Universidad Nacional de La Pampa – Facultad de Ingeniería

Proyecto Final de la Carrera Ingeniería en Sistemas:

Simulación de Algoritmos de Control de Congestión en TCP bajo User Mode Linux

Autor: A.P. Rattalino, Damián José
Tutor: Ing. Crespo, Aldo Abel

Año 2014

GENERAL PICO, LA PAMPA

Índice

Capítulo I - Introducción **3**
- 1.0 - Introducción 3
- 1.1 - Control de Congestión 3
- 1.2 - Control de Congestión: Opciones de Experimentación 5
- 1.3 - Objetivos del Trabajo de Tesis 6
- 1.4 - Estructura del Trabajo de Tesis 6

Capítulo II - User Mode Linux (UML) **7**
- 2.0 - Introducción 7
- 2.1 - Configuración e Instalación de User Mode Linux 8
- 2.2 - Ejecutando UML 11
- 2.3 - Opciones de Conectividad de UML 12
  - *2.3.1 - Conectividad a través de dispositivos TUN/TAP* 12
  - *2.3.2 - Conectividad a través de Virtual Bridge* 13
  - *2.3.3 - Conectividad UML a través de uml_switch* 13
  - *2.3.4 - Conectividad UML a través de vde_switch* 14
- 2.4 - Resumen 15

Capítulo III - Simulación de Algoritmos de Control de Congestión bajo User Mode Linux **16**
- 3.1 - Introducción 17
- 3.2.0 - Control de Congestión 18
  - *3.2.1 - Introducción* 18
  - *3.2.2 - TCP Tahoe* 18
  - *3.2.3 - TCP Reno y TCP NewReno* 19
  - *3.2.4 - TCP BIC* 19
  - *3.2.5 - TCP CUBIC* 20
- 3.3.0 - Análisis Experimental 21
  - *3.3.1 - Topología UML* 22
  - *3.3.2 - Consideraciones de configuración* 22
- 3.4.0 - Resultados Experimentales 24
- 3.5.0 - Conclusiones 25
- 3.6.0 - Referencias 25

Capítulo IV - Topología: Configuración y Ejecución de las Simulaciones **27**
- 4.0 - Introducción 27
- 4.1 - Script para la Ejecución del Laboratorio Virtual 28
- 4.2 - Configuración de la Topología 29
- 4.3 - Generación de Tráfico TCP 36
- 4.4 - Generación de Gráficos 37

Capítulo V - Resultados de la Simulación **38**
5.0 - Introducción 38
5.1 - Simulación de NewReno 38
5.2 - Simulación de BIC 39
5.3 - Simulación de Cubic 40

Capítulo VI: Conclusión **42**
6.0 - Conclusión 42

CAPITULO I
Introducción

1.0 Introducción

El protocolo TCP *(Transmission Control Protocol)* es una pieza clave en la pila de protocolos de la arquitectura TCP/IP. Aunque a primera vista, la modelización de TCP como una máquina de estados finitos se ve relativamente simple, al estudiar en profundidad cada aspecto de TCP se toma real dimensión de la complejidad del protocolo. Las primeras implementaciones de TCP se remontan a la década de 1970, y fueron producto de las investigaciones llevadas a cabo por *Robert Kahn* y *Vinton Cerf*, miembros de un selecto grupo de personas consideradas en la actualidad como los padres de Internet. La especificación de TCP como estándar se publicó en el año 1981 (RFC 793).

Si bien el protocolo TCP consta de numerosas aristas, y en cada una de ellas se evidencia una alta dinámica científica, *este trabajo se focaliza en algoritmos para el control de congestión y específicamente en la utilización de herramientas de software para construir un laboratorio de experimentación virtual que simplifique la cantidad de recursos de hardware necesitados, y que arroje resultados fidedignos*.

A continuación, en la sección 1.1 se introducirá conceptualmente el Control de Congestión aplicados a conexiones TCP. Posteriormente, en la sección 1.2 se describirán distintas técnicas utilizadas para la experimentación en laboratorio. En la sección 1.3 se propondrá un novedoso método, alternativo a los descritos en la sección 1.2. Finalmente en la sección 1.4 se detallará la estructura de este trabajo de tesis.

1.1 Control de Congestión

En la actualidad, la mayoría de las aplicaciones de Internet se basan en TCP. TCP es un protocolo orientado a la conexión, que permite a las aplicaciones de red que utilizan su servicio en capa de transporte, intercambiar bytes en forma confiable a través de una red no orientada a la conexión basada en el protocolo IP *(Internet Protocol)*. A pesar de que el control de congestión no formó parte de la formulación inicial de TCP, constituye un elemento esencial del protocolo; pues define el *throughput* (cantidad de tráfico circulante en la red en un momento determinado) asociado a cada conexión TCP.

La mayoría de las aplicaciones de red que utilizan la arquitectura TCP/IP, son del tipo cliente/servidor, y la frase "aplicación de red" conlleva a tres bloques constitutivos: un software cliente, un software servidor y un protocolo mediante el cual interactúan cliente y servidor.

La característica clave del protocolo TCP es su capacidad para proporcionar un canal virtual, bidireccional, libre de errores, libre de segmentos duplicados, que asegura una entrega a la parte receptora en el orden en que los bytes fueron transmitidos por la aplicación emisora. Por todo lo expresado TCP provee la abstracción de un canal perfecto (libre de errores) a las aplicaciones de red que interactúan en la capa de aplicación.

Dado que el protocolo TCP (en capa de transporte) utiliza los servicios del protocolo IP (en capa de red) e IP proporciona un servicio no orientado a la conexión (de mayor esfuerzo) para la entrega de datagramas a través de Internet, el estándar TCP en su versión original (RFC 793) especificó un mecanismo de control de flujo entre los hosts extremos en una conexión, basado en el concepto de ventana deslizante. Es importante aclarar que el control de flujo tiene como propósito que un emisor TCP no sature el buffer de recepción de la parte receptora. Se trata entonces de un mecanismo de control extremo a extremo *(host to host)* basado en el concepto de ventana deslizante, que no considera el estado de la red.

La Figura 1.1 ilustra una ventana correspondiente a la parte emisora TCP. En primer lugar se puede observar de la gráfica que el tamaño de la ventana en el emisor SWS *(Send Windows Size)*, se corresponde con el tamaño del buffer advertido por el receptor (capacidad de *buffer* disponible en el receptor). La parte izquierda de la Figura 1.1 (que incluye hasta el byte 200), corresponde a bytes transmitidos y reconocidos mediante ACKs por el extremo receptor, y por lo tanto fueron eliminados del buffer de transmisión del emisor TCP. Los bytes 201 hasta el 260 (incluido) fueron transmitidos en uno o más segmentos TCP, y aun no fueron reconocidos por la parte receptora de la conexión, y por esta razón es necesario su almacenamiento en el *buffer* TCP emisor, (se eliminaran cuando se reciban los ACKs correspondientes que contemplen la secuencia), el resto de los bytes en la ventana (SWS) son bytes que pueden transmitirse pero que aún no fueron enviados (bytes 261 a 300).

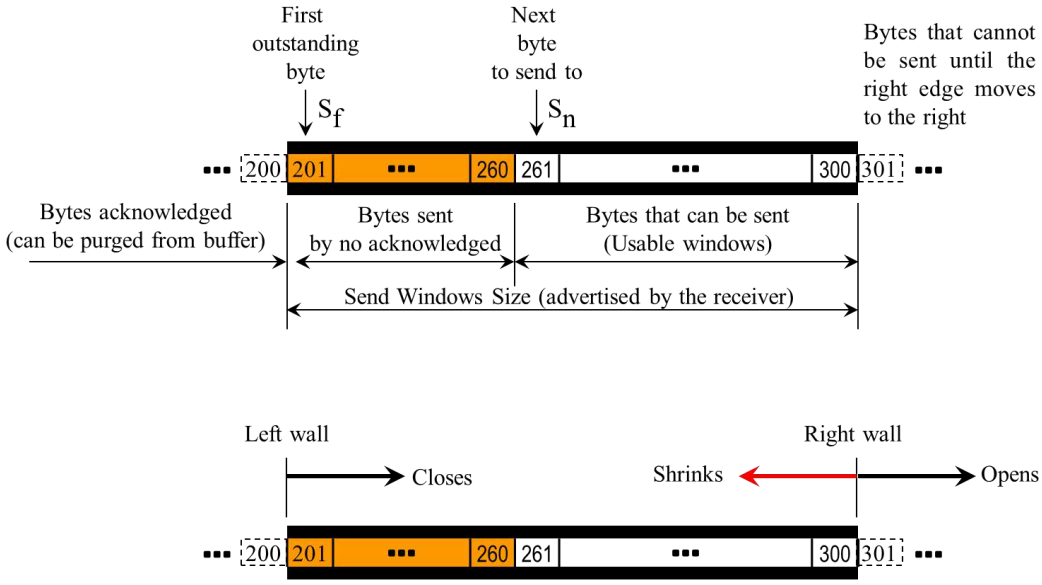

Figura 1.1: Ventana de transmisión de un emisor TCP

El lector comprenderá el término ventana deslizante analizando la parte inferior de la Figura 1.1. Observando la parte izquierda de la Figura 1.1, la ventana se deslizará hacia la derecha conforme se reciben reconocimientos (ACKs) desde la parte receptora. Observando la parte derecha de la gráfica, la ventana se expandirá hacia la derecha cuando el receptor anuncie un tamaño de ventana de recepción *(rwnd)* que comprenda bytes más allá del límite descripto en la Figura 1.1. Si el receptor TCP no actualiza su ventana de recepción, el transmisor solo podrá enviar segmentos TCP que incluyan los bytes 261 a 300 y la ventana se cerrará completamente lo que indica que el emisor agotó su ventana de transmisión y por ende no podrá transmitir segmentos adicionales. La ventana en el emisor TCP sólo se expandirá hacia la derecha cuando el receptor anuncie

un nuevo valor de *rwnd*. Tal como lo ilustra la parte inferior de la Figura 1.1, la ventana en el emisor también puede contraerse *(shrinks)* y este fenómeno puede darse cuando el receptor anuncie un tamaño de *rwnd=x bytes* en el tiempo t_0, e inmediatamente anuncia en t_1 un valor de *rwnd=y*, con $y < x$. La última situación no es deseable y existen recomendaciones para que las implementaciones TCP eviten la circunstancia descrita.

De lo escrito en los párrafos previos de esta sección debe notarse que en la definición de TCP para la Internet incipiente, sólo se consideró la interacción entre los hosts extremos en una conexión TCP (control de flujo), y no fue hasta el año 1986 en que la red de redes (Internet) comenzó a experimentar fallas generalizadas en las conexiones TCP en horarios que evidenciaban una correlación estadística. Lo curioso del fenómeno es que aquellos servicios basados en el protocolo de transporte UDP *(User Datagram Protocol)* no eran afectados.

Se llegó a la conclusión de que la problemática se generaba en los *routers* del núcleo de Internet cuando el volumen de tráfico desbordaba sus *buffers* y producía una situación de *overflow*. No existía por entonces un mecanismo que moderara la velocidad de generación de segmentos TCP en función del estado de la red. En 1988 y a raíz de la problemática descrita, Van Jacobsen propuso, un mecanismo de control de congestión basado en el concepto de ventanas de congestión (*cwnd*).

Ambos algoritmos, para el control de flujo y para el control de congestión deben funcionar de manera simultánea y el valor de la ventana de transmisión permitida en el emisor de una conexión TCP (en bytes) debe ser: $Mínimo\,\{rwnd, cwnd\}$. De esta forma se regula la tasa de transmisión de las fuentes generadoras de tráfico TCP en función de dos variables: una relativa a la capacidad de recepción del host extremo en la conexión, y otra relativa al estado de la red.

A partir del descubrimiento y el impacto que la congestión produce en las conexiones TCP, se produjo una vorágine en la actividad científica tendiente a formular algoritmos que prevengan las situaciones de congestión, que sean equitativos o justos en la asignación de recursos (ancho de banda) entre las distintas conexiones TCP y se adapten a las condiciones de las redes: redes de alta velocidad y elevado retardo con pérdidas de segmentos despreciables (redes basadas en fibra óptica transcontinentales), redes de alta velocidad con elevada tasa de errores (enlaces híbridos, fibra óptica e inalámbricos), etc. De este modo surgieron numerosas definiciones e implementaciones de algoritmos para el control de congestión, que en la actualidad forman parte de los sistemas operativos abiertos o cerrados.

1.2 Control de Congestión: Opciones de Experimentación

En curso típico de Redes de Computadoras es inusual que el tópico control de congestión se experimente. Por lo general se lo aborda de una perspectiva estrictamente teórica y se describen algunos de los algoritmos básicos. En casos excepcionales se construye una topología de experimentación que consta de recursos variados, tales como computadoras, *switchs* y *routers*.

Otro modo de experimentación es a través del uso de complejos simuladores de eventos discretos, tales como NS2 y OPNET, diseñados exclusivamente para el área de Redes y Computadoras. En el último caso, se debe conocer la teoría relativa al área de simulación, y esto no es un hecho menor, por ejemplo, ¿cuantas réplicas de la simulación, basadas en distintas semillas aleatorias deben ser ejecutadas para corroborar

la validez del experimento? ¿Cómo determinar el tiempo mínimo de ejecución de cada replica para que sea representativa de lo que realmente sucede? ¿Cómo extraer resultados finales de las réplicas ejecutadas? ¿Qué distribución probabilística aplicar para generar errores en los segmentos trasmitidos?

Otro método posible de experimentación es a través del acceso a una red WAN, cuyo modelo se debe conocer en detalle. Se debe crear un usuario de la red (WAN) para asignarle un ancho de banda, posteriormente se debe generar tráfico y finalmente obtener las variables que gobiernan el algoritmo de control de congestión para extraer conclusiones. La pregunta es ¿Quién es el afortunado que tiene acceso exclusivo a una red WAN?

También se puede experimentar por medio de un enlace local entre dos *routers* y utilizar un medio de transmisión que por sus características constructiva introduzca un retardo considerable y variable (en función de la longitud) para luego generar errores mediante alguna herramienta de software.

Una posibilidad no explorada es simular el comportamiento de los algoritmos de control de congestión a través de la construcción de topologías virtuales basadas en herramientas para virtualización de computadoras y dispositivos de red virtuales *(switchs y routers)*, y en esta dirección, este trabajo de tesis utilizará *User Mode Linux*, de ahora en más UML, como software para la virtualización de topologías para la experimentación con algoritmos de control de congestión. UML virtualiza la capa de acceso a la red (modelo TCP/IP) y permite la interacción directa y transparente entre protocolos de capa de red, transporte y aplicación. UML es una herramienta que se ejecuta sobre sistemas operativos basados en GNU/Linux.

1.3 Objetivo del Trabajo de Tesis

El objetivo principal de este trabajo es demostrar que es posible experimentar con los diferentes algoritmos de control de congestión soportados por el núcleo Linux sobre una topología de red virtual basada en UML. El trabajo servirá para estudiar el comportamiento de los algoritmos, extraer conclusiones considerando las condiciones dinámicas de las redes WAN, en particular, retardos, variabilidad del retardo *(jitter)* y pérdidas de segmentos. El trabajo propuesto es inédito, y servirá como base para la publicación de artículos futuros relativos a comparación de diferentes algoritmos de control de congestión bajo condiciones idénticas en lo que respecta a retardos y tasa de errores característicos de enlaces en redes de área amplia.

1.4 Estructura del Trabajo de Tesis

Este trabajo de tesis se encuentra organizado del siguiente modo. El capítulo 2 introducirá al lector en aspectos teóricos sobre UML, que comprende los componentes de software necesitados para su instalación, todos los aspectos de la configuración del núcleo Linux para su posterior instalación, y finalmente aspectos de conectividad entre computadoras virtuales. Posteriormente, en el Capítulo 3 se presentará el artículo científico aceptado por CoNaIISI (Congreso Nacional de Ingeniería Informática / Ingeniería de Sistemas). En el Capítulo 4 se detallará en forma exhaustiva la topología utilizada para llevar a cabo el trabajo de experimentación, junto a los scripts utilizados para la generación de los resultados buscados, mientras que en el Capítulo 5 se mostraran los resultados de la simulación obtenidos para los distintos algoritmos de congestión estudiados. Finalmente, el Capítulo 6 detalla las conclusiones obtenidas.

CAPITULO II
User Mode Linux (UML)

2.0 Introducción

User Mode Linux es una poderosa herramienta *open source* basada en Linux que permite virtualizar computadores y redes de computadores. Es ideal para estudiar toda la colección de protocolos TCP/IP y tiene la capacidad de implementar dispositivos de interconexión de redes que se desempeñan en las capas 1, 2 y 3 del modelo de referencia OSI.

Desarrollado por Jeff Dick, UML, permite ejecutar múltiples instancias de Linux bajo Linux y a partir de la versión 2.6.10 fue integrado al *núcleo* de Linux. Desde el punto de vista de usuario, cada instancia Linux creada vía UML tiene su propio núcleo y sistema de archivo *(filesystem)*, independiente del residente en la computadora anfitriona. Desde el punto de vista de la computadora anfitriona UML se ejecuta como un proceso normal de usuario al que se le proporciona memoria virtual y acceso a los dispositivos del host anfitrión.

Inicialmente UML fue concebido para ser utilizado como plataforma de prueba para los desarrolladores de software en Linux y la razón es de fácil comprensión: un fallo en el código de software bajo prueba no provoca ninguna degradación en el sistema operativo de la computadora anfitriona. Posteriormente se visualizó un campo de aplicación mucho más extenso relacionado al área de las Redes de Computadoras. Para tal fin UML soporta interfaces virtuales de red y la creación de dispositivos tales como *hubs*, *switchs*, *bridges* y *routers*. Todos los dispositivos pueden conectarse entre sí a través de las interfaces virtuales de red definidas en tiempo de configuración y de este modo una computadora anfitriona ejecutando distintas instancias de UML se transforma en una herramienta ideal para experimentar con complejos escenarios de red.

Una gran variedad de escenarios de red pueden ser elaborados en base a un curso típico de redes de computadoras. En lo referido a la capa 2 del modelo de referencia OSI, es posible experimentar con protocolos y dispositivos propios de una red Ethernet. En el nivel 3, o capa de red es posible crear escenarios con múltiples redes y subredes vinculadas por medio de *routers* ejecutando enrutamientos estáticos o algoritmos de enrutamientos dinámicos o plantear escenarios que involucren el uso de IPv6. En lo que respecta a capa de transporte se puede experimentar con cualquier protocolo de transporte (TCP o UDP) y en lo relativo a capa de aplicación todos los servicios pueden ser experimentados.

A continuación, en la sección 2.1 se describirán los pasos necesarios para una correcta instalación de UML, la sección 2.2 tratará sobre aspectos de ejecución de instancias UML, en la sección 2.3 introducirá al lector en aspectos de conectividad entre instancias UML y el mundo exterior y finalmente en la sección 2.5 se presentaran las conclusiones de este capítulo.

2.1 Configuración e Instalación de User Mode Linux

Para proceder a la instalación de *User Mode Linux* se necesitan distintos elementos de software. Una versión del núcleo Linux (desde www.kernel.org) independiente del host anfitrión pues el núcleo de Linux tiene la capacidad de ejecutarse en el espacio de usuario como cualquier otro proceso. Un *filesystem* basado en alguna de las numerosas versiones existentes de Linux (igual o distinta a la existente en el host anfitrión) y las denominadas *uml utilities*.

Una vez que se dispone de los elementos de software mencionados, el proceso de instalación es sencillo. A continuación se indican los pasos necesarios (se supone creado el usuario `user`, en la computadora cuyo `hostname` es `PC`):

1. Se descomprime el núcleo en `/home/user`:

```
user@PC:/$ cd /home/user
user@PC:/$ tar -xvf linux-3.2.59.tar.gz
```

2. El resultado del último comando en (1) crea el directorio `linux-3.2.59` en `/home/user` que contiene las fuentes del núcleo para la versión escogida. En este punto es necesario configurar el núcleo Linux para la arquitectura UML. Se deben seleccionar las opciones y los módulos del núcleo de acuerdo a los requerimientos que exige la experimentación con algoritmos de control de congestión. Una estrategia de configuración inicial es a partir de una configuración por defecto introduciendo los siguientes comandos:

```
user@PC:/$ cd linux-3.2.59
user@PC:/$ make menuconfig ARCH=um defconfig
```

Como respuesta emergerá una ventana, tal como se observa en la Figura 2.1, con las opciones por defecto del núcleo Linux para la arquitectura UML. Luego se escoge la opción "guardar archivo de configuración" y en forma automática se creará el archivo `.config`. Se parte de un archivo de configuración por defecto para asegurar un proceso de compilación sin errores, para posteriormente agregar en forma controlada las opciones de configuración requeridas para la experimentación con algoritmos de control de congestión. La estrategia descrita permite determinar unívocamente cuando una opción de configuración causa un error de compilación, y cuando este es el caso, se analizan las causas y el modo de evitarlo.

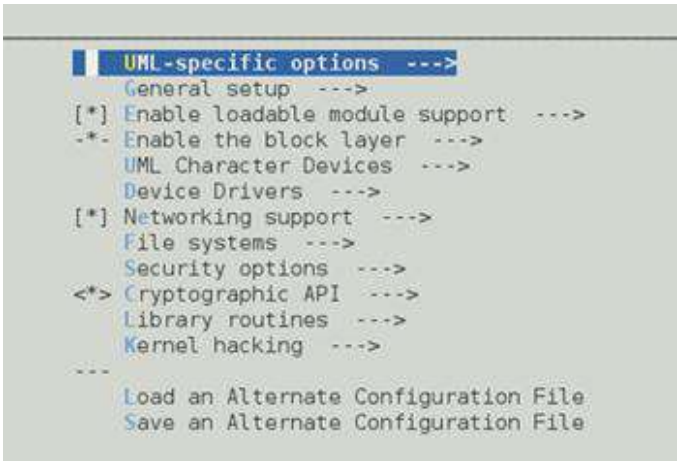

Figura 2.1: Configuración por defecto (defconfig) para UML

Como mecanismo de transporte de tramas Ethernet entre los dispositivos involucrados en una topología de experimentación, fueron escogidas las opciones de configuración `pcap` y `VDE` *(Virtual Distributed Ethernet)*, opciones que requieren de la instalación en el host anfitrión (como usuario root) de la librería `libpcap` y del paquete de software `vde2-2.3.2` (ver Figura 2.2):

```
root@PC:/$ apt-get install libpcap-dev
root@PC:/$ apt-get install vde2-2.3.2
```

Los últimos comandos son los requeridos para instalar software desde los repositorios para una distribución en el host anfitrión basada en Ubuntu.

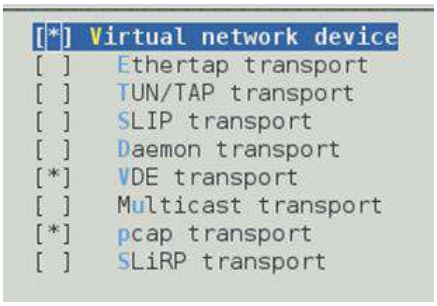

Figura 2.2: Opciones seleccionadas para el transporte de tramas Ethernet

En la sección de configuración *"TCP Advanced Congestion Control"* es necesario habilitar los módulos para los algoritmos de Control de Congestión a experimentar: BIC, Cubic y NewReno. Notar que Bic y Cubic requieren de una habilitación explicita como módulos, aspecto no necesario para NewReno pues en caso de no habilitar ninguno de los módulos, será el algoritmo utilizado por defecto (Figura 2.3).

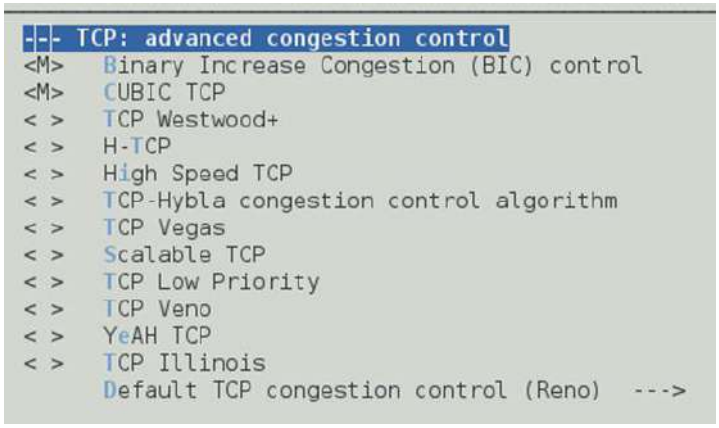

Figura 2.3: Selección de módulos para el control de congestión

Es necesario aclarar que en la versión del núcleo Linux (3.2.59) para ser utilizada con UML, Reno es en realidad NewReno, y ello se puede comprobar a través de la edición del código fuente del algoritmo y leer los comentarios iniciales de la implementación.

Para experimentar con algoritmos de control de congestión será necesario construir una topología de red que involucre distintas redes IP bajo Ethernet, con tasas de transmisión predefinidas (por ejemplo, 10Mbps, 100Mbps o 1Gbps). En un ambiente de computadoras y dispositivos virtuales, la tasa de

trasmisión de tramas Ethernet se define por la frecuencia con que el núcleo Linux real atiende a los procesos de las computadoras y dispositivos virtuales. Ello significa que es necesario utilizar una disciplina de filas (o colas) para adaptar las tasas de transmisión a los valores deseados. Por esta razón es necesario configurar en la sección *"QoS and/or fair queueing"* las opciones, NETEM *(Network Emulator)* y TBF *(Token Bucket Filter)*. NETEM permite emular el comportamiento de enlaces de área amplia (WAN) y TBF es una simple disciplina de filas que permite fijar la tasa de transmisión de las interfaces de red virtuales (Figura 2.4).

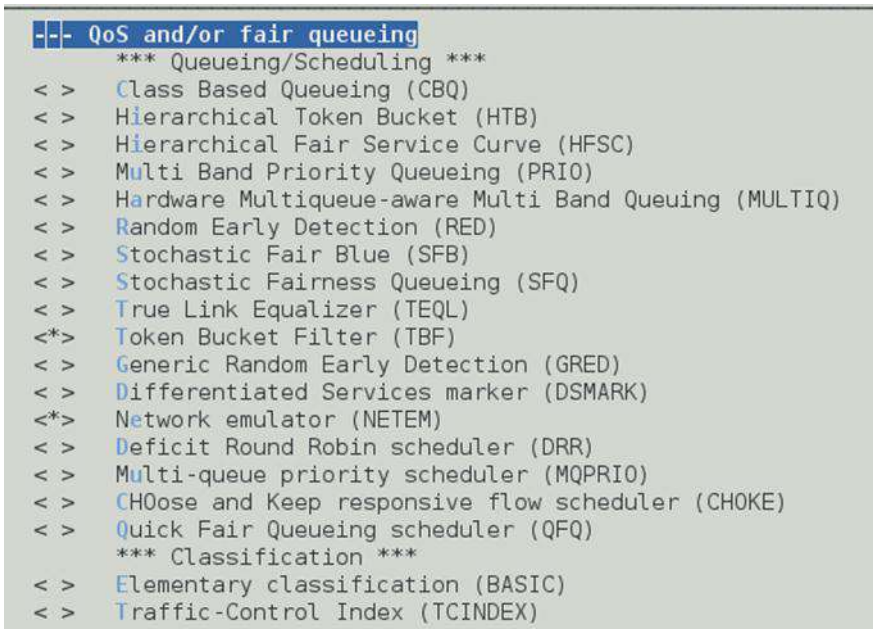

Figura 2.4: Configuración de la sección QoS

Por defecto el núcleo configurado para UML, tendrá la variable de configuración HZ en 100. Ello significa que sólo generará interrupciones al núcleo real en la computadora anfitriona cada 10 ms. Para obtener mayor granularidad en las interrupciones generadas al nucleo real en la computadora anfitriona, será necesario cambiar el parámetro HZ a un valor que permita interrupciones en el orden del milisegundo (ms). Para tal fin se editaran los archivos `./arch/um/Kconfig.common, ./include/asm-generic/param.h`, luego se configura en 1000 la variable HZ y finalmente en el archivo `.config`, se modifica el valor por defecto al nuevo valor de HZ (1000).

Resta salvar la configuración del núcleo y automáticamente se generará el archivo `.config` en `/home/user/linux-3.2.59.`

3. En este punto se debe compilar el núcleo Linux para la arquitectura considerada (ARCH=um)

```
user@PC:/$ make linux ARCH=um
```

4. Como resultado de la compilación en el punto (3) se generará la imagen del núcleo UML (ejecutable binario), en `/home/user/linux-3.2.59`, cuyo nombre es `linux`. La siguiente tarea a realizar es descargar el software `uml_utilities` e instalarlo en la computadora anfitriona:

```
root@PC:/$ tar -xvf uml_utilities.tar
root@PC:/$ make && make install
```

El paquete de software `uml_utilities` contiene diversas herramientas útiles para trabajar con UML. Por ejemplo, provee el comando `uml_console` que entre otras funciones sirve para detener en forma controlada computadoras y/o dispositivos virtuales (routers), provee una implementación de un *switch* virtual a través del comando `uml_switch` y otros comandos útiles a la hora de construir *scripts* para automatizar tareas de ejecución de topologías virtuales.

5. El último paso necesario antes de ejecutar UML, es escoger una versión de un archivo de sistema *(filesystem)* basado en cualquier distribución. En el ámbito de este trabajo de tesis se utilizó un *filesystem* basado en la distribución Slackware versión 12.0.
6. Una vez realizados los pasos descritos, se está en condiciones de ejecutar una o más instancias UML, soportadas por su propio núcleo Linux y *filesystem*. Ello significa que un usuario puede tener privilegios de *root* en una máquina virtual y carecer de idénticos privilegios en el host anfitrión.

2.2 Ejecutando UML

El proceso de arranque de una máquina virtual UML tiene más similitudes que diferencias respecto al que ocurre en un sistema Linux ejecutándose sobre una arquitectura de hardware x86. La principal diferencia es producto de que UML no tiene que inicializar *drivers* para controlar el hardware. Durante el arranque, UML debe montar su propio sistema de archivos *(filesystem)* y para que ello ocurra es necesario indicar el dispositivo de bloque, *Block Device Driver* o dispositivo ubd. Una opción de configuración más explícita debe indicar la cantidad de memoria física suministrada a la instancia UML en el host. De esta forma la ejecución de una instancia UML desde consola inicialmente debe constar de los siguientes comandos:

```
user@PC:/$ ./linux mem=512M ubd0=root_fs
```

La directiva anterior asigna a la instancia UML 512MBytes de memoria *sparse* (archivos que en lugar de ocupar un número fijo de sectores del disco rígido, se definen a partir de un puntero de *offset*, y físicamente ocupan el espacio de memoria definido por la amplitud del *offset*) y un dispositivo de bloque ubd0 desde donde se accesará al sistema de archivos escogido.

Un factor a destacar es que cada instancia UML necesita de su propio *filesystem.* Un escenario de experimentación que por ejemplo consista de 10 máquinas virtuales, en lo relativo al espacio de disco, consumirá 10 veces el tamaño del *filesystem* individual escogido. Por ejemplo, si el tamaño del *filesystem* es de 600 MB, ello implica que las 10 instancias de UML consumirán 6 GB de espacio en disco en el host anfitrión. Para evitar este inconveniente el diseñador de UML contempló el uso de la técnica COW (*Copy On Write*) en que se dispone de un único *filesystem* de solo lectura, para cualquier número de instancias UML ejecutadas, y las modificaciones necesarias que una máquina virtual debería realizar sobre el *filesystem* original se realizan en un archivo COW propio de la instancia. Ello implica un considerable ahorro de espacio en disco, puesto que cada archivo COW es *sparse*. A continuación, los parámetros necesarios

desde línea de comando para ejecutar dos máquinas virtuales utilizando la técnica COW:

```
user@PC:/$ ./linux mem=128M ubd0=root_fs0.cow,root_fs
user@PC:/$ ./linux mem=128M ubd0=root_fs1.cow,root_fs
```

Hasta el momento, las máquinas virtuales están aisladas del host anfitrión y la utilización de un *filesystem* preconstruido para UML no incluirá todos los paquetes de software para todos los fines y perfiles de aplicación de la herramienta virtual. Una forma de hacer visible el *filesystem* del host anfitrión desde una máquina UML es mediante el siguiente comando:

```
uml# mount –t none rootfs /mnt/host
```

Ahora una máquina virtual UML tiene acceso al archivo de sistema del host anfitrión y ello permitirá transferir datos entre ambos *filesystems*. Aunque este es un método simple y sencillo de implementar, es limitado. Una solución más amplia contemplaría la conexión de una máquina virtual al mundo exterior y es precisamente este tópico el que se abordará en la próxima sección de este trabajo.

2.3 Opciones de Conectividad de UML

2.3.1 Conectividad a través de dispositivos TUN/TAP

Aunque UML provee numerosos mecanismos para crear sus interfaces de red, TUN/TAP es el más utilizado y unánimemente recomendado. TUN/TAP se denomina a un *drivers* de red en el núcleo Linux cuyo fin es implementar dispositivos de red virtuales que son enteramente soportados por software a diferencia de lo ocurre con drivers que interaccionan con algún *hardware* de red. TUN/TAP le permite a una instancia UML, intercambiar paquetes con el host y puede ser visto como un dispositivo Ethernet que en vez de recibir tramas desde el medio físico las recibe desde un programa en espacio de usuario y en vez de transmitir tramas al medio físico, las escribe en un programa de espacio de usuario.

Un dispositivo TUN/TAP es similar a un segmento Ethernet que conecta el host con la instancia UML, por lo tanto cada extremo del cable necesita de una dirección IP. Cada datagrama que es encaminado a través del dispositivo tap0 por el host anfitrión, es enviado al proceso que creo la interfaz virtual (máquina UML) en el archivo `/dev/net/tun`.

Es importante destacar que las interfaces TUN/TAP y eth0 de cada instancia UML tienen su propio dominio de *broadcast* diferente al de las interfaces reales en la red con el *host*. Por este motivo, aquellos servicios que para su normal operación se valen de la transmisión a la dirección de difusión podrían no trabajar si erróneamente se considerará un solo dominio de difusión. Para solucionar el problema se deben habilitar en el *host* anfitrión de UML, el servicio de *proxy_arp* y el de reenvío de datagramas o *ip_forward*. En síntesis, TUN/TAP requiere que el host encamine paquetes hacia y desde UML, además de la configuración de los servicios de *proxy_arp* e *ip_forward*.

2.3.2 - Conectividad a través de Virtual Bridge

Una alternativa que conecta instancias UML al mundo exterior y que no exige de tantos requisitos como el citado anteriormente, es mediante la utilización un dispositivo virtual denominado *virtual bridging* (el dispositivo virtual bridge, requiere de la instalación del software *bridge-utilities* en el host anfitrión de UML). Este dispositivo reenvía tramas Ethernet por sus interfaces basado en la dirección MAC destino y su comportamiento es análogo a un *switch* en capa 2, de este modo y a diferencia del primer mecanismo descrito, *virtual bridging* define un único dominio de *broadcast* entre las máquinas virtuales y la red de host. A continuación, se enumeran los pasos necesarios para la configuración de un sistema que consta de un dispositivo *bridge* virtual con dos interfaces que se conectan a una instancia UML y a la interfaz ethernet del host anfitrión respectivamente:

I. Vía línea de comando, se crea una máquina virtual cuyo nombre es PC1:

```
user@PC:/$ ./linux umid=PC1 ubd0=root_fs
```

II. Se crea un dispositivo virtual TUN/TAP y se le asigna el nombre IHMV

```
user@PC:/$ ./tunctl –u root –t IHMV
```

III. Se crea el dispositivo virtual bridge1:

```
user@PC:/$ brctl addbr bridge1
```

IV. Transitoriamente se asigna la dirección IP especial 0.0.0.0 al dispositivo TUN/TAP creado y a la interfaz eth0 del host.

```
user@PC:/$  ifconfig IHMV 0.0.0.0 up
user@PC:/$  ifconfig eth0 0.0.0.0 up
```

V. Se vinculan ambas interfaces al *bridge*:

```
user@PC:/$  brctl addif bridge1 IHMV
user@PC:/$  brctl addif bridge1 eth0
```

VI. En este punto es necesario asociar la interfaz virtual TUN/TAP (creada en el paso II) con la interfaz eth0 de la máquina virtual PC1. Ello se consigue mediante el siguiente comando:

```
user@PC:/$ uml_mconsole PC1 config  /
eth0=tuntap,IHMV,fe:fd:c0:a8:00:fd
```

VII. Por último, se configuran las interfaces de red en el host y en la máquina virtual respectivamente.

2.3.3 - Conectividad UML a través de uml_switch

Los mecanismos hasta aquí descritos constituyen herramientas válidas para conectar instancias UML al mundo exterior. Un mecanismo que permite crear redes virtuales aisladas del mundo exterior, es a través de la utilización del demonio *uml_switch*, el cual es parte de las *utilities* de UML. Este método difiere de los previamente desarrollados en el sentido de que antes de proceder a la creación de las máquinas virtuales, es necesario ejecutar en el host el proceso *uml_switch* quien abrirá un *socket* tipo UNIX cuyo único propósito es comunicar las instancias UML mediante datagramas:

```
user@PC:/$ uml_switch –unix /tmp/uml_1.ctl
```

Ahora se pueden ejecutar máquinas virtuales y conectarlas al dispositivo uml_switch, por ejemplo, el siguente comando, crea una máquina virtual denominada PC1 y la conecta al dispositivo *uml_switch* en el socket uml_1.ctl:

```
user@PC:/$ ./linux umid=PCV1 ubd0=root_fs1.cow,root_fs /
eth0=daemon,,unix,/tmp/uml_1.ctl
```

Un factor a destacar es que *uml_switch* puede configurarse para actuar según dos modos de operación: como *switch* Ethernet, o como *hub* Ethernet. En el modo *switch* memoriza cada una de las direcciones MAC de las máquinas a él conectadas y al reenviar un *frame* sólo lo hace por el puerto vinculado a la dirección MAC destino. En el modo *hub*, los frames son enviados a través de todos los puertos virtuales definidos y constituye una herramienta útil cuando se pretende capturar tráfico de todas las máquinas virtuales en la red. El comando que permite configurar el dispositivo virtual en modo *hub* es:

```
user@PC:/$ uml_switch –hub –unix /tmp/uml_1.ctl
```

No es exagerado afirmar que *uml_switch* constituye una herramienta ideal para crear redes totalmente virtuales a los efectos de experimentar con ellas. Complejas topologías que requerirían de un considerable número de hosts y dispositivos de interconexión se pueden resumir en un solo computador bajo Linux.

2.3.4 - Conectividad UML a través de vde_switch

Este conector es una versión alternativa a *uml_switch*. Su principal ventaja con respecto a este es que permite conectarse a él a cualquier tipo de cliente: QEMU, KVM, UML, Interfaces TUN/TAP e Interconexiones Virtuales. Debido a esta flexibilidad que posee *vde_switch*, es que fue seleccionada para realizar las diferentes pruebas realizadas en este trabajo. Su sintaxis es similar a la utilizada en *uml_switch*. A continuación, se muestra como crear una instancia de este:

```
user@PC:/$ vde_switch -s /tmp/sw1.ctl
user@PC:/$ vde_switch -s /tmp/sw2.ctl -tap tap0 -m 666
```

El primer comando es prácticamente igual al que se utiliza para crear instancias de *uml_switch*. El segundo, por el contrario, crea una instancia de *vde_switch* la cual estará conectada a una interfaz TAP, la cual se corresponde con una interfaz virtual del host anfitrión.

Un comando que crea una computadora virtual (PC) con una interfaz Ethernet conectada al switch vde es la siguiente:

```
host# ./linux umid=PC ubd0=root_fs1.cow,root_fs /
eth0=vde,/tmp/sw1.ctl con0=xterm
```

2.4 Resumen

Este capítulo fue destinado para el lector no familiarizado con *User Mode Linux*. En él se abordaron los principales aspectos relativos a los componentes necesarios para su instalación, ejecución y distintos modos de conectividad entre instancias UML y con la red en el host anfitrión. Quien desee estudiar en profundidad todo lo relativo a UML, la referencia bibliográfica obligada es la numerada como [14] en el artículo principal que compone este trabajo de tesis en el Capítulo 3.

CAPITULO III
Simulación de Algoritmos de Control de Congestión bajo User Mode Linux

Simulación de Algoritmos de Control de Congestión en TCP bajo User Mode Linux

Abstract

La experimentación con protocolos y servicios de aplicación bajo TCP/IP requiere de una costosa infraestructura de laboratorio que debe ser actualizada y/o renovada periódicamente, considerando la alta dinámica científica-tecnológica del área. Por lo general, es necesario crear topologías con múltiples redes y/o subredes pobladas de computadoras, y cuando este es el caso, la experiencia de laboratorio forzosamente debe ser grupal, con las desventajas que ello implica. Una alternativa superadora, que permite mejorar la calidad de enseñanza, es a través de la utilización de tecnologías para la virtualización de computadoras y dispositivos de interconexión, posibilitando la construcción de complejos escenarios de red sin otro requerimiento más que el de disponer de una computadora de medianos recursos. En esta dirección, el presente trabajo muestra como experimentar con algoritmos de control de congestión en TCP, utilizando topologías virtuales construidas en User Mode Linux (UML) [1]. UML es una tecnología que permite virtualizar computadoras y redes de computadoras sobre Linux.

Palabras Claves: TCP/IP, UML, Redes, Algoritmos Control de Congestión, Netem, Kprobes.

3.1. Introducción

Para el dictado de un curso típico de Redes de Computadoras, se requiere de las bases teóricas del área, para luego entrar a otra, en que el dictado se puede transformar es una conjugación muy estrecha entre teoría y práctica. El límite entre ambos modos de enseñanza suele ser el estudio exhaustivo de las funciones de la capa de enlace de datos respecto del modelo de referencia OSI. Luego, todo lo relativo al estudio de protocolos y servicios de aplicación de red puede abordarse desde la perspectiva simultánea antes mencionada. Mediante esta técnica, el alumno puede asimilar conceptos y reafirmarlos desde la experimentación. Si la experimentación es individual, el proceso de enseñanza mejora notablemente, pues desde la óptica docente, permite unívocamente determinar puntos de incertidumbre, que deben ser aclarados para que el proceso de aprendizaje converja al entendimiento definitivo.

Un laboratorio de experimentación basado en tecnologías para la virtualización de computadoras y dispositivos de interconexión que permita la construcción de complejas topologías de red, es la única vía para lograr la modalidad de enseñanza antes descrita. Ha de notarse que las empresas más renombradas en la producción de dispositivos relacionados a las redes de cualquier envergadura, proveen soluciones propietarias a sus clientes para la virtualización de sus productos. De esta forma, un administrador, antes de realizar modificaciones en la configuración real de una infraestructura de red, lo hace sobre una réplica virtual, y si la nueva configuración responde a sus expectativas, finalmente las aplicará en la infraestructura real.

Si bien existen múltiples opciones *open source* [2] respecto a tecnologías de virtualización para construcción de redes de computadoras, en el ámbito de este trabajo se utilizará User Mode Linux, de ahora en más por su acrónimo UML. UML es un núcleo Linux, modificado para ser compilado sobre GNU/Linux. Las computadoras o dispositivos virtuales se soportan sobre un núcleo Linux compilado para la arquitectura UML, que se ejecuta como un proceso de usuario sobre el núcleo Linux convencional en la computadora anfitriona. De esta manera, es posible crear redes virtuales que respondan a topologías arbitrariamente complejas.

Absolutamente todo lo relativo al conjunto de protocolos TCP/IP puede ser experimentado con UML, y en esta

dirección, este artículo muestra cómo experimentar con algoritmos de control de congestión. El control de congestión aplicable a flujos de bytes sobre conexiones TCP es uno de los tópicos que en un curso típico de redes de computadoras se lo aborda por lo general, desde una perspectiva teórica, y este artículo contribuye con un novedoso método de experimentación que no requiere de la utilización de complejos simuladores de eventos discretos, tal como NS [3] y OPNET [4], ni de la variada cantidad de dispositivos necesitados para construir una topología real. A tal fin, se virtualizará con UML una topología de red apta para él estudio de algoritmos para el control de congestión, posteriormente se llevarán a cabo los experimentos considerando las características típicas de una infraestructura de red WAN en lo que se refiere a retardos y variaciones de retardo *(jitter)*.

Este trabajo está organizado del siguiente modo. La sección 3.2.0 tratará sobre aspectos teóricos de algunos de los algoritmos de control de congestión escogidos para la simulación con UML, en la sección 3.3.0 se describirá la topología de experimentación implementada con UML y las herramientas utilizadas para llevar a cabo el análisis experimental, en la sección 3.4.0 se presentarán los resultados experimentales y finalmente en la sección 3.5.0 se presentarán las conclusiones del artículo.

3.2.0 Control de Congestión

3.2.1 Introducción

El término congestión hace referencia a un fenómeno no deseado, generado en una red, o en una parte de su infraestructura, cuando distintas fuentes inyectan un volumen de tráfico superior al que los dispositivos de conmutación de paquetes pueden procesar. En octubre de 1986 se evidenció por primera vez un colapso debido al fenómeno de congestión y a partir de ese evento, la situación se agravaría en la medida que más computadoras se conectaban a Internet.

3.2.2 TCP Tahoe

En 1988 y a raíz de la problemática descrita, Van Jacobsen propuso en [5], un mecanismo de control de congestión basado en el concepto de ventanas de congestión (*cwnd*) y en la definición de tres algoritmos: arranque lento *(slow start)*, prevención de congestión *(congestion avoindace)* y retransmisión rápida *(fast retransmit)*.

A cada extremo en una conexión TCP se le asigna una ventana de congestión, *cwnd*, indicativa del número de segmentos que puede enviar a la red sin esperar por reconocimientos explícitos desde el receptor (ACK). Al inicio, en la fase de operación de arranque lento *(slow start)*, *cwnd* se configura en uno o dos segmentos de máximo tamaño (MSS) y su valor se incrementará en un segmento por cada reconocimiento recibido desde su par receptor (ACK). Durante la fase operativa de arranque lento, *cwnd* experimentará un crecimiento exponencial, tal como se ilustra en la Figura 3.1.

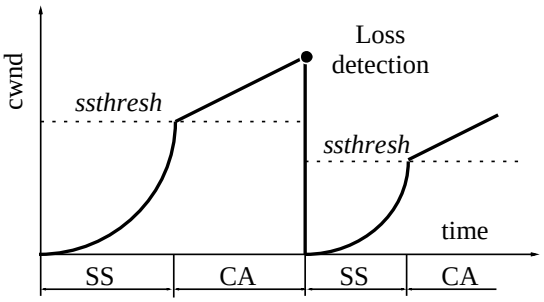

Figura 3.1: TCP Tahoe

Cuando *cwnd* alcanza el valor de la variable "umbral de arranque lento" o *slow start threhold* (*ssthresh*), el mecanismo de control de congestión conmuta al modo *congestion avoindance* y *cwnd* se incrementa linealmente a razón de un segmento por grupo de reconocimientos (ACKs), que se corresponden con el valor actual de la ventana de congestión (Figura 3.1).

Se abandona el estado de *congestion avoindace* cuando el emisor TCP detecta una pérdida por expiración de *timeout* o por

la recepción consecutiva de tres reconocimientos duplicados (ACKs). En ambos casos el emisor reconfigura la variable *ssthrehold* al valor *cwnd/2*, retransmite el segmento perdido *(fast retransmit)*, y conmuta a la fase de operación *slow start* (SS) actualizando *cwnd=1*.

3.2.3 TCP Reno y TCP NewReno

TCP Reno surgió en 1990 como el sucesor de TCP Tahoe y fue implementado por primera vez en el sistema operativo 4.3BSD Unix. Si bien mantuvo las características de su antecesor, incorporó el algoritmo de recuperación rápida o *fast recovery* [6]. El algoritmo de recuperación rápida (FR) evita el ingreso a la fase de arranque lento (SS), cuando se detectan tres reconocimientos duplicados (ACKs) en la fase prevención de congestión (CA) y bajo esta circunstancia el emisor ejecuta las siguientes acciones: reconfigura la variable *ssthresh* al valor *cwnd/2*, retransmite el segmento perdido y actualiza la ventana de congestión a *cwnd=ssthresh+3*, finalmente abandona la fase de recuperación rápida para regresar a la fase de prevención de congestión (CA) (Figura 3.2).

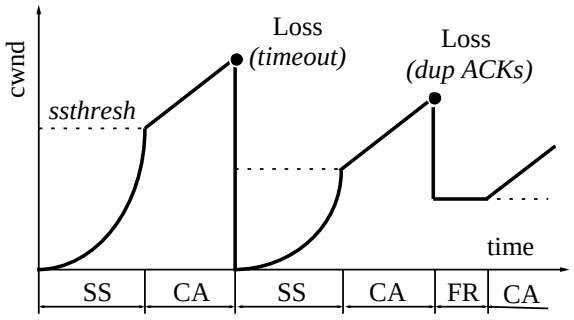

Figura 3.2: Dinámica de cwnd en Reno

Una de las vulnerabilidades de TCP Reno se manifiesta cuando en la fase *fast recovery* (FR) se origina un evento de pérdida de múltiples segmentos para el valor actual de *cwnd*. Bajo esta circunstancia TCP Reno alternará su operación entre las fases CA/FR, y provocará una reducción sucesiva del valor de la variable umbral de arranque lento *(ssthresh)*, y de la ventana de congestión *(cwnd)*. Como consecuencia se produce una importante disminución en el desempeño *(thrughput)* de la conexión TCP.

Atentos a la problemática descrita, Floyd y otros en [7] y [8] introdujeron un simple refinamiento al algoritmo *fast recovery* (FR). En la fase *congestion avoindance* (CA), al recibir tres ACK´s duplicados, ingresa a la fase *fast recovery*, pero a diferencia de TCP Reno, no abandona la fase *fast recovery* hasta que se produzca la confirmación *(acknowledged)* de todos los segmentos de la ventana de congestión *(cwnd)*. Más formalmente, TCP NewReno utiliza una variable de estado adicional para registrar el número de secuencia del último segmento TCP enviado antes de entrar a la fase *fast recovery*.

TCP NewReno resuelve el problema de bajo desempeño ante eventos de múltiples pérdidas en la ventana de congestión *(cwnd)* pero como contrapartida emplea un tiempo excesivo en el proceso de recuperación, pues le toma un *Round Trip Time* (RTT) recuperar cada uno de los segmentos perdidos en la ventana de congestión. Diversas estrategias se formularon para mejorar el desempeño de TCP NewReno en la fase *fast recovery* y una excelente referencia que investiga exhaustivamente numerosas propuestas puede encontrarse en [9].

3.2.4 TCP BIC

El desempeño del protocolo TCP basado en TCP Reno y TCP NewReno no es satisfactorio en redes de alta velocidad, con elevados RTT entre los host que se comunican, y el factor limitante es el comportamiento conservativo de ajuste de la ventana de congestión que gobierna la tasa de transmisión del emisor [10]. Por tal razón surgieron una serie de propuestas basadas en reformular la vía con la cual TCP adapta la ventana de congestión.

Un intento exitoso para crear un control de congestión que pueda escalar en

redes con alto valor BDP *(bandwidth delay product)* fue propuesto por Xu y otros en [11]. BIC *(Binary Increase Congestión control)* extiende TCP NewReno con una fase adicional denominada de convergencia rápida *(Rapid Convergence* o *RC)*. La fase RC, tiene como objetivo descubrir mediante una rápida búsqueda binaria, el valor óptimo de la ventana de congestión en función de los recursos actuales de la red. El proceso de búsqueda binaria es como se ilustra en la Figura 3.3.

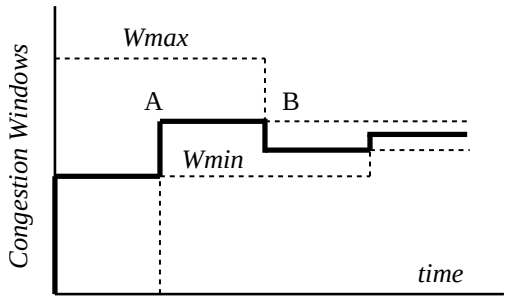

Figura 3.3: Búsqueda binaria de cwnd

Inicialmente se establecen las variables *Wmin=1* y *Wmax* en un valor arbitrariamente alto. Si la red entrega correctamente todos los segmentos emitidos (el emisor recibe todos los ACKs correspondientes al último RTT), la ventana de congestión se actualiza al valor promedio entre *Wmax* y *Wmin* (A en la Figura 3.3) y *Wmin* adquiere el tamaño de la ventana de congestión previa. Cuando se detecta una pérdida, BIC reduce *Wmax* al valor actual de la ventana de congestión (B en la Figura 3.3) y como en TCP NewReno ingresa a la fase de recuperación rápida (FR).

Para incrementar la tasa de convergencia, en ambientes de bajo nivel de pérdidas, BIC reduce el coeficiente de decremento multiplicativo desde 0.5 a 0.125, para lograr mayor velocidad de respuesta.

Las características del algoritmo de búsqueda binaria se traducen en tiempos de convergencia muy bajos que pueden provocar degradación en conexiones TCP. Si la ventana de congestión se incrementa drásticamente, un grupo significativo de segmentos pueden perderse. Por esta razón BIC implementa un algoritmo de arranque lento restringido [12] que limita el incremento en cada paso de *slow start* a un valor máximo de 100 segmentos. En la misma dirección, BIC impone límites al crecimiento de la ventana de congestión en la fase de convergencia rápida cuando el rango de búsqueda binaria es muy amplio, y no permite incrementar la ventana de congestión más allá de un valor predefinido S_{max}.

En el caso opuesto, cuando el rango de búsqueda binaria es pequeño, y por tanto cercano al valor óptimo, BIC define el valor constante S_{min} como un quantum de incremento de la ventana de congestión. Cuando el valor actual de la ventana de congestión alcanza el valor óptimo, o lo excede, BIC conmuta a la fase de arranque lento restringido, no limitado por el valor umbral de *ssthresh (slow start threshold)*. El propósito de esta acción es descubrir un nuevo límite superior para luego reiniciar la fase de búsqueda binaria. La Figura 3.4 ilustra la dinámica de la ventana de congestión de una conexión para TCP BIC.

Figura 3.4: Dinámica de cwnd en BIC

3.2.5 TCP CUBIC

Rhee y Xu en [13] observaron que TCP BIC presenta problemas cuando dos o más flujos de bytes con diferencias significativas en sus RTT compiten por el ancho de banda en un enlace de características restrictivas *(bottleneck link)*. Concretamente en [13] se demostró que flujos de bytes con pequeños RTT escalan rápidamente su ventana de congestión al

valor óptimo y consumen los recursos de la red, en detrimento de flujos con RTT mayores. A raíz de las observaciones experimentales, Rhee y Xu [13] propusieron el mecanismo de control de congestión CUBIC, que constituye una mejora de TCP BIC e implementa una función para el crecimiento de la ventana de congestión en forma independiente del RTT para cada conexión en particular.

TCP CUBIC define una ventana de congestión *(w)* como una función cubica del tiempo transcurrido desde el último evento de congestión (Δ):

$$w = C\left(\Delta - \sqrt[3]{\frac{\beta * w_{max}}{C}}\right)^3 + w_{max}$$

Donde C es una constante predefinida, β es un coeficiente multiplicativo decreciente en la fase de recuperación rápida, y w_{max} es el tamaño de la ventana de congestión inmediatamente antes de que se detecte una pérdida. La función presenta un rápido crecimiento cuando w es pequeña respecto de w_{max}, y es muy conservativa cuando w se aproxima al valor de w_{max} tal como se ilustra en la Figura 3.5.

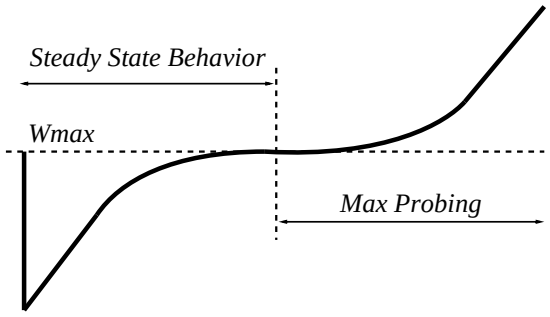

Figura 3.5: Descripción función cúbica

La Figura 3.6 muestra la dinámica de la ventana de congestión de TCP CUBIC. De la observación se puede deducir que, durante el paso inicial "1" se desconoce w_{max} y para descubrir su valor se emplea la sección derecha de la función cubica en la Figura 3.5 *(Max Probing)*, esta etapa finaliza al detectarse una pérdida. A continuación, en la fase 2, se emplea la sección izquierda de la función cúbica, que implica una tasa de crecimiento en la ventana de congestión menor al de la fase 1. La etapa 3 incluye ambas etapas de la función cúbica y finaliza cuando se produce una pérdida; en este caso particular se actualiza el valor umbral de w_{max}

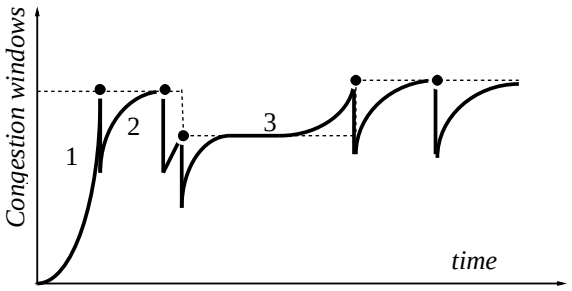

Figura 3.6: Dinámica de cwnd en Cubic

Por último, TCP CUBIC es el mecanismo de control de congestión más utilizado desde que fue incluido por defecto en las distintas versiones de núcleos Linux desde 2006.

3.3.0 Análisis Experimental

Es importante destacar que el objetivo de la experimentación no está dirigido a realizar valoraciones sobre medidas de desempeño entre distintos algoritmos de control de congestión. El propósito central del artículo es convalidar un novedoso e inédito método, para experimentar con algoritmos de control de congestión, basado en la construcción de topologías virtuales utilizando *User Mode Linux.*

El método propuesto es de fácil e inmediata implementación, y puede simular el comportamiento de enlaces WAN con sus características inherentes (retardos, *jitter*, tasa de pérdidas y restricciones de ancho de banda). Permite ahorrar tiempo en la obtención de resultados y es apto para obtener medidas de desempeño para cualquiera de los algoritmos de control de congestión incluidos como módulos en las distintas versiones de núcleos Linux.

A continuación en la sección 3.3.1 se presentará el laboratorio de experimentación a implementar bajo *User*

Mode Linux, se describirán los elementos constitutivos de la topología seleccionada y se expondrán los objetivos finales de la experimentación. En la sección 3.3.2 se abordaran aspectos de configuración del núcleo Linux en la computadora anfitriona, y del núcleo Linux en los dispositivos y computadoras virtuales.

3.3.1 Topología UML

La Figura 3.7 ilustra la topología a implementar en UML y aunque existe una extensa variedad de literatura sobre como ejecutar el escenario de red virtual, la referencia obligada es la de Jeff Dike, autor de la tecnología [14].

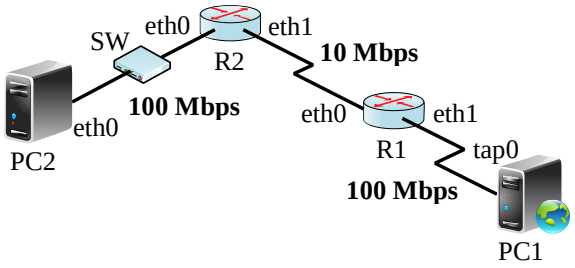

Figura 3.7: Topología virtual en UML

Los dispositivos de conmutación de paquetes R1 y R2 *(routers)*, la computadora PC2 y el dispositivo de conmutación de tramas, SW, constituyen componentes virtuales UML, soportados por el sistema operativo Linux en el host anfitrión PC1.

La vinculación entre los dispositivos virtuales y el host anfitrión (PC1) se define durante la creación de la interfaz de red *eth1* en R1, que al mismo tiempo genera una interfaz TAP en el host anfitrión PC1.

Una interfaz TAP es un enlace virtual punto a punto entre dispositivos Ethernet, que en vez de recibir tramas desde el medio físico, lo hace desde un programa en espacio de usuario; y en vez de transmitir tramas vía el medio físico, las envía al mismo programa en espacio de usuario. Es importante destacar que el núcleo Linux del host anfitrión debe tener habilitado el modulo *TUN/TAP Support* en la sección de configuración, soporte de dispositivos de red *(Network Device Support).*

Antes de llevar a cabo el trabajo experimental se deben realizar una serie de tareas que incluyen, la configuración de la capacidad de los enlaces tal como lo expresa la Figura 3.7, la utilización de herramientas de software para simular el RTT de una red de área amplia (WAN), la selección de un generador de trafico TCP para establecer una conexión entre los hosts extremos de la topología, y repetir la experiencia con los algoritmos de control de congestión TCP NewReno, TCP BIC y TCP CUBIC.

3.3.2 Consideraciones de configuración

Tal como se anticipó en la sección 3.1.0 de este artículo, *User Mode Linux* requiere de un núcleo Linux, configurado y compilado bajo herramientas GNU, para la arquitectura UML. Este núcleo, servirá de soporte para computadoras y dispositivos virtuales y se ejecutará como un programa en el plano de usuario del sistema operativo de la computadora anfitriona que posee su propio núcleo.

Para experimentar con algoritmos de control de congestión se necesitará disponer de un conjunto de variables en el núcleo Linux, y la vía para realizarlo es a través de *tcpprobe*, un módulo que posibilita el acceso a las variables de estado en una conexión TCP. El mecanismo opera insertando una sonda de prueba no invasiva *(kprobe)* en el buffer de recepción *tcp_recv*. Mediante este procedimiento se puede capturar en forma dinámica todas las variables necesarias de una conexión TCP, para cualquier algoritmo de control de congestión.

La tecnología de virtualización UML inhabilita el módulo *tcpprobe* y esto obliga a que el emisor TCP en la conexión entre los hosts extremos de la Figura 3.7 sea obligatoriamente PC1, que se soporta sobre el núcleo Linux del host anfitrión. En

síntesis, el núcleo Linux del host anfitrión (PC1) debe ser compilado con el módulo *tcp_probe* habilitado.

Para formar tráfico *(traffic shaping)* en el ambiente virtual, y limitar el ancho de banda de acuerdo a los valores descritos en la Figura 3.7, es necesario habilitar en ambos núcleos Linux, en la sección de configuración "*QoS and/or fair queueing"* el módulo TBF *(Token Bucket Filter).* TBF es una sencilla disciplina de filas, que no distingue entre clases de tráfico *(clasless)* y que consiste de un *buffer* al que ingresan piezas virtuales de información o *tokens*, a una tasa constante o *token rate* (Figura 3.8). Tal como se ilustra en la Figura 3.8, la salida de un *token* desde el *bucket* habilita la partida de un segmento TCP. Fijando la tasa de *tokens* se controla el ancho de banda de un enlace [15].

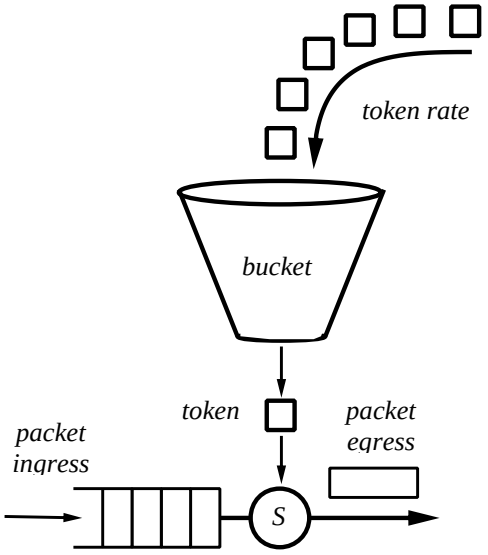

Figura 3.8: Modelo de disciplina TBF

Para emular el comportamiento de un enlace WAN en el ambiente virtual, es necesario habilitar en el núcleo Linux (soporte de los dispositivos virtuales), la opción *Network Emulator* o *NetEm* [16]. Para configurar *NetEm* se necesita del comando, tc *(traffic control),* que es parte del paquete de software *iproute2* [17].

Aunque NetEm posibilita una gran variedad de opciones respecto de su funcionalidad, en el marco de este artículo sólo se utilizará para generar retardos y deviaciones del retardo (*jitter),* para replicar en el ambiente virtualizado las condiciones de un enlace WAN cuyas características topológicas son similares a las de la Figura 3.7 y sus características temporales son: RTT promedio 50ms con ± 6 ms de *jitter*.

El objetivo perseguido es replicar en el ambiente virtual las condiciones de una red de área amplia (WAN). No se inducirán pérdidas mediante NetTem, pues se pretende que los algoritmos de control de congestión funcionen en base a la limitación originada por la restricción del ancho de banda del enlace WAN (10 Mbps), considerando que el medio físico utilizado es fibra óptica y el nivel de pérdidas de segmentos es despreciable.

Un aspecto a tener en cuenta para lograr granularidad en los retardos introducidos mediante NetEm es relativo a la opción de configuración CONFIG_HZ en el núcleo Linux de los dispositivos virtuales. Esta opción determina la frecuencia con que el núcleo Linux atiende a los eventos generados por procesos en el plano de usuario. El valor por defecto de CONFIG_HZ en núcleos compilados bajo GNU para la arquitectura UML es 100 Hz, y sólo se puede modificar mediante la aplicación del código referenciado en [18]. Si se busca granularidad en el orden de 1 ms, el valor de CONFIG_HZ debe ser modificado a 1000 Hz.

Una cuestión relativa a los dispositivos de conmutación de tramas *(switchs)* utilizados para construir topologías virtuales. UML puede emplear una de dos herramientas de software para implementarlos, *uml_switch* [19] o *vde_switch* [20]. En el trabajo experimental se utilizó *vde_switch* al comprobar que no presentaba pérdida de tramas en condiciones de tráfico intenso. Por el contrario, la utilidad *uml_switch* ante

idénticas circunstancias presentó perdidas de tramas (capa 2) y este factor afectaría el comportamiento de los algoritmos de control de congestión experimentados.

Para generar tráfico entre ambos extremos de la topología de la Figura 3.7 se utilizó Iperf [21], con PC1 como emisor y PC2 como el receptor de segmentos TCP. A continuación en la sección 3.4, se presentaran las características de hardware y del sistema operativo de la computadora anfitriona y las características de software de las computadoras y dispositivos virtuales; y se presentaran los resultados de la simulación.

3.4.0 Resultados Experimentales

Las pruebas experimentales para la simulación de los algoritmos de control de congestión se realizaron en una computadora con las siguientes características: procesador AMD (FX)-8128 de 8 núcleos con 8 GB de memoria RAM, ejecutando un sistema operativo GNU/Linux Ubuntu v10.04 basado en la versión del núcleo 2.6.32.28. Las computadoras y dispositivos virtuales UML, se ejecutaron sobre un sistema operativo GNU/Linux Slackware v12 soportado por la versión de núcleo Linux 3.2.59, y 1GB de memoria para cada dispositivo virtual.

Las simulaciones se realizaron, en un todo de acuerdo con la topología ilustrada en la Figura 3.7, considerando un RTT de 50 ms, un *jitter* de $\pm$ 6 ms, sin introducir perdidas ni duplicaciones de segmentos. Los segmentos generados por Iperf son de máxima longitud (MSS) para un MTU de Ethernet de 1500 bytes.

La Figura 3.9 muestra los resultados de la simulación para el algoritmo de control de congestión TCP New Reno. A diferencia de lo analizado en la sección 3.2.3, el valor inicial de la variable umbral de arranque lento es elevado *(ssthresh)* y la ventana de congestión crece exponencialmente en la fase *slow start* hasta que se detecta la pérdida de un segmento por la recepción de tres ACKs duplicados, a continuación se actualiza el valor de variable *ssthresh*, TCP NewReno reingresa a la fase de arranque lento *(slow start)* y aproximadamente a los 10s el crecimiento de la ventana de congestión *(cwnd)* alcanza el valor de *ssthresh* y conmuta a la fase de prevención de congestión (CA). En esta fase de operación *cwnd* se incrementa de acuerdo a lo expresado en la sección 3.2.3 y en ausencia de pérdidas permanece en CA.

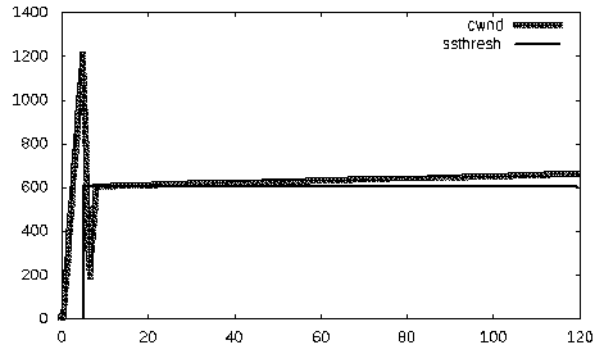

Figura 3.9: Simulación NewReno

La Figura 3.10 ilustra el resultado de la simulación para TCP BIC. Tal como se observa en la gráfica, el algoritmo tiene una fase de descubrimiento rápido al inicio, para detectar la capacidad de la red para la conexión. La variable *ssthresh* se corresponde con el parámetro w_{min} estudiado y descrito en la sección 3.2.4, y su dinámica es de rápida convergencia, mostrando su naturaleza de búsqueda binaria.

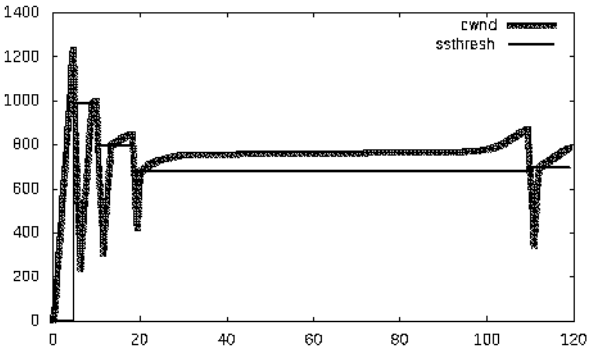

Figura 3.10: Simulación BIC

El comportamiento dinámico del algoritmo de control de congestión TCP Cubic se muestra en la Figura 3.11, tal como se describió en la sección 3.2.5, la simulación muestra una etapa inicial de descubrimiento del límite del ancho de banda de la red, al que se aproxima utilizando la parte derecha de la función cúbica que sustenta el funcionamiento del algoritmo. Una vez que se produce una condición de pérdida, el algoritmo disminuye su ventana de congestión *(cwnd)* y se aproxima a *ssthresh* utilizando la parte izquierda de la función cúbica, tal como fue tratado en la sección 3.2.5.

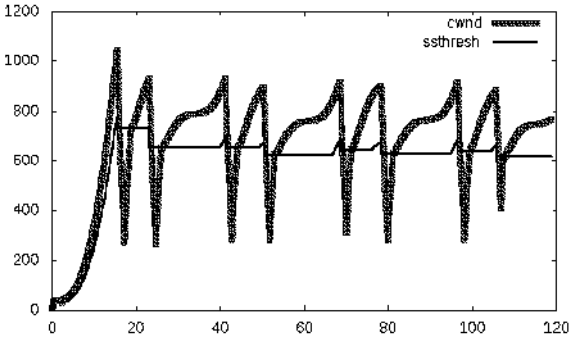

Figura 3.11: Simulación Cubic

3.5.0 Conclusiones

A pesar de la extensa variedad de artículos relativos al tópico control de congestión, no existen al respecto antecedentes de implementaciones ejecutadas íntegramente en ambientes virtualizados. Ello significa que el método de experimentación basado en *User Mode Linux*, para la simulación de algoritmos de control de congestión es inédito, y de gran utilidad para el estudio, análisis, y comparación de cualquiera de los algoritmos implementados en las distintas versiones de núcleos Linux.

El método propuesto es de fácil e inmediata implementación, y puede simular el comportamiento de enlaces WAN con sus características inherentes (retardos, *jitter*, tasa de pérdidas y restricciones de ancho de banda).

Permite ahorrar tiempo en la obtención de resultados y es apto para obtener medidas de desempeño.

El esquema propuesto no requiere de la utilización de complejos simuladores de eventos discretos, ni de la variada cantidad de dispositivos necesitados para construir una topología real; solo se necesita disponer de una computadora de medianos recursos para obtener idénticos resultados.

Finalmente, y para concluir, los resultados obtenidos vía simulación bajo UML para los mecanismos de control de congestión TCP NewReno, TCP BIC y TCP Cubic, se corresponden exactamente con los principios teóricos que rigen la dinámica de los algoritmos tratados en la sección 3.2 del artículo.

3.6.0 Referencias

[1] The UML Kernel Home Page; http://user-mode-linux.sourceforge.net

[2] O. S. Initiative, http://opensource.org/docs/osd

[3] NS, http://www.isi.edu/nsnam/ns/

[4] OPNET, http://www.riverbed.com/

[5] Jacobson, V., “Congestion avoidance and control”. In Symposium Proceedings on Communications Architectures and Protocols (Stanford, California, United States, August 16 - 18, 1988). V. Cerf, Ed. SIGCOMM ‘88. ACM, New York, NY, 314-329.

[6] Jacobson, V., "Modified TCP Congestion Avoidance Algorithm", end2end-interest mailing list, April 30, 1990.

[7] S. Floyd and T. Henderson, “RFC2582 the New Reno modification to TCP’s fast recovery algorithm”, RFC, 1999

[8] S. Floyd, T. Henderson, and A. Gurtov, “RFC3782 the NewReno modification to TCP’s fast recovery algorithm”, RFC, 2004.

[9] Alexander Afanasyev, Neil Tilley, Peter Reiher, and Leonard Kleinrock. “Host-to-Host Congestion Control for TCP”. IEEE Communications Surveys & Tutorials, 12(3):304–342, 2010.

[10] Sangtae Ha, Yusung Kim, Long Le, Injong Rhee, and Lisong Xu, “A step toward realistic evaluation of high-speed TCP protocols”, 2006.

[11] L. Xu, K. Harfoush, and I. Rhee, “Binary Increase Congestion Control (BIC) for fast long-distance networks, Proceedings of IEEE INFOCOMM 2004, vol. 4, March 2004, pp. 2514-2524.

[12] Floyd, S., "Limited Slow-Start for TCP with Large Congestion Windows", RFC 3742, March 2004

[13] L. Xu, and I. Rhee, “CUBIC: A New TCP-Friendly High-Speed TCP Variant”.

[14] Dike Jeff, “User Mode Linux”, Publisher: Prentice Hall, April 12, 2006

[15] QoS (Quality of Service) concepts, http://lartc.org/howto/lartc.cookbook.fullnat.intro.html

[16] Stephen Hemminger, “Network Emulation with NetEm”, In Linux Conf AU, 2005

[17] IPROUTE2, Linux Foundation Page, http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2

[18] Config clock in User Mode Linux, http://ehc.ac/p/mrvopensource/linux-ppc-2.6/ci/7281ff952c7b3cbefb14847460e5fe73a2d240e4

[19] Switch daemon for UML - uml_switch, http://manpages.ubuntu.com/manpages/lucid/man1/uml_switch.1.html

[20] VDE - Virtual Distributed Ethernet, http://vde.sourceforge.net/

[21] http://sourceforge.net/projects/iperf/, Iperf - The TCP/UDP, Bandwidth Measurement Tool, 2013.

CAPITULO IV
Topología: Configuración y Ejecución de las Simulaciones

4.0 Introducción

En este capítulo se detallaran exhaustivamente los pasos llevados a cabo para realizar las experiencias de laboratorio bajo UML con los diferentes algoritmos de Control de Congestión abordados en el artículo principal, objeto del trabajo de tesis. La topología de red para la experimentación se ilustra en la Figura 4.1.

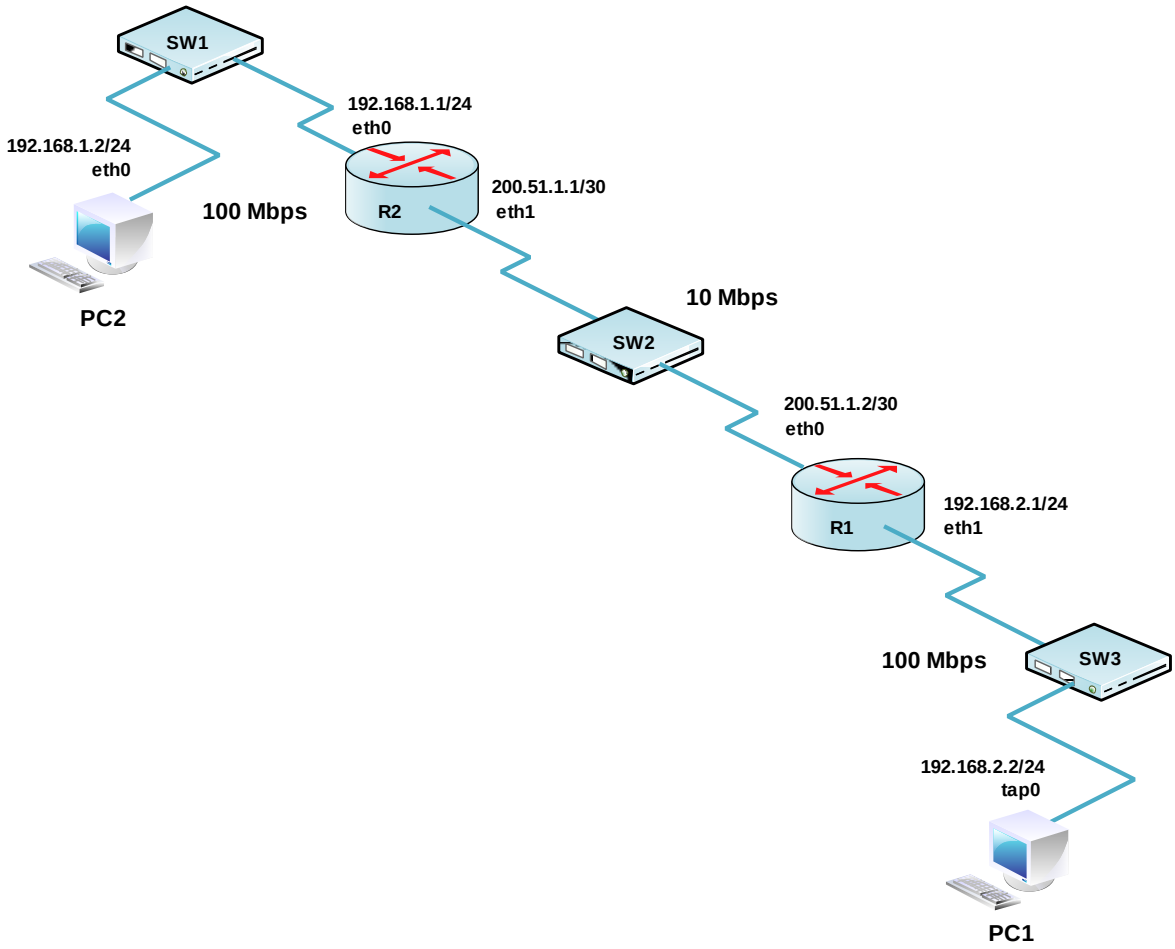

Figura 4.1: Topología de Red para la Experimentación

A continuación, en la sección 4.1 se explicará el script basado en el intérprete de comandos *Bourne Shell* utilizado para facilitar la ejecución de todos los componentes en la topología virtual. La sección 4.2, tratará sobre los aspectos de configuración de la topología (Figura 4.1), posteriormente en la sección 4.3 se muestran los comandos para generar tráfico TCP entre dos equipos. Finalmente, la sección 4.4 detalla cómo generar gráficos de las experiencias realizadas por medio de la herramienta *gnuplot*.

4.1 Script para la Ejecución del Laboratorio Virtual

Para ejecutar la topología descrita por la Figura 4.1 bajo UML, se construyó el script `inicio.sh`, y se lo almacenó en el mismo directorio donde se encuentra el núcleo compilado para la arquitectura UML (ver Capitulo II):

```
#-------------------------------------------------------------------------------------------------
#                                Script inicio.sh
#-------------------------------------------------------------------------------------------------
#!/bin/sh
#*****************************************************************
# Los siguientes comandos detienen los procesos activos, y elimina
# archivos temporales, producto de una ejecución anterior del script
#*****************************************************************
killall linux
killall xterm
killall /usr/lib/uml-port-helper
killall vde_switch
killall /usr/sbin/console-kit-daemon
rm /tmp/sw1.ctl -R
rm /tmp/sw2.ctl -R
rm /tmp/sw3.ctl -R
#*****************************************************************
# VDE provee el comando vde_switch para ejecutar switches virtual
# (dispositivos de conmutación de tramas)
#*****************************************************************
vde_switch -s /tmp/sw1.ctl &
vde_switch -s /tmp/sw2.ctl -tap tap0 -m 666 &
vde_switch -s /tmp/sw3.ctl &
#*****************************************************************
#Ejecución de una máquina y dispositivos virtuales (routers)
#contemplando las interfaces de red definidas en la topología de
#experimentación
#*****************************************************************
./linux umid=PC2 mem=900M ubd0=PC2.cow,root_fs eth0=vde,/tmp/sw1.ctl \
con0=xterm &
./linux umid=R1 mem=900M ubd0=R1.cow,root_fs eth0=vde,/tmp/sw1.ctl \
eth1=vde,/tmp/sw3.ctl con0=xterm &
./linux umid=R2 mem=900M ubd0=R2.cow,root_fs eth0=vde,/tmp/sw3.ctl \
eth1=vde,/tmp/sw2.ctl con0=xterm
#*****************************************************************
#                    Fin Script inicio.sh
#*****************************************************************
```

Antes de explicar cada sección del script `inicio.sh`, es necesario aclarar al lector que todas las instancias de configuración de interfaces de red, de formación de tráfico y de modificación de parámetros de interés del protocolo TCP (en las instancias virtuales bajo UML) se realizan a través de scripts específicos que se ejecutarán en tiempo de inicio (boot) de máquinas y dispositivos de red virtuales. Por el motivo señalado, una vez ejecutado el script `inicio.sh`, no es necesario la detención de cada elemento en la topología mediante el comando `halt`. Por una cuestión de practicidad se utilizó el comando `killall` para terminar con todas las instancias de software activas en memoria, procesos, derivadas de una ejecución previa del script `inicio.sh`.

En el ámbito del trabajo de tesis se utilizó como dispositivos de conmutación de tramas (capa 2 del modelo de referencia OSI de la ISO) el software VDE *(Virtual Distributed Ethernet)*. Tal como se observa en la sección relativa a VDE en el script `inicio.sh`, la ejecución del comando "`vde_switch -s`" creará un *switch* virtual y el argumento `(s)` sirve para especificar el directorio donde se almacenarán los archivos para el socket generado. En el caso particular de la topología de la Figura 4.1, se necesitan tres dispositivos de conmutación de tramas Ethernet, uno por cada red IP. Dos de ellos, `sw1.ctl` y `sw3.ctl` sirven al propósito de vincular PC2 con R2 y R2 con R1, mientras que el *switch* en `sw2.ctl` necesita de un argumento adicional `(-tap)` para crear una interfaz `tap0` desde el lado del host anfitrión tal como se explicó y aclaró en la sección 3.1 del artículo principal que compone este trabajo de tesis.

Los dispositivos de conmutación de tramas deben ser ejecutados antes de iniciar las instancias UML correspondientes a PC2, R2 y R1, porque tal como se desprende de la sección "ejecución de máquina y dispositivos virtuales *(routers)*" en `inicio.sh`, el comando para ejecutar una máquina virtual o un *router* virtual requiere de la definición de las interfaces de red y su asociación a un determinado *switch*.

A continuación, se explicará en detalle el comando utilizado para crear las instancias UML mostradas en la Figura 4.1.

En primer lugar el comando `./linux` invoca en el directorio actual a la versión del núcleo Linux compilado para la arquitectura UML. El argumento `umid` es una directiva para asignarle el nombre a la máquina virtual (PC2) o los dispositivos virtuales R1 y R2, el argumento `mem` permite asignar el tamaño de la memoria física, en este caso 900 MB. Durante el arranque, el núcleo linux (UML) debe montar su propio *filesystem* y para que ello ocurra es necesario indicarle el *Block Device Driver* o dispositivo `ubd` (unidad de disco virtual). Para ahorrar espacio en disco, UML utiliza la técnica COW *(Copy On Write)* en que se dispone de un único *filesystem* de solo lectura para cualquier número de instancias UML ejecutadas. Las modificaciones necesarias que una máquina virtual o dispositivo virtual debería realizar sobre el *filesystem* original, las realizan sobre su propio archivo COW, creado en tiempo de ejecución. Ello implica un considerable ahorro de espacio en disco, puesto que cada archivo COW es construido utilizando la tecnología *sparse*. Finalmente, se definen las interfaces de red (asociadas a los *switchs* que correspondan) para cada máquina o dispositivo virtual de acuerdo a la topología mostrada en la Figura 4.1.

4.2 Configuración de la Topología

Mediante la ejecución del script `inicio.sh`, todos los componentes de la topología en la Figura 4.1 estarán activos y el usuario tendrá acceso a la máquina virtual PC2 y a los dispositivos virtuales R1 y R2, vía terminales de comandos (xterm). El paso siguiente, es configurar los parámetros de red, con las condiciones definidas para realizar las pruebas de laboratorio de acuerdo a lo expresado en la sección 3.1 del artículo principal. Ello incluirá la formación de tráfico para limitar la velocidad de los enlaces de acuerdo a los valores mostrados en la topología de la Figura 4.1, establecer los valores de latencia y *jitter* para establecer el RTT (tiempo de ida y vuelta) entre los hosts extremos, y modificar algunas variables del núcleo Linux relacionadas con el protocolo TCP. A los efectos de simplificar la tarea, todas las acciones de configuración descritas se realizaron mediante la creación de los siguientes scripts ejecutados en tiempo de inicio *(booting)*:

```
#-------------------------------------------------------------------
#                          Script config_PC1.sh
#-------------------------------------------------------------------
#!/bin/sh
#*******************************************************************
# Configuración de la Interfaz de Red en PC1
#*******************************************************************
ip link set tap0 up
ip addr add 192.168.2.2/24 brd 192.168.2.255 dev tap0
ip route add default via 192.168.2.1 dev tap0
#*******************************************************************
# Control de Tráfico
#*******************************************************************
tc qdisc del dev tap0 root
tc qdisc add dev tap0 root handle 1:0 tbf rate 100Mbit burst 100Kb /
limit 100000
#*******************************************************************
# Configuración Parámetros TCP
#*******************************************************************
sysctl -w net.ipv4.tcp_no_metrics_save=1
echo 4096 87380 10000000 > /proc/sys/net/ipv4/tcp_rmem
echo 4096 16384 10000000 > /proc/sys/net/ipv4/tcp_wmem
#-------------------------------------------------------------------
```

```
#-------------------------------------------------------------------
#                          Script config_R1.sh
#-------------------------------------------------------------------
#!/bin/sh
#*******************************************************************
# Configuración de las Interfaces de Red en R1
#*******************************************************************
ip link set eth0 up
ip addr add 200.51.1.2/30 brd 200.51.1.3 dev eth0
ip route add default via 200.51.1.1 dev eth0
ip link set eth1 up
ip addr add 192.168.2.1/24 brd 192.168.2.255 dev eth1
ip route add default via 200.51.1.2 dev eth0
#*******************************************************************
# Control de Tráfico
#*******************************************************************
tc qdisc del dev eth0 root
tc qdisc del dev eth1 root
tc qdisc add dev eth0 root handle 1:0 tbf rate 10Mbit burst 10Kb   /
limit 100000
tc qdisc add dev eth1 root handle 2:0 tbf rate 100Mbit burst 100Kb /
limit 100000
tc qdisc add dev eth1 parent 2:1 handle 20:0 netem delay X1ms Y1ms  /
loss Z1%
#-------------------------------------------------------------------
```

```
#-------------------------------------------------------------------
#                        Script config_R2.sh
#-------------------------------------------------------------------
#!/bin/sh
#*******************************************************************
# Configuración de las Interfaces de Red en R2
#*******************************************************************
ip link set eth0 up
ip addr add 192.168.1.1/24 brd 192.168.1.255 dev eth0
ip route add default via 200.51.1.1 dev eth0
ip link set eth1 up
ip addr add 200.51.1.1/30 brd 200.51.1.3 dev eth1
ip route add default via 200.51.1.2 dev eth1
#*******************************************************************
# Control de Tráfico
#*******************************************************************
tc qdisc del dev eth0 root
tc qdisc del dev eth1 root
tc qdisc add dev eth1 root handle 1:0 tbf rate 10Mbit burst 10Kb   /
limit 100000
tc qdisc add dev eth0 root handle 2:0 tbf rate 100Mbit burst 100Kb /
limit 100000
tc qdisc add dev eth0 parent 2:1 handle 20:0 netem delay X1ms Y1ms  /
loss Z1%
#-------------------------------------------------------------------
```

```
#-------------------------------------------------------------------
#                        Script config_PC2.sh
#-------------------------------------------------------------------
#!/bin/sh
#*******************************************************************
# Configuración de la Interfaz de Red en PC2
#*******************************************************************
ip link set eth0 up
ip addr add 192.168.1.2/24 brd 192.168.1.255 dev eth0
ip route add default via 192.168.1.1 dev eth0
#*******************************************************************
# Control de Tráfico
#*******************************************************************
tc qdisc del dev eth0 root
tc qdisc add dev eth0 root handle 2:0 tbf rate 100Mbit burst 100Kb /
limit 100000
#*******************************************************************
# Configuración Parámetros TCP
#*******************************************************************
echo 4096 87380 10000000 > /proc/sys/net/ipv4/tcp_rmem
echo 4096 16384 10000000 > /proc/sys/net/ipv4/tcp_wmem
#-------------------------------------------------------------------
```

Los scripts `config_R1.sh`, `config_R2.sh` y `config_PC2`, se crearon en el directorio `/etc/rc.local`. Considerando que se utilizó un archivo de sistema *(filesystem)* Slackware, cualquier comando o script escrito o almacenado en ese *path* se ejecutara en tiempo de inicio de la máquina o dispositivo virtual. El script `config_PC1.sh` se ejecuta en el host anfitrión (PC real).

Cada uno de los scripts mostrados en esta sección, consta de un primer grupo de comandos destinados a configurar los parámetros de red con valores acordes a lo que muestra la topología de la Figura 4.1. El comando `ip` es uno de los dos comandos que proporciona el paquete de software iproute2, y de acuerdo con el argumento que le sigue, permite configurar aspectos relativos a la capa de enlace (`ip link`), o a la capa de red (`ip addr`, `ip route`). Sintéticamente:

1. `ip link set {dev DEVICE} [{up | down}]`, activa o desactiva una interfaz de red.
2. `ip addr {add | del} dir_IP/prefijo brd {dir_broadcast} dev {STRING}`, asigna o elimina una dirección IP a la interfaz asociada con el dispositivo *string*.
3. `ip route {add} default via dir_GW dev {STRING}`, crea una ruta por defecto via la pasarela de salida o gateway (GW)

El segundo grupo de comandos en los scripts es relativo al control de ancho de banda en cada uno de los enlaces como se ilustra en la Figura 4.1. En el ámbito virtual, la velocidad de transmisión en bits por segundo en las interfaces de red, no se corresponde con el de las tecnologías reales basadas en Ethernet 10Mbps/100Mbps/1Gbps, pues UML virtualiza la capa física y distintos aspectos de la capa de enlace de datos. Por el motivo antes señalado, la velocidad de transmisión de una interfaz virtual tiene que ver con la frecuencia con que el sistema operativo atiende los procesos de lectura y escritura de tramas. En el ámbito de este trabajo, y por las características del hardware utilizado en la computadora anfitriona (ver sección 4.0 del artículo principal), es necesario limitar las tasas a los valores señalados en la topología de la Figura 4.1.

El comando `tc` (provisto por el paquete de software iproute2) permite a los administradores de una organización construir políticas de QoS en una red utilizando para ello el soporte de Linux. En el ámbito de las redes, QoS se refiere al control de métricas (latencia, ancho de banda, etc) que afectan directamente a los servicios que se ejecutan en un ambiente de red. Para comprender aspectos relativos al control de tráfico es conveniente conocer cómo el núcleo Linux procesa los paquetes TCP/IP (Figura 4.2). Los datagramas IP (que ingresan encapsulados en tramas Ethernet vía la tarjeta de interfaz a la red) son examinados para determinar si deben ser reenviados hacia la red *(forwarding)* o si deben ser transferidos hacia las capas superiores en la pila de protocolos TCP/IP. A su vez, las capas superiores del modelo TCP/IP pueden generar datos hacia las capas inferiores para su transmisión. Una vez que se realiza la acción de reenvío *(forwarding)* los paquetes son puestos en una fila *(queue)* y es aquí donde el control de tráfico puede aplicarse, decidiendo entre otras cosas qué paquetes se transfieren a la fila y qué paquetes son descartados (por ejemplo si la fila ha alcanzado algún nivel de ocupación o si el tráfico excede alguna tasa límite), también puede decidir el orden en que los paquetes son enviados desde la fila en base a prioridades asignadas a distintos flujos de paquetes o también puede demorar la transmisión de paquetes para cumplimentar con un limitación respecto de la tasa de envío (control de ancho de banda).

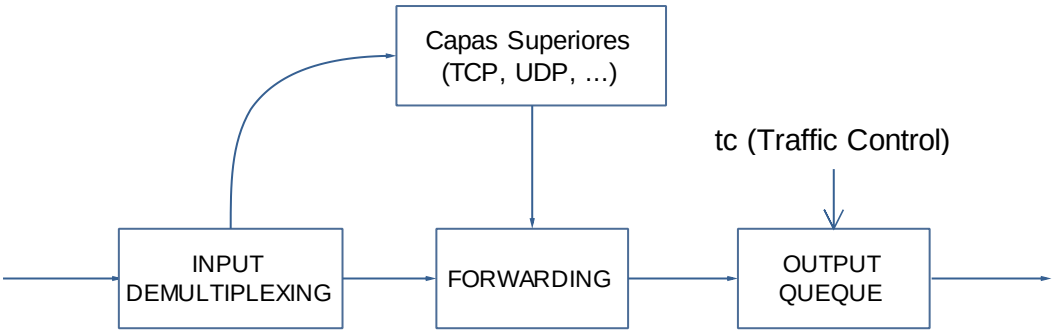

Figura 4.2: Ámbito de operación de `tc` *sobre una interfaz de red*

El control de tráfico en el núcleo Linux consiste de los siguientes componentes conceptuales:

- *queueing disciplines* (disciplina de la fila)
- *classes* (clases dentro de una disciplina de fila)
- *filters* (filtros)
- *policing* (políticas)

Cada dispositivo de red (interfaz) tiene una disciplina de fila asociada que define como los paquetes serán tratados. Por ejemplo una disciplina elemental puede consistir de una simple fila donde los paquetes se almacenan por el orden de llegada y luego se extraen considerando ese orden (FIFO). Disciplinas más elaboradas pueden usar filtros para distinguir entre distintas clases de paquetes y luego procesarlos según el nivel de prioridad asignada a cada una de las clases. La Figura 4.3 muestra un ejemplo de tal disciplina de fila, en ella se pueden distinguir un filtro cuyo objetivo es separar el tráfico en una clase definida para posteriormente procesarla en forma específica y darle prioridad sobre el tráfico complementario.

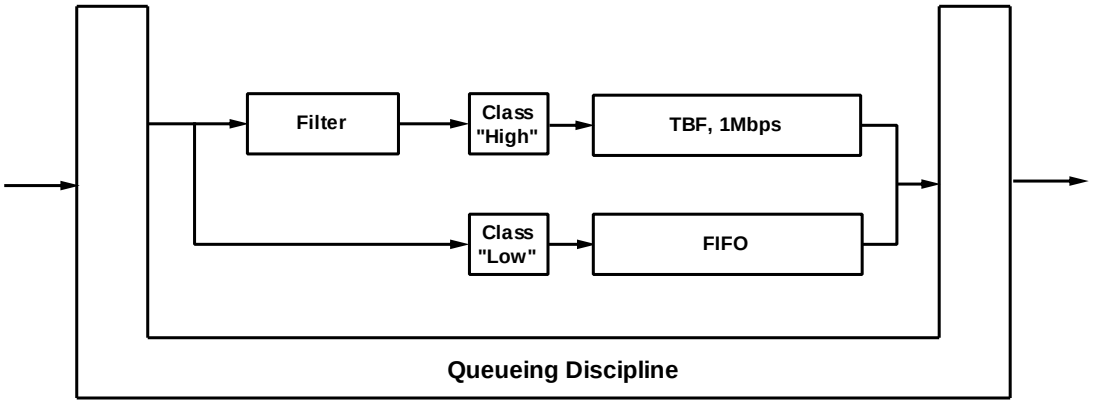

Figura 4.3: Ejemplo de control de tráfico con dos disciplinas de filas

La Figura 4.4 ilustra el caso en que varios filtros pueden mapear paquetes a una misma clase.

Los conceptos de clase y disciplina de fila están íntimamente relacionados, la presencia de clases y su semántica son propiedades fundamentales en una disciplina de fila. Una disciplina de fila expresa la forma en que los paquetes serán seleccionados para su tratamiento. Los filtros pueden ser combinados arbitrariamente con disciplinas

de filas y clases, también como lo ilustra Figura 4.4, múltiples filtros pueden mapearse a una misma clase.

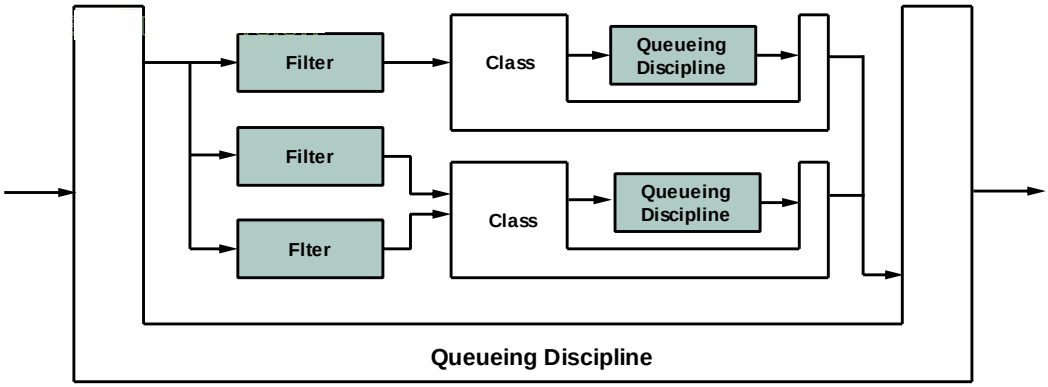

Figura 4.4: Ejemplo de control de tráfico con dos disciplinas de filas

Finalmente no es obligatorio que se definan clases y en ese caso todos los paquetes que ingresan a la fila serán tratados por la disciplina configurada, por este motivo se clasifican en disciplinas que soportan clases y disciplinas sin clases.

En el ámbito de este trabajo de tesis, se utilizó una disciplina sin clases *Token Bucket Filter* (TBF) al sólo efecto de realizar el control de ancho de banda que los enlaces de la topología requieren (Figura 4.1).

TBF es una disciplina de filas muy sencilla y consiste de un buffer al que dinámicamente ingresan piezas virtuales de información o *tokens* a una tasa constante o *token rate*. Cada *token* en el *buffer* tomará un paquete de datos de la fila y se eliminará del *bucket* y de acuerdo a ello las situaciones que se pueden generar son las siguientes:

- Los paquetes ingresan a una tasa igual a la tasa de generación de los *tokens*. En este caso cada uno de los paquetes que ingresa tiene su *token* y pasa a la fila sin retardo.
- Los paquetes ingresan a una tasa menor respecto a la tasa de generación de *tokens*. En este caso los *token* en exceso se acumularan en el buffer y servirán para atender eventuales ráfagas en el flujo de ingreso de los paquetes.
- Los paquetes ingresan a una tasa mayor respecto a la tasa de generación de *tokens*, entonces los paquetes que ingresan deberán esperar cada uno por un *token* y potencialmente se producirán pérdidas de paquetes.

Los parámetros de configuración de la disciplina TBF son los siguientes:

`limit` / `Latency`	`limit` es el número de bytes que pueden ser encolados a la espera de que haya *tokens* disponibles. También puede especificar esto estableciendo el parámetro latency, que indica el período máximo de tiempo que puede pasar un paquete en el TBF. Este último cálculo tiene en cuenta el tamaño del *bucket*, la tasa y posiblemente el *peak rate* (la tasa de picos, si es que se ha configurado).
`burst` / `buffer` /	Tamaño del *bucket*, en bytes. Esta es la máxima cantidad de bytes para los que puede haber *tokens* disponibles instantáneamente. En general, grandes tasas precisan grandes búferes. Para 10 Mbps sobre Intel, se necesitará al menos un búfer de 10kbyte si se desea alcanzar la tasa que

`maxburst`	ha configurado. Si el búfer es demasiado pequeño, se descartarán paquetes debido a que llegan más tokens por tick del temporizador de los que caben en el *bucket*.
`Mpu`	*Minimum packet size*. Parámetro necesario como consecuencia de que un paquete de tamaño 0 bytes se traduce en un *frame* de 64 bytes en Ethernet y consume tiempo de transmisión.
`Rate`	Tasa elegida para el tráfico que parte de la qdisc (disciplina de fila).
`peakrate`	Cuando existen *tokens* en el *bucket* y los paquetes arriban, ellos son enviados inmediatamente por la disciplina TBF. Cuando este es el caso la rapidez de agotamiento de *tokens* en el *bucket* es una función de la tasa de paquetes en la interfaz considerada. Una forma de establecer un control adicional sobre la rapidez con que tokens son viculados a paquetes es a través de este parámetro si es que se desea disminuirla. Sin embargo, debido a la resolución por defecto de 10 ms del reloj en Unix la tasa máxima estará limitada a 1Mbps (considerando paquetes de 10.000 bits en promedio).
`Mtu / minburst`	Una *peakrate* de 1 Mbps no es un valor muy elevado. Es posible tener valores mayores de *peakrate* creando un segundo *bucket* que por defecto tiene un solo *token*.

Hasta aquí se ha explicado como formar tráfico utilizando la disciplina de fila TBF, para fijar la tasa de transmisión de las interfaces a los valores mostrados en la topología de la Figura 4.1. Resta introducir los retardos, desviaciones del retardo (*jitter*) y porcentaje de pérdidas mediante NETEM en los dispositivos R1 y R2; para emular el comportamiento de un enlace de red de área amplia (WAN). Tal como se puede leer en los scripts `config_R1.sh` y `config_R2.sh`, se define una disciplina TBF y a continuación se asocia NETEM para introducir el retardo (`X1ms`) y su correspondiente desviación (Y1ms) junto al nivel de pérdidas porcentuales que se desea emular (Z_1%):

```
tc qdisc add dev eth0 root handle 2:0 tbf rate 100Mbit burst 100Kb /
limit 100000
tc qdisc add dev eth0 parent 2:1 handle 20:0 netem delay X1ms Y1ms  /
loss Z1%
```

Nótese que a los efectos de introducir un RTT (tiempo de ida y vuelta) de `Wms` se decidió repartirlo entre los dispositivos virtuales (*routers*) R1 y R2 para que los buffers de transmisión no se vean sobrecargados de segmentos TCP, concretamente `2*X1ms=Wms` corresponde al `RTT` y `2*Y1ms` al *jitter*.

En PC1 y en PC2 se introducen los comandos para que TCP auto sintonice los valores correspondientes a los buffers de recepción y de transmisión para la conexión:

```
echo 4096 87380 10000000 > /proc/sys/net/ipv4/tcp_rmem
echo 4096 16384 10000000 > /proc/sys/net/ipv4/tcp_wmem
```

Finalmente y de acuerdo a lo expresado en el artículo principal (sección 3.2), PC1 será la fuente generadora de tráfico TCP y PC2 reconocerá mediante ACK´s los segmentos TCP recibidos desde PC1. Ello implica que la obtención de las variables que describen el comportamiento de los algoritmos de Control de Congestión se realizaran en PC1 y por ese motivo es necesario ejecutar el comando `sysctl -w net.ipv4.tcp_no_metrics_save=1`, que instruye al núcleo a no almacenar valores

previos de conexiones TCP, tal como *cwnd* y *ssthresh*, para no alterar los de la conexión actual.

El último paso que resta realizar, es la selección del algoritmo de control de congestión que se desea experimentar. Para ello, se debe ejecutar el siguiente comando en el host anfitrión PC1:

```
sysctl -w net.ipv4.tcp_congestion_control= {BIC | Cubic | Reno}
```

En este punto están dadas todas las condiciones necesarias de configuración para experimentar con los algoritmos de control de congestión.

4.3 Generación de Tráfico TCP

La generación de tráfico TCP entre los host extremos de la topología dada en la Figura 4.1 se realiza mediante el software iperf, y para ello es necesario definir el cliente (generador de segmentos TCP) y la parte servidora de la aplicación. A tal fin, en PC2 se ejecuta el comando que inicia el proceso servidor iperf en el puerto 9112 de la computadora virtual:

```
iperf -s -p 9112
```

En la computadora PC1 (host real) se ejecuta el script captura.sh:

```
#*************************************************************
#                      Script captura.sh
#*************************************************************
#!/bin/sh
#*************************************************************
# Ejecución del módulo tcp_probe
#*************************************************************
modprobe tcp_probe port=9112 full=1
cat /proc/net/tcpprobe > ./captura.out &
#*************************************************************
# Ejecución del cliente iperf
#*************************************************************
iperf -c 192.168.1.2 -p 9112 -i 10 -t 119 –m
#-------------------------------------------------------------
```

El comando `modprobe tcp_probe port=9112 full=1`, habilita como módulo a la herramienta tcp_probe, indicándole que debe monitorear todo lo que tenga como destino el puerto 9112. Una vez ejecutado, se inicia un proceso tcpprobe, que redirige su salida al archivo `captura.out`. Finalmente, el comando "`iperf -c 192.168.1.2 -p 9112 -i 10 -t 119 –m`" inicia el proceso iperf cliente, generando tráfico hacia el socket destino `192.168.1.2:9112` durante 119 segundos e informando vía `stdin` cada 10 segundos. La opción `–m` es para que en las iperf muestre el máximo tamaño de segmento (MSS).

4.4 Generación de Gráficos

Una vez finalizada la ejecución del script `captura.sh`, el archivo `captura.out`, contendrá los valores obtenidos por `tcp_probe`, y el formato será el que muestra la Figura 4.5.

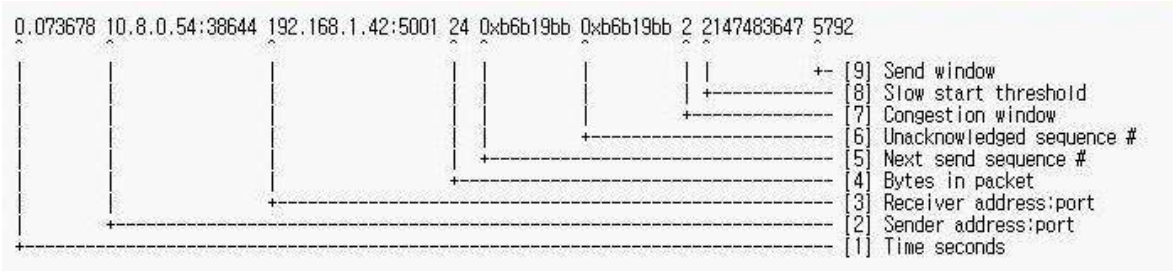

Figura 4.5: Formato de salida de tcp_probe

En base a estos valores, y por medio del software `gnuplot`, se pueden obtener gráficas que muestran la dinámica de los diferentes algoritmos experimentados. Para ello, se debe ejecutar el script `plot.sh`:

```
#*****************************************************************
#                      Script plot.sh
#*****************************************************************
#!/bin/sh
algoritmo="Cubic"
#*****************************************************************
# La opción -persist mantiene la grafica, aún cuando el proceso
# gnuplot se detenga
#*****************************************************************
gnuplot -persist <<EOF
#*****************************************************************
set data style lines
set title "$cubic"
set xlabel "time (seconds)"
set ylabel "Segments (cwnd, ssthresh)"
#*****************************************************************
# Generar el gráfico desde captura.out usando los datos dados en
# las columnas 1 y 7, y , 1 y 8 de la Figura B.5
#*****************************************************************
plot "./captura.out" using 1:7 lt rgb "red" lw 6 title "cwnd", /
"./captura.out" using 1:(\$8>=2147483647 ? 0 : \$8) lt rgb     /
"black" lw 2 title "ssthresh"
EOF
#-----------------------------------------------------------------------------------------------
```

CAPITULO V
Resultados de la Simulación

5.0 Introducción

En este capítulo se muestran las pruebas experimentales realizadas para los algoritmos NewReno, Bic y Cubic, considerando un RTT (tiempo de ida y vuelta) de 50ms y un *jitter* de ± 6 ms. Con los parámetros de dados se contempló el caso de un enlace sin pérdidas y otro en el que se inducen perdidas mediante NETEM. Específicamente un 0,01% de pérdida o el equivalente la pérdida aleatoria de un segmento cada 10.000 segmentos transmitidos. Los aspectos teóricos sobre el funcionamiento de los algoritmos de control de congestión experimentados pueden leerse en la sección 3.2.0 del artículo principal que forma parte de este trabajo de tesis. La topología empleada para las simulaciones es la ilustrada en la Figura 4.1 en el Capítulo IV, o en la Figura 3.7 del artículo publicado en CoNaIISI.

5.1 Simulación de NewReno

La Figura 5.1 muestra el caso de un enlace sin pérdidas. Tal como se observa en la figura, inicialmente el algoritmo incrementa su ventana de congestión *(cwnd)* según la fase de arranque lento *(slow start)* para un valor relativamente alto inicial de *ssthresh* (umbral de arranque lento) que depende de la implementación del algoritmo en Linux. Resulta fácil observar que se genera una pérdida por la limitación del ancho de banda (enlace de 10 Mbps entre R1 y R2), por lo que el algoritmo entra en la fase de recuperación rápida de los segmentos transmitidos (aproximadamente a los 5 segundos en la Figura 5.1), modifica *sshthresh* e ingresa a la fase de prevención de congestión, en la cual la ventana de congestión *(cwnd)* experimenta un crecimiento lineal (cwnd = cwnd + 1 segmento por RTT).

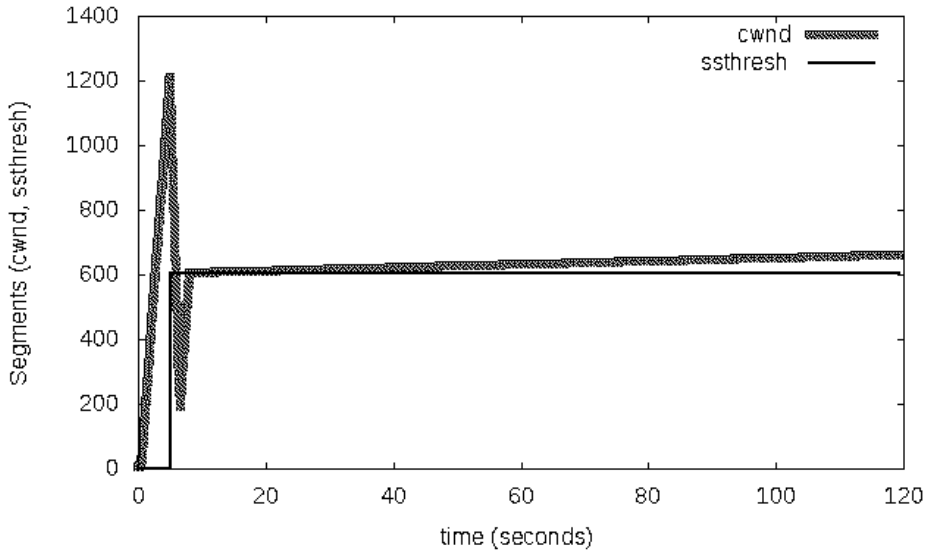

Figura 5.1: Simulación de NewReno 0% de pérdidas

Los resultados de la Figura 5.1 dejan en evidencia que el algoritmo NewReno no se adapta a redes con elevado producto entre el retardo y ancho de banda. El lector, al

observar la pendiente de crecimiento lineal de la *cwnd* comprenderá el excesivo tiempo que le tomará a NewReno redescubrir el límite impuesto por las condiciones de la red y la consecuente traducción en el bajo desempeño *(thrughput)* que impone a la conexión TCP.

La Figura 5.2 muestra el resultado de la simulación cuando se introducen pérdidas aleatorias en el tráfico a una tasa aproximada de 1 cada 10.000 segmentos. Se puede observar que NewReno disminuye consecutivamente la variable *ssthresh* para ingresar a la fase de *Congestion Avoindace* (CA) en que la ventana de congestión experimenta un crecimiento lineal. Bajo la circunstancia descrita el *throughput* para la conexión TCP es ínfimo y el algoritmo no se adapta a los parámetros impuestos por la red.

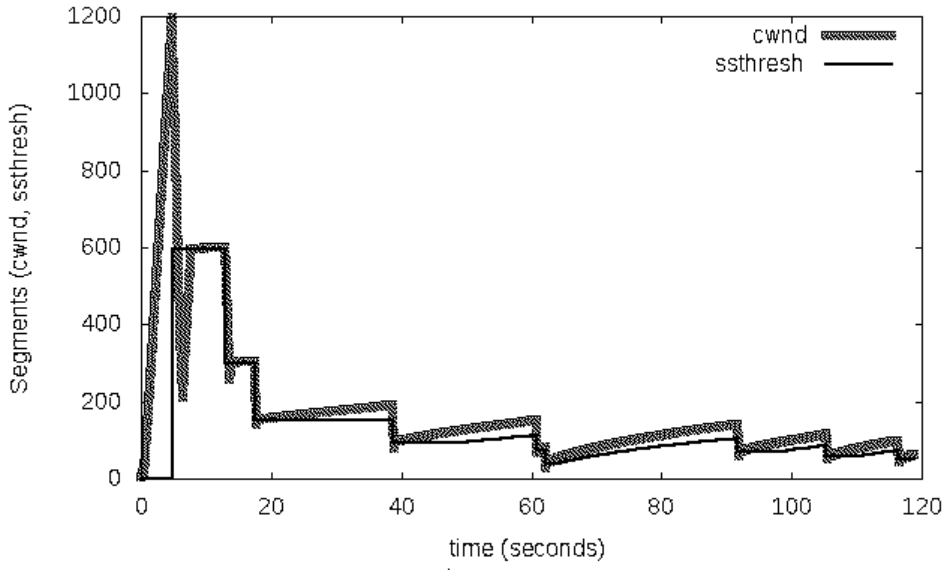

Figura 5.2: Simulación de NewReno 0,01% de pérdidas

5.2 Simulación de BIC

La Figura 5.3 muestra la dinámica de la ventana de congestión de BIC en ausencia de pérdidas. Nótese el comportamiento agresivo del algoritmo en la fase de descubrimiento del límite que impone el enlace entre R1 y R2 (10 Mbps). Luego de la etapa antes señalada y de acuerdo a lo expresado en la sección 3.2.4 del artículo principal, mantiene su ventana de congestión en un valor aproximado a los 800 segmentos de MSS *(Maximun Segment Size)*. El *throughput* bajo estas condiciones es más elevado que en NewReno y se puede calcular considerando el número de envíos según *cwnd* por segundo de tiempo.

La Figura 5.4 muestra el resultado de la simulación cuando se introducen pérdidas aleatorias en el tráfico a una tasa aproximada de 1 cada 10.000 segmentos. De la comparación entre NewReno y BIC, para esta circunstancia, se puede concluir que BIC reacciona agresivamente para adaptarse a los límites que impone la red, pero como contrapartida tiene la desventaja que fuentes con menor RTT se apoderaran de los recursos de la red (ancho de banda en bps) en detrimento de fuentes con valores más elevados de RTT.

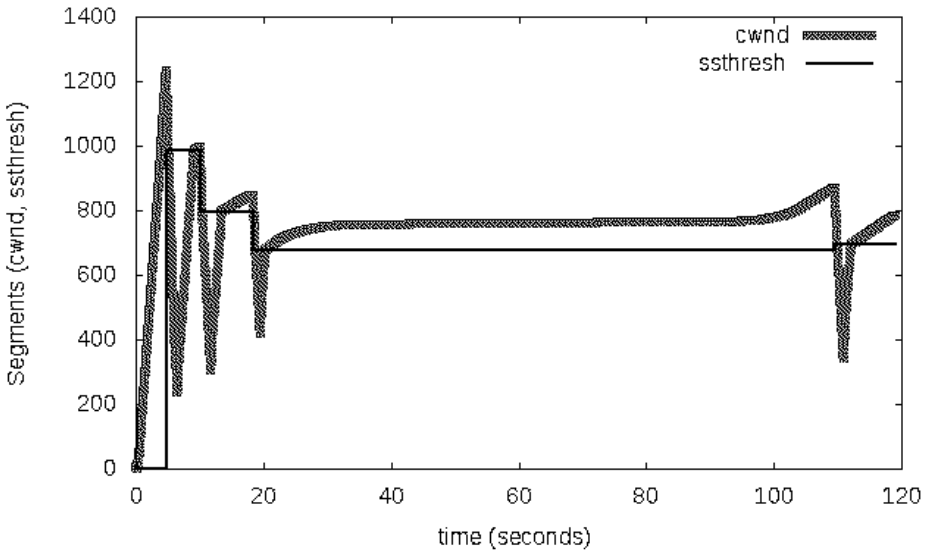

Figura 5.3: Simulación de BIC con 0% de pérdidas

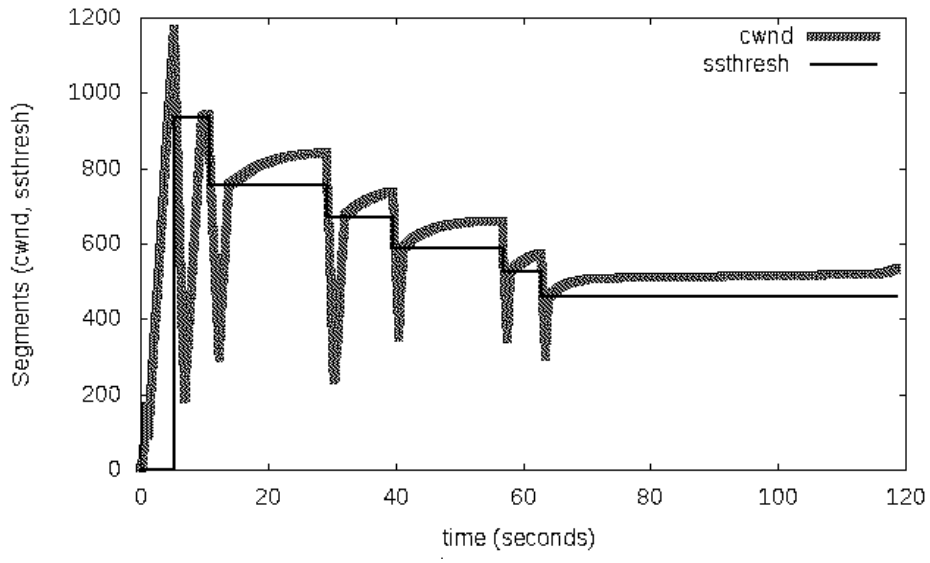

Figura 5.4: Simulación de BIC 0,01% de pérdidas

5.3 Simulación de Cubic

El principio de funcionamiento de Cubic fue tratado en la sección 3.2.5 del artículo principal que forma parte del trabajo de tesis. La función cúbica en la que basa su operación es menos agresiva que BIC y resuelve el problema de inequidad en la asignación del ancho de banda entre fuentes generadoras de tráfico TCP con distintos RTT. La Figura 5.5 ilustra la dinámica de Cubic en ausencia de pérdidas de segmentos y la Figura 5.6 muestra el comportamiento del algoritmo ante la presencia de pérdidas en igual porcentaje que los casos anteriores.

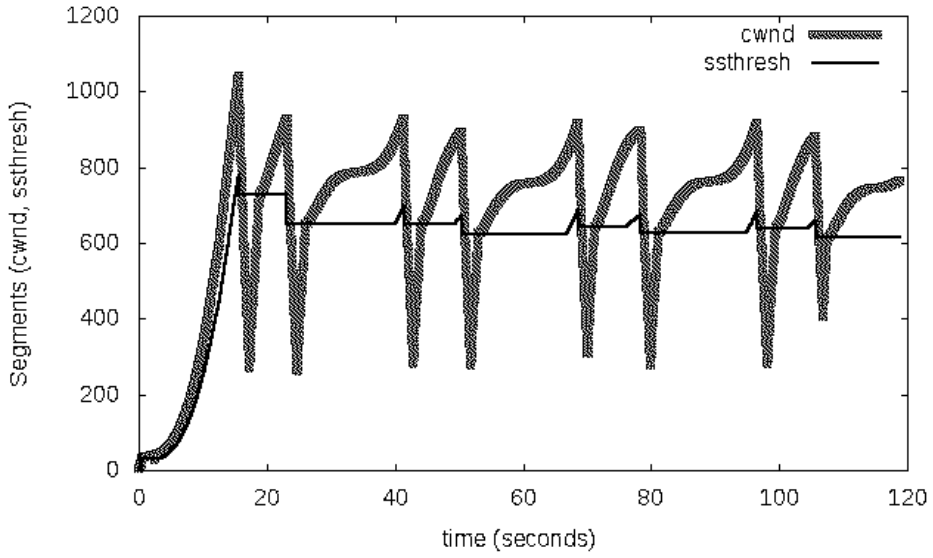

Figura 5.5: Simulación de Cubic con 0% de pérdidas

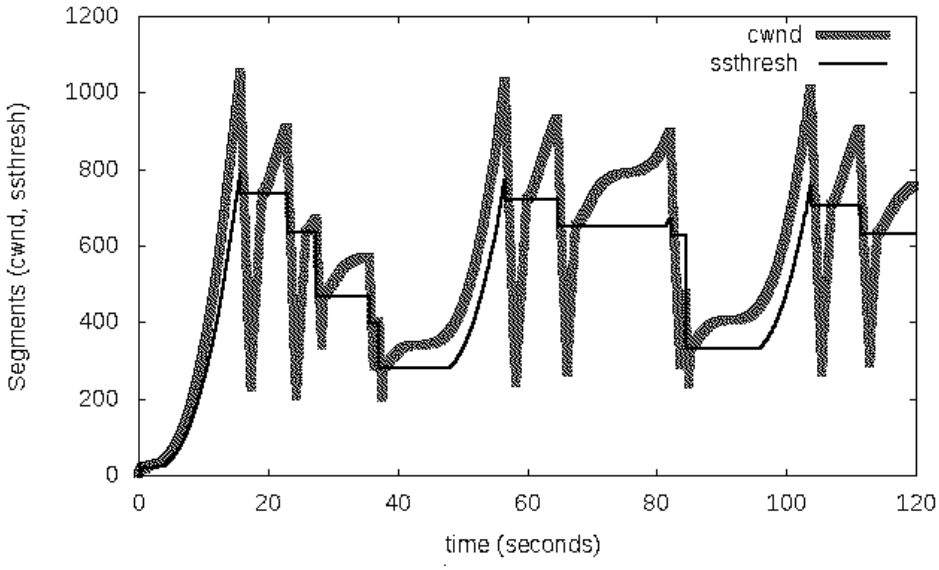

Figura 5.6: Simulación de Cubic con 0,01% de pérdidas

CAPITULO VI
Conclusión

6.0 Conclusión

El trabajo presentado constituye una novedad en al ámbito científico. No existe una publicación que aborde el tópico Control de Congestión y se realicen experiencias de laboratorio sobre un ambiente puramente virtual (UML).

El trabajo desarrollado habilita una línea de investigación que permitirá enriquecer la producción científica del área de Redes de Computadoras de la Facultad de Ingeniería de la UNLPam. Se podrán efectuar comparaciones entre los numerosos algoritmos de Control de Congestión existentes, así como efectuar modificaciones en el código de los módulos de los mismos para luego observar su comportamiento sobre diferentes escenarios. Además, permitirá la experimentación individual del tópico tratado en las clases de la asignatura Redes y Comunicaciones I.

Todo lo aportado permitirá un modo de experimentación sencillo que no requiere de componentes reales para configurar una topología compleja sobre ambientes WAN.

La tarea llevada a cabo no fue sencilla, se trabajó de manera ardua durante más de un año y hubo frustraciones intermedias, felizmente salvables. En los primeros intentos de experimentación el valor de la ventana de congestión *(cwnd)* no se condecía con los valores que debía tomar. Se consumió mucho tiempo en determinar que el elemento que producía tal alteración en las medidas era el componente virtual *uml_switch* y hubo que reemplazarlo por otra herramienta que lo supliera.

El dimensionamiento de los buffer de transmisión y recepción, relativos a TCP en los núcleos Linux (real y virtual) fue otro de los aspectos a contemplar. Se tuvieron que configurar a valores que aseguraran condiciones de no *overflow*, para que no afecte el comportamiento de los algoritmos experimentados.

Otro aspecto a destacar tiene que ver con el resultado de las medidas mostradas para los distintos algoritmos de Control de Congestión ¿Fueron corroboradas? ¿Las curvas mostradas en el trabajo de tesis, se corresponden con las de un ambiente sobre una infraestructura de red WAN? La respuesta es: si, pues de otro modo el trabajo tendría un nivel de incertidumbre no tolerable para un trabajo final de tesis.

La Figura 6.1 muestra la topología de una red de área amplia (WAN) real, a la que se tuvo acceso a fin de corroborar los resultados con los del laboratorio virtual.

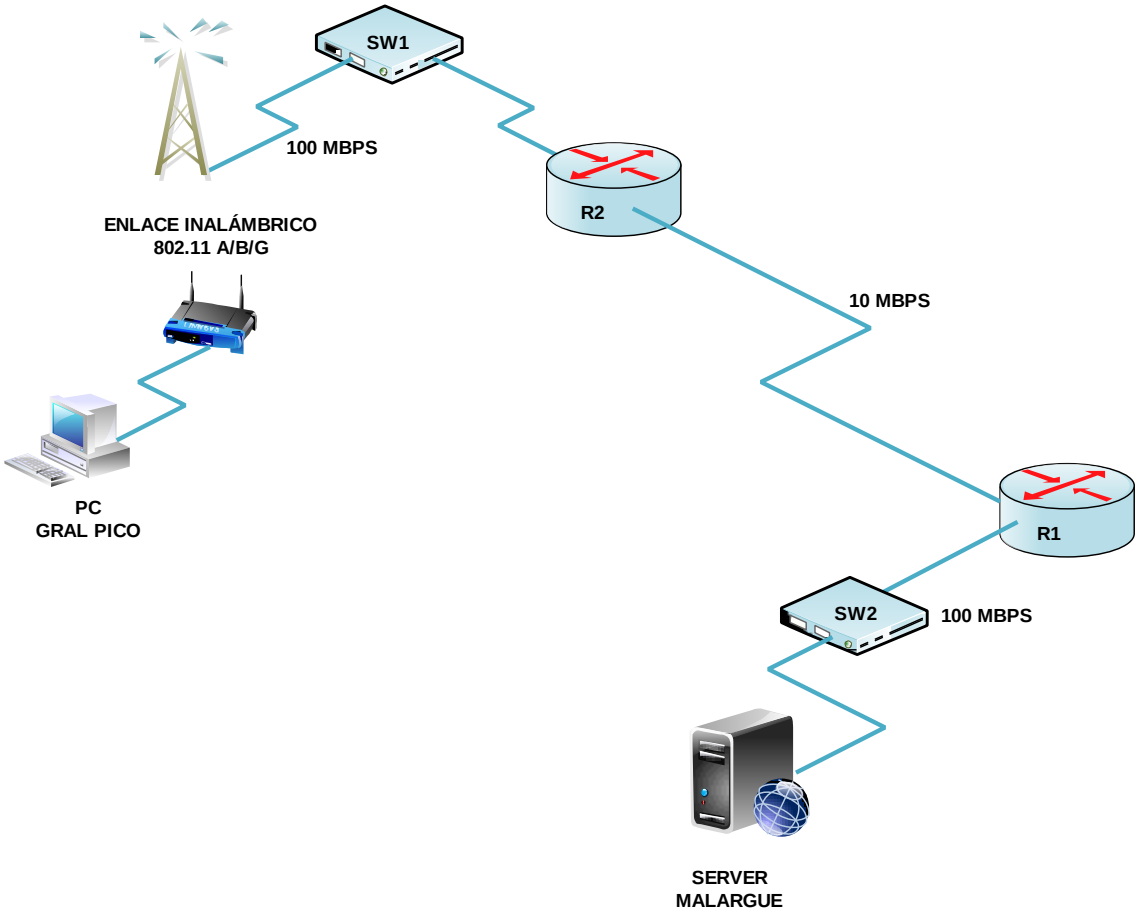

Figura 6.1: Topología red de área amplia

Es importante aclarar, que si bien el enlace simplificado entre las ciudades de General Pico y Malargüe en Mendoza se basa en Fibra Óptica, con un BER *(Bit Error Rate)* del orden de los 10^{-13} o aún menor, el emisor TCP (PC General Pico) se conecta a la WAN a través de un enlace inalámbrico cuyo BER es del orden de los 10^{-5}, aunque esta elevada tasa de errores de bit es atenuada por los mecanismos FEC *(Forward Errror Correction)* que emplean las tecnologías inalámbricas basadas en 802.11. El tiempo de ida y vuelta (RTT) de la red es de 51 ms con ±5 ms de *jitter*.

Lo escrito en el párrafo anterior sugiere que existirán perdidas de segmentos TCP debido a la tecnología de acceso a la red WAN y bajo estas condiciones, la aleatoriedad con que se distribuyen los errores, dará como resultado que las curvas real y virtual sean ligeramente distintas, pero mantendrán una coherencia respecto de los valores de la ventana de congestión *(cwnd)*.

La Figura 6.2 muestra el resultado del algoritmo NewReno para la topología real de la Figura 6.1 y la Figura 6.3 muestra la misma representación para la topología virtual. De ambas gráficas se puede deducir que la dinámica es exactamente la misma. Examinando en profundidad ambas gráficas se puede deducir que la ventana de congestión crece más en su etapa inicial en el ambiente virtual que en el real. Ello es

producto de un segmento de error en el enlace imperfecto descrito en el primer párrafo de esta página. Respecto de la diferencia en la variable umbral de arranque lento (ssthresh), se explica por el valor máximo alcanzado por la ventana de congestión, puesto que ssthresh=cwnd/2.

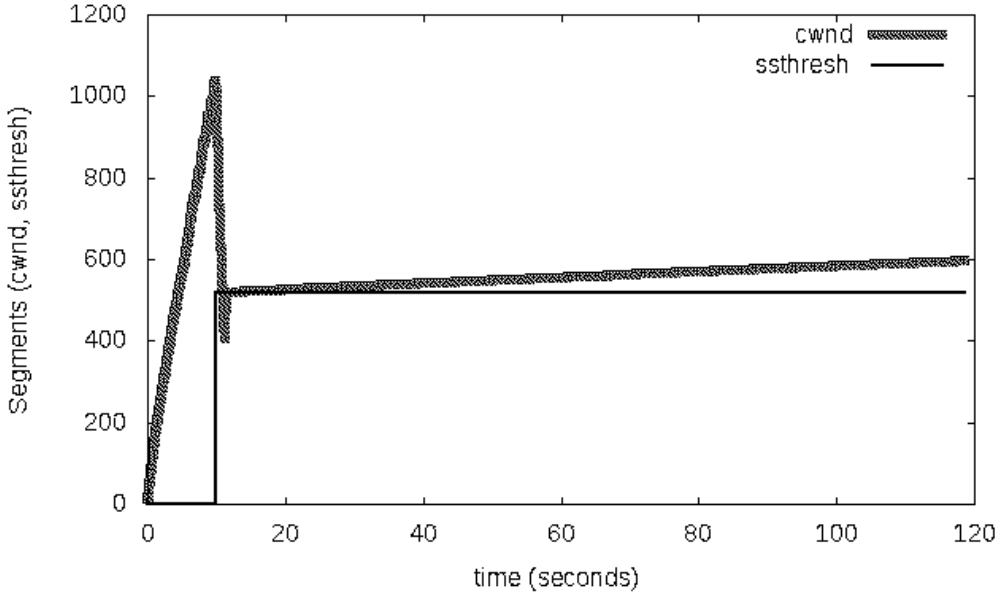

Figura 6.2: Dinámica de NewReno en la WAN real

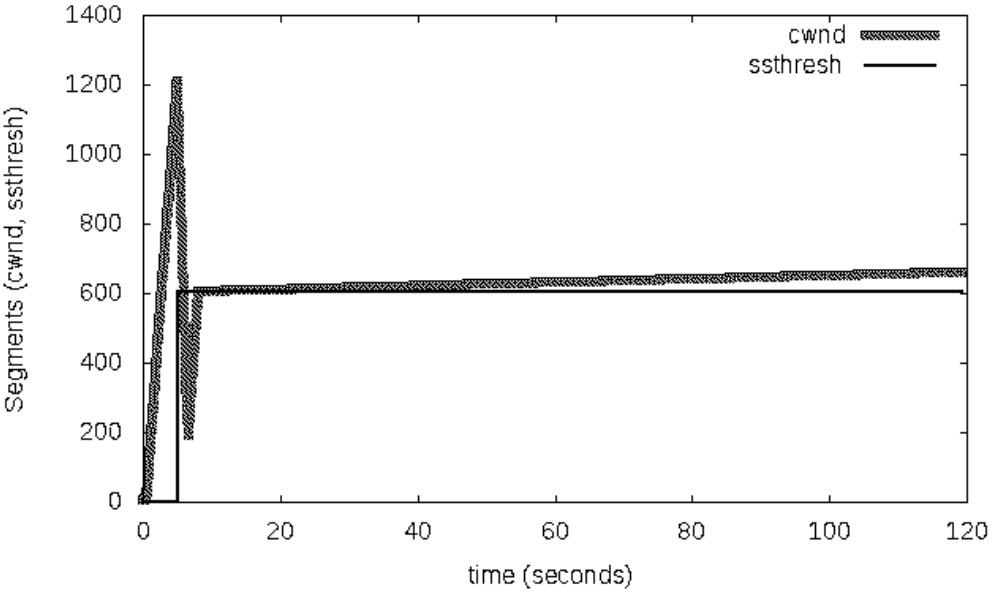

Figura 6.3: Dinámica de NewReno en la topología virtual

La Figura 6.4 ilustra la dinámica de la ventana de congestión cuando se experimenta Cubic sobre la red WAN real (Figura 6.1) y la Figura 6.5 expresa la misma representación sobre la topología virtual. De la observación de ambas graficas se puede concluir que la dinámica es prácticamente la misma, y los valores picos de *cwnd* y de *ssthresh* son muy aproximados.

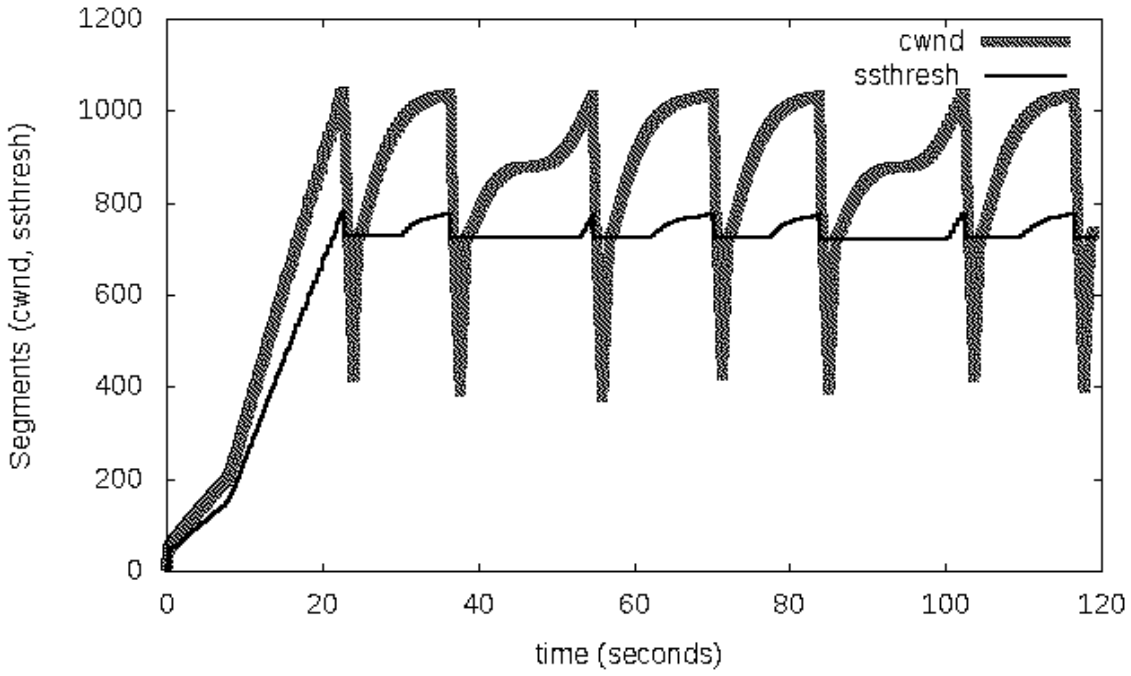

Figura 6.4: Dinámica de Cubic en la WAN real

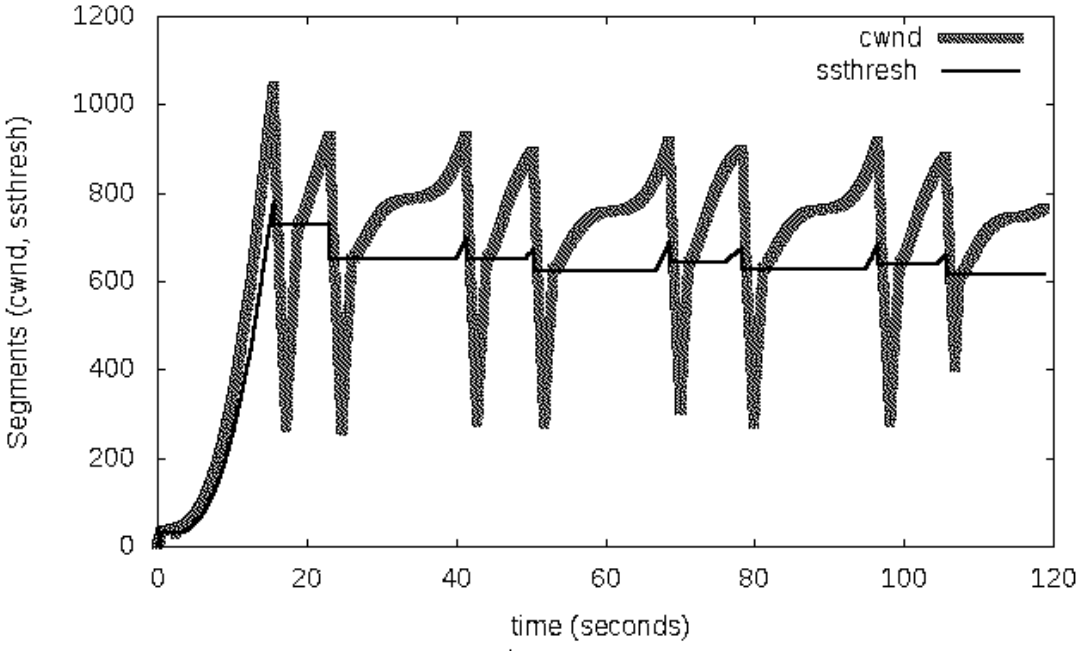

Figura 6.4: Dinámica de Cubic en el ambiente virtual

Por todo lo expresado, se corrobora que el resultado de las pruebas en el ambiente virtual se condice absolutamente con las mismas pruebas en un ambiente WAN (real).