Departamento de Ciencias de la Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Disertación entregada como requisito parcial para
obtener el título de Licenciado en Ciencias de la Computación

Un nuevo protocolo de transporte SCTP-RR

Autor: *Julio Jacobo Kriger* (LU: 207/96)

Director:
Dr. Claudio Enrique Righetti, Universidad de Buenos Aires

Jurados:
Dra. Patricia Borensztejn, Universidad de Buenos Aires
Dr. Ing. Marcelo Risk, Universidad de Buenos Aires

Fecha de entrega:
9 de Noviembre de 2009

Resumen

Stream Control Transmission Protocol (SCTP) es un protocolo de transporte confiable, recientemente estandarizado que presenta varias mejoras con respecto a TCP. Como todo protocolo de transporte, en particular los orientados a conexión, debe enfrentar los problemas de congestión, perdida de paquetes, duplicación, cambios bruscos del Round Trip Time (RTT) y reordenamiento de paquetes, entre otros.

A SCTP le incorporamos mecanismos para mejorar el rendimiento del mismo cuando se presentan dos problemas muy comunes, por ejemplo en redes inalámbricas: cambios bruscos del RTT y reordenamiento de paquetes, los cuales de acuerdo a las simulaciones realizadas demostraron mejorar el rendimiento. En este trabajo presentamos los resultados de utilizar SCTP-RR en un ambiente emulado con condiciones reales y controladas.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

El reordenamiento de paquetes[1] y el cambio brusco del Round Trip Time (RTT)[2] son anomalías frecuentes en las redes IP. Estos comportamientos de la red subyacente producen retransmisiones espurias que llevan a la degradación del rendimiento de los protocolos de transporte confiables, tales como TCP, SCTP[3]. Si bien las estadísticas sobre reordenamiento de paquetes y su influencia en el rendimiento de la red han sido causa de debate en el pasado, actualmente se consideran las mismas un parámetro de QoS a tener en cuenta en la actual ingeniería de redes. Los primeros trabajos que reportan el reordenamiento de paquetes [4] [32] ya coincidían en que es una propiedad de la redes IP.

Cuando la granularidad del reordenamiento de paquetes excede el umbral de reconocimientos duplicados del algoritmo Fast Retransmit [1] puede causar que el emisor retransmita datos y dispare el mecanismo de control de congestión innecesariamente. Un ejemplo muy típico son los enlaces inalámbricos que presentan reordenamiento de paquetes y variaciones bruscas de RTT debido a los protocolos de nivel de enlace, o asignación reactiva de recursos entre otras causas[4].

Además el reordenamiento de paquetes y variaciones bruscas de RTT pueden causar la expiración del Timeout de Retransmisión (RTO) produciendo retransmisiones y disparando el mecanismo de control de congestión.

El protocolo de transmisión de control de flujo SCTP (Stream Control Transmission Protocol) [7] es un estándar de nivel de transporte propuesto por el IETF e implementando en varias distribuciones de Linux y versiones comerciales de UNIX. Trabajos recientes demuestran los beneficios de SCTP en diversas aplicaciones como ser en computación paralela [19], aplicaciones de Internet [30], comunicación multihoming [14] [8], aplicaciones para dispositivos móviles [9], comunicaciones con tecnología WiFi y WiMax [31], seguridad en las asociaciones [12], transferencias multicaminos concurrentes [33] [14], manejo dinámico de direcciones IP [35] [40], entre otras.

En este trabajo presentamos la continuación de la evaluación de las modificaciones propuestas en [3] para el protocolo SCTP. Se desarrollo un algoritmo adaptativo con el fin de disminuir la ocurrencia de los denominados falsos Timeout y falsos Fast Retransmit, que producen degrada-

[1] Es la llegada a destino de paquetes en un orden distinto al que fueron enviados por el origen. Existen definiciones más estrictas sobre el reorden de paquetes en [27] y [18].

[2] Es el tiempo transcurrido desde la transmisión de un paquete hasta la recepción de su Ack correspondiente. No se toma en cuenta cuando incurrió en retransmisiones.

[3] En el caso de UDP se produce la degradación del servicio.

[4] Según [23], existen cinco grandes causas para el reordenamiento de paquetes: ruteo multicamino a nivel de paquete, route fluttering, paralelismo inherente de Router de alta velocidad, retransmisiones a nivel de la capa de enlace y treguas.

ción de rendimiento y la subutilización de recursos críticos: ancho de banda y buffer.

Se introdujeron las modificaciones propuestas en una librería de SCTP [49] y luego se realizaron varios experimentos en un ambiente de prueba controlado, emulando condiciones de reordenamiento de paquetes y cambios bruscos del RTT, utilizando los algoritmos de disciplinas de cola y herramientas disponibles en Linux (tc, NetEm, entre otras).

1.1. Los problemas

Los problemas fundamentales que surgen a partir de la incorporación de nuevas tecnologías subyacentes en las subredes son:

- Reordenamiento de Paquetes
- Cambios Bruscos del RTT

1.2. Objetivos

El objetivo principal de esta disertación es contribuir en los aspectos relacionados con la congestión en las redes IP, llevando al mundo real los cambios propuestos por [3] para así constatar la efectividad del mismo frente a problemas reales causados por el reordenamiento de paquetes IP y cambios abruptos en los valores del RTT.

Dicho objetivo es considerado primordial porque actualmente SCTP no cuenta con un mecanismo que haga frente a estas patologías de redes que provocan una seria degradación en el rendimiento del protocolo. Sin embargo, si estas modificaciones no son realizadas y probadas en el mundo real, su utilidad quedará restringida al mundo académico, y no contribuirá con necesidades actuales de las redes de computadoras.

Un objetivo secundario, pero fundamental y necesario, es poder emular las distintas condiciones de red necesarias para comprobar el correcto funcionamiento del algoritmo. Dichas condiciones deben ser posibles de emular en un ambiente real y controlado, para así poder obtener mediciones objetivas y conocer la verdadera magnitud de la mejora que propone SCTP-RR.

1.3. Contribuciones del trabajo

Una vez detallados los objetivos de la presente disertación, se enumeran las principales contribuciones de la misma:

- Análisis de los trabajos relacionados existentes para SCTP y SCTP-RR.
- Análisis de distintas herramientas para emular distintas condiciones de red.
- Comparativas de la efectividad del algoritmo y conocimientos la magnitud de su mejora en un ambiente real controlado.

1.4. Etapas del trabajo

Durante la primera etapa se investigan los problemas descritos, la congestión en las redes globales, sobre el protocolo SCTP, librerías que implementan SCTP y la emulación de condiciones de red.

Luego se revisan diversos trabajos previos con respecto a mejoras sobre SCTP y formas de emular las distintas condiciones de red. A continuación se explica el algoritmo propuesto SCTP-RR, desde la detección de Fast Retransmit y Timeout espurios, recuperación y adaptación a los cambios en la red.

Posteriormente se describen las disciplinas de colas y varias herramientas que provee el sistema operativo Linux para poder emular distintas condiciones de red necesarias para el algoritmo propuesto por SCTP-RR.

A partir de aquí se utilizan las herramientas provistas por Linux para emular las distintas condiciones de red y se realizan varios experimentos con distintas condiciones de red para conocer la magnitud de las mejoras en el rendimiento del protocolo.

Por último se expone un análisis de los experimentos, las conclusiones obtenidas y el trabajo a futuro.

1.5. Guía del trabajo

El presente capítulo 1 presenta una introducción a la problemática a resolver y etapas del presente trabajo, los problemas que se estudiaron y una descripción global de la estructura del trabajo.

Del capítulo 2 al capítulo 8 se presenta el desarrollo del trabajo en sí.

El capítulo 2 explica el problema de la congestión en la redes globales, luego se hace una breve descripción de SCTP, se comenta sobre la elección e implementación de SCTP elegida, y por último se define que es la emulación de redes de computadoras.

El capítulo 3 presenta los distintos trabajos previos sobre SCTP y las distintas posibilidades para emular distintas condiciones de red.

El capitulo 4 se revisan las modificaciones propuestas a SCTP.

El capítulo 5 explica qué son las disciplinas de colas de Linux y se describen las distintas herramientas de Linux a utilizar para emular distintas condiciones de red.

El capítulo 6 presenta como interactúan las disciplinas de cola y las distintas herramientas de Linux para lograr la emulación de las distintas condiciones de red, en especial como se reproduce el reordenamiento de paquete y los cambios bruscos de RTT que generan falsos Timeout y Fast Retransmit.

El capítulo 7 presenta los distintos experimentos realizados sobre distintas condiciones de red en un ambiente real controlado, los resultado obtenidos y el análisis de los mismos.

El capítulo 8 presenta las conclusiones del presenta trabajo y, finalmente, el capítulo 9 presenta el trabajo a futuro.

Capítulo 2

Los problemas, la congestión y SCTP

El presente capítulo explica los problemas, y la congestión en las redes globales. Luego se presenta la librería del protocolo SCTP elegida. Finalmente definimos que es la emulación de red.

2.1. Los problemas

SCTP, como todo protocolo de transporte, debe enfrentar los problemas de congestión, perdida de paquetes, duplicación, cambios bruscos del Round Trip Time (RTT) y reordenamiento de paquetes, entre otros. En redes inalámbricas, los problemas más comunes son el reordenamiento de paquetes y los cambios bruscos del RTT. A continuación describimos estos dos grandes problemas y sus impactos.

2.1.1. Reordenamiento de Paquetes

El reordenamiento de paquetes hace referencia al comportamiento de la red, en la cual el orden relativo de algunos paquetes, en el mismo flujo de datos, es alterado cuando estos paquetes viajan por la red. Existen cinco grandes causas para el reordenamiento de paquetes [23]:

Ruteo multicamino a nivel de paquete Ruteo multicamino es un técnica para balancear la carga del trafico de paquetes para propagar la carga del trafico a través de la red para aliviar la congestión de la red. Ruteo multicamino a nivel de paquete permite que el tráfico de paquetes de un mismo flujo sean enviados a destino por multiples rutas para lograr el balanceo de carga en redes de intercambio de paquetes. Este balance de carga produce que estos paquetes sufran reordenamiento al arribar al destino debido a diferencias de demora de las distintas rutas utilizadas.

Route fluttering Es un fenómeno de la red donde la ruta de transmisión a cierto destino oscila entre un conjunto de rutas disponibles al destino. Esto es el resultado de enlaces inestables, cargas pesadas o a grandes ráfagas de tráfico, donde la metrica para el costo de enlace empleada en el algoritmo de ruteo está relacionada con los retrazo o la congestion experimentada en los enlaces de la red. Esto trambién resulta en cambios topologicos en ambientes inalámbricos. Al igual que en ruteo multicamino a nivel de paquete, esto produce

que paquetes sufran reordenamiento al arribar al destino debido a diferencias de demora de las distintas rutas utilizadas.

Paralelismo inherente en routers modernos de alta velocidad Los routers modernos soportan la creación de bandas de paquetes para que los paquetes de un mismo flujo puedan ser reenviados a través de routers de menor capacidad, pero a costos mucho más barato por el uso en paralelo de varios enlaces al siguiente router de ese flujo. Para cambiar los paquetes a alta velocidad, este router trabaja generalmente de manera conservadora, a fin de que sus puertos de salida conectados al siguiente router estén inactivos cuando no hay paquetes pendientes a reenviar a ese router. Dado que los paquetes pueden ser de diferentes tamaños y los vínculos pueden ser de diferentes anchos de banda, los paquetes pueden tomar diferentes tiempos de transmisión, y por lo tanto, llegar al router vecino en un orden diferente del de su envío original.

Retransmisiones a nivel de la capa de enlace Los mecanismos de retransmisiones a nivel de la capa de enlace han sido propuestos para recuperar, de manera eficiente, pérdidas de transmisión debido a altas tasas de error de canal en redes inalámbricas. Estos paquetes se envían retransmitidos sólo después de que se detecten las pérdidas. Estos paquetes pueden entonces ser intercalados con otros paquetes que pertenecen a la misma el flujo de tráfico.

Treguas Algunos routers pueden detener su actividad de reenvío de paquetes en buffer cuando se procesa una actualización de las tablas de enrutamiento. Estos paquetes almacenados en el buffer se intercalan con los recién llegados, lo que causa el reordenamiento de paquetes.

2.1.2. Cambios Bruscos del RTT

El Round Trip Time (RTT) es el tiempo transcurrido desde la transmisión de un paquete hasta la recepción de su Ack correspondiente. Este intervalo incluye la propagación, encolado en buffer, procesamiento y otros retrasos en los routers y maquina destino. El rendimiento de TCP y SCTP es inversamente proporcional a su RTT. En las implementaciones de TCP, se espera que el protocolo mida y se adapte a los cambios de RTT para que su comportamiento de retransmisión balancee el rendimiento hacia el usuario y el uso eficiente de la red. Vale notar que no se toma en cuenta la medición cuando se incurrió en retransmisiones.

2.2. El impacto de los problemas

Los problemas arriba descritos inciden en el rendimiento tanto de TCP como de SCTP. A continuación definimos los impactos sufridos, tanto para cuando hay reorden de paquetes como cuando hay cambios bruscos de RTT.

2.2.1. Reordenamiento de Paquetes

Retransmisiones espurias El reordenación de paquetes hace que el número inicial octeto de datos de algunos segmentos difieran de los esperados en un destino. En otras palabras, el destino encuentra un agujero de la secuencia en la recepción de un segmento. A continuación, el destino genera Ack duplicados y los envía al origen. Cuando el origen recibe tres Ack duplicados consecutivamente, se retransmite el *deducido* segmento perdido (aunque en realidad no existe dicha pérdida). El persistente y sustancial reorden de paquetes hace a menudo que algunos segmentos de TCP sean retransmitidos espuria e innecesariamente, llevando al colapso por congestión.

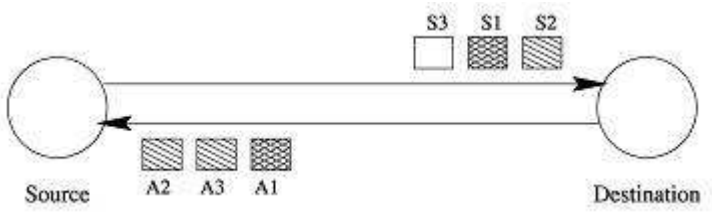


Figura 2.1: Ejemplo de *forward-path reordering* y *reverse-path reordering*

Ventana de congestión innecesaria pequeña Fast Recovery se activa siempre con un Fast Retransmit. Un Fast Retransmit espurio no sólo genera una mayor carga de trabajo innecesario a la red y al destino, sino que también reduce la ventana de congestión. Por lo tanto, la ventana de congestión se mantiene pequeña en relación con el ancho de banda disponible de la ruta de transmisión, con un persistente y sustancial reordenamiento de paquetes.

Pérdida de *Reloj Ack* El reorden de paquetes causa que no sólo segmentos de datos, sino también Ack lleguen a un destino fuera de orden. El primer fenómeno se llama *forward-path reordering*, mientras que el segundo se conoce como *reverse-path reordering*. Un ejemplo de *forward-path reordering* y *reverse-path reordering* se puede ver en la figura 2.1. Supongamos que los segmentos se envían desde el origen en orden S1, S2, S3, pero el segmento S1 llega después de segmento S2 al destino. Esto representa un *forward-path reordering*. Ahora el destino enviá los Ack correspondientes, y el Ack A1 llega después de los Ack A2 y A3 al origen. Esto representa un *reverse-path reordering*.

El *reloj Ack* se refiere a la propiedad del destino que no puede generar Ack más rápido que los segmentos de datos que pueden viajar por la red. Incluso cuando no hay reorden de segmento de datos, Ack desordenados pueden llevar al origen a transmitir varios segmentos juntos en lugar de uno o dos segmentos por Ack. Esto ocasiona la pérdida de *reloj Ack* y rafagas de tráfico mucho más explosivas, lo que puede dar lugar a la congestión transitoria de la red y el colapso por congestión por los paquetes no entregados.

2.2.2. Cambios Bruscos del RTT

Un problema muy común en TCP es la ambigüedad en la retransmisión de paquetes, debido a que las implementaciones de TCP no pueden distinguir, al momento de realizar una medición de RTT, cuando llega un Ack de un paquete original o de paquete retransmitido. El algoritmo de Karn [20] alivia el problema descartando todas las medidas de RTT hasta recibir un Ack de un segmento que no se haya retransmitido. Los cambios bruscos del RTT causan la expiración del Timeout de Retransmisión (RTO) produciendo retransmisiones y disparando el mecanismo de control de congestión, siendo muchas de estas retransmisiones espurias, con su consecuente disminución del rendimiento de la conexión. Si el RTO es grande entonces el origen sera menos sensible a la pérdida de paquetes debido a la congestión.

2.3. Congestión en redes globales

El control de congestión es un problema de constante estudio en redes de conmutación de paquetes no orientadas a conexión[1], como ser Internet. En una primera aproximación, podemos

[1] En realidad también ocurre en redes orientadas a conexión, como ser ATM.

definir la *congestión* como *al estado sostenido de sobrecarga de una red donde la demanda de recursos (enlaces y buffer) se encuentra al límite o excede la capacidad de los mismos*, o también como *una red se dice que está congestionada desde la perspectiva de un usuario si la calidad de servicio percibida por el usuario disminuye debido a un aumento en la carga de la red.*

2.3.1. Control de congestión y asignación de recursos

La asignación de recursos y control de la congestión han sido temas de constante estudio desde que estos problemas fueron detectados[2]. Uno de los factores que incrementa la complejidad de estos problemas, es que no están aislados a un único nivel del modelo de arquitectura de redes[3].

La asignación de recursos, principalmente buffer y ancho de banda de los enlaces, se encuentra en parte implementada en los Router o Switch dentro de las redes; y en parte en los protocolos de transporte que se ejecutan en los host. Existen protocolos de red que trabajan anunciando los requerimientos de recursos que necesita para establecer una comunicación a los nodos intermedios, quienes responden con información acerca de la disponibilidad de los mismos. La asignación de recursos es proceso por el cual los elementos dentro de una red intentan alcanzar las demandas competitivas que las aplicaciones poseen en cuanto a recursos de red; principalmente ancho de banda y espacio de buffer en Router o Switch. En general, no es posible lograr satisfacer todas las demandas de los usuarios, lo que significa que algunas aplicaciones de usuario recibirán menos recursos de red que lo requerido. Esto lleva a la necesidad de decidir cuando y como ceder o no estos recursos.

El objetivo del mecanismo de *control de congestión* es la utilización de la red tan eficientemente como se pueda, es decir, obtener el máximo rendimiento posible manteniendo un bajo costo de mantenimiento y pequeñas demoras. La congestión debe ser evitada. Control de congestión se utiliza para describir los esfuerzos que realiza un nodo de red, un Router o un host, para prevenir o responder a condiciones de sobrecarga. La existencia de congestión dentro de la red es un mal síntoma, por lo cual es conveniente tomar medidas preventivas y no correctivas al respecto.

Sobreprovisionamiento

Hoy en día, la elección más común de los ISP (Internet Service Provider) para resolver el problema de la congestión dentro de sus propias redes es aumentando el ancho de banda, *sobreprovisinando* las necesidades de los usuarios. Esto se debe a:

- El ancho de banda es barato. Conviene más sobreprovisionar la red si el costo es significativamente menor que el dinero que el ISP podría perder en caso de quejas de los usuarios
- Es más difícil controlar una red que tiene el ancho de banda justo, que una red sobreprovisionada. Aquí hay que considerar el costo de administradores de red especializados y posibles quejas de usuarios en caso de fallas en la red.
- Con sobreprovisionamiento, el ISP está preparado para un futuro con mayor cantidad de clientes.

[2] Nagle detecta y reporta congestión en 1984 [28], y en una primera aproximación propone un mecanismo centrado en los Router. Durante 1986 y 1987 se producen varios fenómenos de congestión, y quienes mantenían el Backbone, BBN, proponen como una solución parcial aumentar el ancho de banda de los enlaces. Finalmente Jacobson y Karels intenta mitigar el problema de congestión introduciendo los algoritmos de control de congestión detallados en [17].

[3] En [41] se detalla como afectan a la congestión las políticas de los protocolos de nivel 2, 3 y 4.

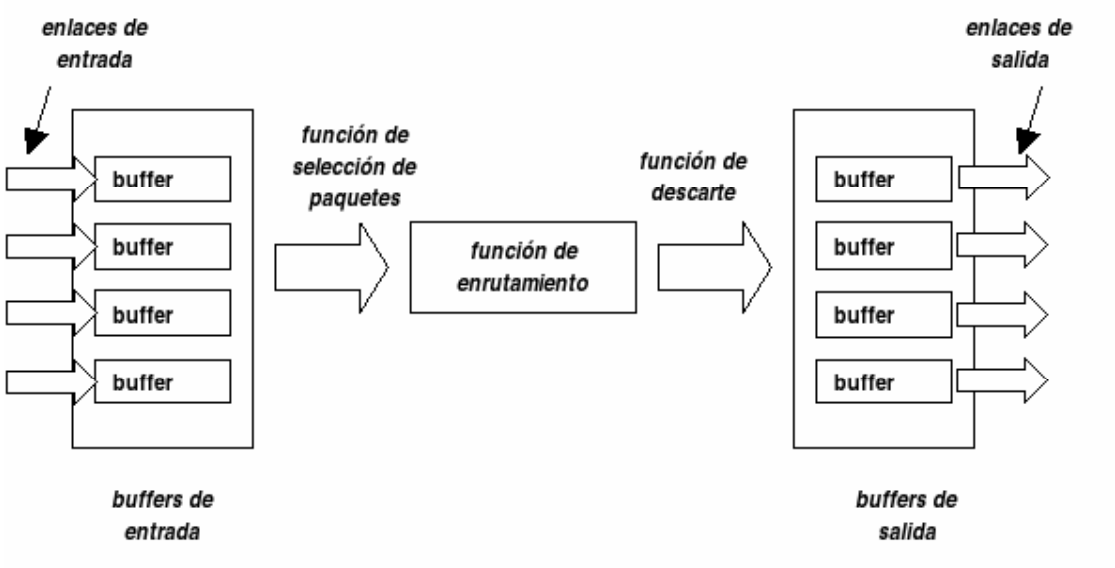


Figura 2.2: Arquitectura básica de un Router

Vale notar que el sobreprovisionamiento de ancho de banda no resuelve el problema de la congestión, si ésta se utiliza la red descuidadamente, las colas de los Router crecerá y aparecerá demora y perdida de paquetes.

2.3.2. Esquema de funcionamiento de los Router

Todos los enlaces en una red se encuentran unidos por medio de Router. Éstos reenvían paquetes de sus enlaces de entrada a enlaces de salida apropiados, para que el paquete sea enviado a destino. La figura 2.2 muestra la arquitectura básica de un Router.

Generalmente los enlaces de entrada y salida son apareados para formar canales full-duplex, donde los datos fluyen en ambas direcciones simultáneamente, o canales half-duplex, donde los datos fluyen en una única dirección a la vez. Los paquetes entrantes son almacenados en buffer de entrada, y luego son seleccionados por una función de selección de paquetes y pasados a una función de enrutamiento. Esta función de enrutamiento determina a cual enlace de salida el paquete debe ser enviado para que llegue a su destino. Esto se logra mediante una tabla de encaminamiento. Una vez seleccionado el enlace de salida, la función de enrutamiento pasa el paquete a una función de encolamiento de paquetes, que los encola en el buffer de salida del enlace. Cuando el paquete alcanza cabeza de la cola es transmitido por el enlace al próximo Router o a su destino final.

La función de selección de paquetes elige los paquetes que se encolados en los buffer de entrada para ser encaminados por la función de enrutamiento según su propia lógica. Normalmente se utiliza el método FIFO (First-In First-Out)[4], sin embargo existen otros métodos de selección útiles.

Existen dos cuellos de botella fundamentales en esta arquitectura de Router:

- El tiempo mínimo requerido por un Router para decodificar la cabecera de un paquete entrante, determinar por cual enlace saliente enviarlos y pasarlo al mismo.

[4] Existen otras políticas de encolado, tales como FQ, WFQ, etc.

- El tiempo de espera que sufre un paquete en los buffer de entrada, hasta que es procesado por el Router; y salida, hasta que es transmitido por el enlace saliente.

Por definición, los buffer de entrada y salida de los Router son finitos[5]. Si un buffer se llena entonces no podrán encolarse mas paquetes y, por lo tanto, el Router tendrá que empezar a descartar paquetes que lleguen. Esta pérdida de datos en un flujo entre un origen/destino causa la retransmisión de los mismos.

Si el Router debe encolar un paquete, ya procesado, pero el buffer de salida está lleno, el Router puede puede descartar el paquete no encolado, o descartar un paquete ya encolado, para así poder encolar el nuevo. Esta elección es ejecutada por la función de enrutamiento de paquetes[6]. Esto no puede hacerse para los buffer de entrada, ya que el paquete no es almacenado en la memoria interna del Router hasta que sea encolado en dichos buffer. Como consecuencia, los paquetes pueden ser descartados en su arribo, y en tal caso el Router no tiene control sobre qué paquetes son los descartados.

Finalmente, los buffer de los Router pueden compartir un espacio de memoria común, o pueden tener espacios de memoria individuales. En el primer caso, ningún buffer puede llenarse hasta que todo el espacio de memoria del Router sea utilizado. En el segundo caso, la utilización de un buffer no influye en la utilización de cualquier otro buffer.

2.3.3. Taxonomías de asignación de recursos

Existen muchos y diversos mecanismos de asignación de recursos. A continuación se enumeran los principales tres.

Centrada en los Router vs. Centrada en los Host Los mecanismos de asignación de recursos pueden ser clasificados en dos grupos: aquellos que tratan el problema adentro de la red (Router o Switch) y aquellos que tratan el problema desde los bordes de la red (host).

En un diseño centrado en los Router, cada Router toma la responsabilidad de decidir cuando los paquetes son reenviados y seleccionar qué paquetes serán descartados, y a la vez informar a los host que generan el tráfico de red cuantos paquetes se les permite enviar.

En un diseño centrado en los host, son ellos los que observan las condiciones de la red (como ser cuantos paquetes obtiene de la red correctamente) y ajustan su comportamiento acorde a esta información.

Es importante destacar que estos dos grupos no son mutuamente excluyente.

Basada en Reserva vs. Basada en Retroalimentación La asignación de recursos se puede hacer mediante una reserva de los mismos o por retroalimentación de información de los recursos.

En un sistema basado en reserva de recursos, el host le solicita a la red ciertos requisitos al momento de establecer un flujo. Luego, cada Router asigna recursos suficientes (buffer y/o porcentaje de ancho de banda del enlace) para satisfacer el requerimiento. Si el mismo no puede ser satisfecho en algún Router, es rechazado.

En un sistema basado en retroalimentación, los host comienzan a transmitir datos sin previamente solicitar requisitos alguno, y luego empiezan a ajustar su tasa de transmisión acorde a la información de retroalimentación recibida. La misma puede ser explícita (Router

[5] Aún siendo infinitos el fenómeno de congestión en ese caso empeora [29].

[6] Normalmente los Router tienen una política de encolado FIFO y un descarte Tail Drop (se encola hasta cierta cantidad, descartando los paquetes excedentes), si embargo existen técnicas más refinadas de descarte tales como RED y WRED.

congestionados envían un pedido de reducción de emisión de tráfico) o implícita (los host ajustan la tasa de transferencia de acuerdo a los comportamientos externos observados en la red, como ser pérdida de paquetes o aumento del RTT).

Es importante notar que un sistema basado en reserva siempre implica un mecanismo de asignación de recursos centrado en los Router. Esto se debe a que cada Router es responsable del seguimiento de sus recursos, y para asegurarse de que cada host se mantenga dentro de la reserva que realizó.

Por otro lado, un sistema basado en retroalimentación puede llegar a implicar un mecanismo centrado en los Router o centrado en los host.

Si la retroalimentación es explícita, el Router está involucrado en lo que respecta al esquema de asignación de recursos.

Si la retroalimentación es implícita, la mayoría del trabajo recae sobre los host; los Router descartan los paquetes silenciosamente frente a congestión.

Basado en Ventana vs. Basado en Tasa Los mecanismos de asignación de recursos pueden ser clasificados en dos grupos: si están basados en ventana o basados en tasa.

Los mecanismos de asignación de recursos necesitan una manera de informar al emisor cuantos datos le es permitido transmitir. Lo protocolos de transporte basados en ventana, como ser TCP, informan al emisor una ventana de transmisión de datos. Esta ventana se corresponde con el espacio de buffer que el receptor posee, y limita la cantidad de datos a transmitir por el emisor.

También es posible controlar el comportamiento del emisor usando una tasa, es decir, cuantos bits por segundo el receptor o la red son capaces de manejar. Los flujos basados en tasas son una elección lógica en un sistema basado en reserva que soporta diferentes calidades de servicios, el emisor realiza una reserva para cierta cantidad de bits por segundo, y cada Router a lo largo de la ruta determina si puede soportar esa tasa.

2.3.4. Modelo de red

A continuación se describen tres de las principales características de la arquitectura de red sobre la cual se basa el presente trabajo.

Redes de conmutación de paquetes Una red de conmutación de paquetes consiste de múltiples enlaces y Router. Los host son los encargados de observar las condiciones de la red (por ejemplo, la pérdida de paquetes y cambios en el tiempo de ida y vuelta de un paquete) para ajustar su comportamiento a la red subyacente. Por último, los protocolos de transporte utilizan mecanismos de ventana deslizante para informar la capacidad de sus buffer.

El modelo de red aquí presentado se puede clasificar dentro de una taxonomía de asignación de recursos centrada en los host, con retroalimentación implícita y basada en ventana.

En este ambiente, Internet, un emisor puede tener capacidad más que suficiente para enviar un paquete en un enlace de salida, pero en otro lugar de la red sus paquetes pueden tener que compartir un enlace con otros emisores.

En la figura 2.3 podemos observar el siguiente escenario: dos enlaces de gran ancho de banda están alimentando un enlace de un ancho de banda muy inferior.

Por último, los paquetes pueden ser de tamaño variable y son encaminados de un enlace a otro por medio de Router.

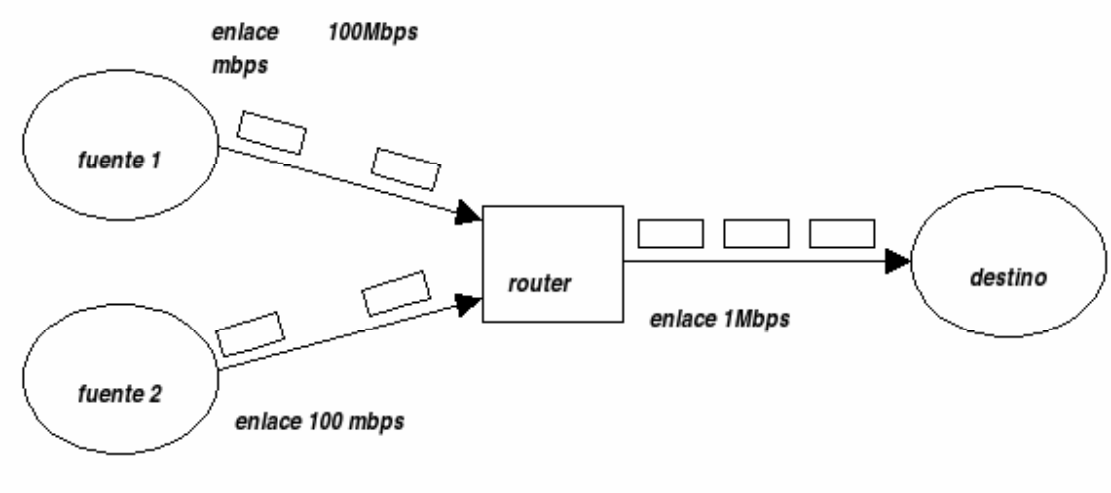


Figura 2.3: Congestión en una red de conmutación de paquetes

El presente trabajo se basa en una arquitectura de red con esta característica, redes de conmutación de paquetes.

Flujos no orientados a conexión La actual Internet es una red de flujos[7] no orientados a conexión. El protocolo *IP* provee un servicio no orientado a conexión de envío de datagramas, y el protocolo *TCP* implementa una capa de abstracción que brinda un servicio orientado a conexión punto a punto.

Por lo tanto es necesario definir más claramente el término *no orientado a conexión.*

La clasificación entre redes que son *orientadas a conexión* o *no orientadas a conexión* es demasiado estricta, pues existen clasificaciones intermedias. En particular, la implicancia de una red no orientada a conexión donde todos los datagramas son completamente independientes es muy fuerte. Aunque los datagramas sean conmutados en forma independiente, es común que pasen por un conjunto de Router en particular.

Dado que por los Router viajan muchos paquetes pertenecientes a distintos flujos, es muy útil mantener estados de información para cada flujo, con el fin de poder tomar buenas decisiones en lo que respecta a asignación de recursos para los distintos flujos de paquetes. Este tipo de almacenamiento de información por parte de los Router suele denominarse *soft state.*

Esta es una de las principales diferencias con las redes orientadas a conexión, donde se establecen *canales* o *circuitos virtuales* por medio de un mecanismo de señalización donde los Router mantienen información por cada uno de ellos. Esta información que han de guardar los Router se define como *hard state.*

Soft state es un punto intermedio entre redes no orientadas a conexión, que no mantienen estado en los Router; y redes orientadas a conexión, que mantienen hard state dentro de los Router. El correcto funcionamiento de la red no depende de la existencia de soft state dentro de los Router, ya que cada paquete puede ser encaminado correctamente sin hacer uso de esta información. Sin embargo, cuando un paquete pertenece a un flujo para el cual

[7] Un *flujo* es una secuencia de paquetes enviada entre dos host que siguen una misma ruta a través de la red.

el Router mantiene soft state, este Router puede tomar mejores decisiones de enrutamiento, como ser eficiencia en el enrutamiento, balance de carga entre los flujos y Quality of Service (QoS) en el nivel de red.

Los flujos pueden llegar a ser *definidos implícitamente* o *establecidos explícitamente*. En el primer caso, cada Router inspecciona los paquetes que viajan entre un mismo par de nodos origen/destino y trata a estos paquetes como pertenecientes a un mismo flujo con el fin de evitar congestión. En el segundo caso, en un flujo establecido explícitamente el origen necesita enviar un *mensaje de inicialización* a través de toda la red, declarando que un flujo de paquetes está por comenzar. Vale notar que en este último caso, un flujo no implica ninguna semántica punto a punto, y en particular no implica el envío ordenado y confiable de un circuito virtual; solo existe con el propósito de asignación de recursos.

El presente trabajo se basa en una arquitectura de red con esta característica, flujos no orientados a conexión.

Modelo de servicio *mejor esfuerzo* En Internet se utiliza el modelo de servicio de *mejor esfuerzo*. En este modelo, cada paquete es tratado de la misma manera, donde ningún host tiene la oportunidad de requerir a la red una determina calidad de servicio para sus flujos, como ser ancho de banda, buffer, demora, o confiabilidad sobre el arribo de los paquetes. Sin embargo, si bien en la actualidad se puede brindar QoS a determinados flujos, por ejemplo Voz sobre IP (VoIP) (en este caso se puede acotar el demora), el mismo tiene un alcance limitado en la Internet global.

El presente trabajo se basa en una arquitectura de red con esta característica, Modelo de servicio *mejor esfuerzo.*

2.3.5. Congestión en redes de conmutación de paquetes no orientadas a conexión

Una red se considera congestionada cuando uno o más de sus nodos deben descartar paquetes debido a la falta de espacio en sus buffer. En el modelo de red sobre el cual se basa el presente trabajo, la congestión ocurre cuando un host emisor de datos no puede reservar ancho de banda en su camino al destino, por lo que es incapaz de determinar qué tasa de transmisión de datos puede mantener con el destino. Si el origen transmite datos a una tasa superior a la posible entre el emisor y receptor, uno o más Router intermedios comenzarán a encolar los paquetes en sus buffer. De mantenerse el encolamiento de paquete, los buffer pronto se llenarán y empezarán a ser descartados, con la consecuente pérdida de datos. Si el emisor quiere garantizar la confiabilidad en la transmisión, deberá retransmitir de los datos descartados, con lo cual empezará a perder rendimiento en la comunicación. La figura 2.4 muestra el rendimiento de un red en función de su carga.

Como se puede observar, a medida que la *carga*[8] de la red aumenta, el *rendimiento (throughput)*[9] se incrementa linealmente. Sin embargo, a medida que la carga alcanza la capacidad máxima de la red, los buffer en los Router comienzan a llenarse. Esto causa el incremento del tiempo de *respuesta*[10] y disminuye el rendimiento.

Vale notar que una vez que los buffer de los Router comienzan a sobrecargarse ocurre la pérdida de paquetes. Al incrementar la carga más allá de este punto aumenta la probabilidad de pérdida de paquetes. Bajo cargas extremas, el tiempo de respuesta tiende a infinito y el

[8] La tasa de datos transmitida.
[9] La tasa de datos que alcanzan el destino.
[10] El tiempo que tardan los datos en atravesar la red entre el origen y destino

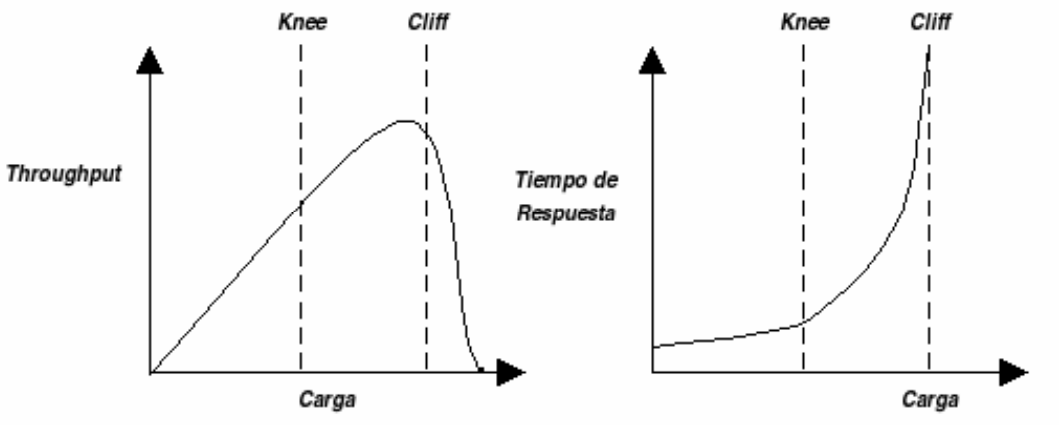


Figura 2.4: Rendimiento de la red en función de la carga

rendimiento tiende a cero; este es el punto del colapso de congestión. Este punto es conocido como el *cliff* debido a la extrema caída en el rendimiento.

La figura 2.5 muestra la potencia[11] en función de la carga de la red.

2.4. Protocolo SCTP

El protocolo de transmisión de control de flujo SCTP (Stream Control Transmission Protocol) [39] es un estándar de nivel de transporte propuesto por el IETF e implementando en varias distribuciones de Linux y versiones comerciales de UNIX. SCTP es producto de los esfuerzos orientados en sus comienzos en el desarrollo protocolo de señalización telefónica para redes IP. Provee provee una conexión full-duplex confiable[12] y mecanismos de control de congestión. También, SCTP ofrece nuevas opciones de envío que son particularmente deseables para señalización telefónica y aplicaciones multimedia:

Entrega de datos fuera de orden confiable En TCP es requerido que la entrega de segmentos sea en orden, SCTP permite que la entrega sea fuera de orden, logrando así entregar datos rápidamente.

Preservación de los límites de mensaje Dada la posible entrega de segmentos fuera de orden en SCTP, éste consta con un mecanismo para identificar claramente que partes corresponden a que mensaje; esto no es posible en TCP ya que está orientado a flujo de bytes.

Soporte para múltiples flujos de datos independientes Mecanismo inexistente en TCP que permite transmitir múltiples flujos de datos en una sola conexión.

Multihoming Permite tener múltiples direcciones IP.

Confiabilidad parcial Permite definir cuan persistente debe ser el protocolo al intentar de entregar un mensaje.

[11] $Potencia = \frac{Throughput^{\alpha}}{Delay}, con(0 < \alpha < 1)$

[12] Si bien existen algunas propuestas como [38] para permitir comunicaciones parcialmente confiable, muy útiles en aplicaciones de audio y video.

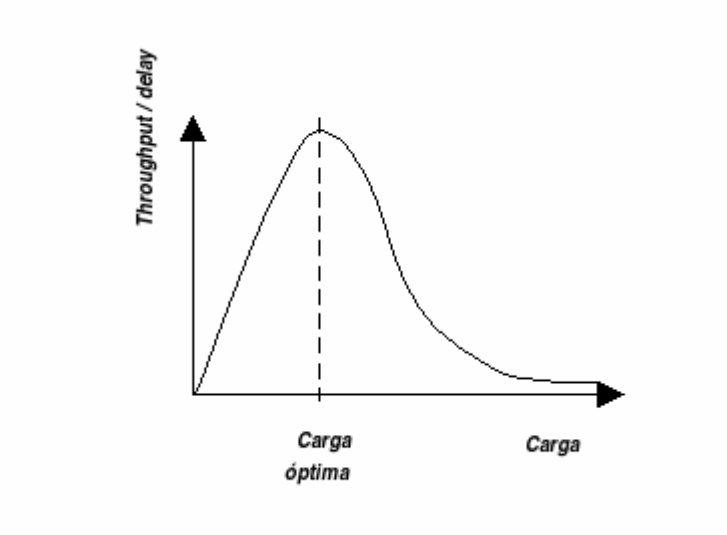


Figura 2.5: Potencia en función de la carga

Una conexión de SCTP se llamada *asociación* y provee un novedoso servicio de *Multihoming*, que le permite a los host de una asociación tener múltiples direcciones IP, y *multistreaming*, que permite el envío de flujos de datos independientes. También provee la importante característica de *four-way-handshake* para establecer una asociación, que lo hace resistente a ataques de negación de servicios como ser *Denial Of Service* (DoS). SCTP provee un servicio de envío de datos orientado a mensajes y además maneja múltiples flujos.

La figura 2.6 muestra el formato de un paquete SCTP.

Un paquete SCTP se compone de una cabecera común y *Chunk*. Múltiples Chunk pueden ser multiplexados en un mismo paquete, hasta alcanzar el límite del *path maximum transfer unit* (PMTU). El PMTU es una estimación de la unidad máxima de transmisión que el emisor SCTP mantiene para cada una de las direcciones destino presentes en una asociación. Un Chunk puede tener datos de control o datos de usuario.

La cabecera común consiste de 12 bytes. Para identificar una asociación, SCTP utiliza el mismo concepto de *puerto* que TCP y UDP. Para detectar errores de transmisión, cada paquete de SCTP tiene 32 bits de *checksum* aplicando el algoritmo *Adler-32*, que es más robusto que el de 16 bits de checksum de TCP o UDP. Aquellos paquetes de SCTP que fallen en el chequeo de checksum son descartados. La cabecera común también contiene un valor 32 bits llamado *tag de verificación*. Éste es específico de una asociación, y son intercambiados entre los host en el momento de creación de una asociación. Existen 2 tag de verificación por asociación.

Cada Chunk empieza con un campo de tipo, que es utilizado para distinguir entre Chunk de datos y de control, seguido por un campo con banderas específicas del Chunk y un campo con el largo del Chunk, ya que el largo de los Chunk son variables. Finalmente, el campo valor contiene los datos actuales del Chunk de la capa de aplicación. SCTP posee la flexibilidad de concatenar diversos tipos de Chunk en un mismo paquete.

Existen diferentes tipos Chunk de control, como ser acuse de recibo de datos (SAck), reporte de errores, fin y comienzo de asociación. SCTP provee un diseño flexible que permite agregar nuevos Chunk de control.

Una de las características principales de SCTP es Multihoming. Esto permite a los host utilizar más de una dirección IP en una asociación. La figura 2.7, muestra un host Multihoming

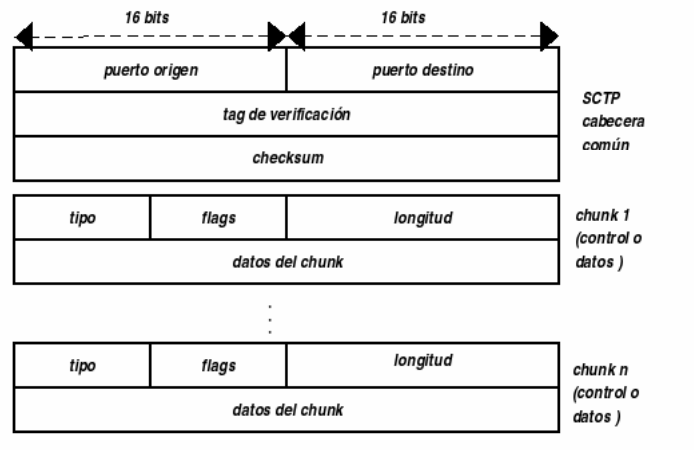


Figura 2.6: Formato de un paquete SCTP

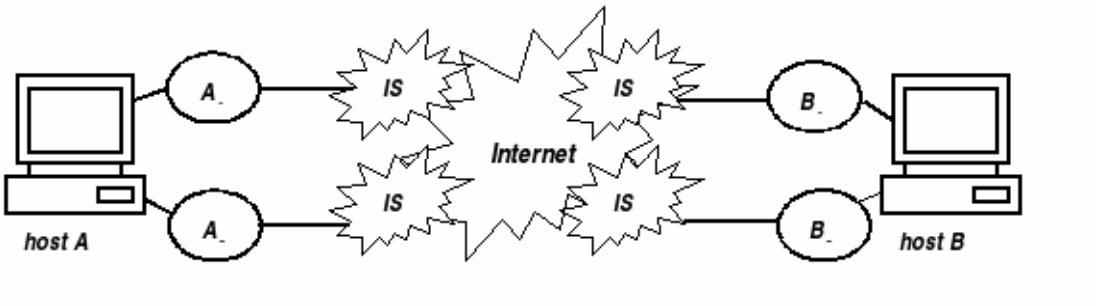


Figura 2.7: Host Multihomed

que es accesible utilizando múltiples direcciones IP, A1 y A2 representan dos direcciones IP del host A. B1 y B2 representan dos direcciones IP para el host B.

Si una de sus direcciones IP falla (posiblemente por una falla en el enlace o en la interfaz) el receptor puede todavía recibir datos utilizando una interfaz alternativa en el origen. Para tomar beneficio de esta redundancia en el nivel de red, SCTP soporta Multihoming en el nivel de transporte.

Cada host elige una dirección destino primaria para el envío de nuevos datos en una transmisión normal. Un host retransmite datos a una dirección alternativa bajo la asunción de que el camino alternativo incrementa la probabilidad de que los Chunk alcancen el destino. El emisor monitorea por medio de Chunk de control de tipo Hearbeat, si cada una de las direcciones IP destino de la asociación es alcanzable.

La tabla 2.1 presenta un breve resumen que compara las características de SCTP frente a TCP [37].

2.4.1. Definiciones

Para el mejor entendimiento de las diferentes técnicas, introduciremos una serie de términos comunes usados por los diferentes algoritmos y mejoras.

CC Para notar Control de Congestión.

Servicios o Características	SCTP	TCP
Transmisión de datos full duplex	Si	Si
Orientado a conexión	Si	Si
Transmisión de datos confiables	Si	Si
Transmisión de datos parcialmente confiables	Opcional	No
Envío de datos fuera de orden	Si	No
Control de flujo y congestión	Si	Si
Soporte para notificación explícita de congestión (ECN)	Si	Si
Ack selectivos	Si	Opcional
Preservación de límite de mensajes	Si	No
Soporte de *PMTU Discovery*	Si	Si
Fragmentación de datos de aplicación	Si	Si
Multistreaming	Si	No
Protección contra ataques de *SYN flooding*	Si	No
Multihoming	Si	No

Cuadro 2.1: Característica de SCTP y TCP

Segmento Es un paquete de cualquier cantidad de datos (bytes) con o sin asentimiento de TCP.

Ack Confirmación o asentimiento de la recepción de un segmento.

Sender Maximum Segment Size (Smss) Es el máximo tamaño que el emisor de un segmento puede enviar en un momento dado. El valor puede estar acotado en el MTU (máxima unidad de transmisión) del enlace de salida desde el host, el resultado del Path MTU Discovery Algorithm [26], Rmss u otros factores. Este tamaño no incluye los encabezados TCP e IP.

Receiver Maximum Segment Size (Rmss) Es el tamaño del mayor segmento que el receptor esta dispuesto a aceptar. Es el valor especificado en la opción MSS cuando se establece la conexión, o en su defecto 536 bytes [6].

Segmento completo Un segmento que contiene la máxima cantidad de bytes de datos permitidos. Por ejemplo: un segmento conteniendo Smss bytes de datos.

FlightSize Cantidad de bytes enviados por un emisor que no fueron asentidos y todavía no se necesita reenviarlos por Timeout.

Receiver Window (Rwnd) La cantidad de bytes permitidos por el receptor, informado en el campo Receiver Window, del encabezamiento de TCP. Es una forma de controlar el flujo entre los dos host que intervienen en una conexión TCP.

Congestion Window (CWnd) Una variable de TCP que limita la cantidad de datos que TCP puede enviar. En cualquier momento, un emisor TCP no debe enviar datos con un número de secuencia más alto que la suma del más alto número de secuencia asentido y el mínimo del CWnd y el Rwnd.

SSThresh Umbral que controla hasta cuando se puede utilizar el algoritmo Slow Start. Una vez sobrespasado el valor de esta variable, se debe utilizar el algoritmo Congestion Avoidance.

Initial Window (IW) El tamaño inicial de la ventana de congestión (CWnd) del emisor, despues del *three-way handshake* (intercambio de tres vías), mecanismo usado para establecer una conexión TCP.

Lost Window (LW) Es el tamaño de CWnd después de que el emisor TCP detectó la perdida de algún paquete por medio del timer de retransmisión.

Restart Window (RW) Es el tamaño de CWnd luego de que TCP vuelve a transmitir después de un largo periodo de inactividad.

Round Trip Time (RTT) Es el tiempo transcurrido desde la transmisión de un paquete hasta la recepción de su Ack correspondiente. No se toma en cuenta cuando incurrió en retransmisiones.

Retransmission Time Out (RTO) Es el tiempo que el emisor esperará por el Ack correspondiente antes de retransmitir datos. El timer de retransmisión debe configurarse con este valor de RTO para algún segmento en particular.

RFC 2581, Slow Start y Congestion Avoidance

Estos algoritmos son utilizados por el emisor en forma obligatoria para controlar la tasa de transmisión en función de la congestión presente en la red. Estos controlan el valor de dos variables: CWnd y SSThresh.

Cuando recién comienza el envío de segmentos, post-establecimiento de la conexión TCP, el emisor no conoce acerca de las condiciones presentes en la red, por lo que debiera evitar enviar una gran cantidad de datos iniciales inapropiadamente y provocar entre otros problemas, congestión en una red como Internet. Slow Start entonces se encargara sobre el comienzo de la transmisión de evitar esto o también cuando se produce pérdida de segmentos detectados por RTO.

Recordemos que un emisor TCP puede enviar datos cada vez hasta el mínimo entre CWnd y Rwnd, menos el FlightSize, la cantidad de bytes sin asentir.

Por empezar, se debe igualar CWnd con IW, que debe ser menor o igual a $2 * Smss$ bytes. SSThresh puede establecerse alto (por ejemplo igual a Rwnd), pero como se verá más adelante, este será reducido en caso de congestión. Slow Start se utilizara siempre que $CWnd < SSThresh$, mientras que Congestion Avoidance se activará cuando $CWnd > SSThresh$, dejando que el emisor elija el algoritmo en caso de igualdad.

Durante Slow Start, TCP incrementará CWnd hasta Smss bytes por cada Ack recibido acerca de nuevos datos, es decir no sobre datos ya asentidos en otro Ack anterior, permitiendo un crecimiento multiplicativo. Como ejemplo si se envía 1KB de datos y suponiendo que CWnd estaba en 1KB, la vez siguiente se podrá enviar 2 KB, pero al ser asentidos estos 2 KB de datos, luego se podrá enviar 4KB y así sucesivamente, hasta alcanzar SSThresh.

Durante Congestion Avoidance, CWnd se incrementará por la siguiente formula: $CWnd+ = Smss * Smss/CWnd$, la cual incrementa a CWnd en forma agresiva si el receptor responde con Ack por cada segmento recibido y no tanto si demora la respuesta del Ack, para hacer uso del *piggybacking* permitido en TCP. Este incremento puede dar lugar a fracciones, por lo cual se redondea para arriba y también se acerca al principio de incrementar CWnd por un segmento completo por vez, linealmente. Es notorio también que el incremento también puede ser tan mínimo como un byte por vez. Y hay que aclarar que solo se usará este algoritmo hasta que se produzca la perdida de un segmento por RTO. Cuando esto se produce, se debe modificar SSThresh al máximo entre $FlightSize/2$ y $2 * Smss$, CWnd se debe igualar a LW, que no debiera

ser mayor a un Segmento Completo y se debe volver a usar Slow Start hasta alcanzar el nuevo valor de SSThresh.

Comentarios adicionales en la RFC 2581

[1] aclara que en periodos ociosos, el emisor debe ajustar el CWnd a RW, que puede ser igual a IW o al mínimo($CWnd, IW$). En función de protocolos (como HTTP persistente) que realizan *keepalives* en conexiones TCP, debe considerarse un período ocioso como aquel en donde no se transmitió datos nuevos por más de un RTO.

Otras consideraciones hablan de los Ack demorados, que hacen un uso efectivo del *piggybacking* y que dejan de proveer un *reloj* de red, a través de los Ack normales enviados con el arribo al receptor de un segmento nuevo. Estos no deberían ser demorados más allá de la recepción de un segundo Segmento Completo o de 500ms del arribo del primer segmento no asentido. Como a veces el concepto de Segmento Completo puede diferir entre el emisor y el receptor y pudiendo sobrepasar el límite establecido de dos Segmentos Completos, el receptor debiera cambiar a dos segmentos como umbral para generar el Ack, más allá del tamaño de los mismos. Tampoco debiera demorar Ack de segmentos fuera de orden, para poder activar mecanismos de Fast Retransmit del lado del emisor.

2.4.2. SCTP y la congestión

Durante una transmisión de datos, se utilizan principalmente dos tipos de Chunk:

- *Data Chunk*, usado por el emisor, que contiene los datos.
- *SAck Chunk*, usado por el receptor, que contiene el acuse de la recepción de los Data Chunk.

La figura 2.8 muestra más en detalle el formato de un paquete SCTP.

Todos los Data Chunk son identificados por un número secuencial llamado *Transmission Sequence Number* (TSN). Entonces, dos Data Chunk consecutivos tendrán TSN consecutivos. Una limitación de SCTP es que una vez enviado un Data Chunk, es imposible dividir el Data Chunk en Data Chunk más pequeños en una retransmisión. Esto es debido a que los SAck Chunk notifican la recepción de los Data Chunk utilizando su TSN.

Cuando un Data Chunk arriba al receptor, éste debe enviar un SAck Chunk reportando la recepción del mismo. El *Cumulative TSN Ack* es utilizado para informar el último TSN recibido antes de encontrar un corte en la secuencia de TSN recibidos. El Ack de SCTP acusa la recepción de todos los TSN hasta el valor del mismo.

A continuación del campo Cumulative TSN Ack, el SAck Chunk contiene los campos *Gap Ack Block*. Estos son utilizados para *informar la recepción de secuencias de TSN recibidas posteriores al corte en la secuencia de TSN recibidos*. Vale notar que los TSN del Gap Ack Block son todo mayores al Cumulative TSN Ack. Entonces, el emisor no necesita retransmitir los TSN informados en el Gap Ack Block.

El SAck Chunk también contiene el *Duplicate TSN*, una lista de TSN duplicados. Por medio de esta lista, el receptor informa al emisor sobre la recepción de TSN que llegaron más de una vez.

SCTP es un protocolo de *ventana deslizante* que regula la tasa de transmisión de datos. Para controlar la tasa de transmisión se basa en el espacio de buffer disponible, y utilizando el valor del *Advertised Receiver Window Credit*. Éste es informado por el receptor al momento de iniciar una asociación y en cada SAck Chunk transmitido.

Para realizar control de congestión, SCTP utiliza los mismos mecanismos de TCP: *Slow Start* y *Congestion Avoidance*. Estos han sido tomados de TCP no solo porque han demostrado tener un

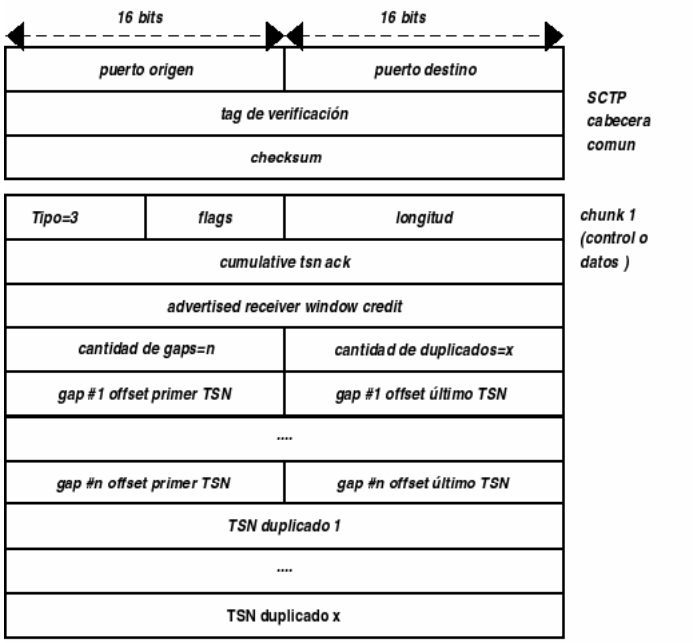


Figura 2.8: Formato de un paquete SCTP en detalle

funcionamiento aceptable, sino también porque se ha teniendo en cuenta la posible convivencia de ambos protocolos en la red. Esto último se debe a que, si SCTP utilizase algoritmos más sensitivos a congestión que TCP, éste se vería en situación de desventaja al momento de utilizar los recursos de la red.

SCTP realiza Slow Start cuando el *Congestion Window* (CWnd) es menor o igual que el *Slow Star Threshold* (SSThresh). Durante un Slow Start, el CWnd es incrementado únicamente si se cumplen las siguientes condiciones: *que el Ack recibido asiente nuevos datos, y que todo el CWnd esté en uso antes de que el Ack arribe.* De ser así, el CWnd es incrementado por el mínimo entre la cantidad de datos confirmados y el *Maximum Transfer Unit* (MTU)[13].

Las variables de CWnd y SSThresh deben mantenerse por cada una de las direcciones destino presentes en una asociación, ya que los algoritmos de control de congestión se ejecutan en cada una de las direcciones. Esto se debe a que diferentes direcciones destinos generalmente implican diferentes caminos, con diferentes estados de congestión y diferentes *Round Trip Time* (RTT).

Al ocurrir un Timeout, el emisor retransmite todos los Data Chunk que no hayan sido confirmados. Todos los Data Chunk que fueron confirmados por medio de Gap Ack Block no serán retransmitidos, evitando la retransmisión de Data Chunk que llegaron fuera de orden.

2.4.3. Medición del *Round Trip Time*

El emisor necesita mantener una estimación del *Round Trip Time* (RTT) de la red con el fin de poder calcular el *Timeout de Retransmisión* (RTO). El cálculo del RTO debe realizarse por

[13]En TCP se utiliza el Sender Maximum Segment Size (Smss) que es el máximo tamaño que el emisor de un segmento puede enviar en un momento dado. Este valor puede estar acotado en el MTU (máxima unidad de transmisión) del enlace de salida desde el host, el resultado del Path MTU Discovery Algorithm, Rmss u otros factores. Este tamaño no incluye los encabezados TCP e IP.

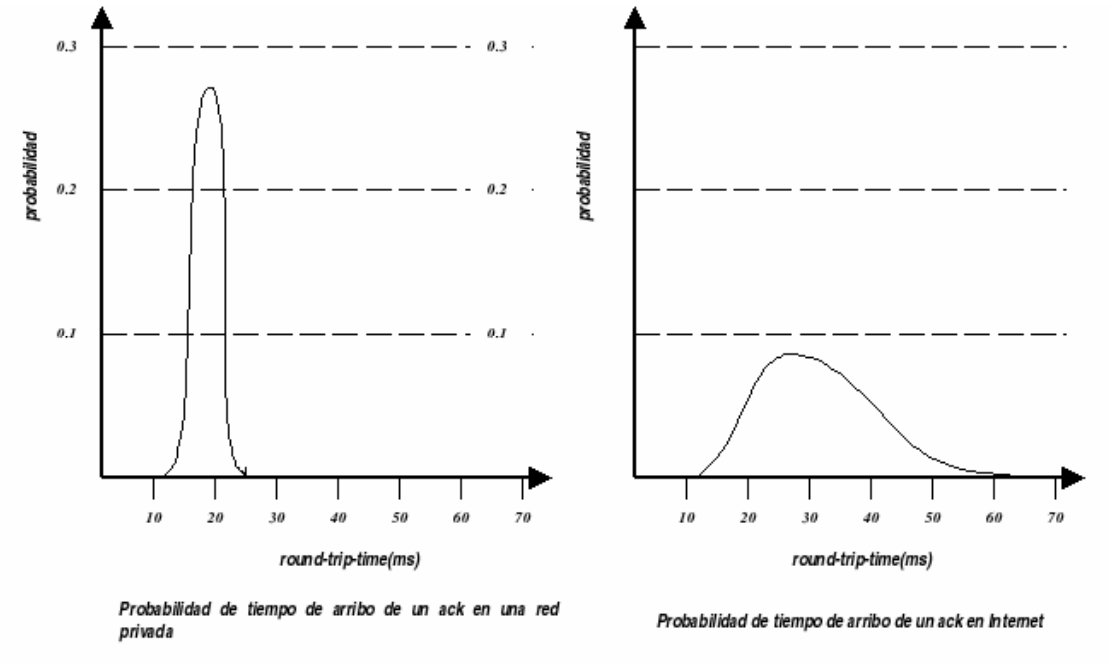


Figura 2.9: Probabilidad de densidad de tiempos de arribos de Ack

cada destino que interviene en la asociación, ya que estos pueden tener distintos valores de RTT. El algoritmo utilizado es el mismo que utiliza TCP, que se detalla a continuación:

- El *Round Trip Time* es tiempo que le toma a un emisor enviar un paquete llegar al receptor y esperar hasta recibir el correspondiente Ack. La importancia de conocer el RTT radica en la necesidad de tener alguna medida que sirva para actualizar el valor del RTO, que es usado por los timers del protocolo. El cálculo del RTT se basa en la medición del tiempo que transcurre desde que un paquete es enviado hasta que su Ack es recibido. Hay algunos puntos que deben ser tenidos en cuenta:

  - Si un paquete es retransmitido, no es posible saber si la transmisión original o la retransmisión fue la que generó el envío del Ack.
  - Debido al uso de Ack demorados, los paquetes recibidos no son confirmados inmediatamente, y esto afecta la precisión en el cálculo del RTT. Es precisamente la variación en el valor del RTT lo que hace el cálculo del RTO más difícil. Si el RTO es pequeño, existe la posibilidad de realizar retransmisiones cuando no es necesario. Si el RTO es grande, se puede demorar demasiado la retransmisión de un paquete perdido. *El problema consiste en encontrar el RTO correcto, teniendo en cuenta que su valor depende del estado de la red y sus cambios.*

  En una red privada no hay mayor inconveniente en el cálculo del RTO, ver figura 2.9. Sin embargo, en Internet, el valor del RTT puede variar rápidamente, dificultando la elección de un buen RTO. Por esta razón surge la necesidad de utilizar algoritmos más elaborados para el cálculo del mismo.

  Las implementaciones de TCP calculan el *Smoothed Round Trip Time* (SRTT), según lo definido en [1], como:

$$SRTT = \alpha SRTT + (1 - \alpha)RTT$$

Las primeras implementaciones calculaban el $RTO = \beta SRTT$, donde $\alpha = 7/8$ y $\beta = 2$. Van Jacobson mostró que el valor fijo de β no era capaz de responder adecuadamente cuando la varianza cambiaba bruscamente, que solo era capaz de adaptarse con cargas de a lo sumo 30 %. Luego se propuso el uso del *desvío medio de los valores del RTT* como una forma fácil y mejorada de aproximar el desvío estándar. El nuevo algoritmo agrega el cálculo del *Round Trip Time Variation* (RTTVAR), previo al del SRTT:

$$RTTVAR = (1 - \beta)RTTVAR + \beta|SRTT - RTT|$$

Finalmente, la estimación del RTO queda como:

$$RTO = SRTT + 4RTTVAR$$

2.4.4. *Fast Retransmit* y *Fast Recovery*

La idea de los algoritmos *Fast Retransmit* y *Fast Recovery* consiste en mejorar en rendimiento en los enlaces cuyo producto entre espera y ancho de banda es alto, hay congestión o perdida de paquetes.

A continuación se describen los algoritmos de Fast Retransmit y Fast Recovery, como son descrito en TCP.

Fast Retransmit y Fast Recovery Fast Retransmit es un algoritmo heurístico que genera la retransmisión de un paquete perdido antes que lo haga el mecanismo regular de Timeout. La idea de Fast Retransmit es la siguiente: cada vez que un paquete arriba al receptor, éste debe responder con un Ack, incluso si el número de secuencia ya ha sido confirmado. Luego, cuando un paquete arriba fuera de orden (esto es, el emisor no puede confirmar el paquete porque paquetes anteriores no han sido recibidos) el receptor reenvía el mismo Ack. Esta segunda transmisión del mismo Ack se denomina Ack duplicado. La idea detrás de este Ack duplicado es poder notificar al emisor, ya que le permite utilizar el algoritmo de Fast Retransmit.

Cuando el emisor recibe un Ack duplicado, infiere que el receptor ha recibido un paquete fuera de orden, lo que sugiere que un paquete anterior pudo haberse perdido. Dado que el emisor no puede determinar si un Ack duplicado fue causado por un paquete que se perdió o porque ocurrió un reordenamiento de los mismos, debe esperar que una cantidad pequeña de Ack duplicados sean recibidos. Se asume que si hay reordenamiento de paquetes en la red, habrá sólo uno o dos Ack duplicados antes de que el paquete reordenado sea procesado, lo cual generará un nuevo Ack. Esta cantidad de reportes de duplicados que se deben recibir para entrar en Fast Retransmit es el *DupThresh*[14]. Luego, el emisor realiza una retransmisión de lo que se presume es un paquete perdido, sin esperar que el RTO expire.

La figura 2.10 muestra como los Ack duplicados llevan a un Fast Retransmit. En este ejemplo, el destino recibe los paquetes 1 y 2, pero el paquete 3 se pierde. Entonces el receptor enviará un Ack duplicado del paquete 2 cuando llegue el 4, nuevamente cuando llegue el 5 y así sucesivamente. Cuando el emisor recibe el tercer Ack duplicado del paquete

[14] En TCP el DupThresh tiene un valor de 3, mientras que en SCTP es 4.

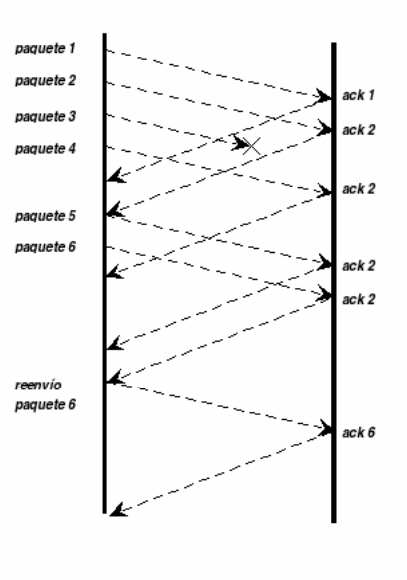


Figura 2.10: Fast Retransmit basado en Ack duplicados

2, por la recepción de los paquetes 4, 5 y 6, retransmite el paquete 3. Notar que cuando la retransmisión del paquete 3 arriba al destino, el receptor envía un Ack Acumulativo por todos los paquetes hasta el 6 inclusive. Vale nota que en este ejemplo del DupThresh vale 3.

Una vez que el mecanismo de Fast Retransmit manejó la recuperación frente a una pérdida, en vez de actualizar el CWnd a un paquete para luego comenzar con Slow Start, es posible utilizar los Ack que se encuentran en tránsito para sincronizar el envío de paquetes. Este mecanismo llamado *Fast Recovery*, logra eliminar la fase de Slow Start que ocurre entre que Fast Retransmit detecta la pérdida de un paquete y el comienzo de Congestion Avoidance. La razón por la cual no realizar Slow Start, es que la recepción de Ack duplicados informa al receptor más que la simple pérdida de paquetes. Puesto que el receptor puede únicamente generar un Ack duplicado cuando algún paquete es recibido, ese paquete ha dejado la red y se encuentra en los buffer del receptor.

Los algoritmos de Fast Retransmit y Fast Recovery son implementados conjuntamente de la siguiente manera:

- Cuando se recibió el tercer Ack duplicado, se actualiza el SSThresh al máximo entre $FlightSize/2$ y $2 * Smss$, pero no menos que dos paquetes. Se retransmite el paquete perdido y se actualiza CWnd al nuevo valor de SSThresh más $3 * Smss$, indicando con esto los tres segmentos fuera de orden que residen en el buffer del receptor y que ya dejaron la red posiblemente, dando lugar a los Ack duplicados.

  Cada vez que algún otro Ack duplicado llega, se incrementa CWnd por Smss, bajo la misma premisa de Fast Recovery, para permitir el uso de recursos de red, aún cuando supuestamente los segmentos residen en buffer en el receptor y todavía no fueron entregados al nivel de aplicación.

  Transmitir un paquete, siempre y cuando el nuevo valor de CWnd y Rwnd así lo permita.

Cuando llega el siguiente Ack que confirma la recepción de nuevos datos (significa que no se perdió nada), se actualiza CWnd al valor del SSThresh y se continua en Congestion Avoidance, aún cuando antes de la situación presentada se estaba en Slow Start. Si no arriba un Ack mencionando al nuevo segmento, se sigue en Fast Recovery.

Aunque estos algoritmo son los más utilizados en la actualidad [36], han demostrado tener serias deficiencias cuando múltiples paquetes se pierden dentro de una misma ventana de datos.

De tres Ack duplicados, el emisor infiere una pérdida de paquete y retransmite. Luego, el emisor podría recibir Ack duplicados adicionales, a medida que el receptor informa la recepción de los paquetes que ya estaban en tránsito cuando el emisor entró en Fast Retransmit.

En el caso de múltiples pérdidas de paquetes dentro de una ventana de datos, la nueva información para el emisor aparece cuando este recibe el Ack del paquete retransmitido por Fast Retransmit. Si hubo una simple pérdida entonces el Ack para este paquete informará la recepción de todos los paquetes transmitidos antes de entrar en Fast Retransmit. Sin embargo, frente a múltiples pérdidas, el Ack para el paquete retransmitido informará la recepción de algunos, pero no todos los paquetes transmitidos antes de Fast Retransmit. Llamaremos a esto un *Ack parcial*.

Frente a esta situación, una mejora es la retransmisión inmediata del paquete siguiente al Ack parcial, logrando realizar múltiples retransmisiones dentro de un RTT. Sin esta modificación, múltiples perdidas dentro de una ventana de datos no pueden recuperarse por medio de Fast Recovery y desembocan en Timeout.

El estado de Fast Recovery se mantendrá hasta la recepción del número de secuencia más alto trasmitido antes de entrar a Fast Retransmit. Para esto, surge la necesidad de almacenar este número de secuencia en una variable, que recibe el nombre de *Recover*.

Se puede extender el estado de Fast Recovery hasta que todos los paquetes trasmitidos dentro de una ventana de datos sean confirmados, y no hasta recibir un Ack confirmando nuevos datos, para evitar múltiples reducciones del CWnd. Esta mejora se encuentra actualmente en etapa experimental, sin embargo ha mostrado un mejor desempeño cuando se utiliza ventanas de gran tamaño, como ser el caso de los enlaces satelitales [10].

Estas ideas se aplican también a SCTP.

El mecanismo que utiliza el receptor para informar la recepción de paquetes fuera de orden es a través de los Gap Ack Block. Estos bloques forman parte del SAck Chunk y el receptor está obligado a notificarlos en caso de poseer en sus buffer TSN no contiguos. Es decir, siempre que el receptor posea "huecos"debido a la recepción de Data Chunk fuera de orden, se ve obligado a informar los mismos en cada SAck Chunk que envía.

Cuando un emisor recibe un SAck Chunk que indica que algún TSN se perdido, éste debe esperar por tres reportes más que informen la pérdida del mismo, antes de tomar alguna acción al respeto.

El conteo de los reportes de TSN faltantes se realiza utilizando el algoritmo denominado *Highest TSN Newly Acked* HTNA [39]. Este algoritmo intenta manejar los paquetes extraviados a causa de reordenamiento de la siguiente manera. Por cada Ack recibido, el TSN más alto que esta siendo confirmado en un Gap Ack Block por primera vez, se convierte en el nuevo marco de referencia para incrementar los reportes faltantes. Por cada uno, solo pueden ser incrementados los reportes de faltante de los TSN anteriores a este punto de referencia. Si el emisor se encuentra

en Fast Recovery y un SAck Chunk avanza el Cumulative TSN Ack Point, se incrementa el reporte de pérdida para todos los TSN reportados como faltantes en el SAck Chunk.

A continuación se detalla brevemente el algoritmo HTNA:

1. Cada vez que un nuevo Data Chunk es transmitido, hay que poner el contador de reporte de faltantes para ese TSN, desde ahora TSN_Missing_Report, en cero.

2. Cada vez que un SAck Chunk reporta Data Chunk extraviados, hay que almacenar el TSN más alto que está siendo confirmado en un Gap Ack Block; llamaremos a este valor HighestTSNInSACK. Cuando un emisor recibe un SAck Chunk que confirma nuevos datos, todos los Data Chunk con TSN menor son calificados como extraviados.

3. Examinar todos los TSN que no fueron confirmados. Si el TSN de un Data Chunk no confirmado es menor que HighestTSNInSACK, para ese Data Chunk hay que incrementar la variable TSN_Missing_Report si no fue retransmitido por Fast Retransmit ni marcado para Fast Retransmit.

Cuando un emisor recibe cuatro reportes consecutivos de TSN perdidos, éste debe realizar los siguientes pasos:

1. Marcar los Data Chunk que hayan sido reportados como faltantes cuatro veces o más para retransmisión.

2. En caso de no estar en Fast Recovery, ajustar el SSThresh y el CWnd de las dirección(es) de destino para las cuales los Data Chunk fueron enviados. Las acciones a tomar son:

$$SSThresh = maximo(CWnd/2, 2MTU) CWnd = SSThresh$$

3. Determinar cuantos de los Data Chunk más viejos a retransmitir entran en un solo paquete, sujetos a la restricción del PMTU de la dirección a la cual el paquete será enviado; llamemos a éste valor K. Retransmitir esos K Data Chunk en un sólo paquete. Cuando Fast Retransmit está por ser ejecutado, el emisor debe ignorar el valor de CWnd y no demorar la retransmisión para éste paquete.

4. Reiniciar el Timer de retransmisión solamente si el último SAck Chunk confirma el recibo del menor TSN pendiente enviado a ese destino, o si el emisor está retransmitiendo el primer Data Chunk pendiente enviado a esa dirección.

5. Marcar los Data Chunk retransmitidos como inelegibles para posteriores retransmisiones por Fast Retransmit. Aquellos TSN marcados para retransmisión que no entraron en el paquete conteniendo otros K TSN, son también marcados como inelegibles para futuras retransmisiones por Fast Retransmit. Sin embargo, como fueron marcados para retransmisión, ellos serán retransmitidos tan pronto como el CWnd lo permita.

6. En caso de no estar en Fast Recovery, entrar a Fast Recovery y marcar el TSN más alto transmitido dentro de una ventana de datos como el punto de salida de Fast Recovery. Cuando un SAck Chunk confirma el recibo de todos los TSN hasta este punto de salida inclusive, se sale de Fast Recovery. Mientras se esté en Fast Recovery, el SSThresh y el CWnd no deberían cambiar para ningún destino.

2.5. Librería SCTP

Existen varias implementaciones del protocolo SCTP[15] que se tomaron en cuenta a la hora de elegir una en la cual trabajar. Se consideraron varios puntos en la elección, como ser facilidad de modificar el código fuente, documentación, facilidad para las pruebas, estabilidad, entre otros. A sabiendas de esto, aparecieron dos importantes candidatos, una librería de SCTP y el módulo del kernel de Linux.

La librería de SCTP [49] soporta la versión actual del protocolo SCTP. Se puede ejecutar sobre Linux, FreeBSD y Mac OS X; y soporta IPv4 e IPv6. Esta librería está en activo desarrollo y fue inicialmente creada en conjunto por Siemens, el Grupo de Tecnología de Redes de Computadora de la Universidad de Essen y la Universidad de Ciencias Aplicadas de Münster.

El módulo del kernel de Linux forma parte del actual kernel de Linux como un módulo experimental. Soporta la versión actual del protocolo SCTP. Solo se corre con Linux y soporta IPv4 e IPv6.

La librería SCTP se ejecuta en el espacio de usuario[16]. LKSCTP se ejecuta dentro del kernel de Linux. Esto hace que LKSCTP tenga mejor rendimiento que la librería de SCTP debido a los diferentes niveles que ejecución respectivos.

La librería de SCTP se utiliza programando a través de la API (Application Program Interface), y luego linkeditando la misma con el programa final. LKSCTP se utiliza se utiliza programando a través la API de sockets propio de Linux, indicando que se utilizará protocolo SCTP (132), a nivel IP, al crear un socket. Esto será de utilidad si se desea cambiar de TCP a SCTP en algún programa, ya que el cambio supuestamente debería ser mínimo. Esta practicidad de cambio de protocolo, sin embargo, presenta la desventaja de llevar esta tecnología a otros sistemas operativos, a diferencia de la librería de SCTP que sí puede llevar a otros sistemas operativos.

La modificación y, consecuentemente, las pruebas necesarias para implementar SCTP-RR fue un factor decisivo en la elección de SCTP. Dado que se LKSCTP es un modulo del kernel de Linux, el compilar el código fuente, instalación del módulo modificado y depuración del mismo, no fue un tema menor que se tuvo en cuenta, ya que cualquier falla del módulo producirá que el kernel falle generando un *coredump*[17] y luego la detención total del sistema. Distintamente, la librería de SCTP, no sufre de estos temas sino que los facilita. Al correr en el espacio de usuario, cualquier falla de la misma generará un *coredump* pero el sistema seguirá funcionando. Además que la depuración (debugging) de un programa que corre a nivel usuario es trivial con relación a la depuración de un módulo del kernel.

Dado este escenario se decidió optar por la librería de SCTP.

A continuación se presenta brevemente la librería de SCTP.

2.5.1. Conceptos generales

La API de la librería fue modelada de acuerdo a la sección 10 del RFC 2960. Además de las funciones de interfase con la Upper Layer Protocol (ULP) y la instancia SCTP, la librería también provee varias funciones de ayuda que pueden ser manejadas a través de funciones *Callback* para ser ejecutadas en cierto momento (basadas en Timer) o abrir y enlazar sockets UDP a un puerto configurable, que puede ser utilizado para comunicación asincrónica entre procesos.

Todas estas funciones pueden ser utilizadas simplemente linkeditando la librería SCTP a una aplicación e incluyendo el archivo *sctp.h*. Como SCTP opera sobre IP, se eligió utilizar un *socket*

[15] En `http://www.sctp.org/` se encuentra una lista de la distintas implementaciones.

[16] Es un espacio de memoria virtual separado de la memoria del kernel disponible para los usuario, este puede bajarse a disco de ser necesario

[17] Un *coredump* es una bajada a disco del estado de la memoria al tiempo que ocurrió una falla en el sistema.

raw para manejar todos los paquetes entrantes que tengan como valor 132 (SCTP) en el byte *id* (byte de identificación) del protocolo IP. Vale notar que se deben tener los derechos de acceso necesarios para poder utilizar este tipo de socket.

Una aplicación que utilice esta librería puede manejar un número grande de asociaciones como también varias instancias de SCTP (que pueden trabajar en diferentes puertos). Los puertos de los paquetes entrantes son chequeados para saber si ya pertenecen a un asociación existente o se debe crear una nueva.

El concepto de la librería es el siguiente: después de registrar la primer instancia SCTP, se abren socket y la aplicación puede registrar eventos basados en Timers, o descriptores de archivo, que se ejecutan asincrónicamente a través de funciones Callback. La aplicación utiliza la lista de funciones Callback a través de la función de registración, y las funciones Callback para los eventos de SCTP son pasados a través de la función *SCTP_ulpCallbacks*[18].

Luego la aplicación puede llamar a dos funciones para procesar los eventos que ocurran:

Función bloqueante*sctp_eventLoop* Con está función la aplicación reaccionará a eventos basados en Timer previamente agendados o a cualquier evento del descriptor de archivo (a través de la ejecución de las funciones Callback registradas). Mientras no ocurra evento alguno la aplicación quedará *dormida* hasta que ocurra algún evento que la despierte. El control del flujo quedará en manos de la librería y la aplicación deberá registrar las funciones Callback para los eventos y Timer apropiados antes de dar el control a la librería (llamando a la función sctp_eventLoop).

Función no bloqueante *sctp_getEvents()* Con esta función la aplicación chequea si un evento de Timer o del descriptor de archivos ocurrió. Si no hay ninguno, el control del flujo retorna inmediatamente a la aplicación. La aplicación debería llamar a está función, de manera recurrente y en cortos períodos de tiempo, para procesar todos los eventos que ocurran.

En la figura 2.11 se puede ver, a groso modo, los módulos que componen la librería y como interactúan entre ellos.

Vale notar que la librería depende del software *glib-1.2*[19] para funciones de lista, tipos de datos portables, etc.

2.6. Emulador

En informática, un emulador es un software que permite ejecutar programas de computadora en una plataforma (arquitectura de hardware o sistema operativo) diferente del cual fueron escritos originalmente. A diferencia de un simulador, que sólo trata de reproducir el comportamiento del programa, un emulador trata de modelar, de manera precisa, el dispositivo que se está emulando.

2.6.1. Emulador de red

La emulación de red es una técnica donde las propiedades de una red existente, proyectada y/o no ideal son simuladas a fin de evaluar el rendimiento, predecir el impacto de un cambio o de otra tecnología de optimización de toma de decisiones

[18] Para más detalles sobre la API consultar el archivo *sctp-api-1.0-1.pdf* que acompaña a la librería

[19] Para mas información sobre el software *glib-1.2* consultar la página del software `http://www.gtk.org/`

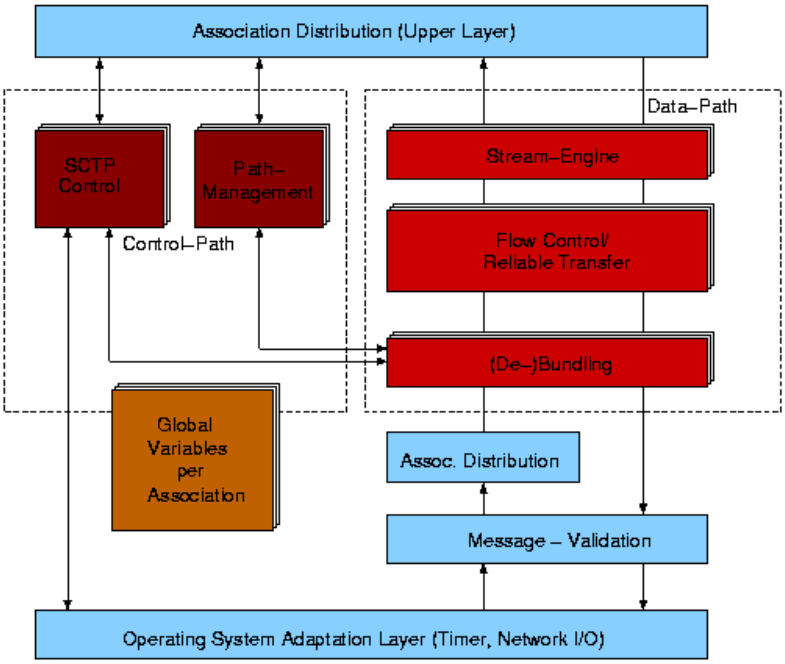


Figura 2.11: Módulos de la librería de SCTP

2.6.2. Métodos de emulación

La emulación de redes se puede lograr por la introducción de un dispositivo, en una LAN, que altera el flujo de paquetes a fin de imitar el comportamiento del trafico de una aplicación en el ambiente siendo emulado. Este dispositivo puede ser una computadora de propósito general ejecutando un software para realizar la emulación de red o un dispositivo dedicado de emulación. El dispositivo incorpora una variedad de atributos de red dentro de su modelo de emulación, incluido el RTT a través de la red (latencia), el ancho de banda disponible, la pérdida de paquetes, la duplicación de paquetes, el reordenamiento de paquetes, y el jitter[20] de la red. Las computadoras se pueden conectar con el ambiente emulado, de manera tal que los usuarios pueden experimentar el desempeño y el comportamiento de las aplicaciones dentro del ambiente emulado. Similarmente, los teléfonos se pueden conectar en el ambiente emulado, a fin de que los usuarios puedan evaluar directamente la calidad de una llamada de VoIP.

2.6.3. Emulación y simulación

A continuación definimos simulación y emulación.

Simulación Un sistema X se dice que simula otro sistema Y cuando los mecanismos o procedimientos internos de X son un modelo matemático (o cualquier otro tipo de modelo) conocido por representar lo mejor posible al actual mecanismo de Y. La simulación se utiliza generalmente cuando se tienen modelos matemáticos de un sistema original (Y) y se

[20] Packet Delay Variation (PDV) es definido en el RFC 3393 como la diferencia de retraso entre los paquetes seleccionados en un flujo, con cualquier pérdida de paquetes ignorados, entre dos extremos.

quiere saber la salida para un determinado conjunto de entradas[21].

Emulación Un sistema X se dice que emula otro sistema Y si el comportamiento de X es exactamente el mismo que el de Y (misma salida para misma entrada en condiciones similares), pero el mecanismo para la salida (a partir de la entrada) es diferente. La emulación es generalmente utilizada cuando no se sabe exactamente el mecanismo interno del sistema original, pero se está familiarizado con el patrón de entrada/salida[22].

[21] Por ejemplo, se puede tener un buen modelo matemático sobre el efecto de la temperatura del agua en la formación de huracanes. Se utiliza este sistema para predecir la naturaleza de los huracanes para las diferentes temperaturas del agua. Cabe señalar que el resultado de la simulación puede ser demostrada. Por ejemplo, el patrón del huracán sugerido por el simulador sobre de la temperatura del agua igual a 50 grados centígrados será difícil de verificar porque es posible que nunca se de tal situación.

[22] Por ejemplo, las redes neuronales se pueden utilizar para emular diferentes sistemas. Las redes neuronales están capacitadas para producir la misma salida de la misma entrada que la del sistema original, aunque el mecanismo o procedimiento para generar los resultados sean muy diferentes.

Capítulo 3

Trabajos Relacionados

A continuación se presenta un resumen sobre los trabajos existentes para SCTP en el área de detección y restauración frente a retransmisiones espurias. Vale aclarar que existen muchos trabajos similares para TCP, pero estos no tienen una implementación directa sobre SCTP debido a las diferencias que poseen ambos protocolos. Se presenta así mismo un resumen de trabajos existentes para emular distintas condiciones de red.

3.1. Mejoras a SCTP

En [13] se presenta una mejora al esquema de control de congestión de SCTP sobre redes con alta latencia y múltiples perdidas de paquetes, condiciones que atentan contra el óptimo rendimiento de SCTP. Una posible solución sería implementar *Scalable TCP* [21] en SCTP.

La implementación de esta posible mejora puede realizarse directamente sobre el actual código de la librería de SCTP. Sin embargo, hay que analizar el impacto que tiene dicha modificación sobre las modificaciones propuestas por SCTP-RR. Dado que ambas propuestas tienen impacto sobre la ventana de congestión CWnd, es necesario una análisis más profundo del tema, el cual no está contemplado en el presenta trabajo.

El trabajo [22] ataca los mismos problemas que SCTP-RR, modificando los datos de la cabecera de SCTP. Propone una extensión de Timestamp para SCTP, con el fin de lograr una versión similar a Eifel[1] [25]. Existe una variación del algoritmo de detección de Eifel que utiliza los bits reservados de la cabecera de TCP, en vez de la extensión de Timestamp. Esta introduce la idea de un bit de retransmisión para indicar retransmisiones y Ack de retransmisiones. El algoritmo de detección es el mismo que para la propuesta original de Eifel. Una ventaja de Eifel frente a los mecanismos de detección basados en los reportes de duplicados, es que puede detectar retransmisiones espurias más rápidamente. Esto se debe a que con solo la recepción de un Ack se puede desambiguar si corresponde a la transmisión original o la retransmisión. El uso de reporte de duplicados implica la espera de al menos un RTT más, pues debe esperar no sólo la recepción del Ack, sino también la recepción del reporte de duplicado. Sin embargo, la desventaja es la sobrecarga que corresponde a los 12 bytes de la opción de Timestamp que debe ser incluida en cada paquete transmitido.

Un tercer esquema es el presentado en [34], llamado *F-RTO*, que modifica ligeramente el comportamiento del emisor SCTP inmediatamente después de un Timeout, y luego observa el

[1] Es un algoritmo que mejora el esquema de recupero de errores, eliminando la ambigüedad de retransmisiones espurias.

patrón de los Ack que recibe. En base a este patrón infiere si la retransmisión fue innecesaria. La ventaja de este algoritmo es que solo necesita ser implementado en el emisor SCTP y no introduce ninguna sobrecarga. Sin embargo, este algoritmo es una heurística que no es robusta a ciertas patologías en la red, como ser el reordenamiento de paquetes claves.

Los dos siguientes trabajos, [16] y [15], atacan los mismo problemas que ataca SCTP-RR, pero cuando se produce un cambio del camino principal de una asociación.

El primer trabajo [16] presenta el algoritmo *Rhein* que ataca al problema retransmisiones innecesarias y el crecimiento "*no amigable a TCP*"de la ventana de congestión del emisor cuando se produce un cambio del camino principal de una asociación. Esto se debe por:

1. El emisor no puede distinguir entre SAck de transmisiones de SAck de retransmisiones.

2. La ventana de congestión del emisor no es consciente del cambio del camino principal de una asociación, y por ende, no puede conocer el reorden producido por el cambio del camino principal de una asociación[2].

El algoritmo Rhein ataca al primer punto previendo el incorrecto crecimiento de la ventana de congestión CWnd habilitando al emisor a distinguir entre SAck de transmisión de los de retransmisión. Esto se logra agregando meta información a los paquetes SCTP para desambiguar los Ack, y así evitar el crecimiento del CWnd debido a retransmisiones espurias.

Aunque el algoritmo Rhein no resuelve el problema de las retransmisiones innecesarias, soluciona el problema del incorrecto crecimiento del CWnd. Este algoritmo es compatible con versiones de SCTP que no incorporen el algoritmo Rhein, pero esto trae una mayor complejidad a SCTP. Otra desventaja es la necesidad de manejar nuevos tipos paquetes SCTP.

El segundo trabajo [15] presenta mejoras atacando al segundo punto, el cual no es solucionado por el algoritmo Rhein. El trabajo presenta el algoritmo *Changeover aware congestion control* (CACC) para solucionar estos puntos. CACC previene el incorrecto crecimiento de la ventana de congestión CWnd eliminando Fast Retransmit espurios. La clave de este algoritmo es que mantiene el estado del emisor para cada camino cuando se produce un cambio de camino. A la llegada de un SAck el emisor incrementa selectivamente el contador de paquetes perdidos para los TSN en la lista de retransmisiones, previniendo incorrectas Fast Retransmit. Este algoritmo tiene 2 variantes:

Conservative CACC (C-CACC) Es la versión conservativa del algoritmo ya que cuando se produce un reordenamiento de paquetes por un cambio de camino observado por el receptor y reportado al emisor, éste último conservativamente elige no incrementar el contador de paquetes perdidos para cualquier TSN dentro del rango de cambios de camino. Al ocurrir una perdida de paquete, el emisor no hará un Fast Retransmit para TSN dentro del rango de cambios de camino. Los TSN dentro del rango de cambios de camino tendrán que esperar por un Timeout de la retransmisión para volver a ser retransmitido. Esta variante no toma en cuenta la posibilidad de múltiples cambios de camino en el emisor.

Esta variante tiene la desventaja de que en caso de perdida de paquetes, un número significante de TSN pueden esperar por un Timeout de la retransmisión para volver a ser retransmitido cuando podrían haber sido retransmitidos por Fast Retransmit.

Split Fast Retransmit CACC (SFR-CACC) Esta variante alivia la limitación de C-CACC, ya que al recibir un SAck, si el emisor puede estimar los TSN que causaron el SAck enviado por el receptor, entonces el emisor puede utilizar el SAck para incrementar el contador de paquetes perdidos dentro del grupo causativo de TSN. En otras palabras, el emisor utiliza

[2]El reorden se produce cuando el nuevo camino elegido es más veloz que el primero.

el SAck selectivamente para aplicar Fast Retransmit a los TSN del grupo causativo. Si más de un grupo es Ack, entonces se aplica Fast Retransmit conservativamente solo a aquellos TSN que están dentro del grupo del actual camino principal.

3.2. Emulación de red

En la actualidad existen muchos simuladores de red, los más populares son NS2, OMNeT++, NetSim, OPNET Modeler y QualNet. Sin embargo, como se explico en 2.6, es deseado que en el presente trabajo se utilicen sólo resultados obtenidos de ambientes reales y controlados, y no son resultados *sintéticos* obtenidos de simulaciones.

Existe una aproximación híbrida, mitad emulador y mitad simulador, llamada *umlsim* [46], que utiliza Linux en modo usuario (UML) para proveer una simulación dirigida por eventos. Esto permite probar la pila del protocolo TCP/IP estándar usando un pseudo-dispositivo que puede simular una red. Aunque es útil para probar el protocolo, *umlsim* está limitado ya que al correr en modo usuario tiene un ambiente virtual que no tiene el misma rendimiento o tiempos del hardware real.

Existen muchos dispositivos dedicados de emulación de red como lo es Simena `http://www.simena.net/`, PacketStorm `http://www.packetstorm.com/`, Gambit `http://www.gambitcomm.com/site/gambit4.shtml` y Anue `http://www.anuesystems.com/`, entre otros. Sin embargo, el análisis de estos productos están fuera del alcance del presente trabajo.

Respecto a emuladores por software, un emulador muy popular es *Dummynet* [44]. Es una parte estándar de FreeBSD [48] y está implementado como una parte del mecanismo de filtrado de paquetes salientes. Con *Dummynet* se puede emular el encolado de paquetes, limitación del ancho de banda, demoras, perdida de paquetes y los efectos de múltiples caminos. También implementa una variante llamada *WF2Q+ - Weighted Fair Queueing*, que es una disciplina de encolado de paquetes. Puede ser utilizado como un Router o puente. Sin embargo, *Dummynet* es autocontenido (viene presentado como una imagen bootable) y por ende difícil de extender (adicionar nuevas funcionalidades).

Otro emulador por software es *NIST Net* [45]. Es una extensión del kernel de Linux que provee complejas demoras, perdidas y otras opciones de emulación. Trabaja a nivel IP, y por ende puede emular condiciones características de una WAN[3]. Sin embargo, *NIST Net* opera sobre los paquetes entrantes antes de que lleguen a la pila del protocolo y utiliza hardware de alta resolución de tiempos. Al igual que *Dummynet*, *NIST Net* hace su propio filtrado y encolado de paquetes.

Un emulador por software muy popular y que ha crecido mucho es *NetEm* [43]. Provee la funcionalidad de emulación de red para realizar pruebas de protocolos, emulando las propiedad de un red WAN. Al ser un modulo del kernel de Linux, es posible agregarle nuevas funcionalidades. Este emulador es que se utilizará en el presente trabajo, y en 5.2 se encuentra una descripción del mismo.

Existe también un emulador por software llamado WANEM `http://wanem.sourceforge.net/`, pero éste está basado en *NetEm*.

[3] Una WAN es una red de comunicaciones que conecta ordenadores dispersos en una amplia área geográfica (conectados por líneas telefónicas u ondas de radio).

Capítulo 4

Algoritmo *SCTP-RR*

A continuación se presenta el algoritmo *SCTP-RR*, que detecta y corrige los problemas causados por el reordenamiento y los cambios abruptos de RTT, con el fin de mejorar el rendimiento de una asociación SCTP.

El mismo está dividido en tres módulos independientes que realizan tareas conceptualmente distintas, y que en conjunto pueden corregir y evitar muchos de los problemas que se pueden presentar en una asociación SCTP.

El resto del presente capítulo describe en detalle cada uno de los módulos del algoritmo y su interacción:

- Módulo de detección de Fast Retransmit y Timeout espurios
- Módulo de Recovery
- Módulo de adaptación dinámica del *DupThresh*[1]

El *módulo de detección* es el encargado de detectar aquellas retransmisiones que hayan sido espurias, ya sea por Fast Retransmit o por Timeout, para que luego el *módulo de Recovery* puede restablecer la tasa de transferencia.

Por último, el *módulo de adaptación dinámica del DupThresh* es el encargado de adaptar este valor, en base al análisis datos históricos, para poder evitar futuras retransmisiones espurias, esquema similar al propuesto en RR-TCP.

4.1. Detección de Fast Retransmit y Timeout espurios

El núcleo del algoritmo de detección se basa en la utilización de los reportes de TSN duplicados contenido dentro de los SAck Chunk.

Cuando en una asociación SCTP ocurre un Timeout, se deben reenviar todos aquellos TSN que no hayan sido confirmados previamente. En cambio, durante Fast Retransmit solo se retransmiten aquellos TSN que hayan sido reportados como faltantes más de tres veces y que no hayan sido retransmitidos por Fast Retransmit.

Un algoritmo de detección basado en la utilización de reporte de duplicados no puede concluir que una retransmisión fue espuria sin antes recibir los reportes de duplicados de cada uno de los paquetes retransmitidos. Sin embargo, la recepción de un reporte de duplicado para algunos de

[1] *DupThresh* es la cantidad de reportes de duplicados que se deben recibir para entrar en Fast Retransmit

ellos, no garantiza que la retransmisión haya sido espuria. Por ende, es necesario mantener un registro de los TSN retransmitidos durante los eventos de retransmisión.

Definimos como *evento de retransmisión* la ocurrencia de un Fast Retransmit o un Timeout.

El algoritmo de detección de SCTP-RR lleva el registro de los TSN retransmitidos utilizando una lista llamada *TSN Retransmision List* (TRL).

Si para cada TSN contenido dentro de la TRL se recibe un reporte de TSN duplicado, se puede concluir que todos los TSN originales llegaron a destino (al igual que sus retransmisiones). Por lo tanto, el evento de retransmisión fue espurio ya que ninguno de los paquetes originales fue perdido en tránsito.

Llegado el momento, la TRL contendrá los TSN de las múltiples retransmisiones que se sucedieron a lo largo de la asociación. El mantener todos los TSN retransmitidos en la TRL, podría causar que el algoritmo no fuera capaz de detectar retransmisiones espurias subsiguientes a una retransmisión que no lo fue, dado que no es posible saber a que retransmisión corresponde un reporte de TSN duplicado. Para solucionar este problema el algoritmo de detección sólo podrá detectar si el último evento que causó la retransmisión fue o no espurio[2].

El algoritmo se basa en dos condiciones:

1. Conocer cuales son los TSN correspondientes a la última retransmisión
2. Mantener en la lista TRL todos aquellos TSN para los cuales aún es posible recibir un reporte de TSN duplicado

Para conocer cuales TSN corresponden a la última retransmisión, alcanza con agregarle a cada uno una marca que indique si el mismo fue agregado en la TRL durante el último evento de retransmisión. Con ésta modificación, al llegar un reporte de TSN duplicado para algún TSN, si existe más de una copia del mismo, se elimina primero alguna copia que no corresponda a la última retransmisión. Cuando la TRL no contenga TSN correspondientes a la última retransmisión, significa que la misma fue espuria.

El mantener en la TRL la lista de los TSN retransmitidos trae el problema de conocer cuan grande puede llegar a ser. Como todo TSN para el cual no se reciba reporte de duplicado no será removido de la lista, ésta puede crecer indefinidamente. Para evitar esto se eliminan de la TRL todos los TSN menores o iguales al Cumulative TSN Ack Point luego de un evento de retransmisión. Dado que estos TSN ya fueron confirmados en forma acumulativa no será necesario retransmitirlos, con lo cual no influirán en la detección del último evento de retransmisión o de los posibles eventos posteriores.

Sin embargo, existe una situación en el cual el algoritmo no detectará algunas retransmisiones espurias. Ésta se presenta cuando ocurre la pérdida de uno o más paquetes retransmitidos y todos los paquetes originales llegan a destino[3].

En SCTP, puede ocurrir un Fast Retransmit estando en *Fast Recovery*[4]. En éste caso, la única diferencia que existe es que no se modificarán las variables de control de congestión. Hasta tanto no se salga del *Fast Recovery*, todos los Fast Retransmit que ocurran serán interpretados como un único evento de retransmisión.

[2] Una alternativa para resolver este problema consiste en vaciar la TRL ante cada nuevo evento de retransmisión. Sin embargo, reportes de duplicados de retransmisiones anteriores pueden ocultar la pérdida de paquetes de la última retransmisión. Esto se debe a la imposibilidad de desambiguar a qué retransmisión corresponde un reporte de TSN duplicado.

[3] Hay que tener en cuenta que aunque la retransmisión fue espuria, la pérdida de paquetes es un mal síntoma del estado de la red. Este comportamiento hace que el algoritmo se comporte de manera conservadora dado que no se realizará ninguna acción agresiva.

[4] El algoritmo de *Fast Recovery* es la ejecución del algoritmo de *Congestion Avoidance* después de un Fast Retransmit sin ejecutar el algoritmo de *Slow Start*.

Para que SCTP-RR funcione correctamente en asociaciones *Multihomed*, se debe ejecutar el algoritmo *por dirección de destino*. Para ello es necesario mantener una TRL por cada destino, ya que los eventos de retransmisión están asociados a un destino.

Ante la retransmisión de un conjunto de TSN, es necesario verificar sobre cual dirección fue transmitido originalmente cada TSN. Los TSN retransmitidos serán almacenados en la TRL asociada a la dirección de destino donde fueron originalmente transmitidos, independientemente de la dirección por la que sean retransmitidos.

Por cada reporte de TSN duplicado, se debe eliminar el TSN en la TRL asociada a la dirección de destino por la que fue transmitido originalmente, siguiendo las pautas de remoción antes detalladas.

Como SCTP no lleva registro de la dirección por la que un TSN fue originalmente transmitido, será necesario almacenar esta información que luego será utilizada el algoritmo.

Conceptualmente, la ocurrencia de un evento de retransmisión está asociada a la dirección de destino por la cual los TSN son transmitidos, y es sobre ésta dirección donde se quiere aplicar el algoritmo de detección. SCTP sugiere que las retransmisiones de los paquetes se realicen, en caso de ser posible, por una dirección alternativa.

Si una retransmisión fue espuria, entonces todos los paquetes transmitidos originalmente fueron recibidos por el receptor. Por cada uno de los paquetes retransmitidos, se recibirá un Ack conteniendo los reportes de duplicados por la dirección de destino por el cual fueron retransmitidos. Aplicando el algoritmo y utilizando la TRL asociada a la dirección de destino (por la que los paquetes fueron transmitidos originalmente), se puede concluir que ningún paquete fue perdido y que los paquetes fueron retransmitidos en forma innecesaria.

4.1.1. Costo Computacional y de Almacenamiento

La TRL puede implementarse como dos listas, una conteniendo los TSN pertenecientes a la última retransmisión, y otra con el resto de los TSN. Las longitudes máximas de las listas no pueden calcularse a priori, dado que la cantidad de paquetes a retransmitir no puede determinarse debido a la variabilidad del CWnd. Cada elemento de la lista es un valor de 32 bits. Está garantizado que una vez que el Cumulative TSN Ack Point avance, ante un evento de retransmisión se eliminarán aquellos TSN menores.

Para facilitar el cálculo de los costos involucrados del algoritmo de detección, es conveniente dividir el mismo en tres etapas:

Evento de retransmisión Aquí el costo computacional está dado por dos operaciones: la copia de la TRL con los TSN de la ultima retransmisión a la lista de históricos y el costo asociado al mantenimiento del la TRL histórica. El costo asociado a la primera operación es lineal. En sí, se debe recorrer la lista con el fin de copiar todos los elementos a la TRL histórica. El costo de la segunda operación también es lineal, ya que se debe recorrer la lista de históricos y eliminar todos los TSN menores. Dado que las longitudes de ambas listas en la práctica se espera que posean longitudes pequeñas, este costo puede despreciarse.

Retransmisión de TSN Cada TSN a retransmitir se debe almacenar en la TRL asociada a la dirección por la cual fue originalmente transmitido. Esta operación es de orden constante, ya que la información para un TSN contiene un puntero a la dirección. Por último, la inserción en la lista es constante.

Recepción de un reporte de TSN duplicado Por cada reporte de TSN duplicado recibido se debe buscar el TSN en la TRL que corresponda. Como posiblemente la información para un TSN no este presente en la ventana de transmisión, se debe buscar el mismo en todas

los TRL y eliminarlo según detalle el algoritmo. Sin embargo, en la mayoría de los casos se ha de esperar que la TRL buscada sea la asociada a la dirección primaria. De nuevo aquí el costo es lineal.

Cabe destacar que el costo computacional y de almacenamiento adicional no influyen en el rendimiento en general si no ocurren eventos de retransmisión, que es donde esta modificación mejora el desempeño del protocolo.

4.2. Recovery

Ante el evento Timeout, el emisor SCTP dispara los algoritmos de control de congestión. Esto produce una reducción significativa del rendimiento. Esta reducción se logra disminuyendo dramáticamente los valores de CWnd y SSThresh. Sin embargo, si el Timeout no fue causado realmente por pérdida de paquetes, sino por cambios bruscos del RTT, se estará penalizando el rendimiento de la asociación en forma innecesaria. El costo producido por un Timeout consta de tres componentes:

1. Período de inactividad.
2. Slow Start.
3. Crecimiento lineal más allá del SSThresh, que sería la mitad del CWnd antes del Timeout.

El mismo problema se presenta con Fast Retransmit, que, ante una simple pérdida que produce que se dispare el algoritmo de Fast Retransmit, se reduce el CWnd a la mitad, y el SSThresh toma el valor del nuevo CWnd, entrando en Congestion Avoidance. En éste caso también se está penalizando el rendimiento de la asociación en forma innecesaria. La reducción del rendimiento en Fast Retransmit no es tan costosa como en Timeout.

Con el fin de minimizar el impacto causado por las penalizaciones innecesarias antes mencionadas, una solución posible es restaurar los valores del CWnd y SSThresh a los valores previos al Timeout o Fast Retransmit que causó su reducción. Sin esto, alcanzar el valor previo del CWnd y el rendimiento anterior podría llevar varios RTT.

Con el fin de implementar el Recovery será necesario, ante un evento de retransmisión, almacenar los valores de CWnd y SSThresh previos a la reducción sobre la dirección que ocurrió el evento.

El objetivo del algoritmo de Recovery consiste en restaurar a los valores previos de CWnd y SSThresh, tan pronto se detecta que un evento de retransmisión fue espurio.

El trabajo [24] muestra que el restaurar simplemente a los valores anteriores puede llevar a la emisión de una ráfaga de paquetes, llevando la red a un posible estado de congestión.

La capacidad de una red puede variar abruptamente en períodos muy cortos. Durante el lapso de tiempo entre que se produjo el evento de retransmisión espurio y su posterior detección, la capacidad de la red pudo haber disminuido considerablemente. Restaurar al valor del CWnd previo, que puede no ser adecuado para el estado actual de la red, podría provocar la saturación de la misma. La solución propuesta por [24] y adaptada para el algoritmo, consiste básicamente en actualizar el SSThresh al valor previo del CWnd para así entrar en Slow Start y comenzar a probar la red nuevamente, luego del período de inactividad. De esta manera, en un par de RTT, se puede alcanzar el valor anterior del CWnd. Si el valor del CWnd actual supera el valor que poseía anteriormente, no se realiza ningún ajuste a las variables. Queda claro que en la mayoría de los casos, la cantidad de RTT necesarios para alcanzar el rendimiento anterior, es mucho

menor que la cantidad necesaria sin aplicar el algoritmo de Recovery, principalmente en enlaces con altos valores de demora.

Sin el algoritmo de Recovery, el valor del CWnd crecerá en forma exponencial hasta el SSThresh, que es igual a la mitad del valor del CWnd previo. Luego crecerá en forma lineal. Con el algoritmo de Recovery, el valor del CWnd crecerá en forma exponencial hasta el SSThresh, pero en este caso el valor es igual al CWnd previo. El crecimiento del valor del CWnd en forma exponencial comenzará tan pronto se detecte que el evento de retransmisión fue espurio. Entonces, mientras más pronto esto se detecte, menor la cantidad de RTT necesarios para alcanzar el rendimiento anterior.

En caso de producirse eventos de retransmisión anidados, es decir, que ocurra un evento antes de poder detectar el anterior como espurio, en caso de serlo, las variables se irán reduciendo a medida que cada uno de estos eventos se sucedan. Bajo estas circunstancias, el algoritmo de Recovery únicamente restaurará los valores de las variables previos al último evento ocurrido. De ésta manera, mientras más eventos anidados ocurran, más conservador será el algoritmo. Una excepción a esto es el caso de los Fast Retransmit anidados. Como ya se mencionó, las variables CWnd y SSThresh no son modificadas si ocurren Fast Retransmit cuando la asociación se encuentra en Fast Recovery. Por este motivo se considera a los Fast Retransmit anidados como un único evento de retransmisión.

4.2.1. Costo computacional y de almacenamiento

El costo de almacenamiento es de 8 bytes por dirección existente en la asociación, que corresponden al almacenamiento del SSThresh y CWnd. El costo computacional asociado al Recovery es despreciable, ya que la única acción a realizarse será la restauración de los valores de CWnd y SSThresh.

4.3. Adaptación dinámica del DupThresh

Cuando un paquete es reportado como faltante en cuatro oportunidades, SCTP dispara el mecanismo de Fast Retransmit[5]. La decisión de elegir el valor del DupThresh no es trivial y está sostenida en la experiencia acumulada en TCP. Este valor tiene influencia en el rendimiento del protocolo cuando la red presenta reordenamiento de paquetes.

Para que un emisor SCTP evite Fast Retransmit espurios luego de que un paquete es reordenado, el DupThresh debe ser mayor o igual al número de reportes de TSN duplicados que fueron generados debido a reordenamiento para un TSN.

El contador de reporte de faltantes (CRF) de un TSN es la cantidad de veces que el mismo fue reportado como faltante, siguiendo el algoritmo de HTNA. Este valor es utilizado para disparar Fast Retransmit, y es mantenido por cada TSN transmitido en una asociación.

Intuitivamente, cuando un paquete es demorado y arriba fuera de orden, hay un hueco en la ventana de recepción; el emisor recibe la información sobre paquetes faltantes que fueron enviados después del paquete demorado.

El rendimiento de SCTP será menor en redes que presenten CRF mayor o igual que cuatro. En este caso, SCTP entrará en Fast Retransmit reiteradamente. La penalización que se debe pagar por esto es alta. Por un lado, SCTP retransmitirá todos los paquetes con CRF mayor que tres. Por otro lado, se producirá una reducción del CWnd y SSThresh. Estos factores tienen como consecuencia directa un período de inactividad de la asociación, donde no se transmiten datos nuevos, causando una gran reducción del rendimiento.

[5] La cantidad de reportes de duplicados que se deben recibir para entrar en Fast Retransmit es el DupThresh

Por esta razón es conveniente que el DupThresh no sea un valor constante, sino que se *adapte dinámicamente a las condiciones de reordenamiento* presentes en la red. El fenómeno de reordenamiento en general no está asociado a ningún patrón ni distribución probabilística. Como muestra [32], uno de los factores que provoca el reordenamiento sobre ciertas arquitecturas de red es la congestión. El ajuste del DupThresh debe realizarse con sumo cuidado. Si el valor es pequeño respecto al CRF medio que experimenta la red, ocurrirán sucesivos Fast Retransmit, con las consecuencias antes mencionadas. En cambio, si el valor es grande respecto al CRF medio, puede ocurrir que una simple pérdida de un paquete no sea solucionada por Fast Retransmit, con lo que se producirá un Timeout que podría haberse evitado. La penalización que se debe pagar en este caso es mayor que en el caso de Fast Retransmit.

En resumen, la elección de un DupThresh pequeño en relación al CRF medio de la red produce que una asociación entre en Fast Retransmit cuando no debiera hacerlo ya que los paquetes solo fueron demorados o reordenados. Por otro lado, la elección de un DupThresh grande, no permitirá la recuperación frente a pérdidas simples por medio de Fast Retransmit, lo que dará lugar a un Timeout, con el costo que esto tiene asociado.

4.3.1. Medición y Seguimiento del contador de reporte de faltantes

Para adaptar el DupThresh en base a las condiciones de reordenamiento que se producen en la red, es necesario realizar una medición del mismo y poder analizar su evolución en función del tiempo. Con el fin de mantener un histórico con los CRF que se fueron presentando en la red se utilizará un histograma.

El histograma contendrá el CRF de cada uno de los paquetes transmitidos tales que su CRF sea mayor o igual que cuatro.

La inserción en el histograma del CRF de un TSN que fue retransmitido deberá demorarse hasta la recepción de un reporte de TSN duplicado para el mismo, pues debemos asegurarnos que el TSN no se haya perdido, de lo contrario dicho valor estará dando información acerca de un reordenamiento que nunca pudo haber ocurrido. Nuevamente, dada la imposibilidad de saber a qué transmisión corresponde la recepción de un reporte de TSN duplicado, sólo se insertan en el histograma los CRF de los TSN correspondientes a la última retransmisión.

Cuando un paquete i es demorado, un reporte de faltante será recibido por el emisor en cada paquete $i+1 \ldots i+k$ que arribe al receptor antes del paquete i. Luego, el número de reporte de faltantes generados por la demora del paquete i es k, la diferencia entre el paquete más alto confirmado y el número del paquete demorado. Se define como *longitud de reordenamiento* al valor de k. Sin embargo, utilizando únicamente la información que brinda SCTP es imposible obtener una medida exacta de la longitud de reordenamiento.

Una forma de obtener la longitud de reordenamiento en forma exacta consiste en la modificación de la parte receptora de SCTP, de forma tal que ésta calcule la longitud del reordenamiento e informe al emisor. Sin embargo, además de la modificación en el código del receptor, es necesario agregar una extensión al protocolo para soportar éste tipo de mensajería.

4.3.2. Actualización del DupThresh

El histograma de CRF sumariza la distribución del reordenamiento experimentado para una asociación. La primera estrategia para evitar Fast Retransmit espurios podría ser seleccionar el percentil deseado de reordenamiento para el cual deseamos que los Fast Retransmit espurios sean evitados, y actualizar el DupThresh de tal manera que éste sea igual al valor del percentil de la distribución acumulada de los CRF del histograma. Es decir, si el 90 % de los eventos de reordenamiento los TSN poseen un CRF de 8 o menos, entonces un DupThresh de 9 evitará

el 90 % de los Fast Retransmit espurios. Aún con la elección de un percentil fijo, el DupThresh variará en el tiempo dado que el contenido del histograma cambiará en relación al comportamiento del reordenamiento en la ruta de conexión.

Incrementar el DupThresh tiene un costo asociado. Para evitar estos problemas, el algoritmo debe ser capaz de reducir el DupThresh en caso de ser necesario.

Basándose en [51], se propone que en vez de variar el DupThresh directamente, se varíe el percentil utilizado para calcular el DupThresh. Definimos a este valor como PEF (*percentil 90* [6] *para evitar Fast Retransmit*).

Incrementar el PEF incrementará el DupThresh, mientras que decrementar el PEF decrementará el DupThresh. El incremento o decremento del PEF es conveniente realizarlo en función del costo relativo de un Fast Retransmit o de un Timeout. Tanto un Fast Retransmit como un Timeout poseen un costo de oportunidad de perdida de transmisión de paquetes. Un Fast Retransmit espurio causa una reducción del CWnd a la mitad y esta ventana reducida prevalece hasta el momento que se detecta que la retransmisión fue espuria, y permite la restauración del valor previo del CWnd. En contraste, un Timeout tiene dos costos principales: un período de inactividad luego de que toda la ventana se haya transmitido [7], y el Slow Start durante el cual el CWnd debe crecer desde uno y será menor que la mitad del CWnd previo por varios RTT.

Un Timeout consta de tres partes: un período de inactividad, Slow Start, y un crecimiento lineal más allá del SSThresh, que sería la mitad del CWnd antes del Timeout.

Un Fast Retransmit reduce el rendimiento menos que un Timeout. Este consiste solamente en la reducción del SSThresh a la mitad y un crecimiento lineal del CWnd ya que ambos valores son iguales. Luego, el costo adicional de sufrir un Timeout en vez de sufrir un Fast Retransmit es solamente el período de inactividad y Slow Start.

Suponiendo que una asociación dada posee el siguiente estado:

- $RW = Ventana\ de\ recepción$
- $W = Cantidad\ máxima\ de\ paquetes\ que\ se\ pueden\ enviar\ en\ un\ momento\ dado$
- $R = Smoothed\ RTT$
- $T = Duración\ de\ la\ retransmisión\ de\ paquetes\ durante\ un\ Timeout$

Y analizando cada uno de los costos antes mencionados tenemos:

Timeout espurio Una vez transmitidos todos los paquetes de una ventana W, se debe esperar que los paquetes sean confirmados antes de poder enviar una nueva ventana. Esta espera es de al menos un RTT. Luego, en un tiempo T se podrán enviar a lo sumo $W * (T/R)$. Si no hubiese existido el Timeout, al menos W paquetes hubieran sido transmitidos. En conclusión, durante un período de inactividad el emisor pierde la oportunidad de transmitir $W * (T/R) - W$ paquetes.

Durante Slow Start y hasta alcanzar $W/2$, el emisor pierde la oportunidad de enviar paquetes.

$$(W-1) + (W-2) + \ldots + (W - W/2 + 1) + (W - W/2)$$

[6] Sea un conjunto de valores ordenados $(x_1, x_2, \ldots, x_i, \ldots, x_n)$ de forma creciente, donde el $P90$ es el valor que se obtiene realizando $i = n * 0{,}9$ siendo $P90 = X_i$.

[7] Pero antes del Timeout en sí.

En Slow Start el CWnd crece exponencialmente desde 1 hasta $W/2$, luego en cada etapa de crecimiento se pueden enviar a lo sumo CWnd paquetes. Adicionalmente, en esa misma etapa el emisor se pierde de enviar $W - CWnd$ paquetes.

Lo expresado anteriormente se puede escribir como (en paquetes):

$$\mathbf{C}(Timeout) = \sum_{i=0}^{\log_2(W-1)} (W - 2^i) = W(\log_2 W - 1) + 1$$

Luego el costo total de un Timeout es aproximadamente:

$$W(T/R + \log_2 W - 2) + 1$$

El costo calculado no incluye el costo de la etapa de Congestion Avoidance, necesaria para alcanzar el valor anterior del CWnd en el momento del Timeout. A diferencia de Slow Start que posee un fin[8], el crecimiento del CWnd durante la etapa de Congestion Avoidance no posee un límite superior más allá del almacenamiento en memoria, por lo que resulta imposible el cálculo exacto de este valor. Sin embargo, se puede observar que este costo es despreciable en relación al costo total del Timeout.

Fast Retransmit espurio El costo de transmitir después de un Fast Retransmit espurio depende del intervalo requerido por el emisor para recibir los reportes de duplicados que identifican el Fast Retransmit como espurio. Para esto es necesario mantener una aproximación de la duración del Fast Retransmit espurio, llamada D. Cuando $D = R$ el costo de un Fast Retransmit espurio es simplemente $W/2$, la ventana ha sido reducida a la mitad innecesariamente por un RTT. Sin embargo, cuando $D > R$ el costo es mayor ya que la ventana se encuentra reducida por un período de tiempo mayor. En cada subsiguiente RTT el costo es menor. El crecimiento lineal del CWnd continúa hasta después de $(W/2)$ RTT, que es cuando la ventana original ha sido restaurada.Luego, para $\lceil k = D/R \rceil$ el costo de un Fast Retransmit espurio es menor o igual que:

$$(W - W/2) + (W - W/2 - 1) + \ldots + (W - W/2 - (k-1))$$

Luego

$$\mathbf{C}(Fast\ Retransmit) \leq \sum_{i=0}^{k-1} (W/2 - i) = k(W - k + 1)/2$$

La aproximación que se utilizará de D se basará en los mismos métodos que se utilizan en el cálculo del RTT.

4.3.3. Actualización del PEF

A continuación se explica como se usan las funciones de costos asociadas a Timeout y a Fast Retransmit espurios, en forma combinada para adaptar el PEF. Definimos como S la medida axiomática para adaptar el PEF. El valor de S elegido debe ser pequeño para permitir un fino ajuste del PEF. Las reglas para adaptar el PEF son:

- Después de cada Fast Retransmit espurio detectado, incrementar el PEF en S.

[8] Indicado por el SSThresh.

- Después de cada Timeout detectado, decrementar el PEF en

$$(\mathbf{C}(Timeout)/\mathbf{C}(Fast\ Retransmit)) * S$$

Estas reglas adaptan heurísticamente el PEF, y consecuentemente el DupThresh, de manera tal de maximizar el rendimiento por asociación que experimenta reordenamiento. Los Fast Retransmit espurios causan un incremento gradual del PEF. Los Timeout disparados por un valor alto del DupThresh causa que el PEF sea decrementado proporcionalmente teniendo en cuenta la reducción del rendimiento que éste produce, y la reducción del rendimiento de un Fast Retransmit espurio.

4.3.4. Costo computacional y de almacenamiento

El costo de almacenamiento está dado por el histograma de CRF, donde cada elemento consta de un CRF de 32 bits, y un Timestamp asociado, también de 32 bits. Esto hace un total de 8 bytes por entrada en el histograma. El histograma definido posee un máximo de 1024 entradas, lo que hace que el tamaño máximo de almacenamiento del histograma sea de 8 Kbytes.

Para facilitar el cálculo de los costos involucrados en el algoritmo de detección, es conveniente dividir el mismo en tres etapas:

Mantenimiento del histograma de CRF La inserción sobre la estructura es de orden constante, ya que se encuentra ordenada por Timestamp en forma ascendente. Ante cada inserción se remueven aquellos elementos que se consideren expirados en base a su Timestamp. Esta última operación es de orden lineal.

Cálculo del PEF Ante un evento de retransmisión, se debe actualizar el PEF en base a las funciones de costos definidas anteriormente. Los costos de dichas funciones son de orden constante.

Actualización del DupThresh En base al PEF se deberá actualizar el DupThresh. Para esto es necesario calcular el *p-percentil* del histograma de CRF. Con tal fin se debe reordenar el histograma por las longitudes de reordenamiento. Esta operación puede realizarse en orden $n\log(n)$ utilizando un algoritmo *quick sort*. Por último, el cálculo del *p-percentil* posee orden constante.

Nuevamente, el costo computacional y de almacenamiento no existe si no ocurren eventos de retransmisión.

Capítulo 5

Las herramientas de Linux

El presente capítulo explica qué son las Disciplinas de Colas de Linux, y se describen las distintas herramientas de Linux, *NetEm* y *tc*, a utilizar que posibilitan la emulación de distintas condiciones de red, necesarias para experimentar con el algoritmo.

5.1. Disciplina de colas (*qdics*)

A partir de la versión 2.2, Linux incorporó todas las herramientas necesarias para gestionar el ancho de banda de manera comparable a los sistemas dedicados de alto nivel.

5.1.1. Colas y disciplinas de cola explicadas

La disciplina de cola (qdisc) es un algoritmo que controla la cola de un dispositivo, sea de entrada o de salida. Con el encolamiento se determina la manera en que se envían los datos, se controla el flujo de los mismos. Es importante tener en cuenta de que sólo se puede dar forma o conformar al tráfico de datos que se transmiten.

Es conocido que Internet se basa en su mayoría en TCP/IP, y tiene algunas características que son utilidad. Sin embargo, TCP/IP no tiene manera de saber la capacidad de la red entre dos Host, de manera que simplemente empieza a enviar datos más y más rápido (Slow Start) y cuando empieza a perder paquetes, porque no hay espacio en los buffer de los Router para enviarlos, reduce la marcha.

Si se tiene un Router y se desea evitar que ciertas máquinas dentro de una red descarguen demasiado rápido sus paquetes, es necesario dar *forma* a la interfaz *interna* del Router, la que envía los datos a las computadoras. También hay que asegurar de que se controla el cuello de botella del enlace[1].

5.1.2. Disciplinas de cola simples, sin clases

Las *disciplinas de cola sin clases* son aquellas que, mayormente, aceptan datos y se limitan a reordenarlos, retrasarlos o descartarlos.

Esto se puede usar para ajustar el tráfico de una interfaz entera, sin subdivisiones.

[1] Si se tiene una Network Interface Card (NIC) de 100Mbit y un Router con un enlace de 256kbit, hay que asegurar de que no envían más datos de los que el Router puede manejar. De otra manera, será el Router el que controle el enlace y ajuste el ancho de banda disponible.

La disciplina más usada para lograr este objetivo, es la *qdisc pfifo_fast*, que se usa por defecto[2]. Cada una de estas colas tiene tanto puntos fuertes como debilidades específicos.

5.1.3. Disciplinas de cola con clases

Las qdisc con clases son muy útiles si se tienen diferentes tipos de tráfico a los que quiere dar un tratamiento separado.

Cuando entra tráfico dentro de una qdisc con clases, hay que enviarlo a alguna de las clases que contiene (se necesita *clasificarlo*). Para determinar qué hay que hacer con un paquete, se consulta a los *filtros*. Es importante remarcar que los filtros se llaman desde dentro de una qdisc, y no al revés.

Los filtros asociados a esa qdisc devuelven una decisión, y la qdisc la usa para encolar el paquete en una de las subclases. Cada subclase puede probar otros filtros para ver si se imparten más instrucciones. En caso contrario, la clase encola el paquete en la qdisc que contiene.

Aparte de contener otras qdisc, la mayoría de las qdisc con clases también realizan *shaping*, dan forma al tráfico. Esto es útil tanto para reordenar paquetes como para controlar tasas. Esto es necesario en caso de tener una interfaz de gran velocidad (por ejemplo, Ethernet) enviando a un dispositivo más lento (un cable módem).

La familia qdisc: raíces, controladores, hermanos y padres

Cada interfaz tiene una *qdisc raíz* de salida. Por defecto se asigna la disciplina de colas pfifo_fast sin clases, que se mencionó anteriormente. A cada qdisc y clase se le asigna un *controlador* (handle), que se puede utilizar en posteriores sentencias de configuración para referirse a la qdisc en cuestión. Aparte de la qdisc de salida, la interfaz también puede tener una qdisc de entrada, que dicta las normas sobre el tráfico que entra.

Los controladores de estas qdisc consisten de dos partes, un número mayor y un número menor: *"mayor":"menor"*. Es costumbre darle a la qdisc de raíz el nombre "1:", que es lo mismo que "1:0". El número menor de una qdisc siempre es "0".

Las clases deben tener el mismo número mayor que sus padres. Este número mayor tiene que ser único dentro de una configuración de salida o entrada. El número menor debe ser único dentro de una qdisc y sus clases.

Desencolado de paquetes

Cuando el kernel decide que necesita extraer paquetes para enviarlos a la interfaz, la qdisc "1:raíz recibe una petición de desencolar, esta petición es pasada a su descendiente "1:1", que a su vez es pasada a sus descendientes "10:", "11:", y "12:", que a su vez transmiten esta petición a sus descendientes, hasta que llegan a una hoja y se realiza el desencolamiento de paquetes.

Vale notar que clases anidadas solo hablan con sus padres qdisc, y no con la interfaz. Solo las qdisc raíz son desencoladas por el kernel.

La consecuencia de esto es que las clases nunca desencolan más rápido de lo que sus padres permiten. Y esto es exactamente lo que se quiere: *de esta manera, se puede tener SFQ como una clase interna, que no hace ajustes, sólo reordena, y tener una qdisc externa, que es la que hace los ajustes.*

Para mayor detalle ver apéndice B.

[2] Esta cola es, como su nombre indica, *First In, First Out* (*FIFO*, el primero que entra es el primero que sale), lo que significa que ningún paquete recibe un tratamiento especial.

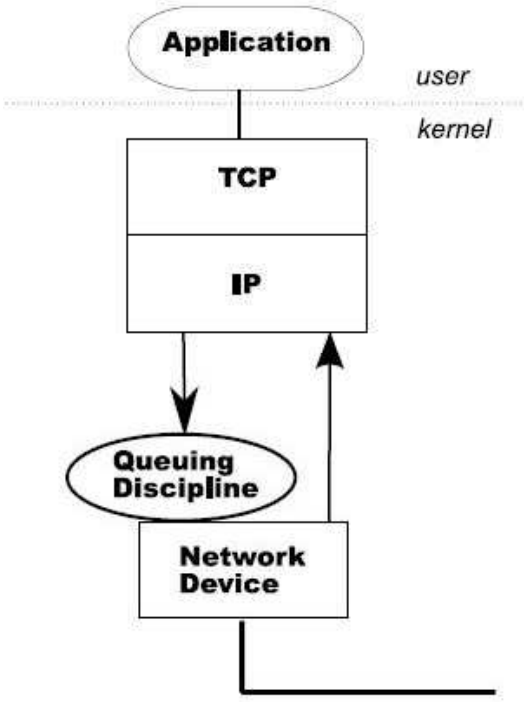


Figura 5.1: Arquitectura básica de NetEm.

5.2. NetEm

El modulo NetEm [43] del kernel de Linux provee la funcionalidad de emulación de red para realizar pruebas de protocolos, emulando las propiedad de un red WAN[3]. La actual versión puede emular demora, perdidas, duplicación, corrupción y reordenamiento de paquetes. NetEm se configura con la herramienta tc.

NetEm es una mejora reciente a las funcionalidades de control de trafico de Linux. Se construyó utilizando las funcionalidades existente en Linux de QoS y Servicios Diferenciados (DiffServ).

La motivación que hay detrás de NetEm es la de proveer una manera de reproducir grandes redes en un ambiente de laboratorio. Primeramente se utilizo para evaluar nuevas mejoras a TCP en Linux.

La figura 5.1 muestra la arquitectura básica de NetEm. Las disciplinas de cola existen entre la salida del protocolo y el dispositivo de red.

Como se explicó anteriormente, una disciplina de cola se puede ver como un objeto simple con dos interfases claves. Una interfase encola paquetes a ser enviados y la otra interfase entrega paquetes al dispositivo de red. Las políticas a utilizar para la entrega de paquetes al dispositivo de red está en la configuración de la disciplina de cola. Se pueden configurar políticas complejas anidando distintas disciplinas de cola.

Internamente, NetEm es una disciplina de cola con clases con colas de paquetes. Una de ella es una cola privada de espera tipo TFIFO y la otra es una disciplina de cola anidada, generalmente TFIFO también. La interfase de encolado toma paquetes, les asigna un Timestamp con el tiempo

[3]Una WAN es una red de comunicaciones que conecta ordenadores dispersos en una amplia área geográfica (conectados por líneas telefónicas u ondas de radio).

de envío y los pone en la cola de espera. Un reloj interno mueve los paquete de la cola de espera a la disciplina de cola anidada. La interfase de desencolado toma paquetes de la disciplina de cola anidada. *Al entrar paquetes a la cola y asignarse un Timestamp, la cola al ser de tipo TFIFO es reordenada según el Timestamp, los paquetes con menor Timestamp, o tiempo de entrega, se encuentran al principio de la cola y los paquetes con mayor Timestamp se encuentran al final de la cola.*. Esto es muy importante ya que dependiendo de como se configure NetEm, los paquetes pueden sufrir un reorden cuando no este no es deseado.

Sin embargo, NetEm presentó un problema ya que no soporta anidamiento consigo mismo, y por ende no es posible configurar una demora fija y una demora variable al mismo tiempo con la versión actual de NetEm. Por tal razón, se modificó NetEm para agregar una nueva opción de configuración, *fixed_ delay*. Esta opción permite fijar una demora fija a todos paquetes, además de permitir la demora variable original. Esta opción es útil al momento de definir las condiciones de red en los experimentos.

Para mayor detalle ver apéndice C.

5.3. *tc*

La herramienta *tc* es utilizada para configurar las colas y disciplinas de cola en el kernel de Linux, para dar formato al flujo de datos. Es parte del paquete de herramientas de *iproute2* [42]. El control de tráfico consiste configurar, a través de tc, las siguientes opciones:

Conformado Cuando se conforma el tráfico de red, se toma control de la tasa de transmisión de la misma. EL conformado puede ser algo más que la disminución del ancho de banda disponible, es también utilizado para suavizar las ráfagas de tráfico de manera tal que se consiga un mejor comportamiento de la red. El conformado del tráfico ocurre en los egresos.

Programación Con la programación de la transmisión de paquetes es posible mejorar la interactividad del tráfico que se requiera, al tiempo que se garantiza el ancho de banda para transmisiones en masa. Al reordenamiento también se lo llama priorización, y solo ocurre en los egresos.

Policing Mientras el modelado se ocupa de la transmisión del tráfico, el policing tiene que ver con la llegada del mismo. Solo ocurre en los ingresos.

Eliminación El tráfico que exceda un ancho de banda establecido podrá ser eliminado inmediatamente, tanto en ingresos como en egresos.

El procesamiento del tráfico es realizado dentro del kernel de Linux, por la configuración de las colas y disciplinas de cola.

Este software también tuvo que ser modificado para que acepte el nuevo parámetro fixed_delay *de la nueva configuración de NetEm.*

Para mayor detalle ver apéndice D.

Capítulo 6

Emulando condiciones de red

El presente capítulo explica como interactúan la Disciplina de Cola y las distintas herramientas de Linux para lograr la emulación de las distintas condiciones de red, en especial como se emula el reordenamiento de paquete y los cambios bruscos de RTT, que generan falsos Timeout y Fast Retransmit.

Anteriormente hemos explicado que son el reordenamiento de paquetes y los cambios bruscos de RTT. A continuación explicamos como se pueden emular estos comportamientos sobre un único camino de red utilizando las herramientas de Linux descritas, NetEm y tc.

6.1. Reordenamiento de paquetes

Demora promedio de X ms y un *gap* de Y paquetes Esto hará que 1 de cada Y paquetes sufra una demora X ms, mientras que el resto de los paquetes no sufra demora alguna y sean enviados a la interfaz inmediatamente. Esto se logra con el siguiente comando de ejemplo:

```
tc qdisc add dev eth0 root netem delay 30ms gap 5
```

Demora promedio de X ms y un desvío estándar Y ms Esto hará que los paquetes sean demorados X $ms \pm Y$ ms. Como internamente NetEm ordena los paquetes según su tiempo de egreso, se dan situaciones donde un paquete A sufre una demora mayor que la sufrida por otros paquetes que ingresaron posteriormente. Entonces estos últimos paquetes serán enviados a la interfaz antes que el paquete A. Esto se logra con el siguiente comando de ejemplo:

```
tc qdisc add dev eth0 root netem delay 30ms 15ms
```

Para entender mejor esta situación se presenta el siguiente ejemplo:

- A un paquete se le asigna una demora aleatoria de 30 ms ($30\ ms - 0\ ms$), luego a un siguiente paquete se le asigna una demora estándar de 20 ms ($30\ ms - 10\ ms$). Al sufrir este último una demora menor que la del primero, se invierte el orden de egreso de los paquetes[1].

[1] Esto se debe a que la disciplina de cola *tfifo* interna de NetEm, que mantiene ordenados los paquete por su tiempo programado de envió y por lo tanto los paquetes se reordenan por su tiempo dentro de NetEm y luego son enviados a la interfaz de salida.

Demora promedio de X ms y un porcentaje Y % de reorden Esto hará que Y por ciento de los paquetes sufran una demora de X ms, mientras que el resto de los paquetes no serán demorados y sean enviados a la interfaz de salida inmediatamente. Aquí también se puede configurar un desvío estándar Y ms a la demora promedio. Esto se logra con el siguiente comando de ejemplo:

```
tc qdisc add dev eth0 root netem delay 50ms reorder 25%
```

6.2. Cambios bruscos de Round Trip Time

Para el presente trabajo estudiamos procesos aleatorios donde el resultado de un experimento aleatorio se asocia con una función de tiempo. Los procesos aleatorios también se denominan procesos *estocásticos*. Así, un proceso aleatorio asigna una función aleatoria de tiempo como el resultado de un experimento aleatorio.

Para emular distintas condiciones de red debemos comprender que son los procesos determinísticos y no determinísticos [11].

Procesos determinísticos Un proceso determinístico es aquel en el que los valores futuros de la función de ejemplo se saben si se conoce el valor actual.

Procesos no determinísticos Un proceso aleatorio no determinístico es aquel en el que los valores futuros de la función de ejemplo no se pueden saber si se conoce el valor actual.

La naturaleza misma de los cambios bruscos de RTT es estocástica. Por eso necesitamos emular este comportamiento de dos maneras: 1) determinística, 2) aleatoria o no determinística.

Determinística y controlada Presupone cambios en el RTT de manera determinada. Aquí el valor del RTT cambia de manera controlada, manteniendo un mismo valor de RTT por tiempo determinado, luego este valor cambia a uno distinto por un tiempo determinado. Estos cambios controlados y determinados de RTT se repiten sin fin[2].

Esto se puede emular haciendo un pequeño programa que configure NetEm para que durante Y_0 ms aplique una demora de X_0 ms a los paquetes, y luego configure NetEm para que durante Y_1 ms aplique una demora de X_1 ms a los paquetes, repitiendo este ciclo de cambios del RTT incesantemente.

Aleatoria o no determinística No presupone cambios en el RTT de manera determinada, sino que el RTT cambia constantemente aleatoriamente. Aquí el valor del RTT cambia de manera aleatoria, manteniendo el mismo valor por un tiempo no determinado y luego este cambio a otro valor aleatorio del RTT durante un tiempo no determinado.

Esto se puede emular configurando NetEm para que aplique a los paquetes una demora por X ms, un desvío de Y ms y una correlación de Z %. Configurando valores grandes para Y y Z, se dan situaciones donde ráfagas de paquetes sufren una misma demora pequeña y otros una misma demora mayor.

Vale notar que, en ambas configuraciones de NetEm, como efecto secundario, se producen reorden de paquetes.

[2] Por ejemplo, una comunicación establecida puede tener inicialmente un RTT de 100 ms durante x ms, luego cambia el RTT a 400 ms durante y ms, luego cambia el RTT al valor original de 100 ms para que luego de x ms cambie nuevamente el RTT a 400 ms durante y ms, este ciclo de cambios del RTT se repite indefinidamente en el tiempo.

6.3. Control de Tasa

Para emular un red física utilizable en la vida real, es necesario poder reproducir ciertas características de la misma, como ser el ancho de banda, demora, tamaño de paquetes, ráfagas de paquetes, etc.

Dado que NetEm no tiene un control de tasa interno, se analizo recurrir a la disciplina de cola *Token Bucket Filter* para que realice el control de tasa. Esto se logra con el siguiente comando de ejemplo:

```
tc qdisc add dev eth1 root tbf rate 10mbit burst 1500 mtu 1500
latency 50ms
```

Con esta configuración se obtiene una red que puede transmitir hasta 10 *mbit* de información, la unidad máxima de transmisión es de 1500 *bytes*.

Sin embargo, luego de realizar varios experimentos con y sin control de tasa, se decidió no utilizar esta herramienta ya que se pudo verificar que el comportamiento no era el esperado y esto podría generar cierto *ruido* que influiría en los experimentos a realizar. Por lo tanto no utilizamos control de tasa en nuestros experimentos.

6.4. Integrando las herramientas

Para poder emular una red con las condiciones necesarias para el presente trabajo, es preciso considerar todos de sus aspectos y, de ellos, sintetizar aquellos que atañen a esta disertación. Para ello es necesario utilizar las herramientas disponibles para emular el reordenamiento y los cambios brusco de RTT que sufrirán los paquetes. Esto se puede lograr integrando las distintas herramientas aquí presentadas:

- Posibilidad de anidar distintas clases de disciplina de cola
- Uso de la disciplina de cola NetEm para el reordenamiento y cambios bruscos de RTT

A los ejemplos de configuración de NetEm arriba descritos, hay que agregar la configuración de demora fija necesaria para el algoritmo aquí presentado. A continuación se presentan unos ejemplos de configuración de NetEm con demora fija:

- ```
  tc qdisc add dev eth1 root netem limit 999999 \
          delay 15ms 5ms 90% reorder 90% fixed_delay 50ms
  ```

  En este ejemplo, la disciplina de cola NetEm controla que los paquetes sufran una demora fija de 50 ms más una demora variable de 15 $ms \pm 5$ ms con una correlación del 90 %, con un límite de ventana de 999999 paquetes y un reorden del 90 %. Con estas condiciones, el 90 % de los paquetes sufrirán una demora de 50 ms y serán enviados en orden, mientras que el 10 % de los paquetes restantes sufrirán una demora de 50 $ms + (15\ ms \pm 5\ ms)$ y se reordenarán según su tiempo de egreso.

- ```
  tc qdisc add dev eth1 root netem limit 999999 \
          delay 200ms 150ms 90% fixed_delay 50ms
  ```

  En este ejemplo, la disciplina de cola NetEm controla que los paquetes sufran una demora fija de 50 ms más una demora variable de 200 $ms \pm 150$ ms, con una correlación del 90 % entre las demoras variables y un límite de ventana de 999999 paquetes. Con estas condiciones, todos los paquetes sufrirán cambios bruscos en RTT.

Capítulo 7

Experimentos

El presente capítulo presenta los experimentos realizados sobre distintas condiciones de red y los resultados obtenidos de los mismos.

Como se explicó en el capítulo 6, se utilizaron distintas herramientas para emular diferentes condiciones de red donde se realizaron los experimentos, con y sin el algoritmo aquí presentado. Esto se debe a que actualmente no existen otros algoritmos para mejorar el desempeño de SCTP frente a reordenamiento y cambios bruscos de RTT.

7.1. Configuración del hardware

A continuación se detalla la configuración del hardware utilizado para realizar los experimentos. Éstos se realizaron utilizando tres computadoras, siendo una de ellas, la que realizó la emulación de las diferentes condiciones de red necesarias, haciendo de *puente* entre las otras dos máquinas. De las restantes dos, una maquina actuó como cliente y otra como servidor. Estas dos últimas no tenían ninguna conexión directa entre sí, sino que se conectaban a través de la maquina puente, que poseía dos placas de red de 100 Mb, una para conectarse con la maquina cliente y otra para conectarse con la maquina servidor. La figura 7.1 muestra una gráfico explicativo de la configuración utilizada.

La maquina puente permite la conexión entre el cliente y el servidor de manera transparente, es decir, que ninguna conoce que existe una tercer maquina entre ellas.

Según su definición, un *puente* es una manera de conectar dos segmentos Ethernet juntos, de una manera independiente del protocolo de red. Los paquetes son redirigidos según la dirección Ethernet de su destino en vez de la dirección IP (como un Router). Como esta redirección es

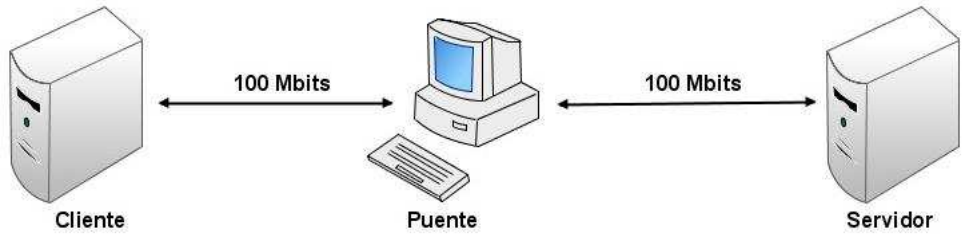


Figura 7.1: Conexión de maquinas

hecha en la segunda capa, a nivel IP, todos los protocolos superiores pueden comunicarse de una manera transparente.

En la maquina *puente* se emularon las condiciones de red en ambas placas Ethernet. De esta manera las distintas condiciones de red fueron simétricas tanto para la maquina cliente como para la maquina servidor.

7.2. Escenarios

Se configuraron diversos escenarios, con distintas condiciones de red en ambientes emulados, para experimentar con SCTP y SCTP-RR. Podemos dividir en dos los escenarios, de acuerdo a los objetivos buscados en los mismos.

Un escenario tuvo como objetivo medir el *rendimiento en tiempo*, que consistió en medir cuanto tarda, en segundos, la maquina cliente en transmitir 3 MB de información al servidor, en mensajes de 110592 bytes de largo a la maquina servidor.

Otro escenario tuvo como objetivo medir el *rendimiento en bytes por segundo*, que consistió medir la cantidad de bytes que puede transmitir la maquina cliente a la maquina servidor durante 60 segundos, con mensajes de 16384 bytes y respuestas de 1024 bytes de largo.

En todos los experimentos que se realizaron, se emularon condiciones de reorden de paquetes y cambios bruscos de RTT sobre enlaces de 100Mb.

Sobre cada uno de los dos escenarios, se emularon tres tipos de condiciones de red, que se detallan a continuación.

Demora fija más demora variable por porcentaje de reorden Se realizaron varios experimentos donde los paquetes IP sufrieron una demora fija de 50, 100, 200 y 300 ms. Además el 0, 5, 10 y 15 por ciento de los paquetes IP sufrieron una demora variable de 100, 200 y 300 ms que era sumada a la demora fija anterior.

El reorden de paquetes se logra ya que un $Y\,\%$ de los paquetes sufren una demora fija y variable, mientras que el resto de los paquetes solo sufren una demora fija. El cambio brusco de RTT, se logra de manera aleatoria, mayormente cuando la demora variable es grande.

En las figuras 7.2 y 7.3 se pueden observar los resultados de los experimentos realizados con una demora fija de 200 ms y una demora variable de 100 ms para distintos valores de reorden.

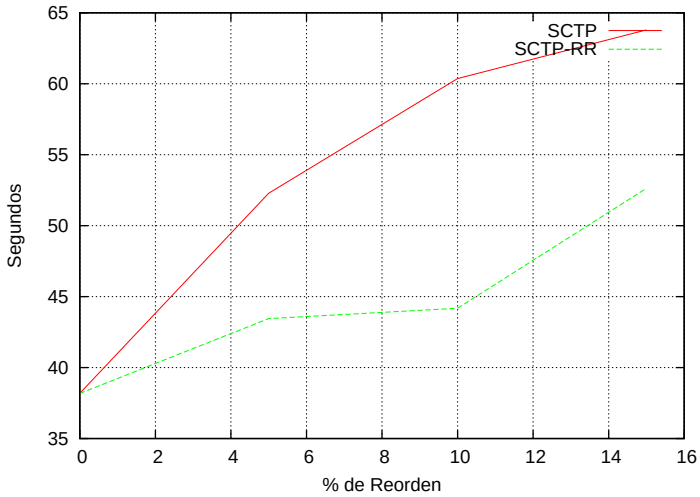


Figura 7.2: Demora fija de 200 ms, demora variable de 100 ms, por tiempo

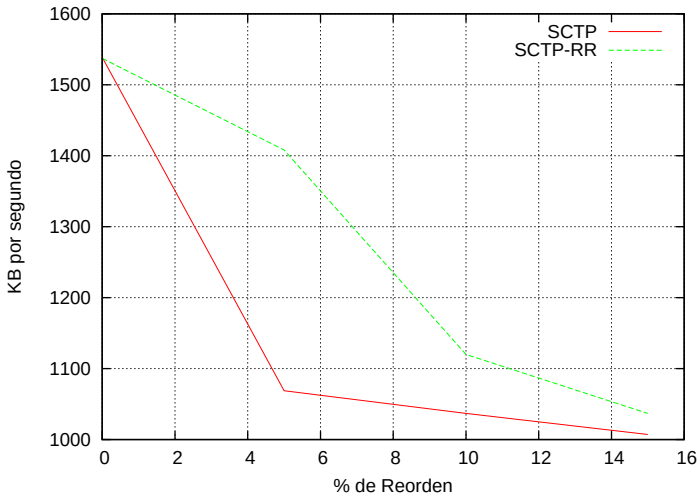


Figura 7.3: Demora fija de 200 ms, demora variable de 100 ms, bytes por segundo

Demora fija más demora variable Se realizaron varios experimentos donde los paquetes IP sufrieron una demora fija de 100, 200, 300 ms, más una demora variable de 50, 100, 150 $ms \pm 50$, 100, 150 ms.

El reorden de paquetes se logra ya que, dependiendo de la demora fija más la demora variable que sufra un paquete, NetEm los reordena con su nuevo tiempo de egreso (distinto al de arribo). El cambio brusco de RTT se logra de manera aleatoria mayormente cuando la demora variable es grande.

En las figuras 7.4 y 7.5 se pueden observar los resultados de los experimentos realizados con una demora fija de 300 ms y una demora variable de 100 ms para distintos valores de reorden.

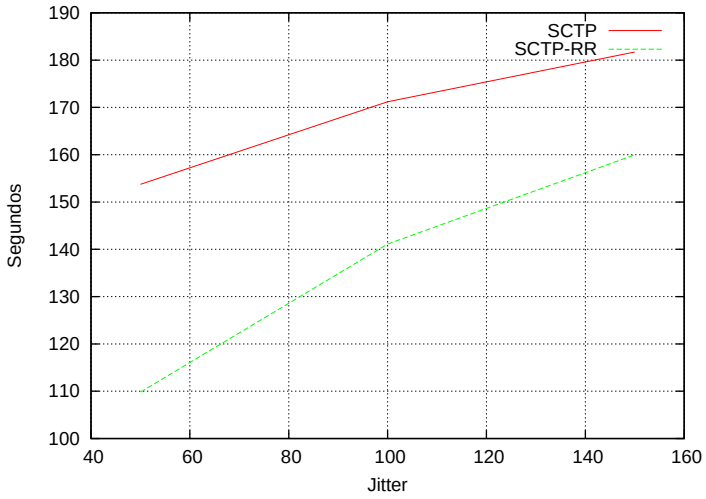


Figura 7.4: Demora fija de 300 ms, demora variable de 100 ms (Jitter), por tiempo

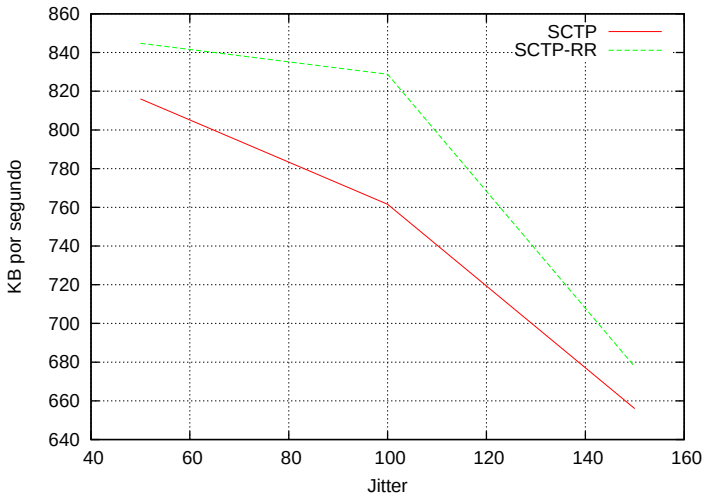


Figura 7.5: Demora fija de 300 ms, demora variable de 100 ms (Jitter), bytes por segundo

Demoras fijas por cambio de tiempo Se realizaron varios experimentos donde, cada 250, 500, 750 y 1000 ms, los paquetes IP sufrían una demora fija que cambiaba de 1 ms a 200 ms, 100 ms a 300 ms y de 200 ms a 400 ms. Por ejemplo, durante 500 ms la demora fija era de 100 ms y luego de 500 ms demora fija cambiaba a 300 ms durante 500 ms, y luego la demora fija volvía cambiar a 100 ms durante 500 ms, este ciclo se repetía indefinidamente durante todo el experimento.

El reorden de paquetes se logra al momento del cambio de la demora fija. El cambio brusco de RTT se logra de manera determinística, al momento de cambio de tiempo de la demora fija (ya que el RTT bajaba o subía 200 ms a cada cambio de la demora fija).

En las figuras 7.6 y 7.7 se pueden observar los resultados de los experimentos realizados

con un cambio de tiempo de cada 500 ms.

Figura 7.6: Tiempo de cambio 500 ms, por tiempo

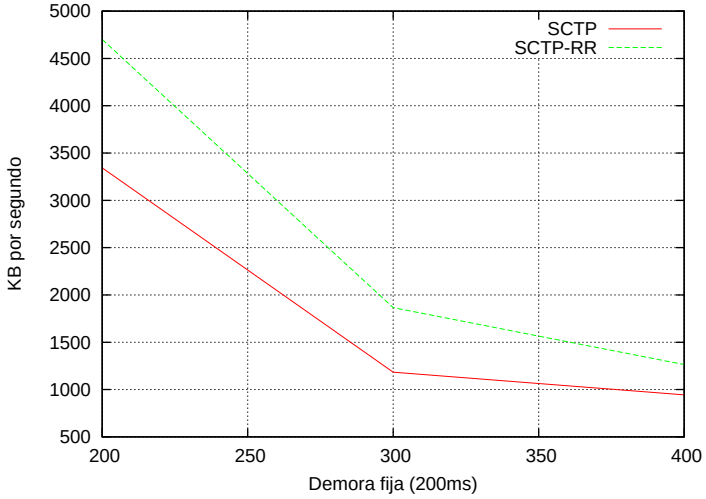


Figura 7.7: Tiempo de cambio 500 ms, bytes por segundo

Cabe aclarar que no se realizaron experimentos de comparación con TCP dado que dichas pruebas no tendrían valor, ya que TCP corre, y esta implementado, a nivel del kernel de Linux mientras que SCTP corre a nivel de usuario, no está implementado con instrucciones nativas del kernel, utiliza sockets *raw* para montar el protocolo SCTP. Estas diferencias hacen que el rendimiento entre ambos sea dispar y arroje resultados poco fiables a la hora de realizar una comparativa.

Para el presente trabajo se realizaron muchos experimentos, sin embargo, y para no hacer extensivo en demasía y mayor claridad expositiva, se decidió mostrar solamente algunos gráficos

de los resultados obtenidos. En el anexo A se pueden observar más gráficos de los resultados de los experimentos realizados.

Para el presente trabajo se realizaron muchos experimento, sin embargo, para no hacer extensivo en demasía y para mayor claridad expositiva, se decidió mostrar solamente algunos gráficos de los resultados obtenidos. En el anexo A se pueden observar más resultados de los experimentos realizados.

7.3. Análisis

Al analizar los gráficos resultantes de los experimentos, para los dos escenario y los tres tipos de condiciones de red, podemos observar resultados congruentes de la mejora introducida por SCTP-RR. A continuación analizamos los resultados en ambos escenarios.

Medición del rendimiento en tiempo De los gráficos resultantes con el primer tipo condición de red observamos que al aumentar el porcentaje de reorden de paquetes, el rendimiento de SCTP bajaba al igual que SCTP-RR. Sin embargo, SCTP-RR tiene un mejor rendimiento con respecto a SCTP, en el sentido que tarda menos tiempo en trasmitir 3 MB que SCTP. Observamos una mejora de rendimiento en escenario con reordering de paquetes.

Vale notar que a medida que aumenta el porcentaje de reorden, la caída de rendimiento tiende a nivelarse, esto sucede tanto con todos los tiempos de demora fija utilizados en los experimentos.

De los gráficos resultantes de los experimentos con el segundo y tercer tipo de condición de red, observamos que SCTP-RR tiene un mejor rendimiento que SCTP. A medida que aumenta la demora variable o la demora fija, el rendimiento de SCTP-RR y SCTP decae (aumenta el tiempo en que tardan en transmitir 3 MB). Pero a pesar de la perdida de rendimiento por parte de SCTP-RR y SCTP, SCTP-RR continua teniendo un mejor rendimiento.

Medición del rendimiento en bytes por segundo De los gráficos resultantes con el primer tipo de condición de red observamos el resultado opuesto al rendimiento en tiempo. A medida que aumenta el porcentaje de reorden la cantidad de bytes por segundo disminuye para SCTP-RR y SCTP. Sin embargo, SCTP-RR continua teniendo mejor rendimiento que SCTP.

Finalmente de los gráficos resultantes con el segundo y tercer tipo condición de red observamos el mismo comportamiento. Baja el rendimiento de ambos protocolos, pero SCTP-RR supera en rendimiento a SCTP.

Vale notar que para las tres tipos de condiciones de red, sin importar el escenario de experimento, a medida que aumenta el porcentaje de demora, la demora variable o la demora fija, las curvas de los gráficos resultantes tienden a nivelarse. Sin embargo, en todos los experimentos realizados SCTP-RR supera en rendimiento a SCTP.

Capítulo 8

Conclusiones

Durante el desarrollo del presente trabajo se presentó un algoritmo para mejorar el rendimiento de SCTP ante cambios bruscos de RTT y reordenamiento de paquetes, compuesto por tres módulos. El módulo de detección le permite a un emisor distinguir si un Timeout o un Fast Retransmit fue espurio, basándose en los reportes de duplicados. El módulo de Recovery restaura los valores de las variables de control de congestión cuando un evento de retransmisión fue detectado como espurio. El módulo de adaptación dinámica del DupThresh se encarga de ajustar el mismo en base a información sobre el reordenamiento existente en la asociación, con el fin de evitar futuros Fast Retransmit espurios. La elección del DupThresh se basa en la utilización de una función de costos, de forma tal de maximizar el rendimiento de la asociación. Se realizaron las modificaciones propuestas por este algoritmo sobre una librería actual del protocolo SCTP.

Así mismo se presentaron modificaciones al módulo NetEm del kernel de Linux y a la herramienta tc para poder emular distintas condiciones de red, necesarias para evaluar SCTP y SCTP-RR en un ambiente real y controlado.

Se realizaron distintos experimentos sobre distintos escenarios significativos para realizar un análisis, en un ambiente real y controlado, comparando el rendimiento de SCTP y SCTP-RR. En todos los escenarios propuestos SCTP-RR demostró tener un mejor rendimiento que SCTP. Vale notar que a medida que se incrementa el deterioro en las condiciones de red, ambos algoritmos bajan en su rendimiento y el gráfico su curva tiende a nivelarse.

Capítulo 9

Trabajo Futuro

Como trabajo a futuro creemos que sería provecho realizar distintas aplicaciones, utilizando la librería, y utilizarlas en el mundo real. Ciertamente midiendo el rendimiento de las mismas mientras son utilizadas.

Utilizar hardware especial para emular las distintas condiciones de red sería un punto importante para corroborar los resultados presentados en este trabajo.

Finalmente, y para que este algoritmo pueda ser aprovechado por todos y no solamente con fines científicos, se propone implementar SCTP-RR en el módulo LKSCTP, para luego si poder realizar comparativas de rendimiento entre TCP y SCTP-RR.

Bibliografía

[1] M. Allman, V. Paxson, and W. Stevens. RFC 2581: TCP congestion control. *Internet RFCs*, 1999.

[2] P. Almquist. RFC 1349: Type of service in the Internet Protocol suite. *Status: PROPOSED STANDARD*, July 1992.

[3] Righetti Claudio E. Barletta Patricio O., Veiga Paulo. SCTP: robusto a retransmisiones espúreas, 2006.

[4] J.C.R. Bennett, C. Partridge, and N. Shectman. Packet reordering is not pathological network behavior. *IEEE/ACM Transactions on Networking (TON)*, 7(6):789–798, 1999.

[5] Y. Bernet, S. Blake, D. Grossman, and A. Smith. An informal management model for diffserv routers. *Arbeitsdokument, IETF-Diffserv Working Group*, 2001.

[6] RT Braden. RFC 1122: Requirements for Internet Hosts–Communication Layers. *Internet Requests for Comments*, 1122, 1989.

[7] A.L. Caro Jr, J.R. Iyengar, P.D. Amer, S. Ladha, and K.C. Shah. SCTP: a proposed standard for robust internet data transport. *Computer*, 36(11):56–63, 2003.

[8] L. Coene et al. RFC 3257, Stream control transmission protocol applicability statement, 2002.

[9] T. Dreibholz, A. Jungmaier, and M. Tuxen. A new scheme for IP-based Internet-mobility. In *Conference On Local Computer Networks*, volume 28, pages 99–108. IEEE, 2003.

[10] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the Internet. *IEEE/ACM Transactions on Networking (TON)*, 7(4):458–472, 1999.

[11] F. Gebali. *Analysis of computer and communication networks*. Springer Verlag, 2008.

[12] C. Hohendorf, E. Unurkhaan, and T. Dreibholz. Secure SCTP. *draft-hohendorf-secure-sctp-00. txt (work in progress)*, 2005.

[13] A. Ishtiaq, Y. Okabe, and M. Kanazawa. Improving Performance of SCTP over Broadband High Latency Networks. *IEEE LCN 2003*, 2003.

[14] JR Iyengar, PD Amer, and R. Stewart. Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths. *IEEE/ACM Transactions on networking*, 14(5):951–964, 2006.

[15] J.R. Iyengar, AL Caro, P.D. Amer, G.J. Heinz, and R.R. Stewart. Making SCTP more robust to changeover. *SIMULATION SERIES*, 35(4):37–49, 2003.

[16] J.R. Iyengar, A.L. Caro Jr, P.D. Amer, G.J. Heinz, and R. Stewart. SCTP congestion window overgrowth during changeover. *Proc. SCI2002, Orlando*, 2002.

[17] V. Jacobson. Congestion avoidance and control, Symposium proceedings on Communications architectures and protocols, 1988.

[18] A. Jayasumana, N. Piratla, A. Bare, T. Banka, R. Whitner, and J. McCollom. RFC 5236: Improved packet reordering metrics. Technical report, IETF, June 2008.

[19] H. Kamal, B. Penoff, and A. Wagner. SCTP versus TCP for MPI. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Washington, DC, USA, 2005.

[20] P. Karm and C. Partridge. Improving round-trip time estimates in reliable transport protocols. *Computer Communication Review*, 17(5):2–7, 1987.

[21] T. Kelly. Scalable TCP: Improving performance in highspeed wide area networks. *First International Workshop on Protocols for Fast Long Distance Networks*, 2003.

[22] S. Ladha, S. Baucke, R. Ludwig, and P.D. Amer. On making SCTP robust to spurious retransmissions. *ACM SIGCOMM Computer Communication Review*, 34(2):123–135, 2004.

[23] K.C. Leung, VOK Li, and D. Yang. An overview of packet reordering in transmission control protocol (TCP): problems, solutions, and challenges. *IEEE Transactions on Parallel and Distributed Systems*, 18(4):522–535, 2007.

[24] R. Ludwig and A. Gurtov. RFC 4015: The Eifel response algorithm for TCP. *Request for Comments*, February 2005.

[25] Reiner Ludwig and Randy H. Katz. The eifel algorithm: making tcp robust against spurious retransmissions. *SIGCOMM Comput. Commun. Rev.*, 30(1):30–36, 2000.

[26] JC Mogul and SE Deering. RFC 1191: Path MTU discovery. *Internet RFCs*, 1990.

[27] A. Morton, L. Ciavattone, G. Ramachandran, S. Shalunov, and J. Perser. RFC 4737: Packet reordering metrics, 2006.

[28] J.Ñagle. RFC 896: Congestion Control in IP/TCP Internetworks. *Request For Comments*, 1984.

[29] J.Ñagle. On packet switches with infinite storage. *Communications, IEEE Transactions on [legacy, pre-1988]*, 35(4):435–438, 1987.

[30] P.Ñatarajan, J.R. Iyengar, P.D. Amer, and R. Stewart. SCTP: an innovative transport layer protocol for the web. In *Proceedings of the 15th international conference on World Wide Web*, pages 615–624. ACM New York, NY, USA, 2006.

[31] V.Ñelson, J. Tania, and H. Yezekael. Performance of SCTP in Wi-Fi and WiMAX networks with multi-homed mobiles. In *Proceedings of the 3rd International Conference on Performance Evaluation Methodologies and Tools*, page 71. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[32] Vern Paxson. End-to-end internet packet dynamics. *IEEE/ACM Trans. Netw.*, 7(3):277–292, 1999.

[33] F. Perotto, C. Casetti, G. Galante, and D. di Elettronica. SCTP-based transport protocols for concurrent multipath transfer. In *IEEE Wireless Communications and Networking Conference, 2007. WCNC 2007*, pages 2969–2974, 2007.

[34] P. Sarolahti and M. Kojo. F-RTO: An Algorithm for Detecting Spurious Retransmission Timeouts with TCP and SCTP. *Work in Progress*, 2003.

[35] C.S.L. Shieh, I.C. Lai, and W. Kuang. Improvement of SCTP Performance in Vertical Handover. In *Intelligent Systems Design and Applications, 2008. ISDA'08. Eighth International Conference on*, volume 3, 2008.

[36] W. Stevens et al. RFC 2001: TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms, 1997.

[37] R. Stewart, L. Ong, I. Arias-Rodriguez, K. Poon, P. Conrad, A. Caro, and M. Tuexen. Stream control transmission protocol (SCTP) implementerŠs guide. *draft-ietf-tsvwg-sctpimpguide-10 (work in progress)*, 2003.

[38] R. Stewart, M. Ramalho, Q. Xie, and M. Tuexen. RFC 3758: Stream Control Transmission Protocol (SCTP) Partial Reliability Extension. *Request For Comments*, May 2004.

[39] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. RFC 2960: Stream Control Transmission Protocol. *RFC Editor United States*, 2000.

[40] R. Stewart, Q. Xie, M. Tuexen, S. Maruyama, and M. Kozuka. RFC 5061: Stream control transmission protocol (SCTP) dynamic address reconfiguration, 2007.

[41] A.S. Tanenbaum. *Computer networks*. Prentice Hall, 2002.

[42] `http://developer.osdl.org/dev/iproute2/`. IP routing utilities.

[43] `http://developer.osdl.org/shemminger/netem/`. Netem, Network Emulator.

[44] `http://info.iet.unipi.it/\~luigi/ip\_dummynet/`. DummyNet.

[45] `http://snad.ncsl.nist.gov/nistnet/`. NIST Net.

[46] `http://umlsim.sourceforge.net/`. UML Simulator.

[47] `http://www.die.net/doc/linux/man/man8/tc.8.html`. tc - Linux man page.

[48] `http://www.freebsd.org/`. Free BSD.

[49] `http://www.sctp.de/sctp.html`. SCTP and a prototype implementation.

[50] `http://www.tcpdump.org/`. tcpdump.

[51] M. Zhang, B. Karp, S. Floyd, and L. Peterson. RR-TCP: A reordering-robust tcp with dsack. In *11th IEEE International Conference on Network Protocols, 2003. Proceedings*, pages 95–106, 2003.

Apéndice A

Resultados

A continuación se presentan los resultados de los experimentos realizados. En los mismos se comparan los resultados de los experimentos con y sin SCTP-RR.

Demora fija más demora variable por porcentaje de reorden Se presentan los resultados de los experimentos de demora fija más demora variable por porcentaje de reorden.

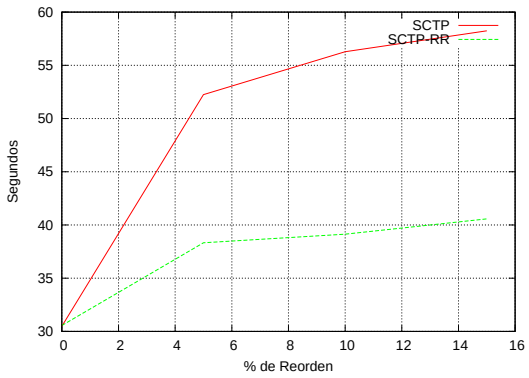


Figura A.1: Demora fija de 50 ms, demora variable de 100 ms, por tiempo

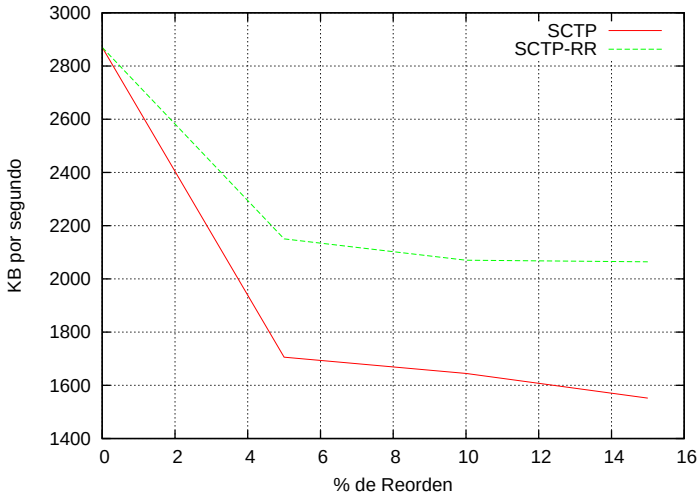


Figura A.2: Demora fija de 50 ms, demora variable de 100 ms, bytes por segundo

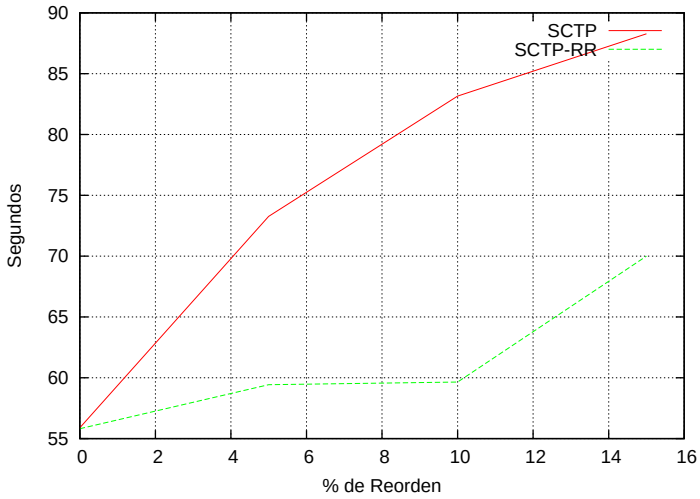


Figura A.3: Demora fija de 50 ms, demora variable de 200 ms, por tiempo

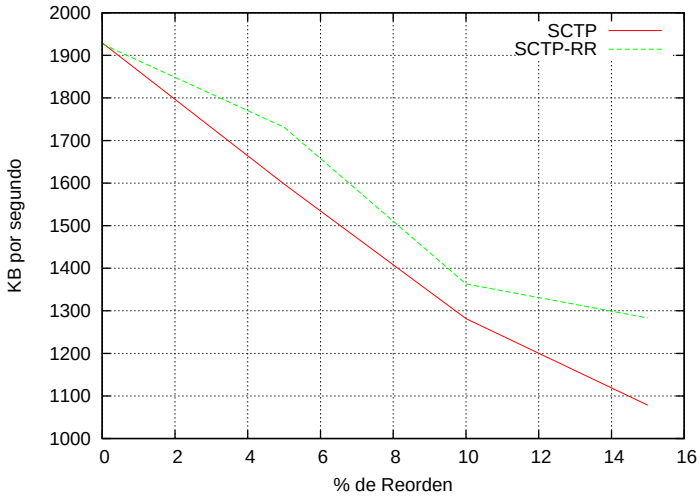


Figura A.4: Demora fija de 50 ms, demora variable de 200 ms, bytes por segundo

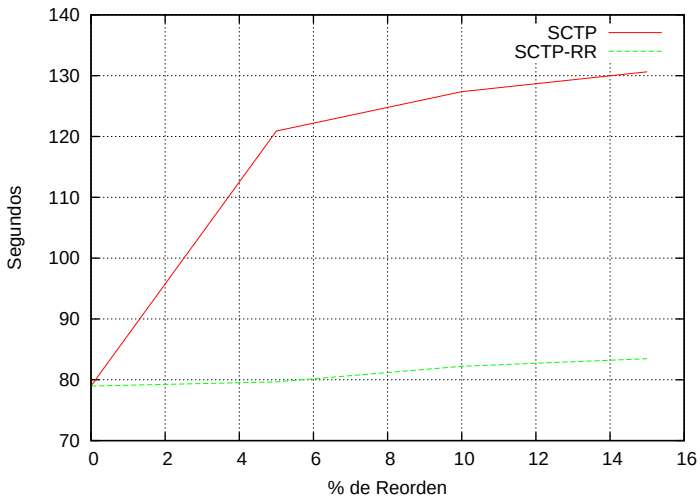


Figura A.5: Demora fija de 50 ms, demora variable de 300 ms, por tiempo

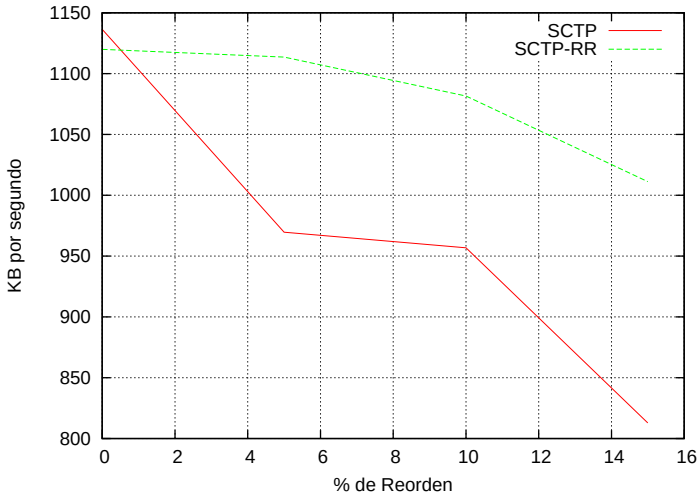


Figura A.6: Demora fija de 50 ms, demora variable de 300 ms, bytes por segundo

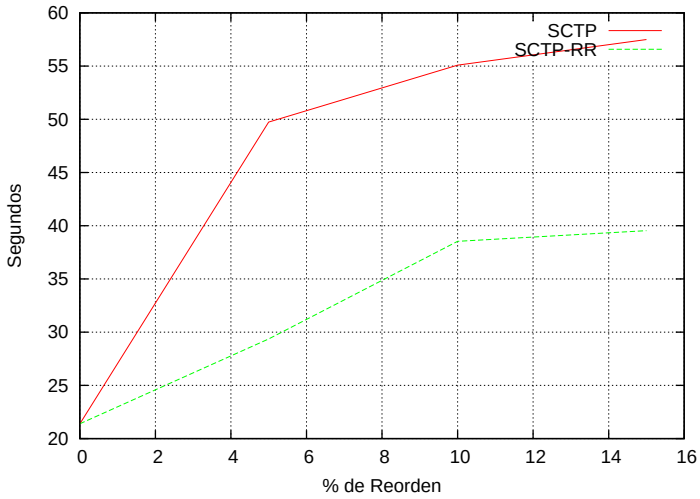


Figura A.7: Demora fija de 100 ms, demora variable de 100 ms, por tiempo

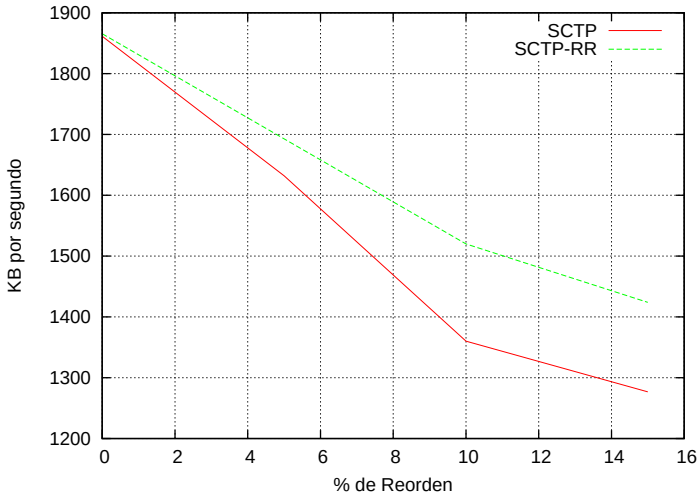


Figura A.8: Demora fija de 100 ms, demora variable de 100 ms, bytes por segundo

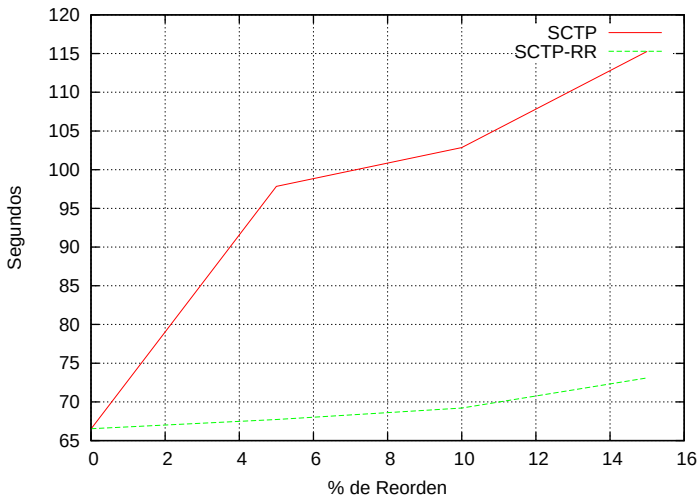


Figura A.9: Demora fija de 100 ms, demora variable de 200 ms, por tiempo

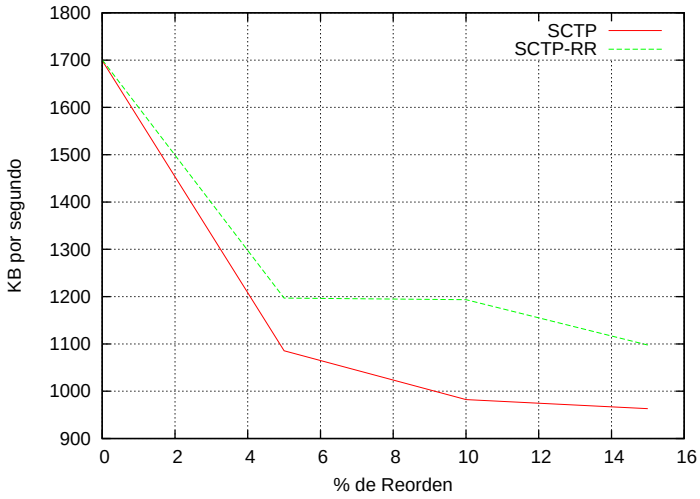


Figura A.10: Demora fija de 100 ms, demora variable de 200 ms, bytes por segundo

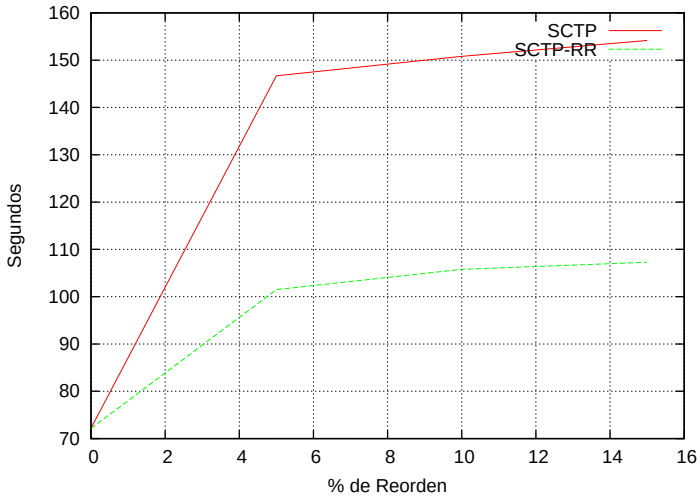


Figura A.11: Demora fija de 100 ms, demora variable de 300 ms, por tiempo

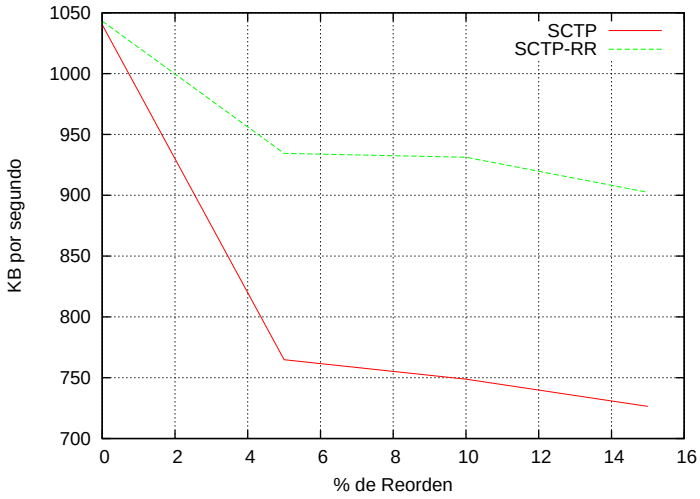


Figura A.12: Demora fija de 100 ms, demora variable de 300 ms, bytes por segundo

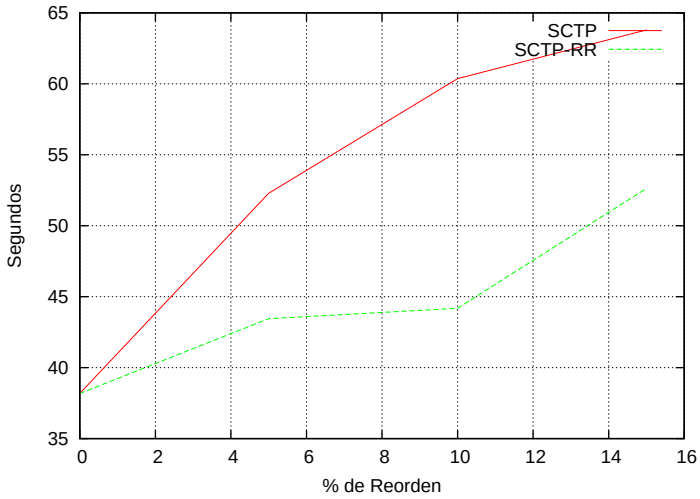


Figura A.13: Demora fija de 200 ms, demora variable de 100 ms, por tiempo

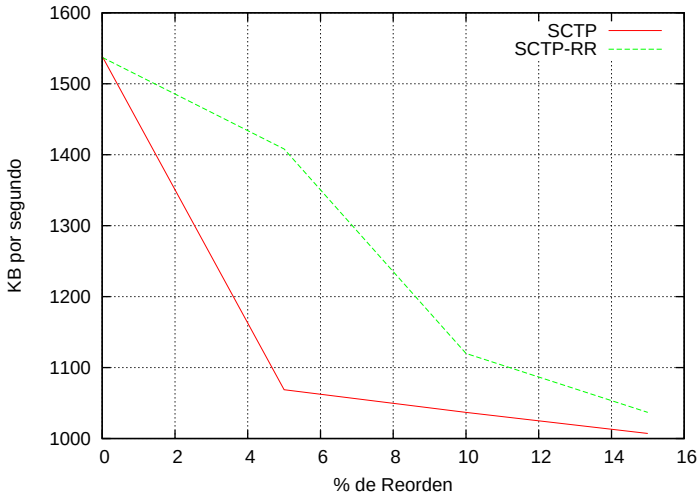


Figura A.14: Demora fija de 200 ms, demora variable de 100 ms, bytes por segundo

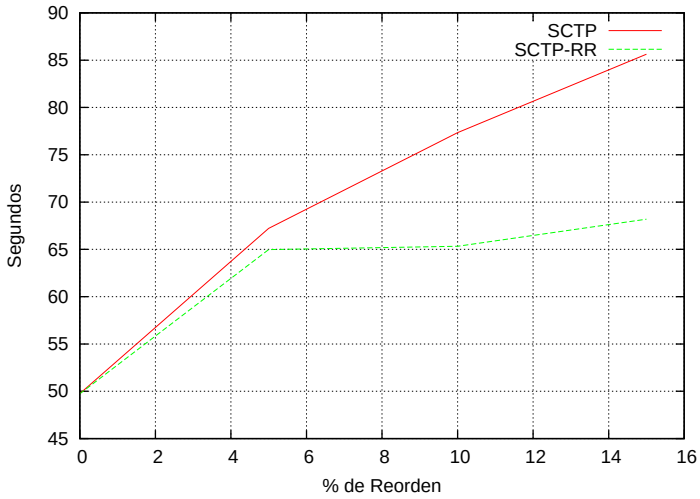


Figura A.15: Demora fija de 200 ms, demora variable de 200 ms, por tiempo

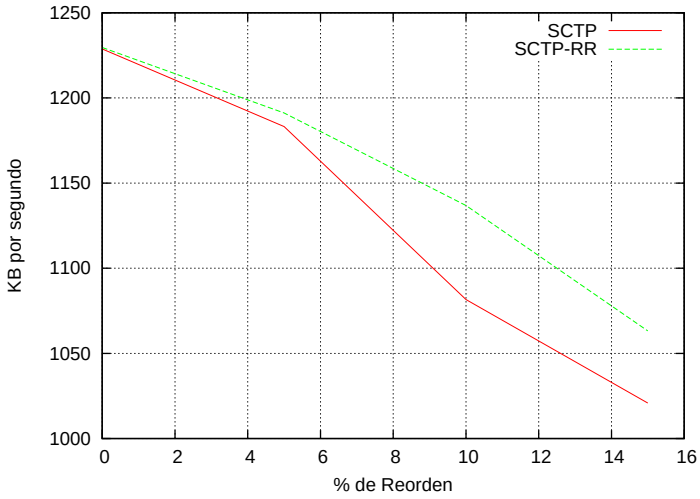


Figura A.16: Demora fija de 200 ms, demora variable de 200 ms, bytes por segundo

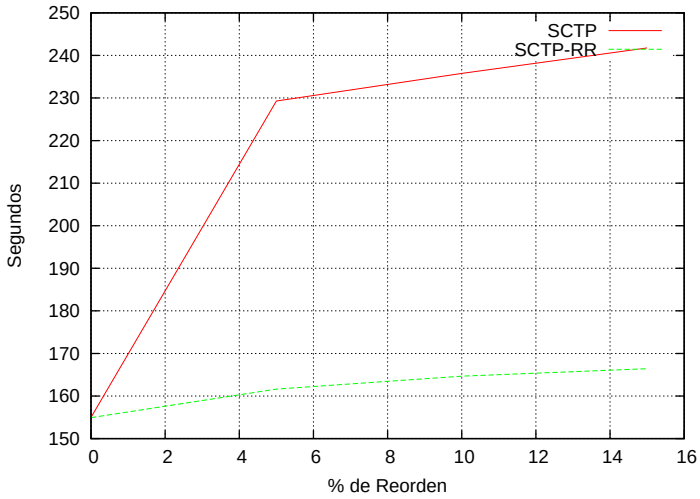


Figura A.17: Demora fija de 200 ms, demora variable de 300 ms, por tiempo

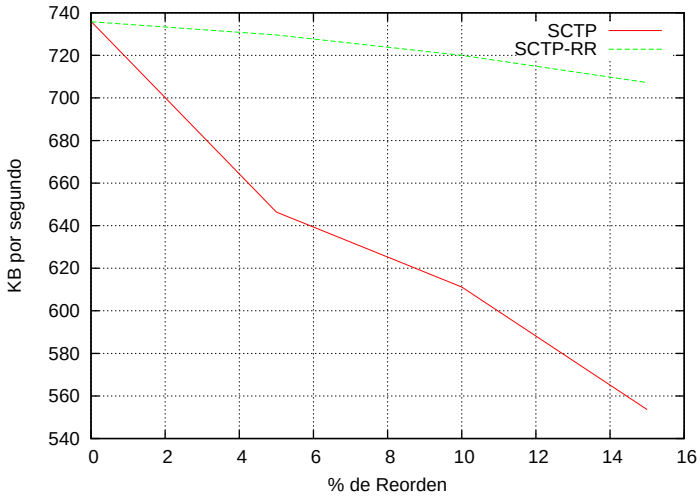


Figura A.18: Demora fija de 200 ms, demora variable de 300 ms, bytes por segundo

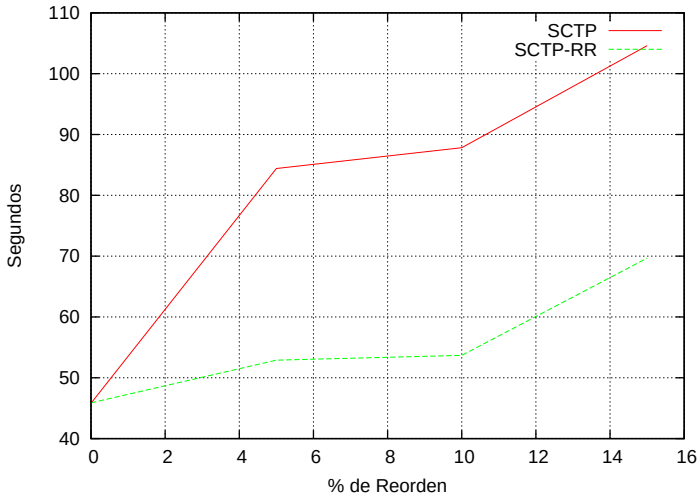


Figura A.19: Demora fija de 300 ms, demora variable de 100 ms, por tiempo

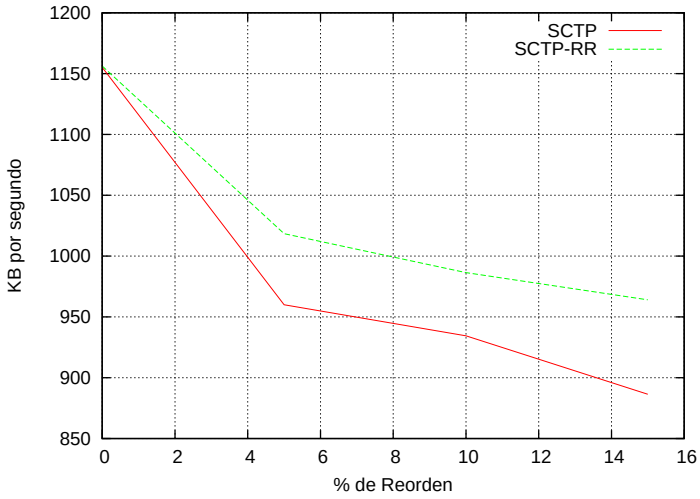


Figura A.20: Demora fija de 300 ms, demora variable de 100 ms, bytes por segundo

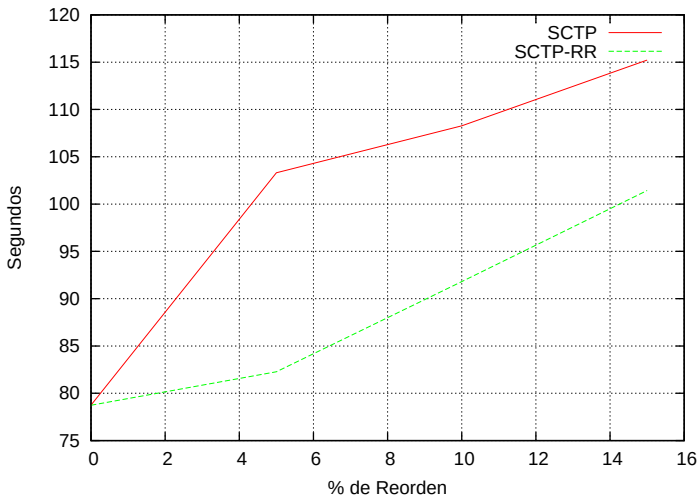


Figura A.21: Demora fija de 300 ms, demora variable de 200 ms, por tiempo

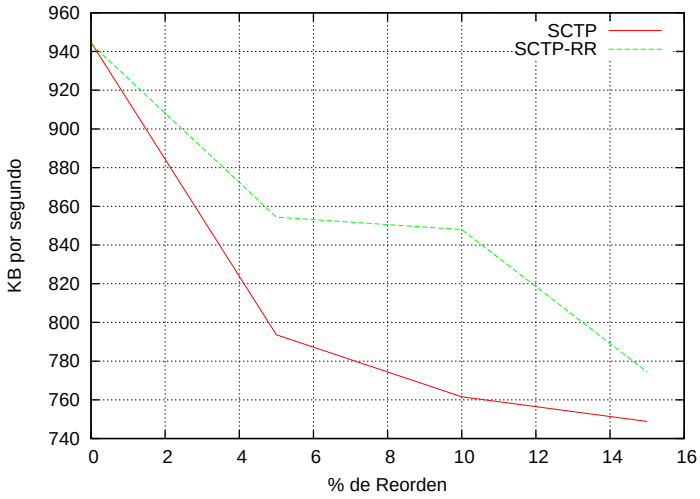


Figura A.22: Demora fija de 300 ms, demora variable de 200 ms, bytes por segundo

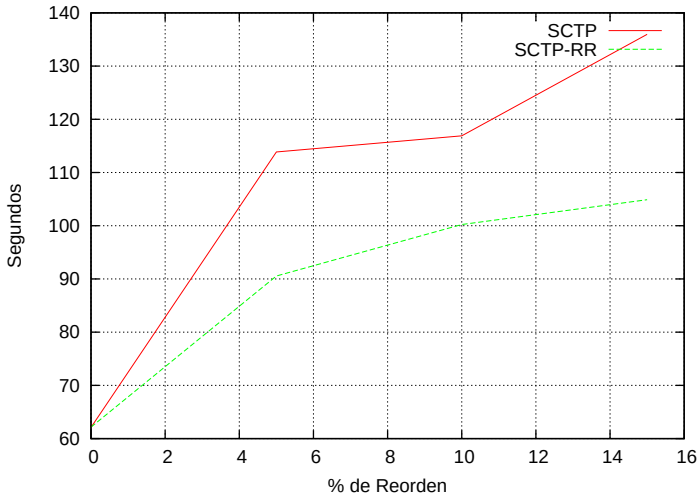


Figura A.23: Demora fija de 300 ms, demora variable de 300 ms, por tiempo

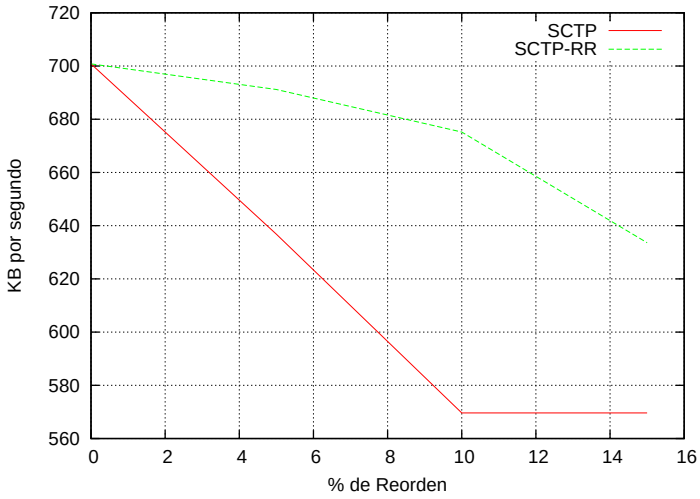


Figura A.24: Demora fija de 300 ms, demora variable de 300 ms, bytes por segundo

Demora fija más demora variable Se presentan los resultados de los experimentos de demora fija más demora variable.

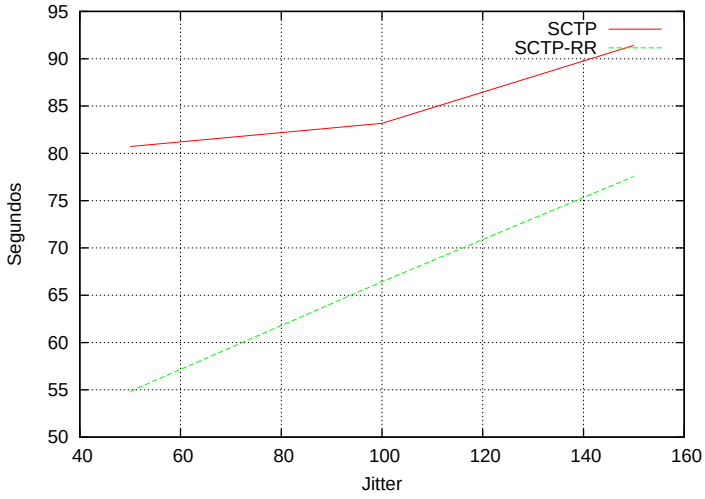


Figura A.25: Demora fija de 100 ms, demora variable de 50 ms, por tiempo

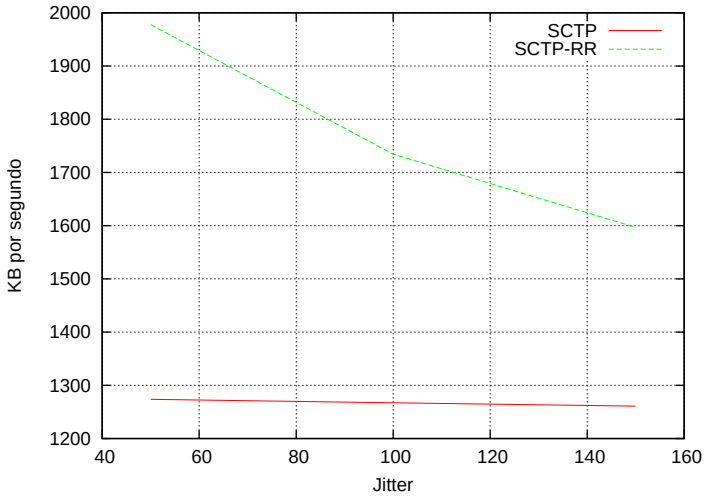


Figura A.26: Demora fija de 100 ms, demora variable de 50 ms, bytes por segundo

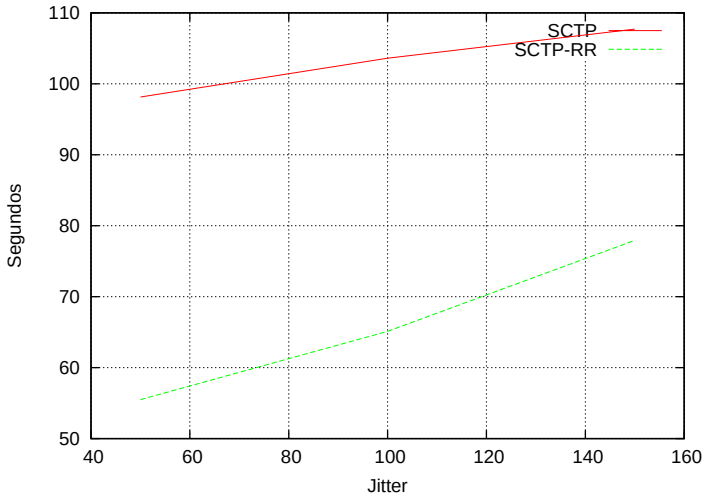


Figura A.27: Demora fija de 100 ms, demora variable de 100 ms, por tiempo

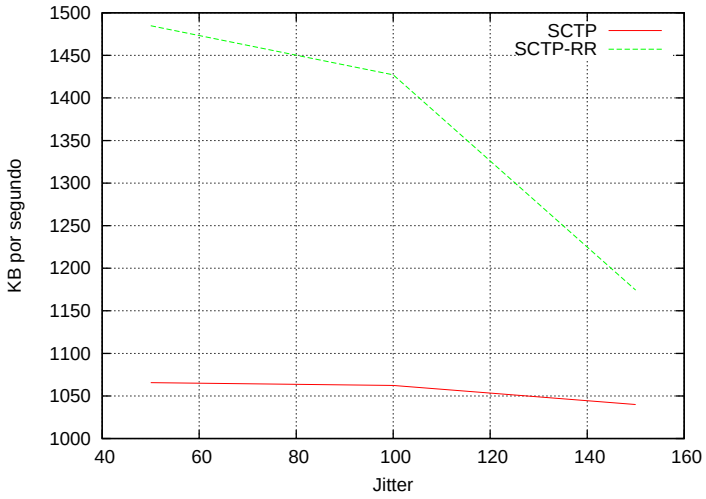


Figura A.28: Demora fija de 100 ms, demora variable de 100 ms, bytes por segundo

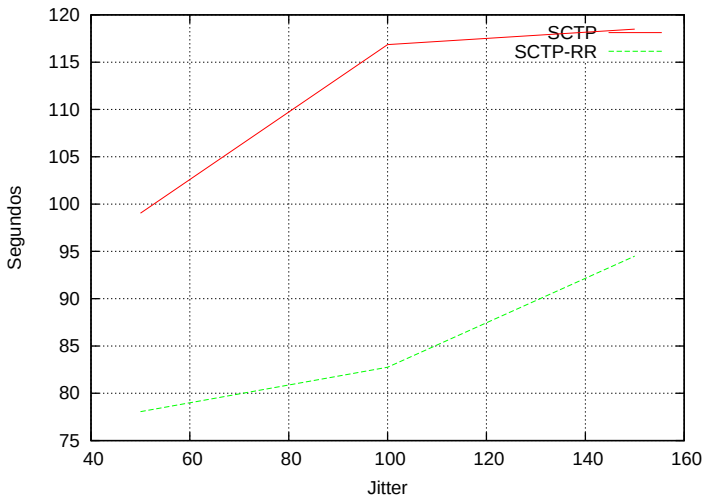


Figura A.29: Demora fija de 100 ms, demora variable de 150 ms, por tiempo

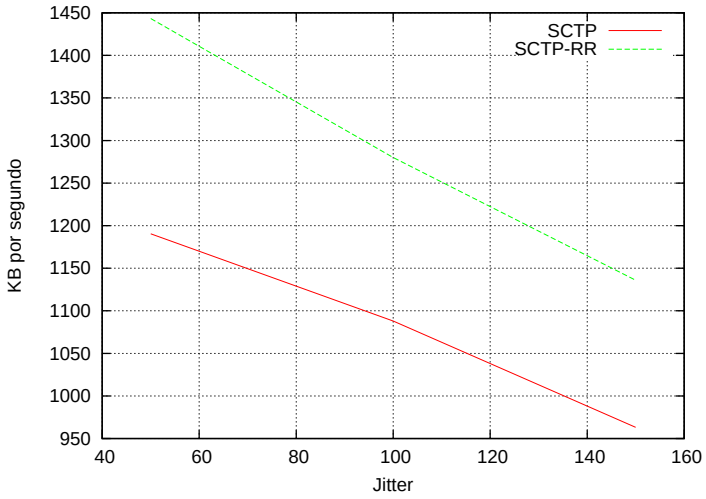


Figura A.30: Demora fija de 100 ms, demora variable de 150 ms, bytes por segundo

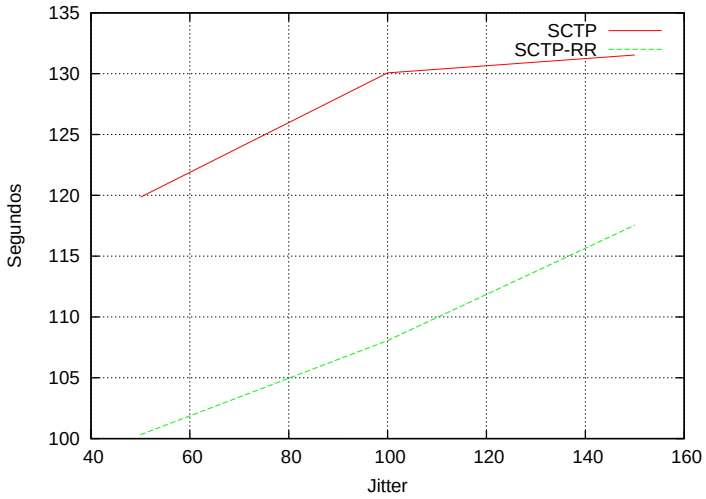


Figura A.31: Demora fija de 200 ms, demora variable de 50 ms, por tiempo

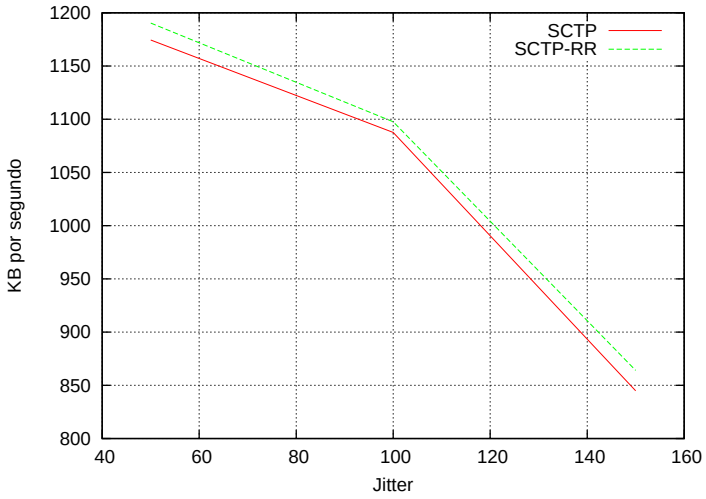


Figura A.32: Demora fija de 200 ms, demora variable de 50 ms, bytes por segundo

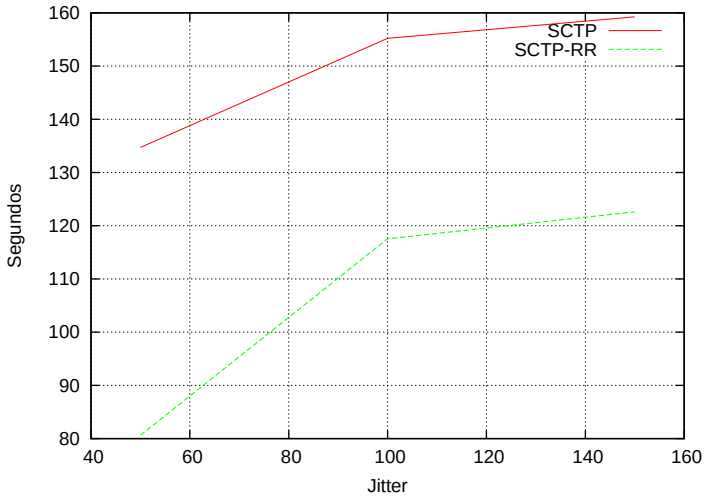


Figura A.33: Demora fija de 200 ms, demora variable de 100 ms, por tiempo

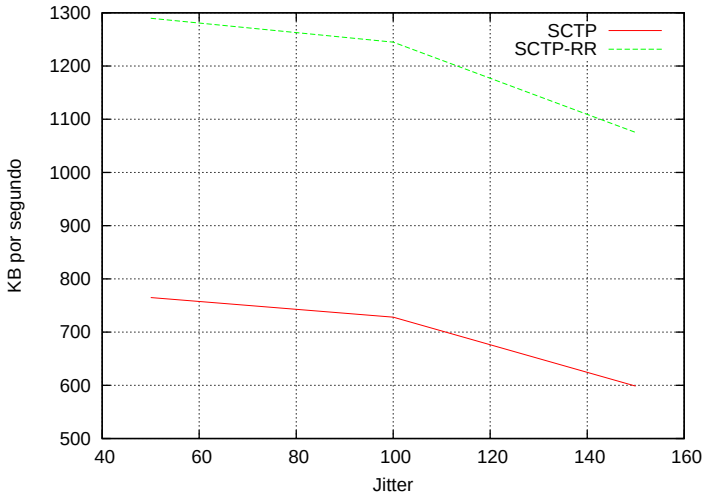


Figura A.34: Demora fija de 200 ms, demora variable de 100 ms, bytes por segundo

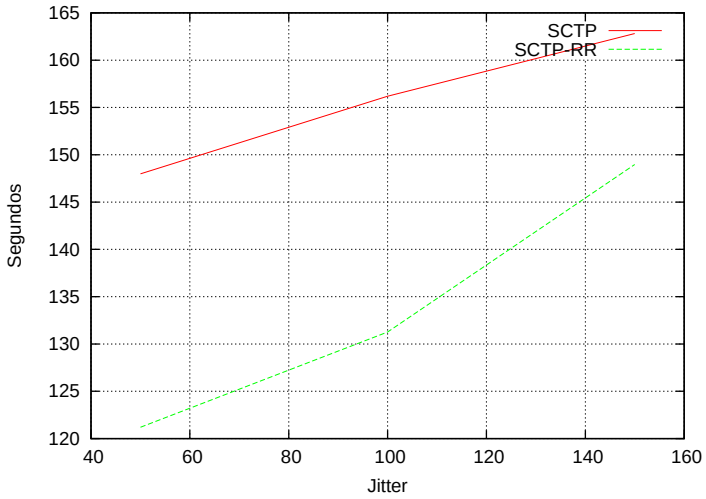


Figura A.35: Demora fija de 200 ms, demora variable de 150 ms, por tiempo

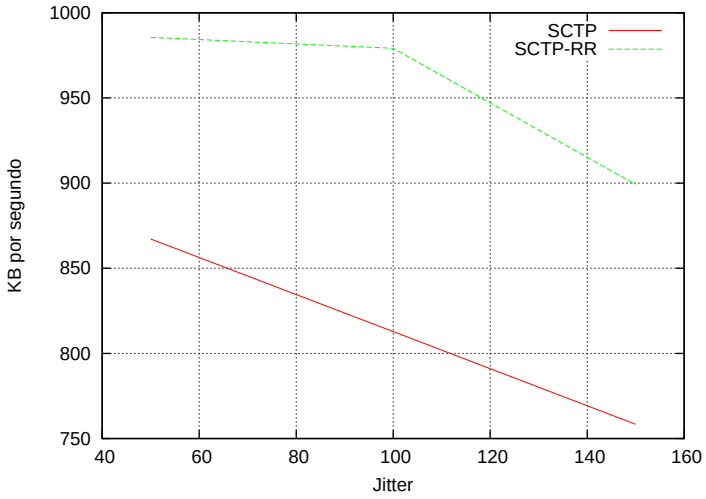


Figura A.36: Demora fija de 200 ms, demora variable de 150 ms, bytes por segundo

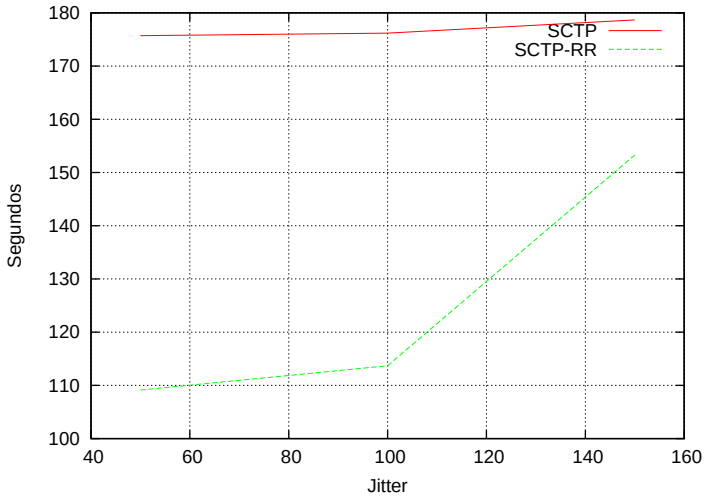


Figura A.37: Demora fija de 300 ms, demora variable de 50 ms, por tiempo

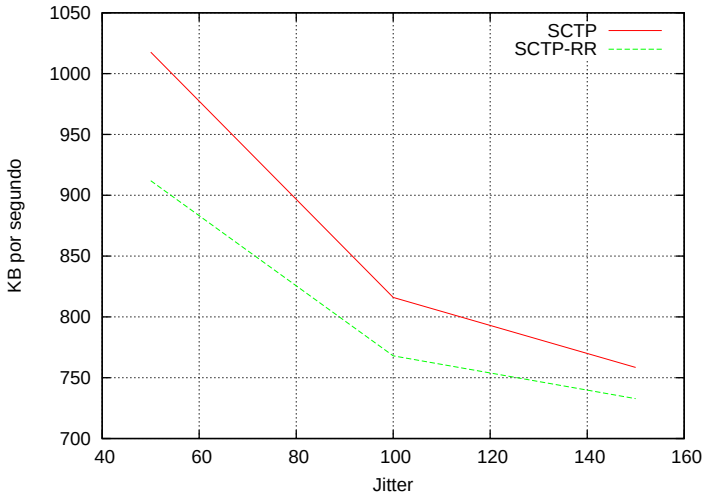


Figura A.38: Demora fija de 300 ms, demora variable de 50 ms, bytes por segundo

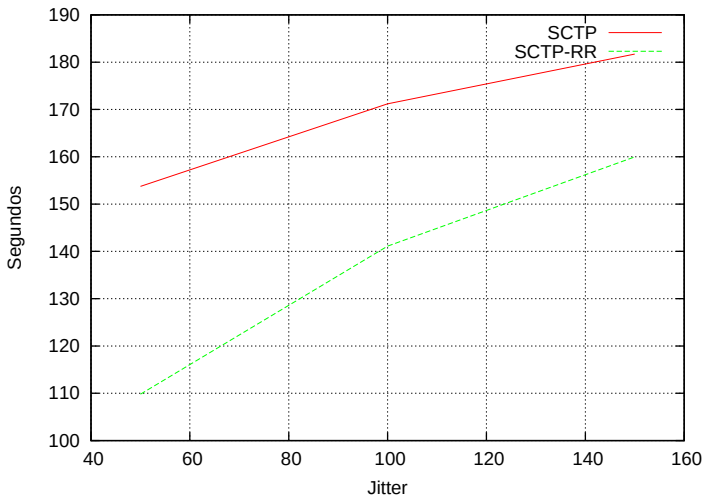


Figura A.39: Demora fija de 300 ms, demora variable de 100 ms, por tiempo

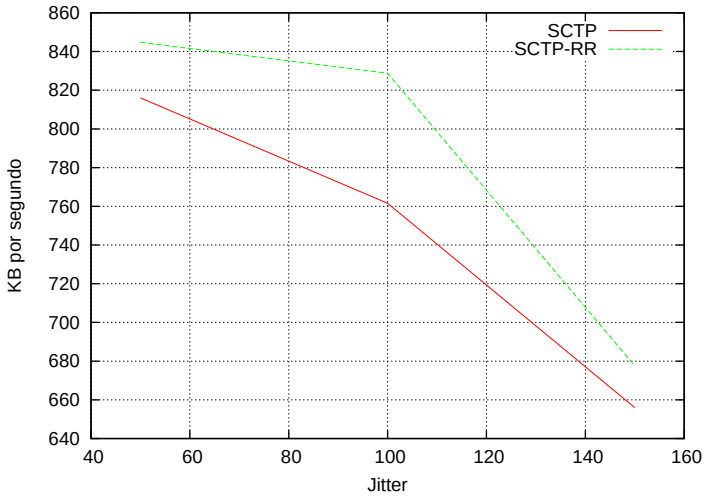


Figura A.40: Demora fija de 300 ms, demora variable de 100 ms, bytes por segundo

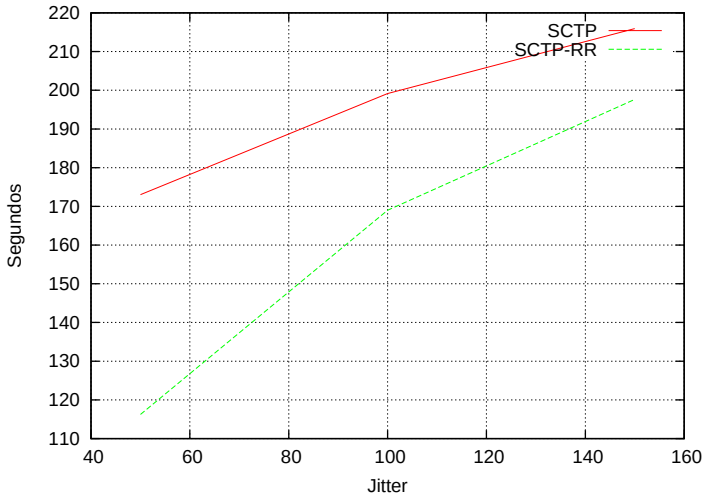


Figura A.41: Demora fija de 300 ms, demora variable de 150 ms, por tiempo

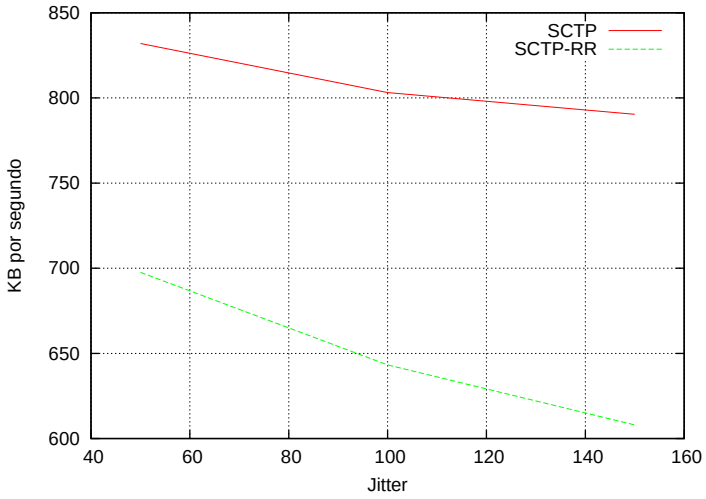


Figura A.42: Demora fija de 300 ms, demora variable de 150 ms, bytes por segundo

Demoras fijas por cambio de tiempo Se presentan los resultados de los experimentos de demora fija por cambio de tiempo.

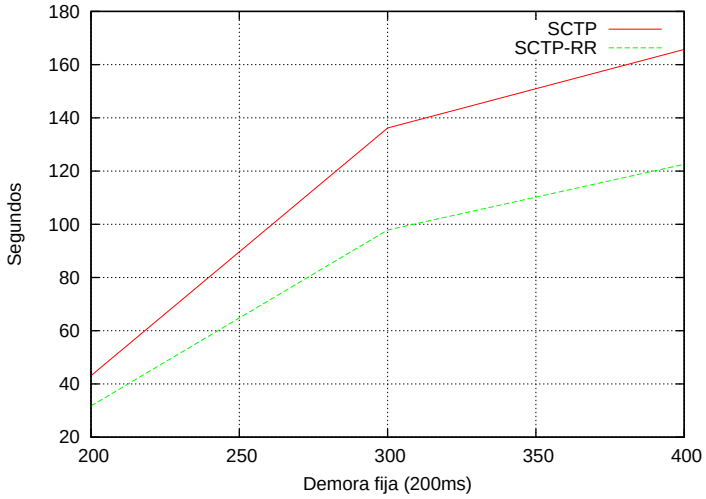


Figura A.43: Tiempo de cambio 250 ms, por tiempo

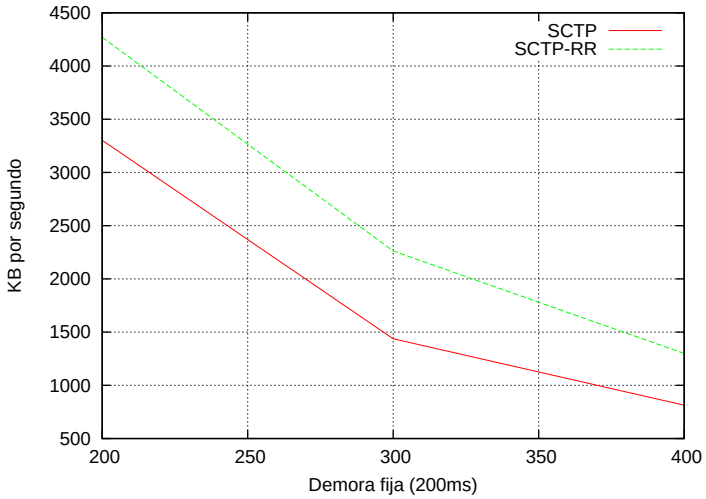


Figura A.44: Tiempo de cambio 250 ms, bytes por segundo

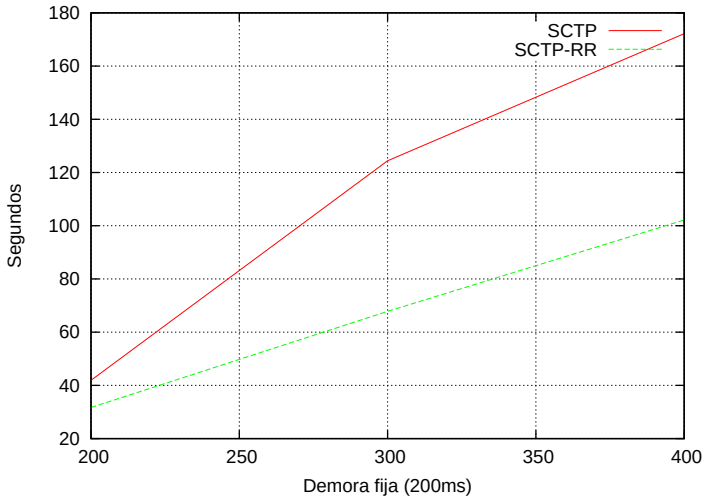


Figura A.45: Tiempo de cambio 500 ms, por tiempo

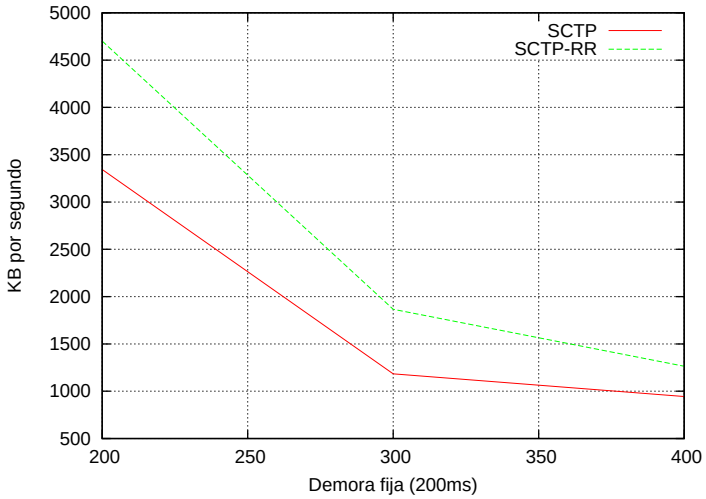


Figura A.46: Tiempo de cambio 500 ms, bytes por segundo

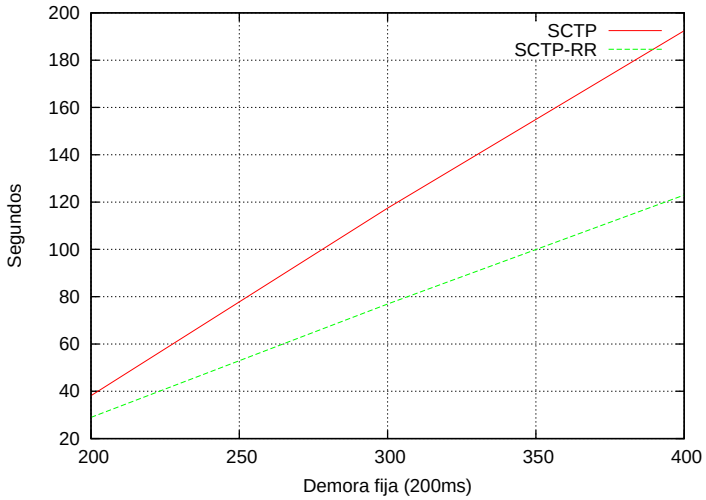


Figura A.47: Tiempo de cambio 750 ms, por tiempo

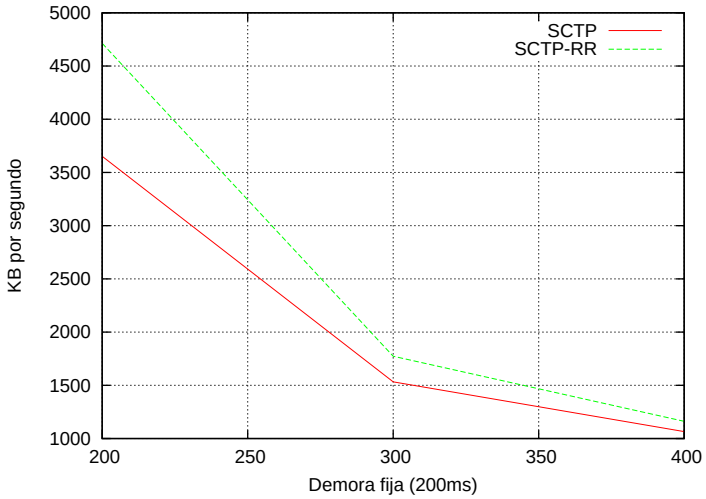


Figura A.48: Tiempo de cambio 750 ms, bytes por segundo

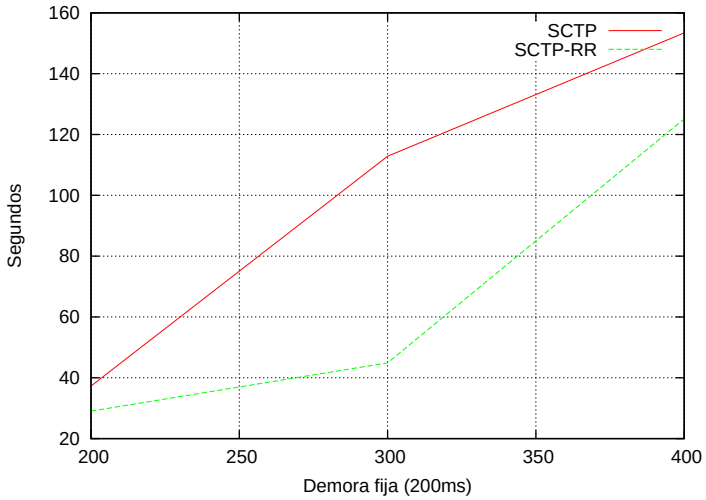


Figura A.49: Tiempo de cambio 1000 ms, por tiempo

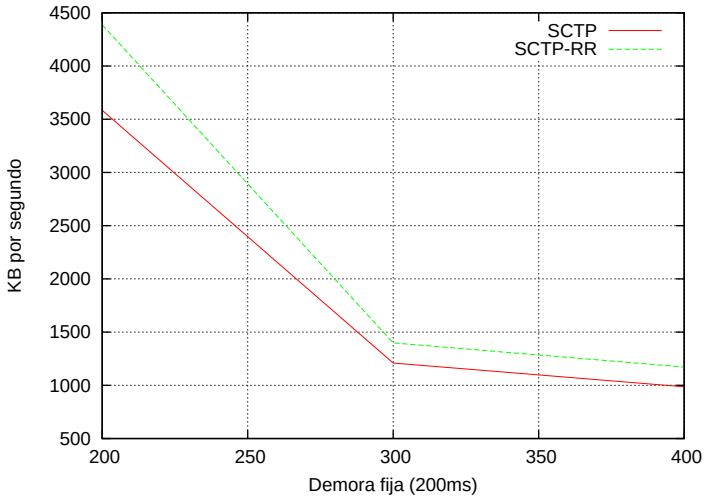


Figura A.50: Tiempo de cambio 1000 ms, bytes por segundo

Apéndice B

Disciplina de colas (*qdisc*)

Desde la versión, Linux incorporó todas las herramientas necesarias para gestionar el ancho de banda en formas comparables a los sistemas dedicados de alto nivel para gestión de ancho de banda. Linux va incluso más allá de lo que proporcionan Frame Relay y ATM.

Se utilizarán las siguientes reglas para la especificación de ancho de banda:

```
mbps = 1024 kbps = 1024 * 1024 bps => byte/s mbit = 1024 kbit =>
kilobit/s. mb = 1024 kb = 1024 * 1024 b => byte mbit = 1024 kbit =>
kilobit.
```

Al imprimir las tasas se usa la siguiente regla:

```
1Mbit = 1024 Kbit = 1024 * 1024 bps => byte/s
```

B.1. Colas y disciplinas de cola explicadas

Con el encolamiento se determina la manera en que se *envían* los datos. Es importante tener en cuenta de que sólo se puede dar forma a los datos que se transmiten.

Es conocido que Internet se basa en su mayoría en TCP/IP, y tiene algunas características que son utilidad. Sin embargo, TCP/IP no tiene manera de saber la capacidad de la red entre dos host, de manera que simplemente empieza a enviar datos más y más rápido (Slow Start) y cuando empieza a perder paquetes, porque no hay espacio en los buffers de los routers para enviarlos, reduce la marcha.

Si se tiene un router y se desea evitar que ciertas máquinas dentro de una red descarguen demasiado rápido, es necesario dar *forma* a la interfaz "interna"del router, la que envía los datos a las computadoras.

También hay que asegurar de que se controla el cuello de botella del enlace. Si se tiene una Network Interface Card (NIC) de 100Mbit y un router con un enlace de 256kbit, hay que asegurar de que no envían más datos de los que el router puede manejar. De otra manera, será el router el que controle el enlace y ajuste el ancho de banda disponible.

B.2. Disciplinas de cola simples, sin clases

Las *disciplinas de cola sin clases* son aquellas que, mayormente, aceptan datos y se limitan a reordenarlos, retrasarlos, o descartarlos.

Esto se puede usar para ajustar el tráfico de una interfaz entera, sin subdivisiones.

La disciplina más usada para lograr este objetivo, es la *qdisc pfifo_fast* y se usa por defecto. Cada una de estas colas tiene tanto puntos fuertes como debilidades específicos.

B.2.1. pfifo_fast

Esta cola es, como su nombre indica, *First In, First Out* (*FIFO*, .[el] primero que entra es el primero que sale"), lo que significa que ningún paquete recibe un tratamiento especial. Esta cola tiene 3 *bandas* numeradas. Dentro de cada *banda*, se aplican las reglas FIFO. Sin embargo, no se procesará la banda 1 mientras existan paquetes esperando salir en la banda 0. Lo mismo se aplica para las bandas 1 y 2.

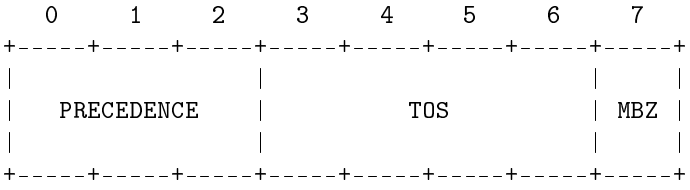
El kernel de Linux obedece la marca llamada *Type Of Service* (TOS, tipo de servicio) que hay en los paquetes, y tiene cuidado de insertar los paquetes de *mínimo retraso* en la banda 0.

No se debe confundir ésta disciplina de cola sin clases con una disciplina de cola con clases PRIO. Si bien comparten similaridades, pfifo_fast es sin clases y no se pueden agregar otras qdisc a la misma. Esto será explicado con mayor detenimiento abajo.

B.2.2. Parámetros y forma de uso

La qdisc pfifo_fast no se puede configurar ya que es la cola por defecto y sus parámetros están fijos. Ésta es la configuración por defecto:

priomap Determina cómo se corresponden las prioridades de los paquetes, tal como las asigna el kernel, a las bandas. El cuadro B.1 correspondencia se basa en el octeto TOS del paquete.



Cuadro B.1: Octeto TOS de un paquete

Los cuatro bits TOS, llamado *campo TOS*, están definidos en la tabla B.2.

Binario	Decimal	Significado
1000	8	Minimizar retraso (md)
0100	4	Maximizar transferencia (mt)
0010	2	Maximizar fiabilidad (mr)
0001	1	Minimizar el coste monetario (mmc)
0000	0	Servicio normal

Cuadro B.2: Cuatro bits del TOS

Como hay 1 bit a la derecha de estos cuatro, el valor real del campo TOS es el doble del valor de los bits TOS. `tcpdump -v -v`[1] muestra el valor del campo TOS completo, no sólo

[1] *tcpdump* [50] es un herramienta que imprime la cabecera de los paquetes de una NIC que cumplan con cierta condición. Puede almacenar en un archivo los paquetes, con los datos, para un posterior análisis, y también leer paquetes de un archivo, en vez de una red.

TOS	Bits	Significado	Prioridad de Linux	Banda
0x0	0	Servicio normal	0 Mejor esfuerzo	1
0x2	1	Minimizar coste monetario	1 Relleno	2
0x4	2	Maximizar fiabilidad 0 Mejor esfuerzo	1	
0x6	3	mmc+mr 0 Mejor esfuerzo	1	
0x8	4	Maximizar transferencia	2 En masa	2
0xa	5	mmc+mt	2 En masa	2
0xc	6	mr+mt	2 En masa	2
0xe	7	mmc+mr+mt 2 En masa	2	
0x10	8	Minimizar retrasos	6 Interactivo	0
0x12	9	mmc+md	6 Interactivo	0
0x14	10	mr+md	6 Interactivo	0
0x16	11	mmc+mr+md	6 Interactivo	0
0x18	12	mt+md	4 Interactivo en masa	1
0x1a	13	mmc+mt+md	4 Interactivo en masa	1
0x1c	14	mr+mt+md	4 Interactivo en masa	1
0x1e	15	mmc+mr+mt+md	4 Interactivo en masa	1

Cuadro B.3: Valor del campo TOS

sus cuatro bits. Esto es el valor que aparece en la primera columna de la tabla B.3.

La segunda columna contiene el valor de los cuatro bits TOS relevantes, seguidos por su significado traducido. Por ejemplo, el 15 significa que un paquete espera un *Mínimo Coste Monetario*, la *Máxima Fiabilidad*, la *Máxima Transferencia* y un *Retraso Mínimo*. La cuarta columna indica la manera en que el kernel de Linux interpreta los bits del TOS, mostrando cual prioridad les asigna. La última columna indica el resultado del priomap por defecto. En la línea de comando, el priomap por defecto se parece a:

```
1, 2, 2, 2, 1, 2, 0, 0 , 1, 1, 1, 1, 1, 1, 1, 1
```

Esto significa que la prioridad 4, por ejemplo, se asigna a la banda número 1. priomap también permite listar prioridades mayores (> 7) que no se corresponden a asignaciones del TOS, sino que se configuran por otros medios.

La tabla B.4 del RFC 1349 [2] indica cómo las aplicaciones deberían setear los bits del TOS:

txqueuelen La longitud de esta cola se obtiene de la configuración de la interfaz, que puede ver y modificar con ifconfig o ip[2]. Para establecer la longitud de la cola en 10, se debe ejecutar:

```
# ifconfig eth0 txqueuelen 10
```

B.2.3. Token Bucket Filter

El *Token Bucket Filter* (TBF) es un qdisc sencillo que se limita a dejar pasar paquetes que lleguen a una tasa que no exceda a la configurada, pero con la posibilidad de permitir ráfagas cortas que excedan esta tasa. TBF es muy preciso, amigable para la red y el procesador.

[2]ifconfig e ip son herramientas de Linux que permiten configurar los NIC. Es utilizado en tiempo de arranque del sistema para configurar los NIC como fuese necesario.

```
TELNET                1000 (minimizar retraso) FTP
    Control           1000 (minimizar retraso)
    Datos             0100 (maximizar transferencia)
TFTP                  1000 (minimizar retraso) SMTP
    Fase de órdenes   1000 (minimizar retraso)
    Fase de datos     0100 (maximizar transferencia)
Domain Name Service
    Consulta UDP      1000 (minimizar retraso)
    Consulta TCP      0000
    Transf. de zona   0100 (maximizar transferencia)
NNTP                  0001 (minimizar coste monetario) ICMP
    Errores           0000
    Peticiones        0000 (la mayoría)
Respuestas            <igual que las peticiones> (la mayoría de la
veces)
```

Cuadro B.4: Bits del TOS según el RFC 1349

La implementación de TBF consiste en un buffer (el *bucket* o *balde*), que se llena constantemente con piezas virtuales de información denominadas *tokens*, a una tasa específica, *token rate*. El parámetro más importante del bucket es su tamaño, que es el número de tokens que puede almacenar.

Cada token que llega toma un paquete de datos entrante de la cola de datos y se elimina del bucket. Asociar este algoritmo con los dos flujos (tokens y datos) nos da tres posibles escenarios:

- Los datos llegan a TBF a una tasa que es igual a la de tokens entrantes. En este caso, cada paquete entrante tiene su token correspondiente y pasa a la cola sin retrasos.
- Los datos llegan al TBF a una tasa menor a la de los token. Sólo una parte de los tokens se borran con la salida da cada paquete que se envía fuera de la cola, de manera que se acumulan los tokens, hasta llenar el bucket. Los tokens sin usar se pueden utilizar para enviar datos a velocidades mayores de la tasa de tokens, en cuyo caso se produce una corta ráfaga de datos.
- Los datos llegan al TBF a una tasa mayor a la de los token. Esto significa que el bucket se quedará pronto sin tokens, lo que causará que TBF se acelere a sí mismo por un rato. Esto se llama una *situación sobrelímite*. Si siguen llegando paquetes, empezarán a ser descartados.

Este último escenarios es muy importante, porque permite ajustar administrativamente al ancho de banda disponible a los datos que están pasando por el filtro.

La acumulación de tokens permite ráfagas cortas de datos extralimitados para que pasen sin pérdidas, pero cualquier sobrecarga restante causará que los paquetes se vayan retrasando constantemente, y al final sean descartados.

En la actual implementación, los tokens corresponden a bytes, no a paquetes.

B.2.4. Parámetros y uso

Aunque probablemente no sea necesario realizar cambios, TBF tiene algunos controles que son ajustables. Primero, los parámetros que estarán disponibles siempre:

limit o latency limit es el número de bytes que pueden ser encolados a la espera de que haya tokens disponibles. También se puede especificar esto estableciendo el parámetro latency, que indica el máximo período 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 peakrate (la tasa de picos, de estar configurada).

burst / buffer / maxburst Estos parámetros son el tamaño del bucket, en bytes. Es la cantidad máxima de bytes para los que pueden haber tokens disponibles de forma instantáneamente. En general, grandes tasas precisan grandes buffers. Si el buffer es demasiado pequeño, se descartarán paquetes debido a que llegarán más tokens, por tick del reloj interno de Linux, de los que caben en el bucket.

mpu Un paquete de tamaño cero no usa cero ancho de banda. En ethernet, ningún paquete usa menos de 64 bytes[3]. El *Minimum Packet Unit* (MPU) determina el tamaño mínimo de tokens por paquete.

rate Es el ajuste de la velocidad.

Si el bucket contiene tokens y se le permite estar vacío, por defecto tendrá velocidad infinita. Si esto no es aceptable, se deberá utilizar los siguientes parámetros:

peakrate Si hay tokens disponibles, y llegan paquetes, por defecto se envían inmediatamente. Esto puede no ser lo que se desea, especialmente si se tiene un bucket grande.

La tasa de picos se puede usar para especificar cuan rápido se le permite al bucket vaciarse. Esto se consigue enviando un paquete, y esperando después lo suficiente antes de enviar el siguiente. Se deberán realizar operaciones para calcular la espera de manera que se envíen justo a la tasa de picos (peakrate). Sin embargo, debido a la resolución por defecto de 10ms del reloj interno de Linux, con paquetes de 10.000 bits de tamaño promedio, se tiene una limitación de una tasa de picos de 1mbit/s

mtu / minburst La tasa de picos de 1mbit/s no es muy útil si la tasa normal es mayor que ésa. Es posible tener una tasa de picos mayor enviando más paquetes por fracción del reloj interno, esto significa efectivamente que se ha creado un segundo bucket. Sin embargo, este segundo bucket contiene por defecto un único paquete, por lo que no es un bucket en sí realmente.

Para calcular la máxima tasa de picos posible, se debe multiplicar la mtu configurada por 100 (o más correctamente, por HZ; que en Intel es 100, y en Alpha es 1024).

B.2.5. Configuración de ejemplo

La siguiente es una configuración muy sencilla, con una cola TBF con una tasa de 220kbit, una demora de 50ms y una ráfaga de 1540 bytes:

```
# tc qdisc add dev eth0 root tbf rate 220kbit latency 50ms burst
1540
```

El comando tc se explica en detalle más abajo. Por el momento, tc es una herramienta de Linux que permite la configuración de las colas qdisc, y visualizar dicha configuración.

[3] Hay que tener en cuenta el tamaño del header del paquete

B.2.6. Stochastic Fairness Queueing

La *Stochastic Fairness Queueing* (SFQ) es una implementación sencilla de la familia de algoritmos de *colas justas* (fair queueing). Es menos precisa que las otras colas, pero también necesita realizar menos cálculos que las demás y seguir siendo una *cola justa*.

La palabra clave en SFQ es *conversación* (o *flujo*), que se corresponde en su mayoría a una sesión TCP o a un flujo UDP. El tráfico se divide en un número bastante grande de colas FIFO, una por cada conversación. Entonces el tráfico se envía de una manera parecida a *round robin*, dando a cada sesión por turnos la oportunidad de enviar datos.

Esto lleva a un comportamiento bastante equitativo y evita que una única conversación ahogue a las demás. SFQ se llama *estocástica* porque realmente no crea una cola para cada sesión, sino que tiene un algoritmo que divide el tráfico en un número limitado de colas usando un algoritmo de *hash* (troceo).

Debido al hash, varias sesiones pueden terminar en el mismo bucket, lo que dividirá por dos las posibilidades de cada sesión de enviar un paquete, reduciendo a la mitad de esta forma la velocidad efectiva disponible. Para evitar que esta situación acabe siendo detectable, SFQ cambia a menudo su algoritmo hash de manera que dos sesiones sólo colisionen durante unos pocos segundos.

Es importante tener en cuenta que SFQ sólo es útil en caso de que la NIC real de salida esté realmente llena, de no ser así, la máquina Linux no encolará paquetes y no se producirá efecto alguno.

Específicamente, configurar SFQ sobre una interfaz ethernet que esté conectada a un cable módem o a un router DSL, no tiene sentido si no se hace algún ajuste más.

B.2.7. Parámetros y uso

SFQ es mayormente auto-ajustable:

perturb Reconfigura el hash una vez cada *n* segundos. Si no se indica, el hash no se reconfigurará nunca, lo cual no es recomendable. 10 segundos es probablemente un buen valor.

quantum Cantidad de bytes de un flujo que se permite sacar de la cola antes de que le toque el turno a la siguiente cola. Por defecto es 1 paquete de tamaño máximo (tamaño MTU). No se debe colocar por debajo del mtu.

limit El número total de paquetes que serán encolados por ésta SFQ, después empezará a descartarlos.

B.2.8. Configuración de ejemplo

Si se tiene un dispositivo que tenga velocidad de enlace y tasa actual disponible idénticas, como una línea telefónica, esta configuración ayudará a mejorar la equitatividad:

```
# tc qdisc add dev eth0 root sfq perturb 10 # tc -s -d qdisc ls

qdisc sfq 800c: dev eth0 quantum 1514b limit 128p flows 128/1024
perturb 10sec
 Sent 4812 bytes 62 pkts (dropped 0, overlimits 0)
```

El número *800c:* es el número del manejador asignado de manera automática, limit indica que pueden haber 128 paquetes en esta cola. Hay 1024 hashbuckets disponibles para contabilidad, de los cuales puede haber 128 activos al mismo tiempo (*no caben más paquetes en la cola*) Los hashs se reconfiguran cada 10 segundos.

B.3. Terminología

Para entender correctamente configuraciones más complicadas se necesita explicar primero algunos conceptos. Dada complejidad del tema en cuestión, se suelen utilizar diferentes palabras para decir lo mismo.

A continuación se exponen una serie de términos tomados libremente de [5]:

Disciplina de colas (qdisc) Un algoritmo que controla la cola de un dispositivo, sea de entrada (ingress) o de salida (egress).

qdisc raíz (root qdisc) La qdisc raíz es la que está adjunta al dispositivo.

qdisc sin clases Una qdisc sin subdivisiones internas configurables.

qdisc con clases Una qdisc con clases contiene múltiples clases. Algunas de ellas contienen otras qdisc, que a su vez pueden ser con clases. De acuerdo con la definición estricta, pfifo_fast *es* con clases, porque contiene tres bandas que son, en realidad, clases. Sin embargo, desde la perspectiva de configuración del usuario, no tiene clases ya que las clases no se pueden configurar con la herramienta tc.

Clases Una qdisc con clases puede tener muchas *clases*, cada una de las cuales es interna a ella. A una clase, a su vez, se le pueden añadir varias clases. De manera que una clase puede tener como padre una qdisc u otra clase. Una *clase terminal* o *clase hoja* (leaf class) es una clase que no tiene clases hijas. Esta clase tiene 1 qdisc adjunta. Esta qdisc es responsable de enviar datos a la clase. Cuando se crea una clase, se le adjunta una qdisc FIFO. Cuando se añade una clase hija, se elimina esta qdisc. Para clases terminales, esta qdisc FIFO puede ser reemplazada con otra más adecuada. Incluso se puede reemplazar esta qdisc FIFO por otra con clases de manera que se puedan añadir más clases.

Clasificador Cada qdisc con clases necesita determinar a qué clase necesita enviar los paquetes. Esto se hace usando el clasificador.

Filtro La clasificación se puede realizar usando filtros. Un filtro contiene varias condiciones que los paquetes deben cumplir para no se descartados.

Scheduling (ordenamiento) Un qdisc puede, con la ayuda de un clasificador, decidir que algunos paquetes necesitan salir antes que otros. Este proceso se denomina *scheduling*, y lo realiza por ejemplo la qdisc qfifo_fast anteriormente mencionada. El scheduling también se denomina reordenamiento".

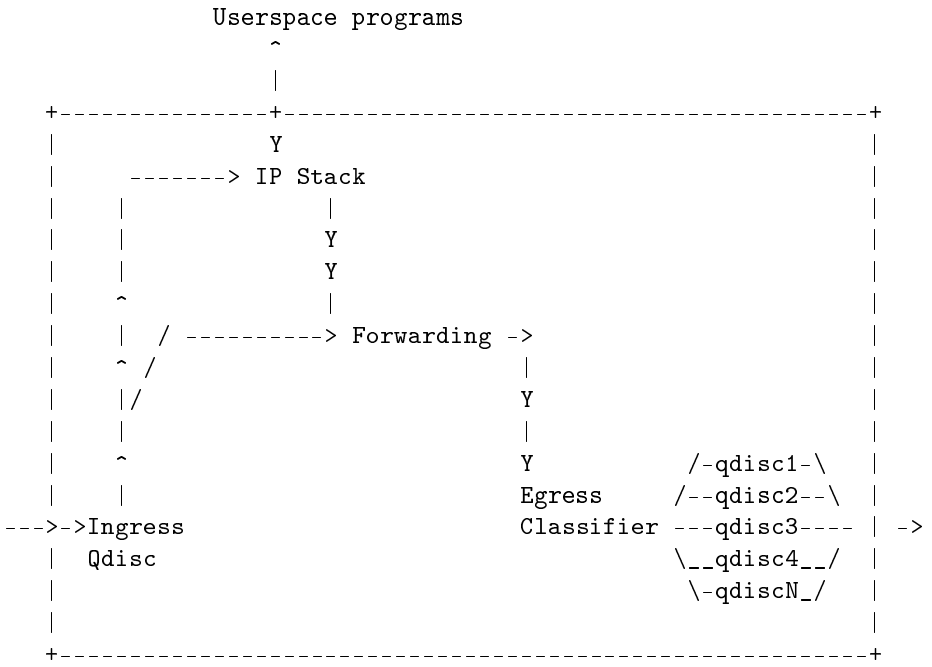
Shaping (ajuste) El proceso de retrasar paquetes antes de que salgan, para hacer que el tráfico sea conforme a una tasa máxima configurada. El *shapping* se realiza durante la salida (egress). Coloquialmente, el descarte de paquetes para ralentizar el tráfico, se le suele denominar shapping.

Policing Retrasar o descartar paquetes para que el tráfico se mantenga por debajo de un ancho de banda configurado. En Linux, el *policing* sólo puede descartar paquetes, no retrasarlos (no hay una cola de entrada).

Conservativa de trabajo Una qdisc conservativa de trabajo (*work-conserving*) siempre distribuye paquetes si los hay disponibles. En otras palabras, nunca retrasa un paquete si el adaptador de red está preparado para enviarlo (en el caso de una qdisc de salida).

No conservativa de trabajo Algunas colas, como la Token Bucket Filter, por ejemplo, pueden necesitar retrasar un paquete durante un cierto tiempo para limitar el ancho de banda. Esto significa que algunas veces rechazarán enviar paquetes, incluso cuando estén disponibles.

El cuadro B.5 muestra como estas colas qdisc encajan en una aplicación del usuario.



Cuadro B.5: Colas qdisc y aplicaciones de usuarios

El gran bloque representa al kernel. La flecha de la izquierda es el tráfico entrando en la máquina desde la red. Entonces se le pasa a la qdisc de entrada (Ingress Qdisc) que puede aplicar filtros a los paquetes, y decidir descartarlos. A esto se le llama *Policing*.

Esto ocurre en una etapa muy temprana, antes de que se haya visto mucho del kernel. Por tanto, es un buen lugar para descartar tráfico sin consumir mucho tiempo de CPU.

Si se le permite continuar al paquete, pueden ocurrir dos cosas:

- El paquete puede estar destinado a una aplicación local, en cuyo caso entra en la pila IP para ser procesado, y se envía al programa en espacio de usuario.
- El paquete puede ser reenviado (Forwarding) sin pasar por una aplicación, en cuyo caso se destina a la salida (egress).

Los programas de espacio de usuario también pueden distribuir datos, que serán examinados y reenviados por el clasificador de salida (Egress Classifier).

Ahí, el paquete es observado y encolado por cualquiera de varias qdisc. En el caso por defecto, sin configurar, sólo hay instalada una qdisc de salida, pfifo_fast, que siempre recibe el paquete. A esto se le llama *encolado* (enqueueing).

Ahora el paquete está en la qdisc, esperando a que el kernel pida que sea retransmitido por la interfaz de salida. A esto se le llama *desencolado* (dequeueing).

Esta estructura también funciona en el caso de que haya un único adaptador de red (las flechas de entrada y salida al kernel no se deben tomar muy literalmente). Cada adaptador de red tiene ranuras tanto para entrada como para salida.

B.4. Disciplinas de cola con clases

Las qdisc con clases son muy útiles si se tienen diferentes tipos de tráfico a los que quiere dar un tratamiento separado. Una de las qdisc con clases *Class Based Queueing* (CBQ), y se la menciona tan frecuentemente que se suele identificar el encolado con clases sólo con CBQ, pero esto no es verdad. CBQ es la qdisc que tiene más tiempo de existencia y la más compleja.

B.4.1. El flujo dentro de las qdisc con clases y sus clases

Cuando entra tráfico dentro de una qdisc con clases, hay que enviarlo a alguna de las clases que contiene (se necesita *clasificarlo*). Para determinar qué hay que hacer con un paquete, se consulta a los *filtros*. Es importante remarcar que los filtros se llaman desde dentro de una qdisc, y no al revés.

Los filtros asociados a esa qdisc devuelven una decisión, y la qdisc la usa para encolar el paquete en una de las clases. Cada subclase puede probar otros filtros para ver si se imparten más instrucciones. En caso contrario, la clase encola el paquete en la qdisc que contiene.

Aparte de contener otras qdisc, la mayoría de las qdisc con clases también realizan *shaping*, dan forma al tráfico. Esto es útil tanto para reordenar paquetes (con SFQ, por ejemplo) como para controlar tasas. Esto es necesario en caso de tener una interfaz de gran velocidad (por ejemplo, ethernet) enviando a un dispositivo más lento (un cable módem). Si se usará sólo SFQ, no debería pasar nada, ya que los paquetes entrarían y saldrían de su router sin retrasos: *la interfaz de salida es mucho más rápida que la velocidad del enlace en sí.* No habrá cola que reordenar.

B.4.2. La familia qdisc: raíces, controladores, hermanos y padres

Cada interfaz tiene una *qdisc raíz* de salida. Y por defecto, es la disciplina de colas pfifo_fast sin clases que se mencionó anteriormente. A cada qdisc y clase se le asigna un *controlador* (handle), que se puede utilizar en posteriores sentencias de configuración para referirse a la qdisc en cuestión. Aparte de la qdisc de salida, la interfaz también puede tener una qdisc de entrada, que dicta las normas sobre el tráfico que entra.

Los controladores de estas qdisc consisten de dos partes, un número mayor y un número menor: *<mayor>:<menor>*. Es costumbre darle a la qdisc de raíz el nombre "1:", que es lo mismo que "1:0". El número menor de una qdisc siempre es 0.

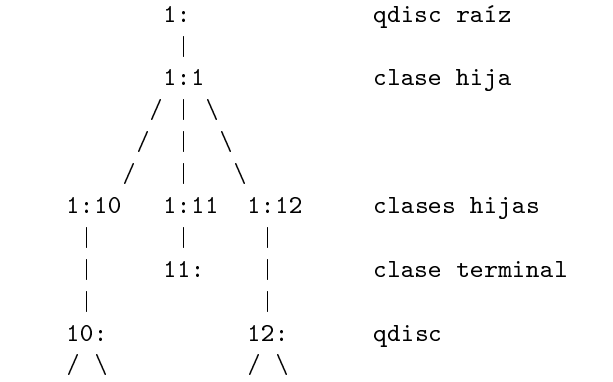
Las clases deben tener el mismo número mayor que sus padres. Este número mayor tiene que ser único dentro de una configuración de salida o entrada. El número menor debe ser único dentro de una qdisc y sus clases.

B.4.3. Cómo se usan los filtros para clasificar el tráfico

Recapitulando, una jerarquía típica de qdisc puede ser como la que muestra el cuadro B.6.

Es importante no imaginarse que el kernel está en la cima del árbol y la red abajo, ya que no es verdad. Los paquetes se encolan y desencolan en el qdisc raíz, que es la única componente con la que habla el kernel.

Un paquete se clasifica en una cadena como ésta:



Cuadro B.6: Jerarquía de una qdisc

```
1: -> 1:1 -> 12: -> 12:2
```

Ahora el paquete reside en una cola de una qdisc asociada a la clase 12:2. En este ejemplo, se asocia un filtro a cada *nodo* del árbol, y cada cual escoge qué rama se toma en su paso. Esto puede tener sentido, sin embargo, también es posible:

```
1: -> 12:2
```

En este caso, un filtro asociado a la raíz decidió enviar el paquete directamente a 12:2.

B.4.4. Cómo se desencolan los paquetes para enviarlos al hardware

Cuando el kernel decide que necesita extraer paquetes para enviarlos a la interfaz, la qdisc 1: raíz recibe una petición de desencolar, que se pasa a 1:1, que a su vez la pasa a 10:, 11:, y 12:, cada una de las cuales consulta a sus descendientes, e intenta hacer `dequeue()` (una función de desencolamiento) sobre ellos. En este caso, el núcleo necesita recorrer todo el árbol, porque sólo 12:2 contiene un paquete.

Vale notar que clases anidadas solo hablan con sus padres qdisc, y no con la interfaz. Solo las qdisc raíz son desencoladas por el kernel.

La consecuencia de esto es que las clases nunca desencolan más rápido de lo que sus padres permiten. Y esto es exactamente lo que se quiere: *de esta manera, se puede tener SFQ como una clase interna, que no hace ajustes, sólo reordena, y tener una qdisc externa, que es la que hace los ajustes.*

B.4.5. La qdisc PRIO

La qdisc PRIO en realidad no hace ajustes, sino que sólo subdivide el tráfico basándose en cómo se hayan configurado los filtros. Se puede considerar la qdisc PRIO como una pfifo_fast con *esteroides*, en la que cada banda es una clase separada, en lugar de una simple FIFO.

Cuando se encola un paquete en la qdisc PRIO, se escoge una clase basándose en los filtros configurados. Por defecto, se crean tres clases. Estas clases contienen qdisc que son puras FIFO sin estructura interna, pero se pueden sustituir por cualquier otra qdisc que haya disponible.

Siempre que se necesite desencolar un paquete, se intenta primero con la clase :1. Las clases más altas sólo se usan si no se ha conseguido un paquete en las clases más bajas.

Esta qdisc es muy útil en caso de que quiera dar prioridad a cierto tráfico sin utilizar sólo el campo TOS sino usando el potencial de los filtros de tc. Se puede agregar cualquier qdisc a las tres clases creadas por defecto, mientras que pfifo_fast está limitada a qdisc de FIFO sencillas.

Como en realidad no hace ajustes, vale la misma prevención de SFQ, esta qdisc es útil en caso de que la NIC real de salida esté realmente llena, o cuando esté dentro de una qdisc con clases que haga ajustes. Esto último se aplica a la mayoría de dispositivos DSL y cable módem.

Se puede decir que la qdisc PRIO es un reorganizador conservativo.

B.4.6. Parámetros y uso de PRIO

tc reconoce los siguientes parámetros:

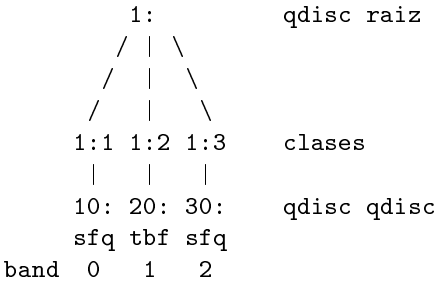
bands Número de bandas a crear. Cada banda es una clase. Si se cambia este número, también se deberá cambiar el parámetro priomap.

priomap Si no se proporcionan filtros de tc para clasificar el tráfico, la qdisc PRIO examina la prioridad TC_PRIO para decidir cómo encolar el tráfico. Esto funciona igual que con la qdisc pfifo_fast mencionada previamente.

Las bandas son clases, y todas se llaman de *mayor*:1 a *mayor*:3 por defecto, de manera que si una qdisc PRIO se llama 12:, tc filtrará el tráfico a 12:1 para garantizar la mayor prioridad.

B.4.7. Configuración de ejemplo

Suponiendo que se tiene el siguiente árbol:



El tráfico masivo irá a 30:, el interactivo a 20: o 10:. Este árbol se configura de la siguiente manera:

```
# tc qdisc add dev eth0 root handle 1: prio
     --> Esto crea "instantáneamente" las clases 1:1, 1:2, 1:3
#
# tc qdisc add dev eth0 parent 1:1 handle 10: sfq
# tc qdisc add dev eth0 parent 1:2 handle 20: tbf rate 20kbit buffer 1600 \
  limit 3000
# tc qdisc add dev eth0 parent 1:3 handle 30: sfq
```

A continuación se muestra como queda la configuración:

```
# tc -s qdisc ls dev eth0

qdisc sfq 30: quantum 1514b
```

```
 Sent 0 bytes 0 pkts (dropped 0, overlimits 0)

qdisc tbf 20: rate 20Kbit burst 1599b lat 667.6ms
 Sent 0 bytes 0 pkts (dropped 0, overlimits 0)

qdisc sfq 10: quantum 1514b
 Sent 132 bytes 2 pkts (dropped 0, overlimits 0)

qdisc prio 1: bands 3 priomap 1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
 Sent 174 bytes 3 pkts (dropped 0, overlimits 0)
```

Como se puede ver, la banda 0 ya ha tenido algo de tráfico, y se envió un paquete mientras se ejecutaba la configuración.

Ahora si se hace alguna transferencia masiva de datos con alguna herramienta[4] que ajuste pertinentemente el campo TOS, se obtiene lo siguiente:

```
# scp tc ahu@10.0.0.11:./ ahu@10.0.0.11's password:
tc                    100% |*****************************| 353 KB 00:00

# tc -s qdisc ls dev eth0

qdisc sfq 30: quantum 1514b
 Sent 384228 bytes 274 pkts (dropped 0, overlimits 0)

qdisc tbf 20: rate 20Kbit burst 1599b lat 667.6ms
 Sent 2640 bytes 20 pkts (dropped 0, overlimits 0)

qdisc sfq 10: quantum 1514b
 Sent 2230 bytes 31 pkts (dropped 0, overlimits 0)

qdisc prio 1: bands 3 priomap 1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
 Sent 389140 bytes 326 pkts (dropped 0, overlimits 0)
```

Se puede observar que todo el tráfico fue al controlador 30:, que es la banda de menor prioridad, tal como se esperaba. Ahora, para verificar que el tráfico interactivo va a bandas más altas, se crea tráfico interactivo con lo cual se obtiene:

```
# tc -s qdisc ls dev eth0

qdisc sfq 30: quantum 1514b
 Sent 384228 bytes 274 pkts (dropped 0, overlimits 0)

qdisc tbf 20: rate 20Kbit burst 1599b lat 667.6ms
 Sent 2640 bytes 20 pkts (dropped 0, overlimits 0)

qdisc sfq 10: quantum 1514b
 Sent 14926 bytes 193 pkts (dropped 0, overlimits 0)

qdisc prio 1: bands 3 priomap 1 2 2 2 1 2 0 0 1 1 1 1 1 1 1 1
 Sent 401836 bytes 488 pkts (dropped 0, overlimits 0)
```

[4] scp es una herramienta que copia, de manera segura y remota, archivos

Se observa que todo el tráfico adicional se ha ido a 10:, que es la qdisc de más alta prioridad. No se ha enviado tráfico a la de más baja prioridad, que recibió anteriormente todos los datos del scp.

B.4.8. La qdisc CBQ

Como se comentó anteriormente, CBQ es la qdisc disponible más compleja y probablemente la más difícil de configurar correctamente. Esto se debe a que el algoritmo CBQ no es tan preciso y realmente no se ajusta del todo a la manera de trabajar de Linux.

Además de ser con clases, CBQ también da forma al tráfico (shaper) y es en este aspecto donde no funciona del todo bien. Debería funcionar de esta manera. Si intenta ajustar a 1mbit/s una conexión de 10mbit/s, el enlace debería estar ocioso el 90 % del tiempo. Si no lo está, se necesita acelerar para que realmente *esté* ocioso el 90 % del tiempo.

Esto es bastante difícil de medir, de manera que en su lugar CBQ deriva el tiempo ocioso de los microsegundos que pasan entre cada petición de más datos por parte de la capa de hardware. Combinados, se pueden utilizar para aproximar cuan lleno o vacío está el enlace.

Esto no siempre lleva a resultados adecuados. Por ejemplo, ¿qué pasaría si la verdadera velocidad del enlace no es capaz de transmitir realmente todos los 100mbit/s de datos, quizá debido a un driver mal implementado? Una tarjeta de red PCMCIA tampoco alcanzará nunca los 100mbit/s debido a la manera en que está diseñado el bus. Entonces, ¿cómo se calcula el tiempo ocioso?.

Se vuelve aún peor si consideramos dispositivos de red no del todo reales, como PPP sobre Ethernet o PPTP sobre TCP/IP. El ancho de banda efectivo en ese caso probablemente se determine por la eficiencia de los canales al espacio de usuario, que es enorme.

Aunque CBQ no es siempre muy preciso, en muchas circunstancias funciona bien.

B.4.9. El *shaping* de CBQ en detalle

Como ya ha sido comentado, CBQ trabaja asegurándose de que el enlace está ocioso sólo lo necesario para reducir el ancho de banda real hasta la tasa configurada. Para hacerlo, calcula el tiempo que debería pasar entre paquetes promedios.

Mientras opera, se mide el tiempo ocioso efectivo usando una *media de movimiento por exponencial proporcional* (EWMA, Exponential Weighted Moving Average), que considera los paquetes recientes exponencialmente más importantes que los pasados. La media de carga de UNIX (*loadaverage*) se calcula de la misma manera.

El tiempo ocioso calculado se resta al medido mediante EWMA, y el número resultante se llama *avgidle*. Un enlace cargado perfectamente tiene un avgidle de cero: los paquetes llegan exactamente una vez cada intervalo calculado.

Un enlace sobrecargado tiene un avgidle negativo y si se vuelve muy negativo, CBQ lo cierra durante un rato y entonces se produce un *sobrelímite*.

Por el contrario, un enlace ocioso puede amasar un avgidle enorme, lo que permitiría anchos de banda infinitos tras unas horas de silencio. Para evitarlo, avgidle se trunca en *maxidle*.

Si hay un sobrelímite, en teoría, la CBQ debería acelerarse a sí misma durante exactamente el tiempo que ha calculado que pasa entre paquetes, entonces pasa un paquete y se acelera nuevamente.

Estos son parámetros que se pueden especificar para configurar el ajuste:

avpkt Tamaño medio de un paquete, medido en bytes. Se necesita para calcular maxidle, que se deriva de maxburst, que va especificado en paquetes.

bandwidth El ancho de banda físico del dispositivo, necesario para cálculos de tiempo ocioso.

cell El tiempo que tarda un paquete en ser transmitido sobre un dispositivo puede crecer de a pasos, basado en el tamaño de los paquetes. Por ejemplo, un paquete de tamaño 800 y uno de 806 pueden tardar lo mismo en ser enviados (esto establece la granularidad). A menudo el valor es 8. Y debe ser una potencia entera de dos.

maxburst Este número de paquetes se utiliza para calcular maxidel de manera que cuando avgidle sea igual que maxidel, se puede enviar una ráfaga de esta cantidad de paquetes medios antes de que avgidle caiga a 0. Se debe poner un número alto si se quiere que sea tolerante a ráfagas. No se puede establecer maxidel directamente, sólo mediante este parámetro.

minburst Como se mencionó previamente, CBQ necesita acelerar en caso de sobrelímite. La solución ideal es hacerlo exactamente durante el tiempo ocioso calculado, y pasar un paquete. Para los kernel de Unix, sin embargo, normalmente es complicado organizar eventos menores a 10ms, de manera que es mejor acelerar durante un periodo algo mayor, y hacer pasar entonces minburst paquetes de una sola tanda, para después dormir minburst veces más.

El tiempo de espera se denomina *offtime*. Los valores altos de minburst llevan a ajustes más precisos a largo plazo, pero a ráfagas más grandes en escala de milisegundos.

minidle Si avgidle está por debajo de 0, estaremos en sobrelímite y se necesitar esperar hasta que avgidle sea suficientemente grande como para enviar un paquete. Para prevenir una ráfaga súbita después de haber detenido el enlace durante un periodo prolongado, avgidle se reinicia a minidle si cae demasiado abajo. minidle se especifica en microsegundos negativos, de manera que 10 significa que avgidle se corta a -10us.

mpu El tamaño mínimo del paquete (necesario porque incluso los paquetes de tamaño cero se rellenan con 64 bytes en ethernet, y por tanto lleva cierto tiempo transmitirlos). CBQ necesita saber esto para calcular de forma adecuada el tiempo ocioso.

rate La tasa deseada de tráfico saliente de la qdisc.

Internamente, CBQ tiene un montón de ajustes más precisos. Por ejemplo, a las clases que se sabe no contienen datos encolados no se les pregunta. Se penaliza a las clases sobre-limitadas reduciendo su prioridad efectiva.

B.4.10. Comportamiento de la CBQ con clases

Aparte de ajustar, usando las aproximaciones de tiempo ocioso ya mencionadas, CBQ también actúa igual que la cola PRIO en el sentido de que sus clases pueden tener diferentes prioridades y que los números pequeños de prioridad se examinan antes que los grandes.

Cada vez que la capa de hardware pide un paquete para enviarlo a la red, se inicia un proceso de round robin por pesos (WRR), comenzando por las clases de prioridad con menor número.

Estas se agrupan y se les pregunta si tienen datos disponibles. En tal caso, se devuelven los datos. Después de que se haya permitido desencolar una serie de bytes a una clase, se prueba con la siguiente clase de esa misma prioridad. Los siguientes parámetros controlan el proceso WRR:

allot Cuando se le pide a la CBQ externa un paquete para enviar por la interfaz, ésta buscará por turnos en todas sus qdisc internas (en las clases), en el orden del parámetro de prio. Cada vez que le toca el turno a una clase, ésta sólo puede enviar una cantidad limitada de datos. allot es la unidad básica de esta cantidad.

prio La CBQ también puede actuar como un dispositivo PRIO. Primero se prueba con las clases internas de mayor prioridad y mientras tengan tráfico, no se mira en las otras clases.

weight weight ayuda en el proceso de *Weighted Round Robin*. Cada clase tiene una oportunidad por turnos para enviar. Si tiene una clase con un ancho de banda significativamente menor que las otras, tiene sentido permitirle enviar más datos en su ronda que a las otras.

Una CBQ suma todos los pesos bajo una clase, y los normaliza, de manera que puede usar números arbitrarios: *sólo las tasas son importantes*. El peso re-normalizado se multiplica por el parámetro allot para determinar cuántos datos se envían en una ronda.

Vale notar que todas las clases dentro de una jerarquía CBQ necesitan compartir el mismo número mayor.

B.4.11. Parámetros de CBQ que determinan la compartición y préstamo de enlaces

Aparte de meramente limitar determinados tipos de tráfico, también es posible especificar qué clases pueden tomar prestada capacidad de otras clases o, al revés, prestar ancho de banda.

isolated / sharing Una clase que está configurada como isolated (*aislada*) no prestará ancho de banda a sus hermanas. Debe ser usada si se tienen varios agentes competidores o mutuamente hostiles sobre el enlace que no quieren dejarse espacio entre sí.

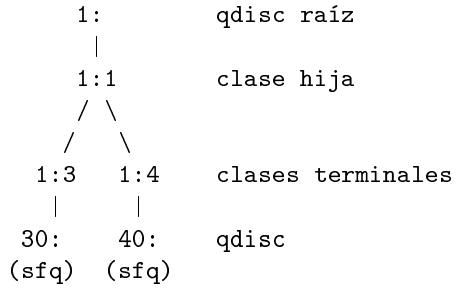
El programa de control tc también conoce el sharing, que es lo contrario que isolated.

bounded / borrow Una clase también puede estar bounded (*limitada*), lo que significa que no tratará de tomar ancho de banda prestado de sus clases hermanas. tc también conoce borrow, que es lo contrario de bounded.

En una situación típica se podría encontrar a dos agentes sobre un mismo enlace que al mismo tiempo son isolated y bounded, lo que significa que están realmente limitadas a sus tasas aisladas, y que no permitirán préstamos entre sí.

Junto a tales clases, puede haber otras que tengan permiso de ceder ancho de banda.

B.4.12. Configuración de ejemplo



Esta configuración limita el tráfico del servidor Web a 5mbit y el SMTP a 3mbit. Juntos, no pueden alcanzar más de 6mbit. Una NIC de 100mbit y las clases pueden tomar ancho de banda prestado unas de otras.

```
# tc qdisc add dev eth0 root handle 1:0 cbq bandwidth 100Mbit \
  avpkt 1000 cell 8
# tc class add dev eth0 parent 1:0 classid 1:1 cbq bandwidth 100Mbit \
  rate 6Mbit weight 0.6Mbit prio 8 allot 1514 cell 8 maxburst 20 \
  avpkt 1000 bounded
```

Esta parte instala la raíz y la clase convenida 1:1. La clase 1:1 está limitada, de manera que su ancho de banda no puede pasar de 6mbit.

La configuración HTB correspondiente es mucho más sencilla.

```
# tc class add dev eth0 parent 1:1 classid 1:3 cbq bandwidth 100Mbit \
  rate 5Mbit weight 0.5Mbit prio 5 allot 1514 cell 8 maxburst 20 \
  avpkt 1000
# tc class add dev eth0 parent 1:1 classid 1:4 cbq bandwidth 100Mbit \
  rate 3Mbit weight 0.3Mbit prio 5 allot 1514 cell 8 maxburst 20 \
  avpkt 1000
```

Estas son nuestras dos clases terminales. Vale notar que se escala el peso con la tasa configurada. Ninguna clase está limitada, pero están conectadas a la clase 1:1, que sí lo está. De manera que la suma de anchos de banda de las dos clases nunca pasará de 6mbit. Por cierto, los identificadores de clase deben estar dentro del mismo número mayor que su qdisc madre.

```
# tc qdisc add dev eth0 parent 1:3 handle 30: sfq
# tc qdisc add dev eth0 parent 1:4 handle 40: sfq
```

Ambas clases llevan una qdisc FIFO por defecto. Pero las hemos reemplazado con una cola SFQ de manera que cada flujo de datos sea tratado de igual forma.

```
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip \
  sport 80 0xffff flowid 1:3
# tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 match ip \
  sport 25 0xffff flowid 1:4
```

Estas órdenes, asociadas directamente a al raíz, envían el tráfico a las qdisc correctas.

Se usó "`tc class add`"para *crear* clases dentro de una qdisc, pero también se usó "`tc qdisc add`"para añadir qdisc a estas clases.

Es interesante preguntar qué sucede con el tráfico que no es clasificado por ninguna de las dos reglas. Parecería que en este caso, los datos serán procesados dentro de 1:0, y estarán sin límites.

Si el tráfico conjunto de SMTP (puerto 25) + Web (puerto 80) intenta exceder el límite impuesto de 6mbit/s, el ancho de banda se dividirá de acuerdo al parámetro de peso, dando 5/8 del tráfico al servidor Web y 3/8 al servidor de correo.

Con esta configuración también podemos decir que el tráfico del servidor Web tendrá como mínimo $5/8 * 6mbit = 3{,}75mbit$.

B.4.13. Otros parámetros de CBQ: split y defmap

Como se mencionó anteriormente, una qdisc con clases necesita llamar a los filtros para determinar en cual clase encolar un paquete.

Aparte de llamar al filtro, CBQ ofrece otras opciones: defmap y split. Como es probable que a menudo sólo se requiera filtrar el Tipo de Servicio (TOS), CBQ proporciona una sintaxis

especial. Cuando CBQ necesita averiguar dónde encolar un paquete, comprueba si la qdisc es una *split node*. Si lo es, una de las sub-qdisc ha indicado que desea recibir todos los paquetes con una determinada prioridad, que se puede derivar del campo TOS, o por opciones de socket establecidas por las aplicaciones. Se hace un *AND* con los bits de prioridad de los paquetes y el campo defmap para saber si coinciden o no. En otras palabras, se hace un atajo para crear filtros muy rápidos, que sólo se ajustan a ciertas prioridades. Un defmap de *ff*, en hexadecimal o 255 en decimal, coincidirá con todo, un defmap de 0, con nada.

```
# tc qdisc add dev eth1 root handle 1: cbq bandwidth 10Mbit allot 1514 \
  cell 8 avpkt 1000 mpu 64
# tc class add dev eth1 parent 1:0 classid 1:1 cbq bandwidth 10Mbit \
  rate 10Mbit allot 1514 cell 8 weight 1Mbit prio 8 maxburst 20 \
  avpkt 1000
```

defmap se refiere a los bits TC_PRIO, que se definen así:

TC_PRIO	**Número**	**Corresponde al TOS**
BESTEFFORT	0	Maximizar fiabilidad
FILLER	1	Minimizar coste
BULK	2	Maximizar transferencia (0x8)
INTERACTIVE_BULK	4	
INTERACTIVE	6	Minimizar retrasos (0x10)
CONTROL	7	

El número TC_PRIO se corresponde a bits, contando desde la derecha.

Ahora, las clases interactiva y masiva:

```
# tc class add dev eth1 parent 1:1 classid 1:2 cbq bandwidth 10Mbit \
  rate 1Mbit allot 1514 cell 8 weight 100Kbit prio 3 maxburst 20 \
  avpkt 1000 split 1:0 defmap c0
# tc class add dev eth1 parent 1:1 classid 1:3 cbq bandwidth 10Mbit \
  rate 8Mbit allot 1514 cell 8 weight 800Kbit prio 7 maxburst 20 \
  avpkt 1000 split 1:0 defmap 3f
```

La *split qdisc* es la 1:0, que es donde se va a hacer la elección. `c0` es el valor binario 11000000, `3F` es 00111111, de manera que entre los dos coinciden con todo. La primera clase coincide con los bits 7 y 6, y por tanto se corresponde con el tráfico *interactivo* y *de control*. La segunda clase coincide con el resto.

El nodo 1:0 tiene una tabla de prioridades como ésta:

Adicionalmente, se puede pasar una *change mask* (máscara de cambio), que indica cuales prioridades se desea cambiar. Sólo se necesita usar esto si se está ejecutando "`tc class change`". Por ejemplo, para añadir los mejores esfuerzos al tráfico de 1:2, se podría ejecutar lo siguiente:

```
# tc class change dev eth1 classid 1:2 cbq defmap 01/01
```

La nueva tabla de prioridades del nodo 1:0 queda así:

Prioridad	Enviar a
0	1:3
1	1:3
2	1:3
3	1:3
4	1:3
5	1:3
6	1:2
7	1:2

Prioridad	Enviar a
0	1:2
1	1:3
2	1:3
3	1:3
4	1:3
5	1:3
6	1:2
7	1:2

B.4.14. Hierarchical Token Bucket

HTB funciona igual que CBQ, pero no recurre a cálculos de tiempo ocioso para realizar los ajustes. En su lugar, es un Token Bucket Filter con clases (de ahí el nombre). Para un mayor detalle sobre HTB, consultar la página Web `http://luxik.cdi.cz/~devik/qos/htb/`. El análisis de esta qdisc escapa a los límites del presente trabajo.

B.4.15. Configuración de ejemplo

Funcionalmente es casi idéntica a la configuración de ejemplo anterior de CBQ:

```
# tc qdisc add dev eth0 root handle 1: htb default 30
# tc class add dev eth0 parent 1: classid 1:1 htb rate 6mbit burst 15k
# tc class add dev eth0 parent 1:1 classid 1:10 htb rate 5mbit burst 15k
# tc class add dev eth0 parent 1:1 classid 1:20 htb rate 3mbit ceil 6mbit \
  burst 15k
# tc class add dev eth0 parent 1:1 classid 1:30 htb rate 1kbit ceil 6mbit \
  burst 15k
```

Es recomendable utilizar sfq por debajo de estas clases:

```
# tc qdisc add dev eth0 parent 1:10 handle 10: sfq perturb 10
# tc qdisc add dev eth0 parent 1:20 handle 20: sfq perturb 10
# tc qdisc add dev eth0 parent 1:30 handle 30: sfq perturb 10
```

Se deben añadir los filtros que dirigirán el tráfico a las clases correctas:

```
# U32="tc filter add dev eth0 protocol ip parent 1:0 prio 1 u32"
# $U32 match ip dport 80 0xffff flowid 1:10
# $U32 match ip sport 25 0xffff flowid 1:20
```

Y eso todo, sin tener números desagradables sin explicación, ni parámetros sin documentar.

Se puede decir que HTB parece maravilloso (si 10: y 20: tienen ambos su ancho de banda garantizado, y todavía queda aún más para dividir, las qdisc tomarán prestado ancho de banda en una relación de 5:3, tal como cabría esperar).

El tráfico sin clasificar va a parar a 30:, que tiene poco ancho de banda por sí mismo, pero que puede tomar prestado todo lo que queda. Como se ha escogido SFQ internamente, se obtiene equitatividad.

Apéndice C

NetEm

A continuación se describe el modulo NetEm [43] del kernel de Linux. NetEm provee la funcionalidad de emulación de red para realizar pruebas de protocolos, emulando las propiedad de un red WAN. La actual versión puede emular demora, perdidas, duplicación y reordenamiento de paquetes.

NetEm se controla a través de la línea de comando, con el comando tc, que es parte del paquete de herramientas de iproute2 [42].

NetEm es una mejora reciente a las funcionalidades de control de trafico de Linux. Se construye utilizando las funcionalidades existente en Linux de QoS y Servicios Diferenciados (DiffServ).

La motivación que hay detrás de NetEm es la de proveer una manera de reproducir grandes redes en un ambiente de laboratorio. Primeramente se utilizo para evaluar nuevas mejoras a TCP en Linux.

Nació del código de TBF y solo soportaba una demora constante, pero fue evolucionando con el tiempo y el esfuerzo de desarrolladores y usuarios.

C.1. Diseño

NetEm consiste de dos partes, un pequeño modulo del kernel de Linux para disciplina de colas y una herramienta, tc, de línea de comando para configurarlo. El modulo está integrado desde la versión 2.6.8 y 2.4.28 del kernel, y la herramienta es parte del paquete *iproute2*.

La comunicación entre la herramienta y el kernel es hecha a través de la interfase de sockets Netlink. Los pedidos son codificados en el formato estándar de mensajes que luego el kernel decodifica.

La figura 5.1 muestra la arquitectura básico de NetEm. Las disciplina de cola existen entre la salida del protocolo y el dispositivo de red.

Una disciplina de cola es un objeto simple con dos interfases claves. Una interfase encola paquetes a ser enviados y la otra entrega paquetes al dispositivo de red. Las políticas a utilizar para la entrega de paquetes al dispositivo de red está en la configuración de la disciplina de cola. Se pueden configurar políticas complejas anidando distintas disciplinas de cola.

Internamente, NetEm es una disciplina de cola con clases con colas de paquetes. Una es una cola privada de espera tipo TFIFO y la otra es una disciplina de cola anidada, generalmente TFIFO. La interfase de encolado toma paquetes, les asigna un Timestamp (con el tiempo de envío) y los pone en la cola de espera. Un reloj interno mueve los paquete de la cola de espera a la disciplina de cola anidada. La interfase de desencolado toma paquetes de la disciplina de cola anidada.

C.1.1. Parámetros

El usuario especifica los parámetros de emulación a través de la herramienta tc. Sin parámetros, NetEm se comporta como una cola TFIFO sin demora, perdida, duplicación o reordenamiento de paquetes.

Demora

La redes de computadoras no muestran siempre la misma demora, ésta varia en base a otros flujos de trafico que viajan por el mismo camino. Por esta razón, la demora se define por el valor promedio (μ), desviación estándar (σ) y correlación (ρ). Por defecto NetEm utiliza una distribución uniforme ($\mu \pm \sigma$), pero es posible utilizar otra tabla de distribución generadas a partir de un modelo matemático o datos experimentales. Éstas se pueden generar con herramientas que provee iproute2. Las distribuciones estándar a utilizar son uniforme, normal, pareto y paretonormal.

Perdida

La perdida de paquete está implementada como la eliminación de un cierto porcentaje de paquetes al azar antes de ser encolados. Se define como un porcentaje de paquetes a perder y su correlación.

Duplicación

En redes con hardware confiable no duplican paquetes, pero con routers redundantes y hardware real la duplicación puede ocurrir. En NetEm la duplicación se realizar clonando un cierto porcentaje de paquetes al azar antes de ser encolados. Se define como un porcentaje de paquetes a duplicar y su correlación.

Corrupción

Ruido aleatorio en los enlaces causa que algunos paquete lleguen corruptos a su destino. NetEm puede generar un simple bit de error en el paquete para corromperlo. Se define como un porcentaje de paquetes a corromper y su correlación.

Reordenamiento

Reordenamiento de paquete ocurre cuando paquetes atraviesan caminos con diferentes demoras. Algunos routers de alta velocidad usan múltiples buses y procesadores que internamente crean caminos alternativos. Los paquetes son reordenados cuando diferentes procesadores y buses tienen diferente demora.

NetEm tiene dos formas posible de reordenamiento. Una forma es definiendo un *gap* que actúa como un agente de seguridad aleatorio, este elige 1 de N paquetes y lo demora. Otra forma es definiendo un porcentaje $X\,\%$ de paquetes a demorar y su correlación. Un $X\,\%$ de los paquetes serán enviados inmediatamente, los restantes sufrirán la demora configurada.

C.2. Control de tasa

Por defecto NetEm usa una disciplina de cola TFIFO para la cola saliente, sin embargo se pueden utilizar otras, como se explicó en Disciplina de colas (*qdisc*).

C.3. Limitaciones

NetEm trabaja en un flujo simple. Sin embargo, redes reales son complejas y la emulación no puede emular todos los casos posibles que se pueden presentar. La granularidad del reloj de Linux afecta la naturaleza de tiempo real de NetEm, la elección del pseudo-generador de números aleatorios impacta en los resultados de la emulación, y los dispositivos de red no siempre están disponibles para ráfaga de paquetes. Los parámetros de NetEm no son suficientes para describir complejas redes con múltiples niveles de complejidad.

C.3.1. Relojes

Al no ser Linux un sistema de tiempo real, esto causa que NetEm sufra ciertas restricciones. Los relojes del kernel están limitados por el tick del sistema a 1000Hz o 1ms en Linux 2.6, o 100Hz o 10ms en Linux 2.4. Por ende, NetEm no puede emular demoras menores a 1ms. Pero este problema no lo sufre solamente NetEm, las disciplinas de control de tasa también lo sufren cuando corren sobre enlaces de alta velocidad. No es posible limitar una red de 10gigabit a 100mbit con precisión sin utilizar relojes de alta resolución.

C.3.2. Números al azar

En el kernel de Linux existen varias fuentes de números pseudo-azarosos, pero ninguno es idóneo para una buena emulación. La segura función criptográfica de generación de números al azar *get_random_bytes()* no puede ser utilizada duramente ya que confía en los eventos del sistema para proveer entropía. La intensión de la misma es la de proveer claves criptográficas y se puede bloquear cuando hay poca entropía hasta que el pool de entropía se llene de nuevos eventos del sistema (búsquedas en disco, llegada de paquetes, movimientos del mouse, etc.).

El código de red tiene una simple función de números al azar de 32bit, *net_random()*, implementada como un generador lineal congruente (LCG). LCG no útiles para simulaciones porque producen patrones en la salida que pueden influenciar los resultados finales. Una mejor alternativa (que finalmente se implementó) fue la utilización de generador Tausworthe combinado y máximamente equi-distribuido encontrado en la librería GNU Scientific Library 1.5.

C.3.3. Dispositivos de red

Los dispositivos de red en Linux tienen un driver de transmisión en forma de anillo que tiene una referencia a los datos disponibles para que el hardware lo procese. Este anillo tiene un tamaño definido, limitado por la disponibilidad de los bloques de control de transmisión. Cuando la carga es alta, NetEm causara ráfaga de paquetes hacia el dispositivo cada 1ms. El anillo de transmisión debe ser lo suficientemente grande para manejar esta ráfaga de paquete o el dispositivo debe manejar el flujo correctamente.

Apéndice D

tc

A continuación se describe en detalles técnicos la herramienta *tc* [47], que permite la manipulación de la configuración del tráfico de red en Linux.

D.1. Sinapsis

```
tc qdisc [ add | change | replace | link ]
  dev DEV [ parent qdisc-id | root ] [ handle qdisc-id ]
  qdisc [ qdisc specific parameters ]

tc class [ add | change | replace ]
  dev DEV parent qdisc-id [ classid class-id ]
  qdisc [ qdisc specific parameters ]

tc filter [ add | change | replace ]
  dev DEV [ parent qdisc-id | root ]
  protocol prio priority
  filtertype [ filtertype specific parameters ]
  flowid flow-id

tc [-s | -d ] qdisc show [ dev DEV ] tc [-s | -d ] class show dev
DEV tc filter show dev DEV
```

D.2. Descripción

La herramienta *tc* es utilizada para configurar el control de tráfico en el kernel de Linux. EL control de tráfico consiste en lo siguiente:

Conformado Cuando se conforma el tráfico de red, se toma control de la tasa de transmisión de la misma. EL conformado puede ser algo más que la disminución del ancho de banda disponible, es también utilizado para suavizar las ráfagas de tráfico de manera tal que se consiga un mejor comportamiento de la red. El conformado del tráfico ocurre en los egresos.

Programación Con la programación de la transmisión de paquetes es posible mejorar la interactividad del tráfico que se requiera, al tiempo que se garantiza el ancho de banda para

transmisiones en masa. Al reordenamiento también se lo llama priorización, y solo ocurre en los egresos.

Policing Mientras el modelado se ocupa de la transmisión del tráfico, el policing tiene que ver con la llegada del mismo. Solo ocurre en los ingresos.

Eliminación El tráfico que exceda un ancho de banda establecido podrá ser eliminado inmediatamente, tanto en ingresos como en egresos.

El procesamiento del tráfico es controlado por tres tipos de objetos: qdisc, clases y filtros.

D.3. *qdisc*

qdisc es la abreviatura de *queueing discipline* (disciplina de cola) y es elemental para comprender el control de tráfico. En cualquier momento que el kernel necesite enviar un paquete a una interfaz, el mismo es encolado en el qdisc configurado para dicha interfaz. Inmediatamente después, el kernel trata de obtener la mayor cantidad de paquetes posibles del qdisc, y así entregarlos al controlador del adaptador de red.

Una qdisc simple, es la *pfifo*, la cual no realiza procesamiento en sí, tan solo se comporta como una cola *first in, first out*. Sin embargo, almacena tráfico cuando la interfaz de red no puede manejarlo momentáneamente.

D.4. Clases

Algunas qdisc pueden contener clases, las cuales pueden contener otras qdisc, el tráfico puede ser encolado en cualquiera de estas qdisc internas, que a su vez están contenidas dentro de las clases. Cuando el kernel trata de desencolar un paquete de algún qdisc con clases el mismo puede provenir de cualquiera de ellas. Por ejemplo, un qdisc puede priorizar algún tipo de tráfico tratando de desencolar algunas clases antes que otras.

D.5. Filtros

Un filtro es utilizado en un qdisc con clases para determinar en cual clase será encolado el paquete. En cualquier momento que el tráfico llegue a una clase con subclases, necesita ser *clasificado*. Se pueden utilizar varios métodos para tal fin, uno de ellos consiste en aplicar filtros. Todos los filtros adjuntados a la clase son invocados, hasta que uno retorna un veredicto. Si no se recibe un veredicto, están disponibles otros criterios. Y esto difiere entre qdisc.

Vale notar que los filtros residen dentro de las qdisc, y no son responsables de lo que sucede.

D.6. *qdisc* sin clases

Las qdisc sin clases son:

pfifo/bfifo Son las qdisc más simples, con un comportamiento puramente first in, first out. Tiene límite en paquetes o en bytes.

pfifo_fast Es la qdisc estándar para el kernel con *router avanzado* habilitado. Consiste en un cola de tres bandas que respeta el flag de Type of Service, como la prioridad se puede asignar al paquete.

RED La *Random Early Detection* (detección temprana aleatoria) simula una congestión física a través de la puesta de paquetes aleatorios cuando se comienza a llegar al máximo ancho de banda configurado. Funciona bien para aplicaciones con un ancho de banda muy amplio.

SFQ La *Stochastic Fairness Queueing* reordena el tráfico encolado de manera que cada *sesion* llegue a enviar un paquete en su turno.

TBF El *Token Bucket Filter* se utiliza para reducir el tráfico y llevarlo a una tasa configurada con precisión. Escala bien a anchos de banda amplios.

D.7. Configurando *qdisc* sin clases

Ante la ausencia de qdisc con clases, las qdisc sin clases sólo pueden ser adjuntadas en la raíz de un dispositivo.

Sintaxis completa: `tc qdisc add dev DEV root QDISC QDISC-PARAMETERS`

Para remover, `tc qdisc del dev DEV root`

El qdisc pfifo-fast es el que está configurado por defecto cuando no se ha configurado alguno.

D.8. *qdisc* con clases

Las qdisc con clases son:

CBQ El *Class Based Queueing* implementa una jerarquía de clases de compartición de enlace. Contiene elementos de modelado como capacidades de priorización. El modelado es ejecutado calculando los tiempos ociosos del enlace basados en el tamaño de paquete promedio y el ancho de banda del enlace subyacente. Aunque este último puede estar mal definido para algunas interfaces.

HTB El *Hierarchy Token Bucket* implementa una jerarquía de clases de compartición de enlace con énfasis en la adaptación de practicas existentes. HTB facilita el garantizado de ancho de banda para las clases, como así mismo permite la especificación de los limites superiores para la intercambio inter-clases. Contiene elementos de modelado, basado en TBF y puede priorizar clases.

PRIO La qdisc PRIO no modela tráfico, es un contenedor para un número configurable de clases, las cuales son desencoladas con un orden. Esto permite una priorización más sencilla del tráfico, donde las clases mas bajas solo puede enviar si las altas no tienen paquetes disponibles. Para facilitar la configuración, los bits de Type Of Service son respetados por defecto.

D.9. Teoría de operación

Las clases forman un árbol, donde cada una tiene un solo padre. Una clase puede tener múltiples hijos. Algunas qdisc permiten agregar clases durante la ejecución (CBQ, HTB) mientras otras (PRIO) se crean con un número estático de hijos.

Las qdisc que permiten agregar de manera dinámica clases pueden tener cero o mas subclases a las cuales se les puede encolar tráfico.

Más aún, cada clase contiene una qdisc hoja, la cual tiene una conducta pfifo por defecto, aunque se le puede adjuntar otra qdisc en su lugar. Esta qdisc puede nuevamente contener clases, pero cada clase solo puede tener una qdisc hoja.

Cuando un paquete ingresa en una qdisc con clases puede ser clasificado a una de las clases internas. Existen tres criterios disponibles, aunque no todas las qdisc usarán todos los tres:

filtros tc Si los *filtros tc* se adjuntan a una clase, primero son consultados para la obtención de instrucciones relevantes. Los filtros pueden hacer match con todos los campos del encabezado de un paquete, como lo hace un firewall implementado por ipchans o iptables.

Type Of Servicio Algunas qdisc contienen reglas integradas para la clasificación de paquetes basadas en el campo TOS.

skb→priority Programas que corren en espacio de usuario pueden codificar un *class-id* en el campo *skb→priority* utilizando la opción *SO_PRIORITY*. Cada nodo dentro del árbol puede tener sus propios filtros, pero filtros de más alto nivel pueden también apuntar directamente a una clase inferior.

Si la clasificación no fue exitosa, los paquetes son encolados a la qdisc hoja adjuntada a la clase.

D.10. Identificadores

Todas las qdisc, clases y filtros tienen *IDs* (identificadores), los cuales pueden ser especificados o asignados automáticamente.

Los ID consisten en un número mayor y un número menor, separados por comas.

qdisc A una qdisc, que potencialmente puede tener hijos, se le asigna un número mayor, llamado *handle*, dejando el espacio de nombres del número menor disponible para las clases hijo. EL handle se expresa como "10:". Se puede asignar explícitamente un handle a qdisc que se espera tengan hijos.

Clases Las clases que residen bajo una qdisc comparten el número mayor del mismo, pero cada una tiene un número menor separado, llamado *classid*, que no tiene relación con su clase padre, sólo con su qdisc padre. La misma convención de nombres de qdisc se aplica a las clases.

Filtros Los filtros tienen un ID de tres partes, el cual es necesario solo cuando se utiliza una jerarquía de filtros hasheada.

D.11. Unidades

Todos los parámetros aceptan un número en punto flotante, posiblemente seguido de una unidad. Anchos de banda o tasas se pueden especificar en:

kbps Kilobytes por segundo

mbps Megabytes por segundo

kbit Kilobits por segundo

mbit Megabits por segundo

bps o un número explícito Bytes por segundo. La cantidad de datos se pueden especificar en:

kb o k Kilobytes

mb o m Megabytes

mbit Megabits

kbit Kilobits

b o un número explícito Bytes. Los intervalos de tiempo se pueden especificar en:

s, sec o secs Segundos

ms, msec o msecs Milisegundos

us, usec, usecs o un número explícito Microsegundos

D.12. Comandos de *tc*

Los siguientes comandos están disponibles por qdisc, clases y filtros:

add Agrega una qdisc, clase o filtro a un nodo. Para todas las entidades, se debe pasar un padre, bien sea pasando su ID o adjuntándolo directamente a la raíz del dispositivo. Cuando se crea una qdisc o filtro, éste puede ser nombrado con el parámetro handle. Una clase se nombra con el parámetro classid.

remove Una qdisc se puede eliminar especificando su handle, el cual también puede ser *root* (raíz). Todas las subclases y sus qdisc hoja se borran automáticamente, así como también cualquier filtro que tengan adjuntado.

change Algunas entidades pueden ser modificadas *en el lugar*. Se comparte la sintaxis del comando *add*, con la excepción de que ni el handle ni el padre se pueden cambiar. En otras palabras, change no puede mover un nodo dentro del árbol.

replace Ejecuta los comando remove y luego add de manera casi atómica en un nodo existente con determinado ID. Si el nodo no existe, se crea.

link Solo disponible por qdisc y ejecuta un replace donde el nodo ya debe existir.