de agrupamiento con la pretensión de reducir la probabilidad de tener muestras ruidosas del ancho de banda disponible.

Una técnica que apunta a la estimación usando paquetes de datos en lugar de paquetes extra de prueba es *ImTCP* Le Thanh Man et al. (2006). Los autores proponen una versión de TCP en la que modifican el sistema operativo para que desde el *kernel* se manipulen los paquetes de datos y se envíen a tasas ajustadas según una herramienta de estimación. Esta técnica, sin embargo, está sujeta a la operación del sistema operativo lo cual no garantiza la prioridad de los procesos de envío de paquetes.

Se puede derivar del estado del arte que las técnicas existentes se basan en envío de paquetes de prueba a través del *software* de estimación. Esto conlleva a diferentes errores mencionados en la literatura por diferentes autores Michaut and Lepage (2005); Paxson (1997); Zhou et al. (2006). La imposibilidad del *software* de

la máquina para dar prioridad al proceso de envío de paquetes hace que se parta de tiempos de transmisión que no corresponde a lo que teóricamente debería darse.

De esta manera, el presente proyecto plantea una novedad que contrasta con lo realizado en la literatura: la manipulación de los tiempos de envío de paquetes desde el *hardware* de red gracias al uso de NetFPGA.

Capítulo 5

Descripción del proceso investigativo

El presente proyecto, se enmarca metodológicamente dentro del enfoque de la investigación aplicada, desarrollada a través del método científico. De tal modo, está basada en la experimentación de una herramienta de estimación de ancho de banda disponible que genera y envía paquetes de prueba a tiempos precisos en un ambiente de red controlado.

El proyecto se desarrollará en cinco etapas las cuales se vinculan con cada uno de los objetivos específicos del proyecto.

5.1. Estudio y caracterización de mecanismos de modificación de tiempos de transmisión

Con el fin de establecer diseños tanto a nivel de *hardware* como de *software* que realicen variaciones en la transmisión de paquetes, se realizó una revisión de los mecanismos existentes que puedan ser base para la implementación en NetFPGA.

5.1.1. Diseños a nivel de *hardware*

De acuerdo a la revisión de literatura realizada se identificaron dos proyectos implementados sobre la NetFPGA los cuales realizan manipulación en los tiempos de envío y permiten la captura de paquetes con exactitud de nanosegundos, lo que cual evita el porcentaje de error agregado a las herramientas de estimación por las limitaciones propias del uso de herramientas a nivel de *software*. Estas herramientas son analizadas en la Sección 5.3.2.

5.1.2. Diseños a nivel de *software*

Las herramientas objetivo a analizar fueron los generadores de paquetes que realizarán procesos de manipulación en el envío de paquetes con el fin de evitar los errores impuestos por las máquinas *hosts*. Dentro de los estimadores analizados (Sección 5.2.1), resaltan la API de red o módulo kernel *OMware* Mok et al. (2015) e *ICIM* Le Thanh Man et al. (2006); Man et al. (2008) las cuales son analizadas en la Sección 6.1.

5.2. Selección y modificación de un estimador de ancho de banda existente

5.2.1. Herramientas de estimación de ancho de banda

Las herramientas para la estimación del ancho de banda disponible o estimadores de ancho de banda disponible (en adelante ABET), son aplicaciones que utilizan *PGM*, *PRM* o la combinación de los dos como metodología de medición. El Cuadro 5.1 muestran las diferentes ABETs analizadas:

Tabla. 5.1

Estimadores de ancho de banda analizados

Nombre	Autores	Año	Técnica		Tipo		Metodología
			PRM	*PGM*	*SW*	*HW*	
Cprobe	Carter, Crovella	1996		•	•		Tren de paquetres
Delphi	Ribiero, Riedi, Sarvotham, Coates, Brent, Baraniuk	2000		•	•		Trenes de paquetes
TOOP	Melander, Bjorkman, Gunningberg	2000	•		•		Trenes de paquetes
Pathload	Jain, Dovrolis	2002	•		•		Self-Loading Periodic Streams
IGI/PTR	Hu, Steenkiste	2002		•	•		Self-Loading Periodic Streams
Spruce	Strauss, Katabi, Kaashoek	2003		•	•		Trenes de paquetes
Abing	Navratil, Cottrell	2003		•	•		Trenes de paquetes
pacthChirp	Ribeiro, Riedi, Baraniuk, Navratil, Cottrell	2003	•		•		Self-Loading Packet Chirps
Netest	Jin, Tierney	2003	•		•		Trenes de paquetes
ProbeGap	Lakshminarayanan, Padmanabhan, Padhye	2004		•	•		Trenes de paquetes
DietTOPP	Johnsson, Melander, Björkman	2004	•		•		Trenes de paquetes
PathMon	Kiwior, Kingston, Spratt	2004	•		•		Self-Loading Packet Chirps
PoissonProb	Xin	2005	•		•		Trenes de paquetes
Yaz	Sommers, Barford, Willinger	2006	•		•		Trenes de paquetes
BART	Ekelin, Nilsson, Hartikainen, Johnsson	2006	•			•	Trenes de paquetes
MoSeab	Zhang, Luo, Li	2006		•	•		Trenes de paquetes
IGMPS	Ali, Lepage	2007		•	•		Trenes de paquetes
FEAT	Wang, Cheng	2008	•		•		Trenes de paquetes
Assolo	Goldoni, Rossi, Torelli	2009	•		•		Reflected Exponential Chirp
PathAB	Roy	2009	•	•	•		Trenes de paquetes
Traceband	Guerrero, Labrador	2010		•	•		Trenes de paquetes

5.2.2. Comparación de las herramientas de estimación de ancho de banda disponible

En la revisión de la literatura de los ABETs analizados en el Cuadro 5.1, se evidencian que *PRM* es una metodología cuyas estimaciones de ab_disponible son más precisas que *PGM* pero, debido a que utilizan el principio de congestión autoinducida, el tiempo para lograr una estimación es considerablemente más alto a comparación de *PGM*, así como la posibilidad de causar una interrupción en los espaciamientos de los paquetes de prueba e invalidar la medición. *PGM* es menos intrusivo que *PRM*, cada par de paquetes de prueba logra una estimación y los tiempos de iteración de la herramienta son cortos Santos (2015). Aunque *Traceband* Guerrero and Labrador

(2010b) logra estimaciones similares a las de *Pathload* Jain and Dovrolis (2002b), en general, las herramientas basadas en *PGM* son menos precisas, tienen problemas para calcular la dispersión de los paquetes y requieren del conocimiento previo de la capacidad del *tight link* para lograr una adecuada estimación de ab_disponible.

Dentro de la revisión de la literatura realizada, las herramientas más representativas de *PGM* y *PRM*, son respectivamente *Spruce* y *Pathload* Guerrero and Labrador (2010b); Hu and Steenkiste (2003); Ali and Lepage (2007); Sommers et al. (2006). Aunque *Pathload* Jain and Dovrolis (2002b) es uno de los ABETs caracterizado por su exactitud y en el cual los trenes de paquetes de prueba varían adaptativamente para calcular el ab_disponible, sus limitaciones están en el alto tiempo empleado para proporcionar una estimación Ribeiro et al. (2003) y la sobrecarga en el enlace es considerable. Cuando ocurre la sobrecarga, hay una mayor posibilidad de generar cambios en el contexto del sistema causando una interrupción en los espaciamientos de los paquetes de prueba que llegan al receptor, e incluso invalidar la medición. Además de lo anterior, *Pathload* tiende a sobrestimar el ab_disponible; es propenso a sesgos de medición y errores en entornos donde el tráfico cruzado varía y donde las condiciones de la red son cambiantes. Según Hu and Steenkiste (2003), al realizarse mediciones en Internet y simulaciones de tipo ns, la medición se ve afectada por factores tales como el tamaño de los paquetes de prueba, la longitud del tren de paquetes y el comportamiento del estimador cuando el cuello de botella es grande, lo que genera errores de estimación que pueden llegar al *30 %*.

Spruce, es un ABET caracterizado por ser considerablemente más rápido y menos intrusivo que *Pathload* Strauss et al. (2003b). Su algoritmo de estimación envía pares de paquetes espaciados de acuerdo a la capacidad del vínculo estrecho, lo anterior supone que la capacidad de enlace ajustado se debe conocer previamente. *Spruce* no admite estimaciones de ancho de banda disponible menores a cero aunque son posibles. Otra de las limitaciones ocurre cuando las condiciones del tráfico cruzado sufren cambios rápidos, situación en la cual *Spruce* no es capaz de reaccionar lo suficientemente rápido. Todo lo anterior hace que *Spruce* sea menos preciso que

Pathload. El uso de estas herramientas se ha limitado a aplicaciones de gestión de red de redes cableadas, donde la precisión, la sobrecarga y los tiempos de convergencia no son tan estrictos como en otras aplicaciones o entornos de red Guerrero and Labrador (2010b).

5.2.3. Herramienta seleccionada: *Traceband*

Traceband Guerrero and Labrador (2010b), es una herramienta cliente-servidor (emisor-receptor) escrito en C que utiliza modelos ocultos de Markov (en adelante HMM) para proporcionar estimaciones de ab_disponible de forma rápida, continua y precisa, bajo el modelo *PGM* descrito en la técnicas para la estimación del ancho de banda disponible 3.1.4. En *Traceband*, un tren de pares de paquetes de prueba se envía desde la aplicación cliente al servidor a la velocidad del *tight link*. Después de interactuar con el tráfico cruzado en el *tight link*, cada par de paquetes en el tren proporcionará un único valor de dispersión que se constituye en una observación en la secuencia del *HMM*. En el servidor, la aplicación procesa la secuencia de observación, estima la secuencia de estados que genera las observaciones y proporciona una única estimación promediada del ancho de banda disponible. Según Guerrero and Labrador (2010b), los resultados experimentales demuestran que el implementar *HMM* permite a *Traceband* proporcionar estimaciones rápidas y precisas con una tasa de tráfico de prueba baja y con la capacidad de reaccionar con rapidez y precisión a las variaciones del tráfico cruzado, como los presentes en enlaces cargados con tráfico a ráfagas.

Luego de realizar la comparación de los estimadores de ancho de banda disponibles presentados en el Cuadro 5.1, desde el punto de vista de la revisión de la literatura, se pudo evidenciar que aunque *Traceband* está basado en *PGM*, permite estimaciones de ab_disponible tan precisas como *Pathload* o los ABETs basados en *PRM*, pero de una manera más rápida y menos intrusiva.

Al igual que las herramientas analizadas en la sección 5.2.2, *Traceband* posee limitaciones que son comunes entre las herramientas analizadas, las cuales comprometen

el rendimiento, el desempeño y la precisión en la estimación Santos (2015). Dentro de las limitaciones encontradas están:

- Rendimiento y precisión impuestos por el sistema operativo. Estos problemas están asociados a la imposibilidad para dar prioridad a la herramienta de estimación, lo que ocasiona que los tiempos de transmisión no correspondan a los tiempos solicitados especificados por el ABET.
- Interfaces de red estándar las cuales son utilizadas por los *Host* que ejecutan las herramientas de estimación. Al no estar diseñadas para para generar paquetes de prueba y enviarlos de forma precisa, ocasionan registros incorrectos en los tiempos de salida o llegada de paquetes e incluso, imposibilitan a transmisión de las interfaces de red a la velocidad requerida por el ABET.
- Recepción fuera de orden, replicación, corrupción de paquetes, el comportamiento de los servicios de colas FIFO o LIFO de los enrutadores.
- Sobrecarga de la red por la transmisión de los paquetes de prueba.
- El comportamiento del tráfico cruzado y la especificación de un conjunto de parámetros antes de ejecutar la aplicación de estimación, afectan la precisión de los ABETs.

5.2.4. Modificaciones al código de *Traceband*

El código de las variaciones a la herramienta de estimación seleccionada y su interacción con la tarjeta NetFPGA 1G son mostradas a detalle en la Sección 6.2.

5.3. Implementación del módulo NetFPGA para variación de tiempos de transmisión

5.3.1. Generadores de tráfico

La generación e inyección de tráfico basadas en software más representativas han sido desarrolladas utilizando la plataforma *GNU/Linux*. Según Blanco et al. (2012), la principal limitación de las herramientas de este tipo está en la falta de granularidad [1] a la hora de planificar los eventos necesarios para el modelador el tráfico. Para solucionar esta limitante, algunas soluciones como *nCap* Deri (2005), se basan en inyectar tráfico sobre la interfaz de red desde aplicaciones de usuario, lo que simplifica el costo computacional que supone el uso del subsistema de red desde el *kernel*. Otras herramientas, como por ejemplo *Pkt-Gen* Olsson (2005), aumentan el rendimiento de la aplicación realizando la generación e inyección del tráfico directamente desde el *kernel*, debido a que los procesos tienen mayor prioridad que los procesos realizados en aplicaciones en espacio de usuario. Aunque las soluciones propuestas por Deri (2005) y Olsson (2005) están orientadas a reducir los retardos en la ejecución de la aplicación relacionados a la carga de la *CPU*, el problema de la granularidad sigue presente, lo que provoca que los tiempos de inyección de los paquetes no sean los deseados e impide además, saturar el enlace a velocidad de línea Botta et al. (2010). La imposibilidad de enviar los paquetes de prueba a los tiempos y capacidad real de las interfaces de red, fue también una limitante encontrada en los ABETs de tipo *software*.

En *hardware*, existen dos tipos de sistemas que responden a dos filosofías opuestas: sistemas comerciales no reprogramables y sistemas reconfigurables. Éstos últimos son los que suelen ser elegidos para el desarrollo de prototipos de investigación Blanco et al. (2012). Dado que no requieren de la ejecución de procesos a nivel de usuario o de *kernel* (salvo parámetros de configuración), los generadores basados en textithardware

[1] Granularidad hace referencia a la cantidad de cómputo con relación a la comunicación.

permiten obtener granularidades del orden de varios nanosegundos. La principal limitación del uso de sistemas comerciales, son la imposibilidad de modificación (especialmente para el desarrollo de proyectos de investigación) y el elevado costo de la solución, los cuales hacen que no siempre sea una opción viable. La principal característica de estos sistemas es que consiguen inyectar el tráfico a la capacidad máxima de la interfaz de red y respetan los tiempos entre paquetes.

Además de las soluciones presentadas previamente, existen soluciones mixtas las cuales integran características de *software* y de *hardware* al mismo tiempo. Entre los sistemas mixtos el más conocido es *DAG (Data Adquisition and Generation)* Tockhorn et al. (2011), el cual está basado en una arquitectura FPGA con acceso *PCI* o *PCI-Express* dependiendo del modelo. *DAG* se configura y se controla desde una aplicación *software* ejecutada en el espacio de usuario y el *hardware* FPGA se encarga del envío del tráfico en los tiempos solicitados. Según *blancoarquitectura*, la aplicación permite inyectar ventanas de tráfico preestablecido, pero no cuenta con la capacidad de modelar los tiempos entre paquetes, la granularidad de la inyección se ve afectada por los problemas de rendimiento de los sistemas *software* y los paquetes que inyecta *DAG* solo implementan el protocolo *Ethernet* y carecen de la capacidad de reprogramar libremente el *hardware* para crear configuraciones personalizadas.

5.3.2. Generación de paquetes de prueba sobre NetFPGA

Dado que dos de los problemas identificados entre los estimadores de ancho de banda disponibles analizados en la Sección 5.2.1 hacen énfasis al rendimiento y precisión impuestos por el sistema operativo asociados a la imposibilidad para dar prioridad al proceso de envío de paquetes, lo que ocasiona que los tiempos de transmisión no correspondan a los tiempos especificados por el ABET e interfaces de red estándar que no están diseñadas para generar paquetes de prueba y enviarlos de forma precisa, la implementación de dispositivos de red basados en NetFPGA es una solución a dicho problema dada su capacidad de generar y transmitir paquetes a la

velocidad de línea e incluso de establecer conexión directa con las herramientas de estimación sin necesidad de pasar por el procesamiento del sistema operativo.

Dentro de los diferentes desarrollos que se han realizado utilizando NetFPGA orientados a la generación de paquetes de prueba, los proyectos *NetFPGA Packet Generator* (en adelante NPG) y *Caliper* son los más representativos. NPG Covington et al. (2009a), implementa un generador de paquetes (en el servidor) y un sistema de captura de tráfico (cliente) en el cual, la aplicación está en capacidad de reproducir con precisión tráfico sensible al tiempo a velocidad de línea con base en un archivo *PCAP* transferido a la memoria local en la tarjeta NetFPGA. En primera instancia, el servidor NPG envía los paquetes a través de los enlaces *Gigabit Ethernet* disponibles en la NetFPGA con base en la tasa transmisión, retardo entre paquetes y el número de iteraciones especificados por el usuario, mientras que el cliente captura el tráfico enviado registrando las marcas de tiempo y el tráfico resultante para luego ser transferido de vuelta al servidor, donde puede ser almacenado utilizando un formato *PCAP* para su posterior análisis. Según Blanco et al. (2012), NPG hace uso del *pipeline de referencia* proporcionado por la NetFPGA, permite replicar tráfico representado en ficheros con formato *PCAP*, inyecta tráfico a una tasa de hasta *1Gbps* y respeta los tiempos entre paquetes marcados en el fichero.

Por otro lado, *Caliper* Salmon et al. (2009), no es un generador de tráfico como tal, sino que pretende introducir una capa intermedia entre un generador de tráfico cualquiera y la interfaz de red, con el fin de mejorar la precisión en los tiempos de envío. El objetivo de esta herramienta es controlar de manera precisa los tiempos de transmisión de paquetes generados dinámicamente en el *host* y enviarlos de manera continua (no requiere de precarga de datos) a la tarjeta NetFPGA. Utiliza la plataforma *NetThreads* que permite la elaboración de aplicaciones C multiproceso sobre NetFPGA. Al igual que NPG, implementa un generador de paquetes (en el servidor) y un sistema de captura de tráfico (cliente).

Los dos proyectos mencionados anteriormente aprovechan la generación y recepción de paquetes de prueba de manera precisa a través del *hardware* NetFPGA e

incluso permiten la manipulación los tiempos de envío de paquetes, tamaños, retardo y el número de iteraciones. La implementación de NetFPGA en el proceso de generación de los pares de paquetes (o trenes de paquetes) de prueba para realizar el proceso de estimar el ab_disponible permitirá eliminar las fuentes de errores asociadas al rendimiento y precisión impuestos por el sistema operativo y las interfaces de red Santos (2015).

5.3.3. Selección de la herramienta a implementar en la NetFPGA

Dado que se requiere integrar una herramienta para la estimación del ancho de banda disponible escrita en C con el *hardware* de la NetFPGA, NPG resalta sobre *Caliper-NetThreads* debido a:

- Proyecto desarrollado por el equipo de desarrollo de la platforma NetFPGA, lo cual brinda una amplia documentación necesaria para el entendimiento del sistema.
- La sencillez en su arquitectura modular permite un rápido desarrollo y a su vez cumple con todos los requerimientos necesarios para alcanzar el objetivo de este proyecto.
- Garantiza el envío y recepción de paquetes de prueba a tiempos precisos dictados por *Traceband*.

La implementación del *NetFPGA Packet Generator* es mostrada en la Sección 6.3.

5.4. Configuración de una red de prueba que permita evaluar la efectividad de la solución propuesta

Para poder evaluar las modificaciones hechas a la herramienta de estimación de ancho de banda es necesario crear y configurar una red de prueba con equipos NetFPGA y PC's actuando como nodos extremos en dos redes comunicadas a través de dos enrutadores.

5.4.1. Componentes de la red de prueba

Los componentes usados en la red de prueba pueden fácilmente dividirse en tres grupos:

1. Nodos finales o End-hosts.
2. Dispositivos de interconexión de redes.
3. Otros elementos (necesarios para la disposición y conexión de los las dos anteriores grupos).

El Cuadro 6.5 hace referencia a los componentes y referencias de los dispositivos usados en el montaje del Testbed.

Tabla. 5.2

Componentes y referencias de los dispositivos usados en el montaje del Testbed

Item	Referencia	Cantidad	Grupo
PC	Dell Optiplex 580	5	1
NetFPGA	Cube 1G	2	
Enrutador	Cisco 1800 Series	2	2
Switch	3Com Baseline 2928-SPF	2	
KVM	KVM	1	
Monitor	Monior Dell	1	3
Cables	Categoría 6	N/A	
Rack	N/A	1	

Todos los anteriores componentes estuvieron disponibles a través del Centro de Investigación en Ingeniería y Organizaciones – CIIO, UNAB y pertenecen a infraestructura mayormente usada por el Grupo de Tecnologías de la Información – GTI[2].

5.4.2. Topología

Una vez disponibles todos los componentes necesarios para la construcción del Testbed se diseñó la topología mostrada en la Figura 5.1, la cual simula el comportamiento de Internet y cumple con los requisitos necesarios para la evaluación de la herramienta.

5.4.3. Configuración general del Testbed

Ya conectados los componentes de la manera definida en la Sección 5.4.2 se procedió a configurar el *software* en los equipos. Dentro del Testbed se crearon redes internas para testear la herramienta bajo un ambiente controlado y evitar la

[2]Información adicional en *http://ciio.unab.edu.co/gti/*

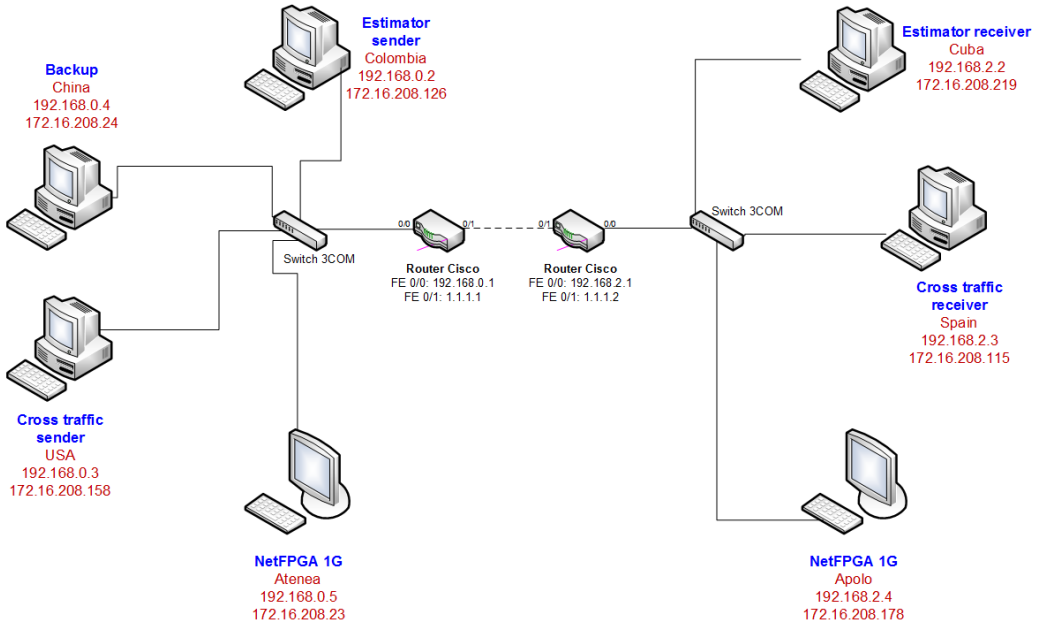


Figura 5.1. Topología del Testbed.

congestión e inyección de paquetes no deseados a la red de la Universidad. La totalidad de los dispositivos finales corren como sistema operativo *GNU/Linux* en diferentes distribuciones (*Fedora* [3], *Arch Linux* [4] y *Debian* [5]), el *switch* corre con la configuración por defecto y tiene las funciones de brindar acceso a Internet a todas las máquinas y permitir la conexión de las mismas a través de las redes internas definidas. En el caso de los enrutadores se definieron rutas estáticas.

Los detalles de esta infraestructura de prueba y evaluación que emula la comunicación en Internet pueden ser consultados en la Sección 6.4.

[3]Disponible en: *https://getfedora.org/en/workstation/download/*
[4]Disponible en: *https://www.archlinux.org/download/*
[5]Disponible en: *https://www.debian.org/distrib/*

5.5. Evaluación de la efectividad de la solución propuesta

Esta etapa corresponde a la validación de la hipótesis primaria del proyecto: Enviando paquetes de prueba desde el *hardware* de la NetFPGA posibilita la reducción de errores de estimación que tienen origen en la imprecisión del tiempo de envío de paquetes a través del *software.*

Ya contanto con el testbed y la implementacion del estimador de ancho de banda *traceband* en conjunto funcionamiento con NetFPGA, se realizaron experimentos para validar el método de ajuste propuesto, los cuales son mostrados en la Sección 6.5.

Capítulo 6

Resultados

Con base en la metodología propuesta en el capitulo 5, a continuación se desarrollarán cada una de las fases de proceso investigativo, así como la solución implementada que permite integrar una herramienta para la estimación del ancho de banda disponible con el *hardware* de red NetFPGA.

6.1. Mecanismos de modificación de tiempos de transmisión

6.1.1. Módulo de red *OMware*

Gran número de proyectos implementan sistemas embebidos basados en Linux para llevar a cabo mediciones de gran escala y experimentos de red. Debido a limitaciones de recursos y el aumento de las velocidades en la red, obtener resultados razonables desde estos dispositivos es muy difícil. *OMware* mejora la exactitud del tiempo de envió de paquetes al permitir a la aplicación de medición programar y pre-enviar el contenido del paquete al *kernel*. Gracias a este método, *OMware* también puede reducir el *overhead* en el marcado de paquetes *(timestamp)* y las interferencias de otros procesos de aplicación.

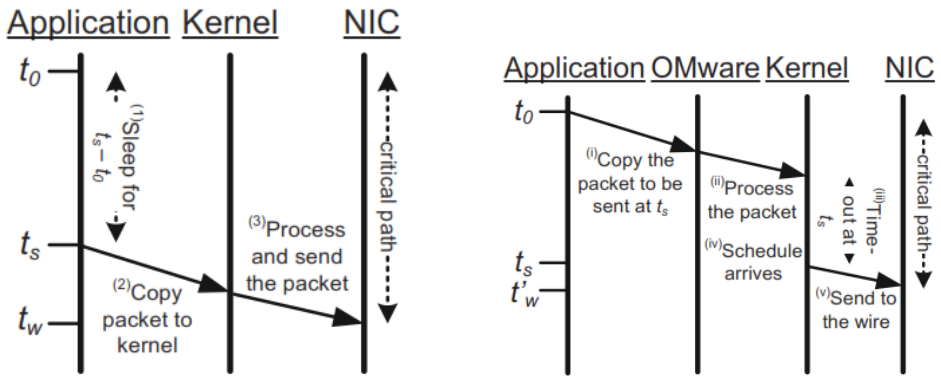
Modelo de programación y pre-envío

En un estudio realizado a varias herramientas de red que aparecen en el Cuadro 6.1, se encontró que estas herramientas son implementadas con el mismo tipo de llamado a funciones de *I/O*, *sleep* y *timestamp*. Al investigar el código fuente y el flujo de programación, estas herramientas usualmente adoptan un modelo de programación secuencial para programar el envío de paquetes. La Figura 6.1a y 6.1b ilustran una comparación entre la línea del tiempo del modelo secuencial y el modelo de pre-envío propuesto, respectivamente. La aplicación en las figuras se refiere a una herramienta de red que se ejecuta en el espacio de usuario. Para ambos modelos, en el tiempo t_0, se asume que la aplicación ha preparado el contenido del paquete para ser enviado en un tiempo futuro, t_s. El paquete aparece en el medio en (t_w,t'_w) en el modelo (secuencial, pre-envío). Por lo tanto, los errores de tiempo de envío son (t_s – t_0) o (t'_s – t_0) para el modelo secuencial o de pre-envío, respectivamente.

Tabla. 6.1

Ejemplos de llamadas a función utilizadas en herramientas de red.

Herramienta	**Función I/O**	**Sleep**	**Timestamp**
D-ITG	Socket POSIX	`select()`	`gettimeofday()`
httping	Socket POSIX	`usleep()`	`gettimeofday()`
Iperf	Socket POSIX	`nanosleep()`	`gettimeofday()`



(a) Modelo secuencial **(b)** Modelo pre-envio

Figura 6.1. Línea de tiempo comparativa entre el modelo secuencial y pre-envío.

Primero se considera el modelo secuencial de la Figura 6.1b. Las aplicaciones que utilizan este modelo se implementan normalmente mediante *sockets POSIX* para E/S de paquetes y una funciones `sleep()` para el espacio entre paquetes. Resumimos este modelo en sus tres pasos principales:

1. La aplicación prepara el contenido del paquete, calcula el período `sleep()` ($t_s - t_0$, para $t_s > t_0$) y entra en modo de suspensión.

2. Después de que el período `sleep()` ha terminado, el contenido del paquete se copia al *kernel* usando *socket*.

3. Las cabeceras de los paquetes son llenadas por la pila de protocolos TCP/IP y finalmente enviados a la tarjeta de red.

Por otro lado, el modelo de pre-envío, como muestra en la Figura 6.1b, divide el proceso de envío de paquetes en dos fases principales. La herramienta primero prepara y copia el paquete al módulo *OMware* antes de programar el tiempo de envío, t_s. A continuación, *OMware* envía el paquete cuando llega t_s. Podemos describir los detalles en cinco pasos:

(I) Una vez que el paquete está listo y se ha determinado el tiempo de envío, la aplicación puede invocar de inmediato la llamada de envío de paquetes en la API *OMware*, que toma un puntero al paquete y el tiempo de envío como entrada.

(II) *OMware* procesa el paquete, que incluye la adición de la cabecera *Ethernet* y la construcción de la estructura `sk_buff`.

(III) Si el momento de enviar el paquete no llega (es decir, el tiempo actual ¡t_s), *OMware* agregará la operación de envío de paquetes como una tarea del *kernel* accionado por un temporizador de alta resolución. De lo contrario, el paquete debe ser enviado inmediatamente.

(IV) Cuando llega el tiempo de envío programado t_s , una interrupción se genera para activar la rutina de llamada y enviar el paquete procesado.

(V) A medida que el paquete es procesado, puede ser puesto en el medio rápidamente.

La principal diferencia entre los dos modelos es cuando el programa inicia a esperar el tiempo programado. El modelo de pre-envío utiliza parte del tiempo de `sleep` para manejar las operaciones que consumen tiempo. Por lo tanto, el sistema puede tomar una ruta crítica más corta en el envío de paquetes y mejorar el rendimiento.

Los resultados muestran que *OMware* puede lograr una precisión a nivel de microsegundos (en lugar de milisegundos en una herramienta ejecutada en espacio de usuario) en el tiempo entre paquetes, incluso bajo trafico cruzado pesado. Además, el retardo en el envío de paquetes puede ser significativamente reducido en 0,2 milisegundos.

6.1.2. ICIM

En redes de alta velocidad, tales como redes de *1Gbps* o de mayores velocidades, los algoritmos de medición de ancho de banda que utilizan intervalos de transmisión/llegada de paquetes, tales como trenes de paquetes y pares de paquetes, tienen

una serie de problemas mencionados anteriormente (Sección 3.1.6). En primer lugar, la medición redes de gran ancho de banda requiere intervalos cortos de transmisión de paquetes, lo que ocasiona una gran carga sobre la *CPU*. En segundo lugar, las *NIC* para redes de alta velocidad suelen emplear interrupción de coalescencia *(IC)*, la cual reordena los intervalos de llegada de los paquetes y causa la transmisión en ráfagas de los paquetes. *ICIM* (Interrupt Coalescence-aware inline measurement) introduce un nuevo método de medición de ancho de banda que supera estos dos problemas. *ICIM* utiliza los paquetes de datos de una conexión TCP activa para la medición. Con el fin de determinar el ancho de banda disponible, en lugar de ajustar los intervalos de transmisión de paquetes, el emisor TCP ajusta el número de paquetes que participan en una ráfaga y comprueba si los intervalos entre las ráfagas correspondientes a paquetes *ACK* se incrementan o no. Los resultados de las simulaciones muestran que *ICIM* puede satisfactoriamente medir el ancho de banda de redes de algunos *Gbps*.

Los aportes del estudio son los siguientes:

1. Algoritmo de medición de ancho de banda disponible que funciona en redes *Gigabit (Interrupt Coalescence-aware Inline Measurement for available bandwidth (ICIM-abw))*.

2. Validación de los resultados de las mediciones de *ICIM-abw* en muchos escenarios simulados. Los resultados mostraron que el algoritmo funciona bien a *1Gbps* y redes de velocidades superiores.

3. Algoritmo para la medición de la capacidad en una red *Gigabit (Interrupt Coalescence-aware Inline Measurement for capacity (ICIM-cap)*. A diferencia de los algoritmos de medición actuales, el algoritmo *ICIM-cap* funciona bien en redes extremadamente cargas. Sin embargo, el algoritmo funciona bien sólo cuando el enlace apretado de la ruta *(tight link)*, el cual tiene el ancho de banda disponible más pequeño, es idéntico al enlace cuello de botella *(bottleneck link)*, que tiene la capacidad más pequeña.

4. La implementación de *ICIM-abw* sobre el sistema *FreeBSD* y los resultados de las mediciones muestran que puede funcionar bien en un sistema real.

6.2. Variaciones a la herramienta de estimación de ancho de banda *Traceband*

6.2.1. Archivos *pcap*

Uno de los sistemas de generación/captura e intercambio de paquetes de red más aceptados es el formato *pcap*, el cual es definido por libpcap para sistemas basados en UNIX y WinPcap para sistemas Windows. Existen varios *ports* para diversos lenguajes de programación como *Perl (Net)*, *Python (python-libpcap)*, *Ruby (Ruby/Pcap)* y *Java (Jpcap)*. Estas bibliotecas han sido utilizadas para crear una amplia gama de herramientas (propietarias y código abierto) que pueden utilizarse principalmente para capturar y analizar paquetes de red.

6.2.2. Estructura de un archivo *pcap*

Para detallar el formato de archivo, se examinará el archivo *connection-termination.cap*[1] mostrado en el Código 6.1:

Código 6.1. Archivo *connection-termination.pcap*

```
00000000  d4 c3 b2 a1 02 00 04 00  00 00 00 00 00 00 00 00  |................|
00000010  ff ff 00 00 01 00 00 00  c2 ba cd 4f b6 35 0f 00  |...........O.5..|
00000020  36 00 00 00 36 00 00 00  00 12 cf e5 54 a0 00 1f  |6...6.......T...|
00000030  3c 23 db d3 08 00 45 00  00 28 4a a6 40 00 40 06  |<#....E..(J.@.@.|
00000040  58 eb c0 a8 0a e2 c0 a8  0b 0c 4c fb 00 17 e7 ca  |X.........L.....|
00000050  f8 58 26 13 45 de 50 11  40 c7 3e a6 00 00 c3 ba  |.X&.E.P.@.>.....|
00000060  cd 4f 60 04 00 00 3c 00  00 00 3c 00 00 00 00 1f  |.O`...<...<.....|
00000070  3c 23 db d3 00 12 cf e5  54 a0 08 00 45 00 00 28  |<#......T...E..(|
00000080  8a f7 00 00 40 06 58 9a  c0 a8 0b 0c c0 a8 0a e2  |....@.X.........|
00000090  00 17 4c fb 26 13 45 de  e7 ca f8 59 50 10 01 df  |..L.&.E....YP...|
000000a0  7d 8e 00 00 00 00 00 00  00 00 c3 ba cd 4f 70 2f  |}............Op/|
```

[1]Disponible a través de *http://packetlife.net/captures/connection%20termination.cap*

```
000000b0  00 00 3c 00 00 00 3c 00  00 00 00 1f 3c 23 db d3  |..<...<.....<#..|
000000c0  00 12 cf e5 54 a0 08 00  45 00 00 28 26 f9 00 00  |....T...E..(&...|
000000d0  40 06 bc 98 c0 a8 0b 0c  c0 a8 0a e2 00 17 4c fb  |@.............L.|
000000e0  26 13 45 de e7 ca f8 59  50 11 01 df 7d 8d 00 00  |&.E....YP...}...|
000000f0  00 00 00 00 00 00 c3 ba  cd 4f db 2f 00 00 36 00  |.........O./..6.|
00000100  00 00 36 00 00 00 00 12  cf e5 54 a0 00 1f 3c 23  |..6.......T...<#|
00000110  db d3 08 00 45 00 00 28  4a a7 40 00 40 06 58 ea  |....E..(J.@.@.X.|
00000120  c0 a8 0a e2 c0 a8 0b 0c  4c fb 00 17 e7 ca f8 59  |........L......Y|
00000130  26 13 45 df 50 10 40 c7  3e a5 00 00              |&.E.P.@.>...|
0000013c
```

Un archivo *pcap* tiene la estructura mostrada en la Figura 6.2.

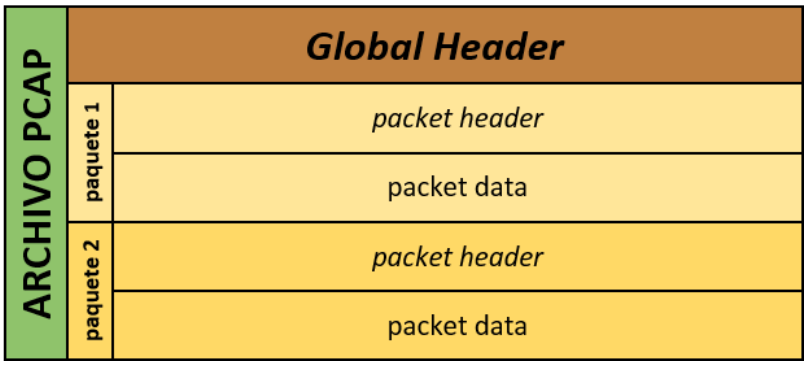


Figura 6.2. Estructura de un archivo *pcap*

Los componentes en *cursiva (Global Header, packet header)* son agregadas por la API Libpcap, los demás son los datos reales del paquete.

La primera parte del archivo es el encabezado global *(Global Header)*, es insertado sólo una vez al inicio del archivo y tiene un tamaño fijo de *24 bytes* (Código 6.2).

Código 6.2. Encabezado global del archivo *pcap*

```
d4  c3  b2  a1  02  00  04  00   00  00  00  00  00  00  00  00
ff  ff  00  00  01  00  00  00
```

Los primeros *4 bytes* `d4 a1 b2 c3` constituyen el número mágico *magic number* que es utilizado para identificar a los archivos *pcap*. Los siguientes *4 bytes* `02 00 04 00` son la versión mayor *(Major version)* - *2 bytes* y la versión menor *(Minor version)* - *2 bytes*, en este caso *2.4*. La codificación utilizada es llamada *little endian*, en la

que el *byte* menos significativo es almacenado en la posición menos significativa. Esto significa que *2* se escribe en *2 bytes* como `02 00`. Para saber que se está codificando con *little endian* en lugar de *big endian*, es utilizado el número mágico, que es el encargado de distinguir el orden de los bits. Si su valor real es `0xa1b2c3d4` significa codificación *big endian*, de lo contrario su valor real será `0xd4c3b2a1` y codificado *little endian*. Los siguientes *bytes* son el desplazamiento de zona horaria GMT menos la zona horaria utilizada en los encabezados (en segundos - longitud de *4 bytes*) y la exactitud de la marca de tiempo en la captura (conocido como *timestamp* - longitud de *4 bytes*). Estos valores son generalmente establecidos en *0* lo que resulta en `00 00 00 00 00 00 00 00`. El siguiente campo es el *Snapshot Length (4 bytes)* que indica la longitud máxima del paquete capturado/generado en *bytes*. En este archivo archivo se encuentra definido en `ff ff 00 00` que es igual a *65535* `(0xffff)`, el valor predeterminado de *tcpdump* y *Wireshark*. Los últimos *4 bytes* del encabezado global especifican el tipo de encabezado de la capa de enlace. Este archivo tiene definido el valor `0x1 (01 00 00 00)`, que indica que el protocolo de capa de enlace es *Ethernet*. Hay muchos otros tipos tales como [2]: PPPoE, USB, Frame Relay... Después del *Global Header*, se encuentran la cabecera y los datos del paquete. El Código 6.3 muestra el primer encabezado del paquete:

Código 6.3. Encabezados *pcap*

```
c2 ba cd 4f b6 35 0f 00  36 00 00 00 36 00 00 00
```

Los primeros *4 bytes* hacen referencia al *timestamp* en segundos. Este corresponde al número de segundos trascurridos desde el inicio del año 1970, también es conocido como *Unix Epoch*. El valor de este campo en el archivo es `0x4fcdbac2`. El Código 6.4 convierte este valor a un formato legible utilizando algunas herramientas UNIX:

Código 6.4. Tiempo de captura del paquete en formato legible

```
4mjaimes@4mjaimes:~$ calc 0x4fcdbac2
1338882754
```

[2] La lista completa está disponible en: http://www.tcpdump.org/linktypes.html

```
4mjaimes@4mjaimes:~$ date --date='1970-01-01 1338882754 sec GMT'
Tue Jun  5 08:52:34 CET 2012
```

Lo cual muestra que el paquete fue capturado el *Martes 5 de Junio de 2012 a las 08:52:34* con zona horaria CET (*Central European Time*).

El segundo campo (*4 bytes*) corresponde a los microsegundos en que fue capturado el paquete. En este caso equivale a `b6 0f 35 00` o *996790* microsegundos. El tercer campo (*4 bytes*) contiene el tamaño *(en bytes)* de los datos del paquete guardado/creado en el archivo. El cuarto campo (*4 bytes)* contiene la longitud del paquete que fue capturado/inyectado por la interfaz. Ambos valores están definidos en `36 00 00 00` *(54 Bytes)* en el archivo ejemplo, pero estos pueden tener valores diferentes en caso donde se establezca la longitud máxima del paquete (cuyo valor es *65535* en el encabezado global del archivo) a un tamaño menor.

Después del encabezado del paquete inicia la data, empezando desde la capa inferior se ve la dirección *Ethernet* destino `00:12:cf:e5:54:a0` seguido de la dirección *Ethernet* fuente `00:1f:3c:23:db:d3`. Desde este punto en adelante, los valores adicionales corresponden a las cabeceras propias de la pila de protocolos TCP/IP y a la data como tal.

6.2.3. Modificaciones al código

El código inicial de *traceband_snd* realiza el envío de paquetes a través de la interfaz de red mediante el uso de los ampliamente conocidos *sockets*. Las modificaciones hechas cambian este comportamiento en la herramienta, está nueva versión crea manualmente todos los campos dentro de los paquetes, mediante el uso de cabeceras incluidas en la librería estándar de C y los almacena en el formato *pcap* llamando funciones propias de Libpcap, lo cual facilita el proceso y a su vez garantiza que dentro de las cabeceras de cada paquete será guardado el tiempo exacto al cual deben ser enviados los paquetes. Este nuevo método es mostrado en el Código 6.8 y descrito a continuación:

Nuevas cabeceras son agregadas al archivo `traceband_fn.h` y permiten el llamado a funciones necesarias para crear de forma correcta el archivo *.pcap* y otras necesarias para llenar las cabeceras propias de cada capa del modelo TCP/IP.

Código 6.5. Definición de librerías

```
1  #include <stdio.h>    // printf() and fprintf()
2  #include <sys/types.h>   // data types used in system calls
3  #include <sys/socket.h>  // definitions of structures needed for sockets
4  #include <netinet/in.h>  // constants and structures needed for internet domain addresses
5  #include <arpa/inet.h>   // for sockaddr_in and inet_ntoa()
6  #include <netdb.h>
7  #include <stdlib.h>      // for atoi() and exit()
8  #include <unistd.h>      // standard unix functions, like alarm()
9  #include <string.h>      // memset()
10 #include <sys/time.h>    // select()
11 #include <math.h>
12 #include <assert.h>
13 #include <unistd.h>      // close()
14 #include <signal.h>      // signal name macros, and the signal() prototype
15 #include <netinet/in.h>
16 #include <netinet/tcp.h>
17 #include <netinet/ip.h>
18 #include <netinet/udp.h>
19 #include <netinet/ether.h>
20 #include <sys/stat.h>
21 #include <fcntl.h>
22 #include <errno.h>
23 #include <sys/uio.h>
24 #include <sys/wait.h>
25 #include <limits.h>
26 #include <float.h>
27 #include <pcap.h>
28 #include <net/if.h>
29 #include <linux/if_packet.h>
```

En las líneas 3 - 14 del archivo `traceband_snd` se definen las direcciones *MAC* origen y destino, acorde a las direcciones de las tarjetas de red de las máquinas a utilizar en el Testbed de prueba.

Código 6.6. Definición de las direcciones *MAC*

```
3  #define MY_SOUR_MAC0    0x00
4  #define MY_SOUR_MAC1    0x1B
5  #define MY_SOUR_MAC2    0x21
6  #define MY_SOUR_MAC3    0x76
7  #define MY_SOUR_MAC4    0xBD
8  #define MY_SOUR_MAC5    0xB1
9  #define MY_DEST_MAC0    0x00
10 #define MY_DEST_MAC1    0x64
11 #define MY_DEST_MAC2    0x40
12 #define MY_DEST_MAC3    0x31
13 #define MY_DEST_MAC4    0xA2
```

```
14 #define MY_DEST_MAC5      0x3B
```

Posteriormente, se declaran las variables necesarias para la lógica del programa y estructuras propias de cada capa de red.

Código 6.7. Variables y estructuras propias de cada capa de red

```
54     int tx_len;
55     char sendbuf[pck_size + 42];
56     struct pcap_pkthdr pcap_hdr;
57     struct ether_header *eh = (struct ether_header *) sendbuf;
58     struct iphdr *iph = (struct iphdr *) (sendbuf + sizeof (struct ether_header));
59     struct udphdr *udph = (struct udphdr *) (sendbuf + sizeof (struct iphdr) + sizeof (struct ↩
           ether_header));
60     struct info_pkt *my_pkt = (struct info_pkt *) (sendbuf + sizeof (struct udphdr) + sizeof (↩
           struct iphdr) + sizeof (struct ether_header));
61
62     pcap_t *handle;
63     pcap_dumper_t *dumper;
64     handle = pcap_open_dead(DLT_EN10MB, 65535);
65     dumper = pcap_dump_open(handle, "./capture.pcap");
```

La línea 55 define el *buffer* donde toda la información del paquete sera guardada. La linea 56 define la estructura `pcap_header` que será la encargada de almacenar las cabeceras *pcap*.

Las lineas 57 - 58 - 59 definen las cabeceras para *Ethernet, IP* y *UDP* respectivamente. La linea 60 es un apuntador a la estructura `info_pkt`, encargado de agrupar la información de las cabeceras y la carga útil del paquete.

La linea 62 - 65 son las encargadas de definir el protocolo de capa de enlace, el tamaño del paquete a capturar y la ruta donde será almacenado el paquete creado.

En las lineas 84 - 137 los campos de las cabeceras correspondientes a cada capa son llenados con información útil

Código 6.8. Modificaciones a *Traceband*

```
84     do {
85         maximum_samples = ((estim_num / 30)*30 == estim_num) ? TRUE : FALSE;
86         samples = maximum_samples ? p_train : min(p_train, 80);
87         gettimeofday(&start_time, NULL); // estimation initial time
88         // send packet pairs every inpair_gap microseconds
89         for (train_cnt = 1; train_cnt <= n_trains; train_cnt++) {
90             for (pck_cnt = 1; pck_cnt <= samples; pck_cnt++) {
91                 pck_num++;
92                 tx_len = 0;
93                 memset(sendbuf, 0, pck_size);
94                 eh->ether_shost[0] = MY_SOUR_MAC0;
```

```
95                  eh->ether_shost[1] = MY_SOUR_MAC1;
96                  eh->ether_shost[2] = MY_SOUR_MAC2;
97                  eh->ether_shost[3] = MY_SOUR_MAC3;
98                  eh->ether_shost[4] = MY_SOUR_MAC4;
99                  eh->ether_shost[5] = MY_SOUR_MAC5;
100                 eh->ether_dhost[0] = MY_DEST_MAC0;
101                 eh->ether_dhost[1] = MY_DEST_MAC1;
102                 eh->ether_dhost[2] = MY_DEST_MAC2;
103                 eh->ether_dhost[3] = MY_DEST_MAC3;
104                 eh->ether_dhost[4] = MY_DEST_MAC4;
105                 eh->ether_dhost[5] = MY_DEST_MAC5;
106                 eh->ether_type = htons(ETH_P_IP);
107                 tx_len += sizeof (struct ether_header);
108
109                 iph->ihl = 5;
110                 iph->version = 4;
111                 //iph->id = htons(random());
112                 iph->frag_off = 0x40;
113                 iph->ttl = 64;
114                 iph->protocol = 17; // UDP
115                 iph->saddr = inet_addr("192.168.2.4");
116                 iph->daddr = inet_addr(rcv_IP);
117                 tx_len += sizeof (struct iphdr);
118
119                 udph->source = htons(snd_echo.sin_port);
120                 udph->dest = htons(SERVER_PORT);
121                 udph->check = 0xe603;
122                 tx_len += sizeof (struct udphdr);
123
124                 my_pkt->num = htonl(pck_num);
125                 my_pkt->size = htonl(pck_size);
126                 gettimeofday(&timestamp, NULL);
127                 my_pkt->sec = htonl(timestamp.tv_sec);
128                 my_pkt->usec = htonl(timestamp.tv_usec);
129                 tx_len += sizeof (struct info_pkt);
130
131                 /* Length of UDP payload and header */
132                 udph->len = htons(tx_len - sizeof (struct ether_header) - sizeof (struct iphdr))↩
                        ;
133                 /* Length of IP payload and header */
134                 iph->tot_len = htons(tx_len - sizeof (struct ether_header));
135                 /* Calculate IP checksum on completed header */
136                 iph->check = csum((unsigned short *) (sendbuf + sizeof (struct ether_header)), ↩
                        sizeof (struct iphdr) / 2);
137
138                 /* Pcap packet header*/
139                 pcap_hdr.ts = timestamp;
140                 pcap_hdr.caplen = sizeof (sendbuf);
141                 pcap_hdr.len = pcap_hdr.caplen;
142
143                 /* Write the packet to pcap file*/
144                 pcap_dump((u_char*) dumper, &pcap_hdr, sendbuf);
```

Las lineas 139 - 141 definen los campos `pcap_hdr.ts` (*timestamp* preciso al cual debe ser enviado el paquete), *pcap_hdr.caplen* (tamaño del paquete a enviar) y `pcap_hdr.len` (cuanto paquete a enviar debe ser enviado). de las cabeceras *pcap*.

En la línea 144 la información del paquete (carga util y cabeceras) es guardada en el archivo *.pcap*.
Este proceso se encuentra dentro de un bucle (linea 89) que es ejecutado `n_trains` veces.

El código completo de *traceband_snd.co* está disponible en el Apéndice A

De está manera se asegura que los tiempos dictados por la herramienta *Traceband* son guardados de manera precisa dentro de la cabecera de los paquetes en el archivo *.pcap*, que posteriormente sera enviado a través de la NetFPGA.

6.3. Implementación del estimador sobre la plataforma NetFPGA

6.3.1. Generador de paquetes

Este proyecto proporciona una manera sencilla de captura y generación de paquetes. Está diseñado para ser utilizado por cualquier persona que desee inyectar paquetes a la red u observar el comportamiento de los paquetes saliendo de la red.

La función de generación de paquetes en la implementación actual (versión 1.1.1) es un mecanismo de "reproducción"; Una secuencia de paquetes es volcado desde un archivo en formato *pcap* [3] al generador que luego procede a transmitir la secuencia de paquetes. Los retardos entre paquetes individuales son controlados estrictamente; por defecto el generador utiliza el retardo especificado en el archivo fuente *pcap*. Opcionalmente puede ser especificada la tasa máxima de datos o fijar un retardo especifico entre paquetes.

El generador de paquetes también puede ser utilizado para reportar estadísticas de los paquetes de salida de la red y, opcionalmente, guardar los paquetes que salen

[3] https://wiki.wireshark.org/Development/LibpcapFileFormat

de la red. Estas funciones son útiles para el análisis de rendimiento a través de un dispositivo y para analizar si los datos que están saliendo de la red son correctos.

Especificaciones

El Cuadro 6.2 presenta los detalles con mas relevancia para la integración del generador de paquetes con la herramienta de estimación:

Tabla. 6.2

Especificaciones relevantes en la integración hardware-software

Item	Descripción
Exactitud del tiempo de reproducción	Típicamente dentro de algunos cientos microsegundos. La ubicación del módulo de retardo dentro del pipeline limita la precisión debido a los buffers que siguen.
Precisión del timestamp de captura	La precisión está dada en nanosegundos
Exactitud del timestamp de captura	Normalmete de 8ns (+ cualquier inexactitud añadida por reloj). El reloj principal de la NetFPGA es corre normalmente 125MHz (periodo de 8ns).

Implementación

Se asumirá que se encuentra correctamente instalada y configurada la plataforma NetFPGA 1G. A continuación se detallan los pasos para la puesta a punto del generador de paquetes:

1. Para el correcto funcionamiento del generador de paquetes es necesario implementar la versión *2.1.1* del *NFP* [4].

   Descomprimir el *NFP*:

   ```
   4mjaimes@4mjaimes:~/Descargas$ tar -xvf netfpga_full_2_1_1.tar.gz
   ```

   Mover la carpeta descomprimida al directorio `home/`:

   ```
   4mjaimes@4mjaimes:~/Descargas$ mv netfpga/ ~/
   ```

[4]Disponible en: https://github.com/NetFPGA/netfpga/wiki/Releases

2. Descargar, descomprimir y copiar el proyecto *packet_generator* [5] a la carpeta `projects/` del *NFP*.

```
4mjaimes@4mjaimes:~/Descargas$ tar -xvf netfpga_packet_generator_1_1_1.tar.gz
```

Mover la carpeta `packet_generator` al *NFP*:

```
4mjaimes@4mjaimes:~/Descargas$ mv netfpga/projects/packet_generator/ ~/netfpga/projects/
```

Mover el *bitfile* del *packet_generator* a la carpeta `bitfiles/` del *NFP*

```
4mjaimes@4mjaimes:~/Descargas$ mv netfpga/bitfiles/packet_generator.bit ~/netfpga/bitfiles/
```

3. Descargar el *bitfile* del generador de paquetes a la *FPGA* de la NetFPGA.

```
4mjaimes@4mjaimes:~$ nf_download ~/netfpga/bitfiles/packet_generator.bit
```

4. Actualizar el archivo *.bashrc*. Apuntar la variable de entorno `NF_DESIGN_DIR` al directorio `packet_generator`

```
if [ "$NF_ROOT" == "" ]; then
  export NF_ROOT=${HOME}/netfpga
fi

if [ "$NF_DESIGN_DIR" == "" ]; then
  export NF_DESIGN_DIR=${NF_ROOT}/projects/packet_generator
fi

if [ "$NF_WORK_DIR" == "" ]; then
  export NF_WORK_DIR=/tmp/${USER}
fi

export PYTHONPATH=${PYTHONPATH}:${NF_ROOT}/lib/python
export LD_LIBRARY_PATH=${NF_ROOT}/lib/java/NetFPGAFrontEnd/bin:${LD_LIBRARY_PATH}

if [ ! -d ${NF_WORK_DIR} ]; then
 mkdir ${NF_WORK_DIR}
fi

if [ ! -d ${HOME}/.qt ]; then
 mkdir ${HOME}/.qt
fi

if [ -f ${NF_ROOT}/bin/nf_profile ]; then
 source ${NF_ROOT}/bin/nf_profile
fi
```

[5] Disponible en https://docs.google.com/file/d/0B4EuVzA5UdPRdmF2ZEI4QmtSSEU

Una vez culminado los anteriores pasos, el generador de paquetes se encuentra configurado y listo para ser usado.

Generación/Captura de paquetes

El proyecto del generador de paquetes es controlado por el comando *packet_generator.pl* escrito en *Perl*, los argumentos que acepta este comando son mostrados en el Código 6.9.

Código 6.9. Argumentos aceptados pro *packet_generator.pl*

```
1  packet_generator.pl
2    [-q]
3    [-r] (Kbps)
4    [-i]
5    [-d] (ns)
6    [-c]
7    [--pad]
8    [--nodrop]
9    [--wait]
10   [--ns]
```

El Cuadro 6.3 muestra una descripción de la la funcionalidad que agrega cada argumento que es pasado a *packet_generator.pl*

Tabla. 6.3

Descripcion de los argumentos del comando packet_generator.pl

Argumento	Descripción
-q <# cola ><archivo pcap >	Especifica el archivo pcap a carga y la cola por la cual se debe enviar.
-r <# cola ><taza >	Especifica la tasa de envío para cada cola (en Kbps).
-i <# cola ><# de iteraciones >	Especifica el número de iteraciones por la cola.
-d <# cola ><retardo paquetes >	Especifica el retardo entre paquetes en ns.
-c <# cola ><archivo captura >	Especifica el nombre del archivo de captura.
–pad	Reduce todos los paquetes a un tamaño máximo de 64 bytes.
–nodrop	No permite la perdida de paquetes por el puerto que se está capturando.
–wait	Espera la señal USR1; ayuda con la sincronización de múltiples generadores.
–ns	Reporta los tiempos con precisión de nanosegundos

Arquitectura del generador de paquetes

La arquitectura del generador de paquetes utiliza como diseño base la NIC de referencia Covington et al. (2009a). Las principales modificaciones realizadas por el generador de paquetes son hechas dentro del *User Data Path* como muestra la Figura 6.3. A continuación se listan las principales modificaciones:

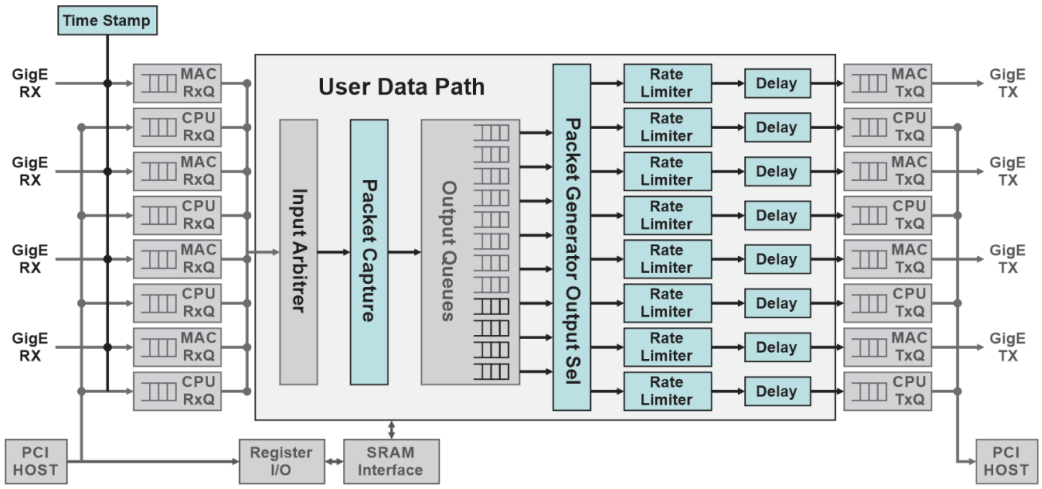


Figura 6.3. Arquitectura del generador de paquetes

- Se añadió un módulo *Time Stamp* (marcado de los paquetes) antes del *MAC RxQ*. Esto permite que los paquetes entrantes sean marcados a medida que se reciben por el *hardware*. Esta marca de tiempo se utiliza luego durante la creación del archivo *.pcap* que permite mayor precisión al ser marcados desde el *hardware*.

- Se agregó el módulo *Packet Capture* dentro del *User Data Path*, el cual realiza dos funciones: la compilación de estadísticas al momento de generación y captura de paquetes (como el número de paquetes recibidos y el tiempo total de captura) y la captura de las marcas de tiempo de los paquetes cuando la función de generación/captura está desactivada. Cuando el generador de paquetes está desactivado, el *bitfile* funciona como una tarjeta *NIC* de cuatro puertos normal.

- Se modificó el *Output Queues* para operar con doce colas de salida y no con ocho como en la *NIC* de referencia. Esto permite que las cuatro nuevas colas se utilicen para almacenar los datos *pcap*, para luego ser transmitidos cuando el Generador de Paquetes está activado.

- Se añadió el módulo *Packer Generator Output Sel* el cual determina cuál de las 12 colas de salida debe ser conectado a las ocho colas de salida del siguiente módulo.

- Cada una de las ocho colas de salida de referencia cuenta con un limitador de tasa y un módulo de retardo. Esto permite agregar un retardo a la velocidad de cada una de las ocho colas de salida individualmente. El retardo y la tasa son establecidos mediante registros escritos por el *software* del generador de paquetes ejecutado en el *host*.

6.3.2. Integración de *traceband_snd.c* con el generador de paquetes

Para lograr el envío de paquetes a tiempos exactos establecidos por la herramienta de estimación de ancho de banda *traceband_snd.c* fue necesario realizar modificaciones adicionales que permitieran la sincronización de estas dos herramientas. Analizaremos el Código 6.10 donde es llevado a cabo este proceso.

Código 6.10. Conexión *traceband_snd.c* con el generador de paquetes

```
150 pcap_dump_close(dumper);
151 system("/home/Apolo/netfpga/projects/packet_generator/sw/packet_generator.pl -q0 ./capture.pcap"↩
        );
152 mysleep(intrain_gap, timestamp); // gap between trains
```

La línea 150 es una función que pertenece a la librería Libpcap, su función es cerrar de manera adecuada el archivo de extensión *.pcap* para que otros programas logren identificar adecuadamente los paquetes almacenados. Una vez cerrado el archivo *pcap*, *traceband* realiza el envío de los paquetes a los cuatro *buffers* adicionales del módulo

Output Queues (Ver Figura 6.3) del generador de paquetes mediante la función definida en la línea *151*. La línea 151 ejecuta la función *system()* incluida en su biblioteca estándar, dentro de la cabecera `<stdlib.h>`. El objetivo de esta función es ejecutar subprocesos o comandos propios del sistema operativo, en este caso la utilidad del generador de paquetes *packet_generator.pl*. la función *system()* recibe como parámetro la ubicación (ruta completa) del comando a ejecutar. De está manera se le indica al generador de paquetes que debe cargar el archivo indicado y realizar el envío de los paquetes contenidos en el archivo *.pcap*. Una vez el generador de paquetes ha enviado la totalidad del archivo, *traceband_snd.c* sigue su ejecución normal.

La Figura 6.4 muestra una captura de pantalla del programa *Wireshark*, el cual es utilizado para abrir el archivo *.pcap* creado con la herramienta modificada *traceband_snd.c* y enviado a la red a través del generador de paquetes. Mediante *Wireshark* en la columna *Time* se evidencia que los paquetes están siendo creados a tiempos exactos dictados por la herramienta de estimación.

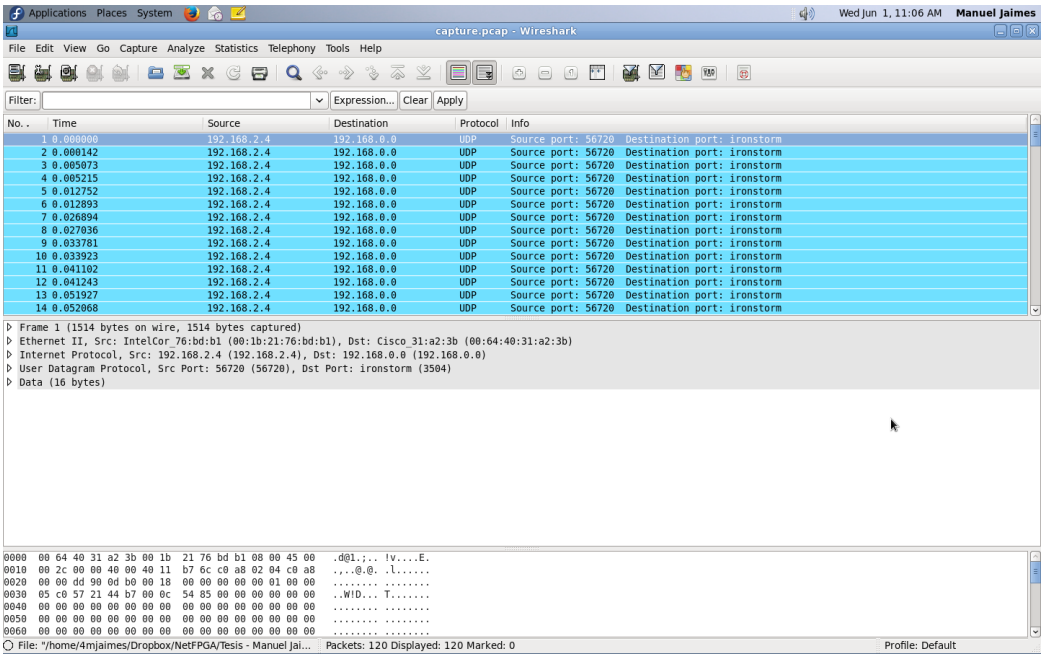


Figura 6.4. pcap generado por *traceband*

6.4. Infraestructura de prueba y evaluación de la herramienta

6.4.1. Definición de Testbed

Un Testbed es una plataforma de red para experimentación de proyectos de gran desarrollo. Los Testbed brindan una forma de comprobación rigurosa, transparente y repetible de teorías científicas, elementos computacionales, y nuevas tecnologías.

El término Testbed es usado también, en varias disciplinas para describir un ambiente de desarrollo que está protegido de los riesgos que conllevan realizar las pruebas en un ambiente de producción. De manera sencilla, es un método o infraestructura para probar un módulo particular de forma aislada.

6.4.2. Testbed en el campo de redes de computadoras

PlanetLab

Es un escenario de pruebas de dimensiones globales, diseñado para apoyar el desarrollo de nuevos servicios en redes académicas avanzadas. Nace en el *2003* liderado por la Universidad de Princeton en Estados Unidos, y es construido gracias a la suma de un gran número de servidores distribuidos a través de las redes académicas del mundo, los que a su vez, forman un laboratorio computacional a escala planetaria; de ahí es atribuido su nombre.

Sobre el conjunto de servidores que componen la red de *PlanetLab* se pueden desarrollar, instalar y ejecutar aplicaciones en un entorno de prueba desplegado sobre una red con condiciones del mundo real. Más de *800* servidores repartidos en *400* sitios de más de *40* países del mundo, albergan la implementación que posibilita la existencia de *PlanetLab* y donan parte de su ancho de banda para que éste efectivamente logre operar. La mayoría de sus servidores están instalados en universidades conectadas a las redes académicas y otros en los centros de operaciones.

ModelNet

Es un emulador de redes a gran escala que permite a los usuarios evaluar sistemas distribuidos en red sobre entornos realistas, como Internet. *ModelNet* permite el ensayo de prototipos sin modificar corriendo sobre sistemas operativos sin modificar a través de diversos escenarios de redes. En cierto modo, se combina la capacidad de repetición de la simulación con el realismo de despliegue en vivo.

La comunidad de usuarios *ModelNet* realizado ayudas en el diseño y prueba de redes nuevas de distribución de contenidos, sistemas *peer-to-peer*, protocolos de capa de transporte, conmutadores basados en contenidos, procesadores distribuidos de flujo, sistemas de archivos distribuidos y herramientas de medición de red.

Web100

Desarrollado por un equipo de *Pittsburgh Supercomputing Center*, para proporcionar una visión de las características de una conexión TCP para desarrolladores de aplicaciones y administradores de sistemas. El proyecto fue creado específicamente para desarrollar una interfaz de gestión avanzada para TCP y exponer así el funcionamiento interno de éste. Ha sido utilizado con gran éxito en la identificación y diagnóstico de problemas de rendimiento de la red.

Web 100 proporciona las herramientas para estudiar y diagnosticar las variables TCP. Garantiza acceso a nivel del *kernel* a variables internas del protocolo TCP para su configuración y la caracterización del funcionamiento y rendimiento.

6.4.3. Componentes de un Testbed

Un Testbed como todo sistema contiene diferentes componentes que se pueden agrupar en unos de tipo *hardware* y otros de tipo *software*. Estos grupos y sus componentes son mostrados en la Figura 6.5.

- Testbed
  - Hardware
    - End-host.
    - Dispositivos de interconexión de redes.
  - Software
    - Sistema operativo.
    - Herramientas o módulo a evaluar
    - Generadores de tráfico
    - Otras herramientas.

Figura 6.5. Componentes de un Testbed.

6.4.4. Descripción general del Testbed UNAB

El Testbed que se muestra en la Figura 6.6 es un ambiente totalmente controlado que permite simular el Internet y cuenta con dispositivos de red que pueden operan a velocidades de 10Mbps, 100Mbps y 1Gbps.

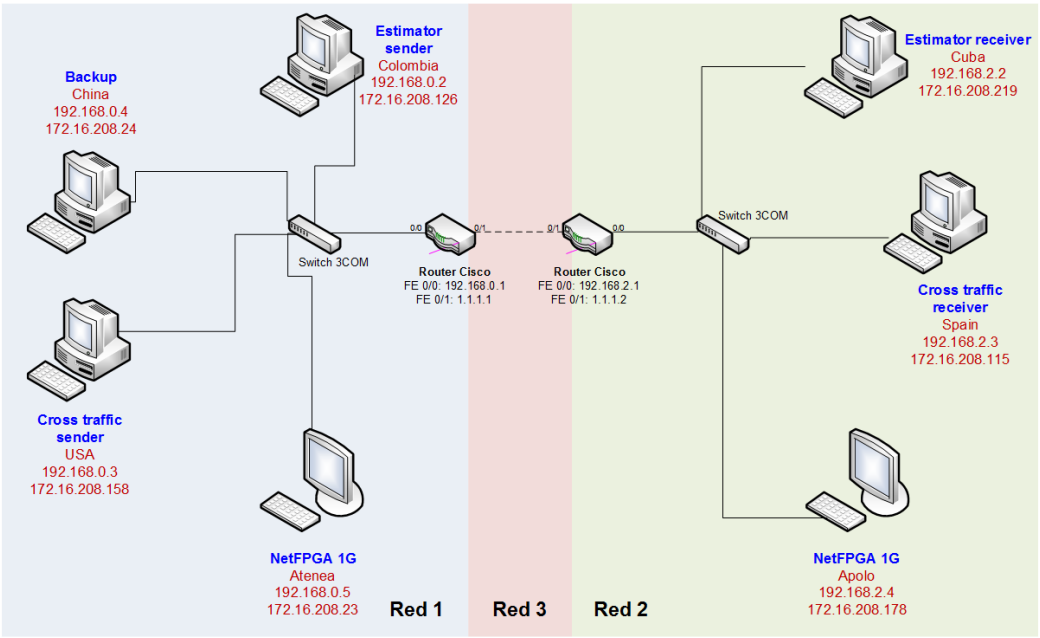


Figura 6.6. Topología del Testbed UNAB

End Host

El Testbed cuenta con 7 computadores correspondientes a los *host*, que permiten interactuar con la red y realizar todas las pruebas que sean necesarias. Las

características de los recursos básicos de *hardware* y *software* con que cuenta cada máquina se encuentran detalladas en el Cuadro 6.4. Donde es de destacar que el sistema operativo de todos los *hosts* es *GNU–LINUX* y a nivel de *hardware* son máquinas dotadas con la suficiente capacidad de procesamiento que los experimentos requieren, las velocidades de *CPU* oscilan entre los *800MHz* y *2.8GHz*.

Tabla. 6.4

Características técnicas de los end-host del Testbed.

Host	S.O	Arquitectura	CPU	RAM	HDD
Colombia	Debian	i686	AMD Athlon II X2 240	2GB	150GB
USA	Archlinux	x86_64	AMD Athlon II X2 240	2GB	150GB
Cuba	Debian	i686	AMD Athlon II X2 240	2GB	150GB
Spain	Debian	i686	AMD Athlon II X2 240	2GB	150GB
China	Archlinux	x86_64	AMD Athlon II X2 240	2GB	150GB
NetFPGA 1	Fedora 13	i686	AMD X2 Quad Core	4GB	500GB
NetFPGA 2	Fedora 13	i686	AMD X2 Quad Core	4GB	500GB

Las Figura 6.7a, muestra el *Rack* donde se encuentran las máquinas instaladas en un ambiente a temperatura controlada. La Figura 6.7c, presenta los *5* equipos *(end-host)* dispuestos en el *Rack*. Los *host* NetFPGA 1 y 2 pueden verse en la Figura 6.7b.

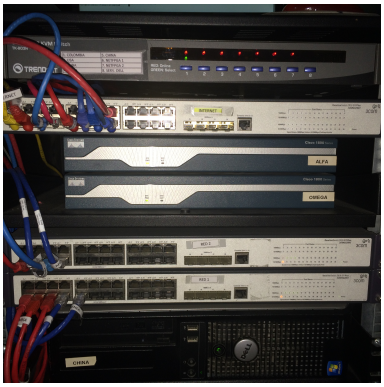
Dispositivos de interconexión

Los *host* Colombia, USA, China, Spain y Cuba cuenta cada uno con dos adaptadores de red, uno *on-board* y otro conectado a un puerto *PCI*.

En cada uno de los extremos del Testbed (Figura 6.6, y para lograr la comunicación de los *host*, cada red dispone de un *Switch 3COM Baseline 2928-SFP Plus*, que operara a velocidades entre *100Mbps* y *1Gbps*. Así mismo para interconectar la Red 1 con la Red 2, cada red dispone de enrutador *CISCO 1800 Serie (100Mbps)*; El

(a) Rack

(b) Hosts NetFPGA



(c) End-Hosts

Figura 6.7. Imagenes del Testbed UNAB

enrutador asignado a la Red 1 se le adjudico el alias *ALFA* y el de Red 2 *OMEGA*. Cada *switch* se encuentra conectado a la interfaz `FE0/0` de su respectivo enrutador, así mismo éstos se interconectan entre sí usando la interfaz `FE0/1`, simulando así una conexión Internet y conformando la Red 3.

Direccionamiento

Como se puede observar en Topología del Testbed 6.6, el direccionamiento IP consta de una configuración dinámica (Administrado por Infraestructura Tecnológica - UNAB) *(dhclient)* y otra estática (Configurada manualmente de acuerdo a los

requerimientos del proyecto); el primero es asignado a las interfaces de red *on-board (eth0)* y el segundo a la tarjeta de red conectada al puerto *PCI (eth1)*. Los enrutadores tiene direccionamiento estático. El Cuadro 6.5 muestra a detalle el direccionamiento en el Testbed.

Tabla. 6.5

Direccionamiento IP del Testbed

Host	**Dirección IP**		**Mascara**	**Red**
	eth0 [6]	*eth1* [7]		
Colombia	172.16.208.126	192.168.0.2	255.255.255.0	1
USA	172.16.208.158	192.168.0.3	255.255.255.0	
China	172.16.208.24	192.168.0.4	255.255.255.0	
Atenea	172.16.208.23	192.168.0.5	255.255.255.0	
Alfa (gateway)	192.168.0.1		—	
Cuba	172.16.208.219	192.168.2.2	255.255.255.0	2
Spain	172.16.208.115	192.168.2.3	255.255.255.0	
Apolo	172.16.208.178	192.168.2.4	255.255.255.0	
Omega (gateway)	192.168.2.1		—	
Alfa	1.1.1.1		—	3
Omega	1.1.1.2		—	

El direccionamiento dinámico sobre las interfaces *eth0* de cada máquina del Testbed, permite que éstos accedan a Internet. El *host* Colombia debido a que permite acceso remoto a través de *ssh* debe ser configurado de manera estática.

[6] En los *host* USA y China, la interfaz de red se llama textitenp2s0

[7] En los *host* USA y China, la interfaz de red se llama textitenp3s0

Enrutamiento

El enrutamiento configurado en cada máquina del Testbed es necesario para poder comunicar la Red 1 con la Red 2 utilizando cualquiera de las dos interfaces de red. La configuración de cada máquina se muestra en la Figura 6.8 .

```
root@Colombia:/home/colombia# route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         Colombia        0.0.0.0         UG    1024   0        0 eth0
172.16.208.0    *               255.255.255.0   U     0      0        0 eth0
192.168.0.0     *               255.255.255.0   U     0      0        0 eth1
192.168.2.0     192.168.0.1     255.255.255.0   UG    1      0        0 eth1
root@Colombia:/home/colombia#
```

(a) Colombia

```
[netperf@USA ~]$ route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         gateway         0.0.0.0         UG    202    0        0 enp2s0
172.16.208.0    *               255.255.255.0   U     202    0        0 enp2s0
192.168.0.0     *               255.255.255.0   U     0      0        0 enp3s0
192.168.2.0     192.168.0.1     255.255.255.0   UG    0      0        0 enp3s0
[netperf@USA ~]$ route
```

(b) USA

```
[netperf@China ~]$ route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         gateway         0.0.0.0         UG    203    0        0 enp2s0
172.16.208.0    *               255.255.255.0   U     203    0        0 enp2s0
192.168.0.0     *               255.255.255.0   U     0      0        0 enp3s0
192.168.2.0     192.168.0.1     255.255.255.0   UG    0      0        0 enp3s0
[netperf@China ~]$ exit
```

(c) China

```
[Atenea@Atenea ~]$ route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
172.16.208.0    *               255.255.255.0   U     1      0        0 eth0
default         172.16.208.1    0.0.0.0         UG    0      0        0 eth0
[Atenea@Atenea ~]$ exit
```

(d) Atenea

```
root@Cuba:/home/cuba# route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         192.168.2.1     0.0.0.0         UG    1024   0        0 eth1
172.16.208.0    *               255.255.255.0   U     0      0        0 eth0
192.168.0.0     192.168.2.1     255.255.255.0   UG    1      0        0 eth1
192.168.2.0     *               255.255.255.0   U     0      0        0 eth1
root@Cuba:/home/cuba# exit
```

(e) Cuba

```
root@spain:/home/spain# route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         192.168.2.1     0.0.0.0         UG    1024   0        0 eth1
172.16.208.0    *               255.255.255.0   U     0      0        0 eth0
192.168.0.0     192.168.2.1     255.255.255.0   UG    1      0        0 eth1
192.168.2.0     *               255.255.255.0   U     0      0        0 eth1
root@spain:/home/spain#
```

(f) Spain

```
[Apolo@Apolo ~]$ route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
172.16.208.0    *               255.255.255.0   U     1      0        0 eth0
link-local      *               255.255.0.0     U     1007   0        0 eth2
link-local      *               255.255.0.0     U     1008   0        0 eth1
default         172.16.208.1    0.0.0.0         UG    0      0        0 eth0
```

(g) Apolo

Figura 6.8. Enrutamiento de las máquinas del Testbed

6.4.5. Accesibilidad

El acceso al Testbed puede efectuarse de tres maneras:

1. Accediendo a cada una de las máquinas directamente en el *Rack*.
2. Remotamente desde algún PC ubicado en la misma *subred (172.16.208.0)*.
3. Remotamente a través de Internet, utilizando *ssh* a una IP pública establecida.

Tabla. 6.6

Credenciales de acceso al Testbed

Host	User	Password	
		user	*root*
Colombia	colombia		
Cuba	cuba		@netfpga
Spain	spain		
USA	netperf	@netfpga	
China	netperf		@NetFPGA
Atenea	Atenea		
Apolo	Apolo		

El Cuadro 6.6, presenta las credenciales de acceso al las máquinas ubicadas en el Testbed:.

6.5. Efectividad de la herramienta de estimación sobre NetFPGA

En esta sección se evalúa y compara la herramienta de estimación de ancho de banda *traceband* con y sin las modificaciones hechas en el presente trabajo, por lo cual se presenta la metodología realizada para tal experimentación. (a) Se presenta un análisis para seleccionar el generador de tráfico. (b) Se configura la herramienta de estimación y el generador de paquetes sobre el testbed de red. (c) Se definen las métricas a evaluar. (d) Se presentan los resultados obtenidos.

6.5.1. Análisis y selección del generador de paquetes

Una revisión de los trabajos donde se evalúan y comparan las herramientas de estimación de ancho de banda disponible más representativas muestra que para la generación de tráfico son utilizadas diferentes herramientas. El Cuadro 6.7 resalta dos importantes aspectos: (a) Las veces que la herramienta fue usada dentro de 27

artículos revisados. (b) El tipo de tráfico que puede generar *(Burst, Poisson, Pareto o Periódico)*.

Tabla. 6.7

Generadores de trafico

Herramienta	**Utilizada en # papers**	**Tipo de trafico**			
		Bursty	*Poisson*	*Periodic*	*Pareto*
D-ITG	4	•	•	•	•
MGEN	5	•	•	•	•
MRTG	2	•	•		
iperf	3	•	•	•	•
Netem-ITGM	1	•	•	•	
Probe Generator	1	•	•		
Tiers topology generator	1			•	
tg	1			•	
Otros	8				

El Cuadro 6.7 muestra que *MGEN*, *D-ITG* e *iperf*, son los generadores más utilizados por los investigadores dentro de los artículos revisados. A continuación se presenta una breve descripción de cada herramienta:

D-ITG Generador de tráfico de gran precisión que permite controlar el tiempo de salida, el tamaño de los paquetes a enviar y trabaja sobre *IPv4* e *IPv6*. Genera gran versatilidad de tipo de tráfico, lo que permite tener más opciones al comparar el comportamiento de las herramientas.

iperf Herramienta de generación de tráfico que permite al usuario experimentar con diferentes parámetros TCP y UDP para ver cómo estos afectan al rendimiento de la red. *Iperf* fue desarrollado por el *Distributed Applications Support Team (DAST)*

en el *National Laboratory for Applied Network Research (NLANR)* y está escrito en *C++*.

MGEN Herramienta de generación de trafico, brinda flexibilidad de configuración para cada tipo de tráfico a generar. Dentro de sus opciones permite replicar una traza de tráfico previamente capturado desde un archivo *pcap* [8].

Con el fin de probar la herramienta con el escenario más cercano a Internet (Topología y trafico) se decidió utilizar una traza de tráfico real de Internet capturada por el *Center for Applied Internet Data Analysis - CAIDA* de tipo: *Direction B*, descargada desde su sitio web en formato *pcap* [9]. Teniendo en cuenta esta razón técnica, el potencial generador a utilizar es *MGEN*, sin embargo, dada la necesidad de escalar el tráfico de la traza para probar el comportamiento de la herramienta en diversos escenarios y la restricción de *MGEN* para realizar esta tarea, se selecciono la herramienta *Tcprepaly* que realiza con gran flexibilidad esta función. *Tcpreplay* es una conocida suite de herramientas para sistemas tipo UNIX (y Windows bajo Cygwin [10]) que da la capacidad de utilizar el tráfico capturado previamente en formato libpcap para probar una variedad de dispositivos y escenarios de red.

6.5.2. Configuración de *traceband*

La configuración del estimador de ancho de banda implica los siguientes pasos:

1. Obtener el código fuente de la herramienta [11].
2. Descomprimir el archivo *tar*, cambiar al directorio raíz, y ejecutar:

Código 6.11. Compilación de *traceband*.

```
make
```

[8] Esto lo hace mediante el parámetro `CLONE`, pero no permite escalar el trafico a introducir en la red. Las opciones de generación de trafico tipo *Poisson* o *Burst* si permite escalar tráfico.

[9] Disponible en: https://www.caida.org/data/monitors/passive-equinix-chicago.xml

[10] Colección de herramientas desarrollada por *Cygnus Solutions* para proporcionar un comportamiento similar a los sistemas UNIX en Microsoft Windows.

[11] La versión original y modificada están disponibles a través del medio digital que acompaña este trabajo.

Los pasos anteriores funcionan en la versión original y modificada de *traceband*.

6.5.3. Configuración de *Tcpreplay*

La configuración e instalación de la herramienta es muy sencilla y puede realizarse de dos maneras:

Instalación desde sus archivos fuente

1. Descargar la herramienta vía *Github*

   Código 6.12. Descarga de *tcpreplay* vía *Github*.

   ```
   git clone https://github.com/appneta/tcpreplay
   ```

2. Compilar el código fuente, pero primero se debe asegurar que el sistema cuenta con las herramientas de compilación y del software pre-requisito instalado. Ejecute:

   Código 6.13. Instalación de herramientas necesarias para *tcpreplay* en *Debian*.

   ```
   sudo apt-get install build-essential libpcap
   ```

   Código 6.14. Instalación de herramientas necesarias para *tcpreplay* en *Archlinux*.

   ```
   sudo pacman -Syu base-devel libpcap
   ```

3. Por ultimo, descomprimir el archivo *tar*, cambiar al directorio raíz, y ejecutar:

   Código 6.15. Configuración e instalación de *tcpreplay*.

   ```
   ./configure
   make
   sudo make install
   ```

Instalación desde el gestor de paquetes

En *Archlinux* ejecute:

Código 6.16. Instalación de *tcpreplay* en *Archlinux*.

```
sudo pacman -Syu tcpreplay
```

En *Debian* ejecute:

Código 6.17. Instalación de *tcpreplay* en *Debian*.

```
sudo apt-get install tcpreplay
```

6.5.4. Definición de las métricas a evaluar

Dentro de los 27 artículos revisados se logró determinar que las métricas más evaluadas en los estudios son (a) Ancho de banda disponible (22/27) (b) Tamaño del paquete (17/27) (c) Tiempo de estimación (16/27) (d) Trafico cruzado y precisión (13/26) (e) *Overhead* (6/26). Debido a que se desea comparar la mejora en la estimación del ancho de banda disponible que supone enviar los paquetes a tiempos exactos, se definen como métricas para evaluar las siguientes variables:

- Tiempo de estimación.
- Error de estimación.
- Ancho de banda disponible.

Diseño y ejecución de experimentos

Para comparar el rendimiento de las dos versiones de la herramienta se diseñaron dos escenarios escalando la traza. El primero replicando el trafico para congestionar el canal (trafico cruzado) al 30 % y el segundo al 60 %. Se realizan 10 experimentos para cada escenario, para tener 20 evaluaciones de cada versión de la herramienta, y así alcanzar un total 40 experimentos.

Para congestionar el canal con al porcentaje deseado se ejecuto el siguiente comando:

Código 6.18. Comando para que *tcpreplay* replique trafico a una taza deseada.

```
tcpreplay  --loop=0 --mbps=30 --intf1=enp3s0 equinix-anon.pcap
```

El comando *tcpreplay* toma como argumentos:

- `--loop=1` Indica el numero de veces que sera replicado el archivo *pcap*, en este caso 1 vez.
- `--mbps=30` Replica el trafico a la taza indicada en *Mbps*, *30* o *60 Mbps* según el escenario.
- `--intf1=enp3s0` Especifica la interfaz a utilizar.
- `equinix-anon.pcap` El último argumento hace referencia al archivo *pcap* a ser replicado.

Para el caso de *traceband*, en el cliente se ejecuto el comando *traceband_snd* que toma como argumento `-s` y la dirección IP del *receiver*.

Código 6.19. Ejecución de *traceband_snd*.

```
./traceband_snd -s 192.168.2.2
```

En el servidor se ejecuto el comando *traceband_rcv* que toma como argumento el nombre del archivo a crear para guardar algunos parámetros de la estimación realizada.

Código 6.20. Ejecución de *traceband_snd*.

```
./traceband_rcv escenario1-10.txt
```

6.5.5. Resultados *traceband* original

Después de la ejecución de los 20 experimentos se obtuvieron los resultados presentados en la Tabla 6.8

Tabla. 6.8

Resultados traceband original

#	Trafico cruzado %	Real_Bw (Mbps)	Est_Bw (Mbps)	Error (%)	Overhead (%)	Tiempo (s)
1	30	70	78,0	11	2,11	0,68
2	30	70	66,0	-6	2,19	0,66
3	30	70	77,0	10	2,30	0,63
4	30	70	79,0	13	2,07	0,70
5	30	70	74,0	6	2,14	0,67
6	30	70	76,0	9	2,17	0,66
7	30	70	65,0	-7	1,96	0,73
8	30	70	65,0	-7	2,06	0,70
9	30	70	77,0	10	2,11	0,68
10	30	70	75,0	7	2,16	0,67
11	60	40	48,0	20	2,17	0,66
12	60	40	44,0	10	2,07	0,69
13	60	40	49,0	23	2,10	0,69
14	60	40	35,0	-13	2,07	0,70
15	60	40	49,0	23	2,13	0,68
16	60	40	48,0	20	2,12	0,68
17	60	40	44,0	10	1,97	0,73
18	60	40	45,0	13	1,90	0,76
19	60	40	36,0	-10	2,27	0,63
20	60	40	46,0	15	2,02	0,71

Convenciones: **Bw_Real:** Ancho de banda disponible real. **Bw_Est:** Ancho de banda disponible estimado por la herramienta. **Error:** Porcentaje de error de la estimación. **Overhead:** Porcentaje de trafico inyectado por la herramienta. **Tiempo:** Tiempo utilizado para realizar la estimación.

6.5.6. Resultados *traceband* modificado

La versión inicial de la herramienta *traceband_snd* realizaba el envío de paquetes mediante *sockets*. Esta nueva versión, se encarga de crear manualmente todos los campos dentro de los paquetes, haciendo uso de cabeceras incluidas en la librería estándar de C (Sección 6.2.3), sin embargo el campo *FCS* [12] de *Ethernet* es calculado por la tarjeta de interfaz de red antes de realizar el envío.

Al crear el paquete manualmente y enviarlo por medio del generador de paquetes de la NetFPGA el campo *FCS* no es calculado, lo cual no permite que los paquetes logren salir de la red. Al llegar al primer enrutador, este descarta todos los paquetes creados y no permite que la comunicación de extremo a extremo de la herramienta sea exitosa.

[12] La secuencia de verificación de trama (FCS) son cuatro octetos de comprobación de redundancia cíclica (CRC) que permiten la detección de datos corruptos dentro de la trama que recibie el receptor.

Capítulo 7

Recomendaciones

- Al plantear un proyecto tan ambicioso como lo fue éste, se desea que día tras día exista una mejora continua del mismo; por lo tanto se recomienda a todos los interesados realizar la mejora del *FCS* calculando la verificación por redundancia cíclica *(CRC)*. Aunque existen numerosas versiones que calculan el *(CRC)*: (a) CRC-reverse (b) CRC32a (c) CRC32b (d) CRC32c (e) CRC32cx (f) CRC32d (g) CRC32f (h) CRC32g (i) CRC32h, el Apendice B presenta una posible implemementación funcional de CRC que debe ser agregada al método encargado de crear manualmente los paquetes.

- Incluir NetFPGA como una herramienta de enseñanza de redes de computadores. Debido a su filosofía abierta, brinda la posibilidad de tratar en detalle aspectos que en el modelo educativo actual apenas son mencionados. A su vez, es una excelente plataforma de prototipado para prueba ideas innovadoras.

Capítulo 8

Conclusiones

- Los mecanismo de modificación de tiempos de transmisión en *software* permiten mejoras considerables en el proceso del marcado de los paquetes, sin embargo las limitaciones propias de la máquina, no permiten que los resultados tengan un desempeño cercano a las soluciones propuestas por el *hardware*.

- Las modificaciones hechas a la herramienta *traceband* cumplen con el objetivo de garantizar el envío de los paquetes a los tiempos determinado, eliminan fuentes de error en la estimación y tratan por primera vez una solución desde la perspectiva *hardware-software*.

- La infraestructura de red configurada para el proyecto brinda un escenario de características muy similares a Internet, lo que permite la ejecución de pruebas y puesta a punto de la herramienta de estimación para su uso en un entorno real.

- La flexibilidad del generador de paquetes de la NetFPGA y la manera en que se estructuraron los cambios en el estimador brinda una interesante base para trabajos futuros, permitiendo experimentar con un juego de variables que hasta el día de hoy no han sido objeto de estudio en al campo de la estimación del ancho de banda.

- Una vez solucionado el error en la creación del paquete, se podrá comparar y demostrar las mejoras en el porcentaje de error y en la exactitud de las estimaciones de ancho de banda.

Bibliografía

Bibliografía

Ali, A. A. and Lepage, F. (2007). Igmps, a new tool for estimating end-to-end available bandwidth in ip network paths. In *Networking and Services, 2007. ICNS. Third International Conference on*, pages 115–115. IEEE.

Beheshti, N., Naous, J., Ganjali, Y., and McKeown, N. (2007). Experimenting with buffer sizes in routers. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 41–42. ACM.

Blanco, I., Morán, A., Ferro, A., Zabala, L., and Pineda, A. (2012). Arquitectura de generación e inyección de tráfico sintético configurable en FPGA. In *XXVII Simposium Nacional de la Unión Científica Internacional de Radio*.

Botta, A., Dainotti, A., and Pescapé, A. (2010). Do you trust your software-based traffic generator? *Communications Magazine, IEEE*, 48(9):158–165.

Carrasquilla, S. M., Ulloque, E., and Guerrero, C. (2006). Evaluación de técnicas de medición de ancho de banda disponible ABET's. In *Revista Comunicación de datos - UNAB*.

Casado, M., Watson, G., and McKeown, N. (2005). Reconfigurable networking hardware: A classroom tool. In *High Performance Interconnects, 2005. Proceedings. 13th Symposium on*, pages 151–157. IEEE.

Covington, G. A., Gibb, G., Lockwood, J. W., and Mckeown, N. (2009a). A packet generator on the netfpga platform. In *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, pages 235–238. IEEE.

Covington, G. A., Gibb, G., Naous, J., Lockwood, J. W., and McKeown, N. (2009b). Encouraging reusable network hardware design. In *Microelectronic Systems Education, 2009. MSE'09. IEEE International Conference on*, pages 29–32. IEEE.

Deri, L. (2005). ncap: Wire-speed packet capture and transmission. In *End-to-End Monitoring Techniques and Services, 2005. Workshop on*, pages 47–55. IEEE.

Garcia, L. M. (2008). Programming with libpcap±sniffing the network from our own application. *Hakin9-Computer Security Magazine*, pages 2–2008.

Gibb, G., Lockwood, J. W., Naous, J., Hartke, P., and McKeown, N. (2008). Netfpga—an open platform for teaching how to build gigabit-rate network switches and routers. *Education, IEEE Transactions on*, 51(3):364–369.

Guerrero, C. D. and Labrador, M. A. (2006). Experimental and analytical evaluation of available bandwidth estimation tools. In *Local Computer Networks, Proceedings 2006 31st IEEE Conference on*, pages 710–717. IEEE.

Guerrero, C. D. and Labrador, M. A. (2008). A hidden markov model approach to available bandwidth estimation and monitoring. In *Internet Network Management Workshop, 2008. INM 2008. IEEE*, pages 1–6. IEEE.

Guerrero, C. D. and Labrador, M. A. (2010a). On the applicability of available bandwidth estimation techniques and tools. *Computer Communications*, 33(1):11–22.

Guerrero, C. D. and Labrador, M. A. (2010b). Traceband: A fast, low overhead and accurate tool for available bandwidth estimation and monitoring. *Computer Networks*, 54(6):977–990.

Hartikainen, E. and Ekelin, S. (2006). Tuning the temporal characteristics of a kalman-filter method for end-to-end bandwidth estimation. In *End-to-End Monitoring Techniques and Services, 2006 4th IEEE/IFIP Workshop on*, pages 58–65. IEEE.

Hartikainen, E., Ekelin, S., and Karlsson, J. M. (2005). Adjustment of the bart kalman filter to improve real-time estimation of end-to-end available bandwidth. In *3rd SNCNW 2005, Halmstad, November 23-24, 2005*, page 56. SNCNW.

Hu, N. and Steenkiste, P. (2003). Evaluation and characterization of available bandwidth probing techniques. *Selected Areas in Communications, IEEE Journal on*, 21(6):879–894.

Jacobson, V., Leres, C., and McCanne, S. (1994). libpcap, lawrence berkeley laboratory, berkeley, ca. *Initial public release June*.

Jain, M. and Dovrolis, C. (2002a). *End-to-end available bandwidth: Measurement methodology, dynamics, and relation with TCP throughput*, volume 32. ACM.

Jain, M. and Dovrolis, C. (2002b). Pathload: A measurement tool for end-to-end available bandwidth. In *In Proceedings of Passive and Active Measurements (PAM) Workshop*. Citeseer.

Jain, M. and Dovrolis, C. (2004). Ten fallacies and pitfalls on end-to-end available bandwidth estimation. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 272–277. ACM.

Lakshminarayanan, K., Padmanabhan, V., and Padhye, J. (2004). Bandwidth estimation in broadband access networks. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement.*, IMC '04, pages 314–321, New York, NY, USA. ACM.

Le Thanh Man, C., Hasegawa, G., and Murata, M. (2006). Icim: An inline network measurement mechanism for highspeed networks. In *End-to-End Monitoring Techniques and Services, 2006 4th IEEE/IFIP Workshop on*, pages 66–73. IEEE.

Man, C. L. T., Hasegawa, G., and Murata, M. (2008). Inline bandwidth measurement techniques for gigabit networks. *International Journal of Internet Protocol Technology*, 3(2):81–94.

McCanne, S. and Jacobson, V. (1993). The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, pages 2–2. USENIX Association.

Melander, B., Björkman, M., and Gunningberg, P. (2000). A new end-to-end probing and analysis method for estimating bandwidth bottlenecks. In *Global Telecommunications Conference, 2000. GLOBECOM'00. IEEE*, volume 1, pages 415–420. IEEE.

Melander, B., Bjorkman, M., and Gunningberg, P. (2002). Regression-based available bandwidth measurements. In *International Symposium on Performance Evaluation of Computer and Telecommunications Systems*, pages 14–19. Citeseer.

Michaut, F. and Lepage, F. (2005). Application-oriented network metrology: metrics and active measurement tools. *IEEE Communications Surveys Tutorials*, 7(2):2–24.

Mok, R. K., Li, W., and Chang, R. K. (2015). Improving the packet send-time accuracy in embedded devices. In *Passive and Active Measurement*, pages 332–344. Springer.

Morillo, D. D. S. (2011). Implementación y evaluación de un algoritmo de clustering en un estimador de ancho de banda disponible. Master's thesis, Universidad Autónoma de Bucaramanga, Colombia.

Naous, J., Gibb, G., Bolouki, S., and McKeown, N. (2008). Netfpga: reusable router architecture for experimental research. In *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, pages 1–7. ACM.

Navratil, J. and Cottrell, R. L. (2003). Abwe: A practical approach to available bandwidth estimation. In *Passive and Active Measurements (PAM) Workshop*. Citeseer.

Olsson, R. (2005). Pktgen the linux packet generator. In *Proceedings of the Linux Symposium, Ottawa, Canada*, volume 2, pages 11–24.

Paxson, V. (1997). End-to-end internet packet dynamics. In *ACM SIGCOMM Computer Communication Review*, volume 27, pages 139–152. ACM.

Prasad, R., Dovrolis, C., Murray, M., and Claffy, K. (2003). Bandwidth estimation: metrics, measurement techniques, and tools. *Network, IEEE*, 17(6):27–35.

Ribeiro, V. J., Navratil, J., Riedi, R. H., Baraniuk, R. G., and Cottrell, L. (2003). pathchirp: Efficient available bandwidth estimation for network paths. In *Presented at*, number SLAC-PUB-9732.

Salmon, G., Ghobadi, M., Ganjali, Y., Labrecque, M., and Steffan, J. G. (2009). Netfpga-based precise traffic generation. In *Proc. of NetFPGA Developers Workshop*, volume 9. Citeseer.

Santos, D. A. R. (2015). Estimación de ancho de banda disponible por generación de paquetes de prueba a través de NetFPGA. Master's thesis, Universidad Autónoma de Bucaramanga, Colombia.

Shriram, A., Murray, M., Hyun, Y., Brownlee, N., Broido, A., Fomenkov, M., et al. (2005). Comparison of public end-to-end bandwidth estimation tools on high-speed links. In *Passive and Active Network Measurement*, pages 306–320. Springer.

Sommers, J., Barford, P., and Willinger, W. (2006). A proposed framework for calibration of available bandwidth estimation tools. In *Computers and Communications, 2006. ISCC'06. Proceedings. 11th IEEE Symposium on*, pages 709–718. IEEE.

Sommers, J., Barford, P., and Willinger, W. (2007). Laboratory-based calibration of available bandwidth estimation tools. *Microprocessors and Microsystems*, 31(4):222–235.

Strauss, J., Katabi, D., and Kaashoek, F. (2003a). A measurement study of available bandwidth estimation tools. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 39–44. ACM.

Strauss, J., Katabi, D., Kaashoek, F., and Prabhakar, B. (2003b). Spruce: A lightweight end-to-end tool for measuring available bandwidth. In *Proc. of the Internet Measurement Conference (IMC)*.

Tockhorn, A., Danielis, P., and Timmermann, D. (2011). A configurable fpga-based traffic generator for high-performance tests of packet processing systems. In *6th International Conference on Internet Monitoring and Protection (ICIMP)*, pages 14–19.

Zhou, H., Wang, Y., Wang, X., and Huai, X. (2006). Difficulties in estimating available bandwidth. In *Communications, 2006. ICC'06. IEEE International Conference on*, volume 2, pages 704–709. IEEE.

Apéndice

Apéndice A

Código *traceband_snd.c*

Código A.1. Código *traceband_snd.c*

```
1  #include "traceband_fn.h"
2
3  #define MY_SOUR_MAC0    0x00
4  #define MY_SOUR_MAC1    0x1B
5  #define MY_SOUR_MAC2    0x21
6  #define MY_SOUR_MAC3    0x76
7  #define MY_SOUR_MAC4    0xBD
8  #define MY_SOUR_MAC5    0xB1
9  #define MY_DEST_MAC0    0x00
10 #define MY_DEST_MAC1    0x64
11 #define MY_DEST_MAC2    0x40
12 #define MY_DEST_MAC3    0x31
13 #define MY_DEST_MAC4    0xA2
14 #define MY_DEST_MAC5    0x3B
15
16 //----- Global variables ------------------------------------------------------
17 int verbose = FALSE; // verbose is off
18 int n_trains = NUM_TRAINS; // Number of trains
19 int p_train = PCK_TRAIN, samples; // Packets per train
20 int target_time = 0; // Tool's running time. Zero means once.
21 int traceband_sock; // Socket descriptor
22 int path_capacity = PATHCAPACITY; // Capacity of the path in bps
23 int link_set = 0;
24 long pck_size = PACKET_SIZE; // Packet size is varied to identify packets
25 double pair_gap; // Gap in a packet pair in us
26 double inpair_gap; // Gap between pairs in us
27 double intrain_gap; // Gap between trains in us
28 char peer_actual[16];
29 char *rcv_IP; // IP address of receiver
30 struct sockaddr_in rcv_echo; // Echo receiver address
31 struct sockaddr_in snd_echo; // Echo sender address
32 struct hostent *rcv_info; // Server information
33
34 //----- Function Prototypes ---------------------------------------------------
35 void args_sender(int argc, char * argv[]);
36 int prep_sockets(void);
```

```
37 unsigned short csum(unsigned short *buf, int nwords);
38
39 //=========================================================================
40 //= Main program                                                          =
41 //=========================================================================
42
43 int main(int argc, char *argv[]) {
44     unsigned int fromSize; // In-out of address size for recvfrom()
45     int rcv_msg_size; // Length of received response
46     int pck_num = 0; // Number of packet to be sent
47     int train_cnt, pck_cnt; // Packet and train counters
48     int estim_num = 0; // number of estimations
49     void *snd_msg; // Data in the packet to be sent
50     char rcv_msg[100]; // Buffer for receiving echoed string
51     struct timeval timestamp, start_time, end_time; // Packet timestamping
52     double run_time = 0, overhead;
53     short int maximum_samples = TRUE;
54     int tx_len;
55     char sendbuf[pck_size + 42];
56     struct pcap_pkthdr pcap_hdr;
57     struct ether_header *eh = (struct ether_header *) sendbuf;
58     struct iphdr *iph = (struct iphdr *) (sendbuf + sizeof (struct ether_header));
59     struct udphdr *udph = (struct udphdr *) (sendbuf + sizeof (struct iphdr) + sizeof (struct ↩
           ether_header));
60     struct info_pkt *my_pkt = (struct info_pkt *) (sendbuf + sizeof (struct udphdr) + sizeof (↩
           struct iphdr) + sizeof (struct ether_header));
61
62     pcap_t *handle;
63     pcap_dumper_t *dumper;
64     handle = pcap_open_dead(DLT_EN10MB, 65535);
65     dumper = pcap_dump_open(handle, "./capture.pcap");
66
67     srand((unsigned int) time((time_t *) NULL)); // to reset the seed
68     args_sender(argc, argv); //
69     prep_sockets();
70     printf("%s:\nSending data to IP : %s via a %i bps path ..\n", argv[0],
71             rcv_IP, path_capacity);
72     double shortest_inpair_gap = (pck_size + 28)*2. * 8. * 1e6 / path_capacity / 0.05;
73     inpair_gap = max(shortest_inpair_gap, inpair_gap);
74     if (verbose) {
75         printf("- Probing packet size : %i\n", pck_size + 28);
76         printf("- Path capacity       : %i\n", path_capacity);
77         printf("- Number of trains    : %i\n", n_trains);
78         printf("- Packets per train   : %i\n", p_train);
79         printf("- Gap in a packet pair: %.2f microseconds\n", pair_gap);
80         printf("- Gap between pairs   : %.2f microseconds\n", inpair_gap);
81         printf("- Gap between trains  : %.2f microseconds\n", intrain_gap);
82     }
83     snd_msg = malloc(pck_size); // allocate memory for packet to be sent
84     do {
85         maximum_samples = ((estim_num / 30)*30 == estim_num) ? TRUE : FALSE;
86         samples = maximum_samples ? p_train : min(p_train, 80);
87         gettimeofday(&start_time, NULL); // estimation initial time
88         // send packet pairs every inpair_gap microseconds
89         for (train_cnt = 1; train_cnt <= n_trains; train_cnt++) {
90             for (pck_cnt = 1; pck_cnt <= samples; pck_cnt++) {
91                 pck_num++;
92                 tx_len = 0;
93                 memset(sendbuf, 0, pck_size);
```

```
94                  eh->ether_shost[0] = MY_SOUR_MAC0;
95                  eh->ether_shost[1] = MY_SOUR_MAC1;
96                  eh->ether_shost[2] = MY_SOUR_MAC2;
97                  eh->ether_shost[3] = MY_SOUR_MAC3;
98                  eh->ether_shost[4] = MY_SOUR_MAC4;
99                  eh->ether_shost[5] = MY_SOUR_MAC5;
100                 eh->ether_dhost[0] = MY_DEST_MAC0;
101                 eh->ether_dhost[1] = MY_DEST_MAC1;
102                 eh->ether_dhost[2] = MY_DEST_MAC2;
103                 eh->ether_dhost[3] = MY_DEST_MAC3;
104                 eh->ether_dhost[4] = MY_DEST_MAC4;
105                 eh->ether_dhost[5] = MY_DEST_MAC5;
106                 eh->ether_type = htons(ETH_P_IP);
107                 tx_len += sizeof (struct ether_header);
108
109                 iph->ihl = 5;
110                 iph->version = 4;
111                 //iph->id = htons(random());
112                 iph->frag_off = 0x40;
113                 iph->ttl = 64;
114                 iph->protocol = 17; // UDP
115                 iph->saddr = inet_addr("192.168.2.4");
116                 iph->daddr = inet_addr(rcv_IP);
117                 tx_len += sizeof (struct iphdr);
118
119                 udph->source = htons(snd_echo.sin_port);
120                 udph->dest = htons(SERVER_PORT);
121                 udph->check = 0xe603;
122                 tx_len += sizeof (struct udphdr);
123
124                 my_pkt->num = htonl(pck_num);
125                 my_pkt->size = htonl(pck_size);
126                 gettimeofday(&timestamp, NULL);
127                 my_pkt->sec = htonl(timestamp.tv_sec);
128                 my_pkt->usec = htonl(timestamp.tv_usec);
129                 tx_len += sizeof (struct info_pkt);
130
131                 /* Length of UDP payload and header */
132                 udph->len = htons(tx_len - sizeof (struct ether_header) - sizeof (struct iphdr))↩
                        ;
133                 /* Length of IP payload and header */
134                 iph->tot_len = htons(tx_len - sizeof (struct ether_header));
135                 /* Calculate IP checksum on completed header */
136                 iph->check = csum((unsigned short *) (sendbuf + sizeof (struct ether_header)), ↩
                        sizeof (struct iphdr) / 2);
137
138                 /* Pcap packet header*/
139                 pcap_hdr.ts = timestamp;
140                 pcap_hdr.caplen = sizeof (sendbuf);
141                 pcap_hdr.len = pcap_hdr.caplen;
142
143                 /* Write the packet to pcap file*/
144                 pcap_dump((u_char*) dumper, &pcap_hdr, sendbuf);
145
146                 if ((pck_cnt / 2)*2 != pck_cnt) mysleep(pair_gap * 1.2, timestamp); // sleep to ↩
                        send second packet
147                     // Exponential separations between pairs (Poisson sampling)
148                 else mysleep(inpair_gap + rint(rand_exp() * inpair_gap), timestamp); // sleep ↩
                        while it is time to send the next pair
```

```
149             }
150             pcap_dump_close(dumper);
151             system("/home/Apolo/netfpga/projects/packet_generator/sw/packet_generator.pl -q0 ./capture.pcap");
152             mysleep(intrain_gap, timestamp); // gap between trains
153         }
154         // Send end of round key (pck_num is 0)
155         *(((long*) snd_msg) + 0) = htonl(0);
156         *(((long*) snd_msg) + 1) = htonl(path_capacity); // to tell the receiver about the path capacity
157         sendto(traceband_sock, snd_msg, 1000, 0, (struct sockaddr *)
158                 &rcv_echo, sizeof (rcv_echo));
159         // Recv a response
160         fromSize = sizeof (snd_echo);
161         signal(SIGALRM, catch_alarm); // set a signal handler for ALRM signals
162         alarm(10); // start a 10 seconds alarm
163         rcv_msg_size = recvfrom(traceband_sock, rcv_msg, sizeof (rcv_msg), 0,
164                 (struct sockaddr *) &snd_echo, &fromSize);
165         alarm(0); // remove the timer, now that we've got the user's input
166         /* null-terminate the received data */
167         printf("%s", rcv_msg); // Print available bandwidth value estimated in the receiver side
168         gettimeofday(&end_time, NULL); // program final time
169         run_time = timeval_diff(&end_time, &start_time);
170         overhead = ((n_trains * samples * 8 * (PACKET_SIZE + 28))*1e6 / run_time) / path_capacity * 100;
171         printf("Time: %.2f s\t", run_time / 1e6);
172         printf("Overhead: %.2f %%\n", overhead);
173         target_time -= (run_time / 1e6);
174         estim_num++;
175     } while (target_time > 0);
176     // Send end of transmission packet key (pck_size is 0)
177     *(((long*) snd_msg) + 1) = htonl(0);
178     sendto(traceband_sock, snd_msg, 1000, 0, (struct sockaddr *)
179             &rcv_echo, sizeof (rcv_echo));
180     close(traceband_sock);
181
182     return 1;
183 }
184
185 //----- Function Definitions --------------------------------------------------
186
187 //-------------------------------------------------------------------------
188 //- Set the sender program arguments                                      -
189 //-------------------------------------------------------------------------
190
191 void args_sender(int argc, char * argv[]) {
192     int opt;
193
194     // Default pck pair gap (us) is the time it takes to transmit a packet in
195     // the bottlenet link specified by "path_capacity"
196     pair_gap = ((double) (pck_size * 8) / path_capacity) * 1e6;
197     inpair_gap = pair_gap*INPAIR_GAP_MUL; // Gap between pairs in us
198     intrain_gap = inpair_gap*INTRAIN_GAP_MUL; // Gap between trains in us
199
200     while ((opt = getopt(argc, argv, "s:c:f:r:g:i:n:t:v")) != -1) {
201         switch (opt) {
202             case 's':
203                 rcv_info = gethostbyname(optarg); // get server IP address
204                 if (rcv_info == NULL) {
```

```
205                     fprintf(stderr, "Host not found: %s\n", optarg);
206                     exit(1);
207                 }
208                 rcv_IP = inet_ntoa(*(struct in_addr *) rcv_info->h_addr_list[0]);
209                 //copy away so future gethost* works?
210                 memcpy(peer_actual, rcv_info->h_addr, sizeof (rcv_echo.sin_addr.s_addr));
211                 break;
212             case 'c': // path's capacity
213             {
214                 int len = strlen(optarg);
215                 int mul = 1;
216                 switch (optarg[len - 1]) {
217                     case 'M':
218                     case 'm':
219                         mul = 1000 * 1000;
220                         break;
221                     case 'k':
222                     case 'K':
223                         mul = 1000;
224                         break;
225                     case 'G':
226                     case 'g':
227                         mul = 1000 * 1000 * 1000;
228                         break;
229                     default:
230                         printf("Capacity units are K/M/G. It will be assumed in M (Mbps)\n");
231                         mul = 1000 * 1000;
232                         break;
233                 }
234                 if (mul != 1) optarg[len - 1] = 0;
235                 path_capacity = atoi(optarg);
236                 path_capacity *= mul;
237                 if (path_capacity <= 0 || path_capacity > 1000000000) {
238                     fprintf(stderr, "illegal path capacity: %d\n", path_capacity);
239                     exit(1);
240                 }
241                 link_set = 1;
242                 pair_gap = ((double) (pck_size * 8) / path_capacity) * 1e6;
243                 inpair_gap = pair_gap*INPAIR_GAP_MUL; // Gap between pairs in us
244                 intrain_gap = inpair_gap*INTRAIN_GAP_MUL; // Gap between trains in us
245                 break;
246             }
247             case 'f':
248                 n_trains = atoi(optarg);
249                 if (n_trains <= 0) {
250                     fprintf(stderr, "Number of trains must be greater than zero\n");
251                     exit(1);
252                 }
253                 break;
254             case 'r':
255                 p_train = atoi(optarg);
256                 if (p_train <= 1) {
257                     fprintf(stderr, "You must send at least two packets per train\n");
258                     exit(1);
259                 }
260                 if ((p_train * n_trains) > MAX_NUMPCK) {
261                     fprintf(stderr, "The maximum number of probing packets is: %d \n", ↩
                            MAX_NUMPCK);
262                     exit(1);
```

```
263                 }
264                 break;
265             case 'g':
266                 pair_gap = atoi(optarg);
267                 if (pair_gap < 0) {
268                     fprintf(stderr, "Gap between packets must be a positive number or zero\n");
269                     exit(1);
270                 }
271                 inpair_gap = pair_gap * 10;
272                 intrain_gap = inpair_gap * 10;
273                 break;
274             case 'i':
275                 inpair_gap = atoi(optarg);
276                 if (inpair_gap < 0) {
277                     fprintf(stderr, "Gap between packets must be a positive number or zero\n");
278                     exit(1);
279                 }
280                 break;
281             case 'n':
282                 intrain_gap = atoi(optarg);
283                 if (intrain_gap < 0) {
284                     fprintf(stderr, "Gap between packets must be a positive number or zero\n");
285                     exit(1);
286                 }
287                 break;
288             case 't':
289                 target_time = atoi(optarg);
290                 if (target_time < 0) {
291                     fprintf(stderr, "Time must be >=0 (seconds)\n");
292                     exit(1);
293                 }
294                 break;
295             case 'v':
296                 verbose = TRUE;
297                 break;
298             case 'h':
299             case '?':
300             default:
301                 snd_usage();
302         }
303     }
304     if (rcv_info == NULL) {
305 
306         fprintf(stderr, "You must specify a receiver host\n");
307         snd_usage();
308     }
309 }
310 
311 //------------------------------------------------------------------------
312 //- Create and establish the UDP socket                                  -
313 //------------------------------------------------------------------------
314 
315 int prep_sockets() {
316     struct protoent *udp;
317     int optval;
318     int optsize;
319 
320     udp = getprotobyname("udp");
321     bzero((void *) &snd_echo, sizeof (struct sockaddr_in));
```

```
322     bzero((void *) &rcv_echo, sizeof (struct sockaddr_in));
323
324     traceband_sock = socket(PF_INET, SOCK_DGRAM, udp->p_proto);
325     if (traceband_sock < 0) {
326         perror("socket2x:");
327         exit(1);
328     }
329
330     snd_echo.sin_family = AF_INET;
331     snd_echo.sin_addr.s_addr = htonl(INADDR_ANY);
332     snd_echo.sin_port = htons(37085);
333
334     rcv_echo.sin_family = AF_INET;
335     memcpy((void*) &(rcv_echo.sin_addr.s_addr), peer_actual,
336             sizeof (rcv_echo.sin_addr.s_addr));
337     rcv_echo.sin_port = htons(SERVER_PORT);
338
339     //12 bytes for payload... will this be combined or not?*/
340     optval = PACKET_SIZE;
341     optsize = sizeof (optval);
342     if (setsockopt(traceband_sock, SOL_SOCKET, SO_SNDBUF, &optval, optsize) < 0) {
343         perror("traceband: SNDBUF");
344         exit(1);
345     }
346     if (bind(traceband_sock, (struct sockaddr*) &snd_echo,
347             sizeof (snd_echo)) < 0) {
348         perror("udp pair bind");
349         exit(1);
350     }
351     if (connect(traceband_sock, (struct sockaddr*) &rcv_echo,
352             sizeof (rcv_echo)) < 0) {
353
354         perror("udp gap connect");
355     }
356     return 0;
357 }
358
359 unsigned short csum(unsigned short *buf, int nwords) {
360     unsigned long sum;
361     for (sum = 0; nwords > 0; nwords--)
362         sum += *buf++;
363     sum = (sum >> 16) + (sum & 0xffff);
364     sum += (sum >> 16);
365     return (unsigned short) (~sum);
366 }
```

Apéndice B

Implementación de *CRC32*

Código B.1. Posible implementación de *CRC32*

```
1  const uint32_t crctable[] = {
2     0x00000000L, 0x77073096L, 0xee0e612cL, 0x990951baL, 0x076dc419L, 0x706af48fL, 0xe963a535L, 0↩
       x9e6495a3L,
3     0x0edb8832L, 0x79dcb8a4L, 0xe0d5e91eL, 0x97d2d988L, 0x09b64c2bL, 0x7eb17cbdL, 0xe7b82d07L, 0↩
       x90bf1d91L,
4     0x1db71064L, 0x6ab020f2L, 0xf3b97148L, 0x84be41deL, 0x1adad47dL, 0x6ddde4ebL, 0xf4d4b551L, 0↩
       x83d385c7L,
5     0x136c9856L, 0x646ba8c0L, 0xfd62f97aL, 0x8a65c9ecL, 0x14015c4fL, 0x63066cd9L, 0xfa0f3d63L, 0↩
       x8d080df5L,
6     0x3b6e20c8L, 0x4c69105eL, 0xd56041e4L, 0xa2677172L, 0x3c03e4d1L, 0x4b04d447L, 0xd20d85fdL, 0↩
       xa50ab56bL,
7     0x35b5a8faL, 0x42b2986cL, 0xdbbbc9d6L, 0xacbcf940L, 0x32d86ce3L, 0x45df5c75L, 0xdcd60dcfL, 0↩
       xabd13d59L,
8     0x26d930acL, 0x51de003aL, 0xc8d75180L, 0xbfd06116L, 0x21b4f4b5L, 0x56b3c423L, 0xcfba9599L, 0↩
       xb8bda50fL,
9     0x2802b89eL, 0x5f058808L, 0xc60cd9b2L, 0xb10be924L, 0x2f6f7c87L, 0x58684c11L, 0xc1611dabL, 0↩
       xb6662d3dL,
10    0x76dc4190L, 0x01db7106L, 0x98d220bcL, 0xefd5102aL, 0x71b18589L, 0x06b6b51fL, 0x9fbfe4a5L, 0↩
       xe8b8d433L,
11    0x7807c9a2L, 0x0f00f934L, 0x9609a88eL, 0xe10e9818L, 0x7f6a0dbbL, 0x086d3d2dL, 0x91646c97L, 0↩
       xe6635c01L,
12    0x6b6b51f4L, 0x1c6c6162L, 0x856530d8L, 0xf262004eL, 0x6c0695edL, 0x1b01a57bL, 0x8208f4c1L, 0↩
       xf50fc457L,
13    0x65b0d9c6L, 0x12b7e950L, 0x8bbeb8eaL, 0xfcb9887cL, 0x62dd1ddfL, 0x15da2d49L, 0x8cd37cf3L, 0↩
       xfbd44c65L,
14    0x4db26158L, 0x3ab551ceL, 0xa3bc0074L, 0xd4bb30e2L, 0x4adfa541L, 0x3dd895d7L, 0xa4d1c46dL, 0↩
       xd3d6f4fbL,
15    0x4369e96aL, 0x346ed9fcL, 0xad678846L, 0xda60b8d0L, 0x44042d73L, 0x33031de5L, 0xaa0a4c5fL, 0↩
       xdd0d7cc9L,
16    0x5005713cL, 0x270241aaL, 0xbe0b1010L, 0xc90c2086L, 0x5768b525L, 0x206f85b3L, 0xb966d409L, 0↩
       xce61e49fL,
17    0x5edef90eL, 0x29d9c998L, 0xb0d09822L, 0xc7d7a8b4L, 0x59b33d17L, 0x2eb40d81L, 0xb7bd5c3bL, 0↩
       xc0ba6cadL,
18    0xedb88320L, 0x9abfb3b6L, 0x03b6e20cL, 0x74b1d29aL, 0xead54739L, 0x9dd277afL, 0x04db2615L, 0↩
       x73dc1683L,
```

```
    0xe3630b12L, 0x94643b84L, 0x0d6d6a3eL, 0x7a6a5aa8L, 0xe40ecf0bL, 0x9309ff9dL, 0x0a00ae27L, 0x7d079eb1L,
    0xf00f9344L, 0x8708a3d2L, 0x1e01f268L, 0x6906c2feL, 0xf762575dL, 0x806567cbL, 0x196c3671L, 0x6e6b06e7L,
    0xfed41b76L, 0x89d32be0L, 0x10da7a5aL, 0x67dd4accL, 0xf9b9df6fL, 0x8ebeeff9L, 0x17b7be43L, 0x60b08ed5L,
    0xd6d6a3e8L, 0xa1d1937eL, 0x38d8c2c4L, 0x4fdff252L, 0xd1bb67f1L, 0xa6bc5767L, 0x3fb506ddL, 0x48b2364bL,
    0xd80d2bdaL, 0xaf0a1b4cL, 0x36034af6L, 0x41047a60L, 0xdf60efc3L, 0xa867df55L, 0x316e8eefL, 0x4669be79L,
    0xcb61b38cL, 0xbc66831aL, 0x256fd2a0L, 0x5268e236L, 0xcc0c7795L, 0xbb0b4703L, 0x220216b9L, 0x5505262fL,
    0xc5ba3bbeL, 0xb2bd0b28L, 0x2bb45a92L, 0x5cb36a04L, 0xc2d7ffa7L, 0xb5d0cf31L, 0x2cd99e8bL, 0x5bdeae1dL,
    0x9b64c2b0L, 0xec63f226L, 0x756aa39cL, 0x026d930aL, 0x9c0906a9L, 0xeb0e363fL, 0x72076785L, 0x05005713L,
    0x95bf4a82L, 0xe2b87a14L, 0x7bb12baeL, 0x0cb61b38L, 0x92d28e9bL, 0xe5d5be0dL, 0x7cdcefb7L, 0x0bdbdf21L,
    0x86d3d2d4L, 0xf1d4e242L, 0x68ddb3f8L, 0x1fda836eL, 0x81be16cdL, 0xf6b9265bL, 0x6fb077e1L, 0x18b74777L,
    0x88085ae6L, 0xff0f6a70L, 0x66063bcaL, 0x11010b5cL, 0x8f659effL, 0xf862ae69L, 0x616bffd3L, 0x166ccf45L,
    0xa00ae278L, 0xd70dd2eeL, 0x4e048354L, 0x3903b3c2L, 0xa7672661L, 0xd06016f7L, 0x4969474dL, 0x3e6e77dbL,
    0xaed16a4aL, 0xd9d65adcL, 0x40df0b66L, 0x37d83bf0L, 0xa9bcae53L, 0xdebb9ec5L, 0x47b2cf7fL, 0x30b5ffe9L,
    0xbdbdf21cL, 0xcabac28aL, 0x53b39330L, 0x24b4a3a6L, 0xbad03605L, 0xcdd70693L, 0x54de5729L, 0x23d967bfL,
    0xb3667a2eL, 0xc4614ab8L, 0x5d681b02L, 0x2a6f2b94L, 0xb40bbe37L, 0xc30c8ea1L, 0x5a05df1bL, 0x2d02ef8dL
};

uint32_t crc32(uint32_t bytes_sz, const uint8_t *bytes)
{
   uint32_t crc = ~0;
   uint32_t i;
   for(i = 0; i < bytes_sz; ++i) {
      crc = crctable[(crc ^ bytes[i]) & 0xff] ^ (crc >> 8);
   }
   return ~crc;
}
```