TRABAJO FINAL DE CARRERA

TÍTULO DEL TFC: Diseño de un entorno para el estudio de los parámetros de funcionamiento del protocolo TCP

TITULACIÓN: Ingeniería Técnica de Telecomunicaciones, especialidad Telemática

AUTORES: Sergio Higón Fernández
Alejandro Feliu Argila

DIRECTOR: Lluís Casals Ibáñez

FECHA: 25 de julio de 2006

Título: Diseño de un entorno para el estudio de los parámetros de funcionamiento del protocolo TCP

Autores: Sergio Higón Fernández
Alejandro Feliu Argila

Director: Lluís Casals Ibáñez

Fecha: 12 de julio de 2006

Resumen

En este proyecto tratamos de mejorar una de las características del simulador de redes llamado Network Simulator (NS).

Por otro lado, es también objeto de estudio el protocolo de transporte TCP.

Surgen muchas propuestas para mejorar la eficiencia de éste en enlaces inalámbricos. Una de estas propuestas es el proxy snoop, cuyo funcionamiento se detalla a lo largo del trabajo.

En concreto, los objetivos del proyecto son, por un lado, crear un entorno para el estudio de los parámetros de funcionamiento del protocolo TCP, por otro lado, plasmar las nuevas funcionalidades del proxy snoop adaptado a GPRS en el simulador NS y por último, realizar una serie de simulaciones con el nuevo módulo en el simulador NS para verificar su funcionamiento.

A lo largo del proyecto, realizamos introducciones a las características y las distintas versiones disponibles del protocolo TCP, al medio *wireless* (problemas que conlleva la utilización de este medio, sus posibles soluciones y la utilización de dicho medio en una red GPRS) y al programa *Network Simulator* utilizado para hacer las simulaciones (funciones y herramientas).

A continuación se explican las modificaciones realizadas en el programa NS para la realización del proyecto y las características críticas encontradas.

Por último, se presentan los escenarios que se utilizaran para realizar las simulaciones, los resultados de dichas simulaciones y las conclusiones extraídas de dichos resultados.

Title: Create of a environment for the study of the parameters of operation of the TCP protocol

Authors: Sergio Higón Fernández
Alejandro Feliu Argila

Director: Lluís Casals Ibáñez

Date: July, 12th 2006

Overview

In this project we try to improve the characteristics of the program Network simulator or NS.

The aim of the study is the transport protocol TCP. There were a lot of proposals to improve the efficiency of this protocol in wireless links. One of these is the proxy snoop, which is detailed in the project.

The objectives of the project are firstly to create an environment for the study of the operating parameters of the TCP protocol , secondly, to capture the news performance of the adapted proxy snoop to GPRS in the NS simulator, and thirdly, to do a group of simulations with the new module in the simulator NS to check this operation.

Lengthways the project we have made an introduction to the characteristics and the different versions of the TCP protocol, to the wireless medium (problems of his utilization, his possible solutions and the utilization of this medium in a GPRS network) and to the Network Simulator program (functions and tools).

Later we explain the modifications to be made in the program NS for the achievement of the project and the critical characteristics found.

Finally we present the stages which will be used to do the simulations, the outcomes if the simulations and the conclusions obtained through the results.

ÍNDICE

INTRODUCCIÓN

Las redes inalámbricas cada vez son más populares. Su utilización va desde el ámbito de redes de área local hasta enlaces de telefonía móvil, pasando por muchas más aplicaciones prácticas.

Uno de los aspectos a tener en cuenta en este tipo de redes son los protocolos que utiliza su capa de transporte. Surgen muchas propuestas para mejorar su eficiencia, como la de TCP (*Transmission Control Protocol*).

El protocolo TCP fue creado pensando en medios guiados donde la tasa de error es muy baja y el principal problema que puede ocasionar perdidas de paquetes es la congestión. Por eso, los mecanismos que invoca TCP para hacer frente a cualquier pérdida, están pensados para prevenir la congestión. Estos mecanismos de prevención de congestión no son adecuados cuando el principal problema de perdida de paquetes es la tasa de error de la red, que es lo que ocurre en redes inalámbricas. Hay diversas propuestas con la finalidad de mejorar la eficiencia de TCP en enlaces inalámbricos. Los reconocimientos selectivos, notificaciones explícitas de pérdida o conexiones partidas son algunas de estas propuestas [5]. La propuesta que nosotros vamos a estudiar esta basada en el proxy snoop [3].

No obstante, el proxy no siempre mejora las prestaciones del sistema, en este caso la eficiencia de TCP. En unos estudios existentes, basados en una implementación real con el proxy snoop y varias operadoras GPRS [3], quedó demostrada la inconveniencia de usar el proxy snoop en este tipo de redes. En los documentos consultados se describe también un propuesta de adaptación del proxy snoop a una red GPRS. Algunas de las características del proxy snoop adaptado son su temporizador de retransmisiones constantes y reconocimientos que abren y cierran la ventana de transmisión del emisor TCP.

Los objetivos del proyecto son, por un lado crear un entorno para el estudio de los parámetros de funcionamiento del protocolo TCP, por oto lado plasmar las nuevas funcionalidades del proxy snoop adaptado a GPRS en el simulador NS mejorando sus características, y por último, realizar un análisis mediante simulaciones con el nuevo módulo en el simulador NS para verificar su funcionamiento.

En el primer capítulo de nuestro trabajo hacemos una introducción a las características y a las distintas versiones disponibles del protocolo TCP.

En el segundo capítulo hacemos una introducción al medio *wireless,* los problemas que conlleva la utilización de este medio, sus posibles soluciones y la utilización de dicho medio en una red GPRS.

En el tercer capítulo hacemos una introducción al programa *Network Simulator* utilizado para hacer las simulaciones, describiendo sus funciones y herramientas.

En el cuarto capítulo se explican detalladamente las modificaciones realizadas en el programa NS para la adaptación de la nueva versión de snoop y algunas características críticas encontradas.

En el quinto capítulo se presentan los escenarios que se utilizaran para realizar las simulaciones.

En el sexto capítulo se presentan los resultados obtenidos de dichas simulaciones.

En el séptimo y último capítulo se explican las conclusiones a las que se ha llegado a través de los resultados obtenidos.

CAPÍTULO 1. INTRODUCCIÓN A TCP

TCP (*Transmisión Control Protocol*) es un protocolo de nivel de transporte dentro de la arquitectura de capas del modelo de referencia TCP/IP [13].

Es un protocolo fiable que ofrece un servicio de transmisión de altos volúmenes de datos extremo a extremo.

El modelo de refencia TCP/IP se compone de diferentes capas. Cada una de ellas ofrece unas funcionalidades. El protocolo TCP se sitúa en la capa de transporte. Podemos ver en qué lugar de la arquitectura se encuentra la capa de transporte en la figura 1.1.

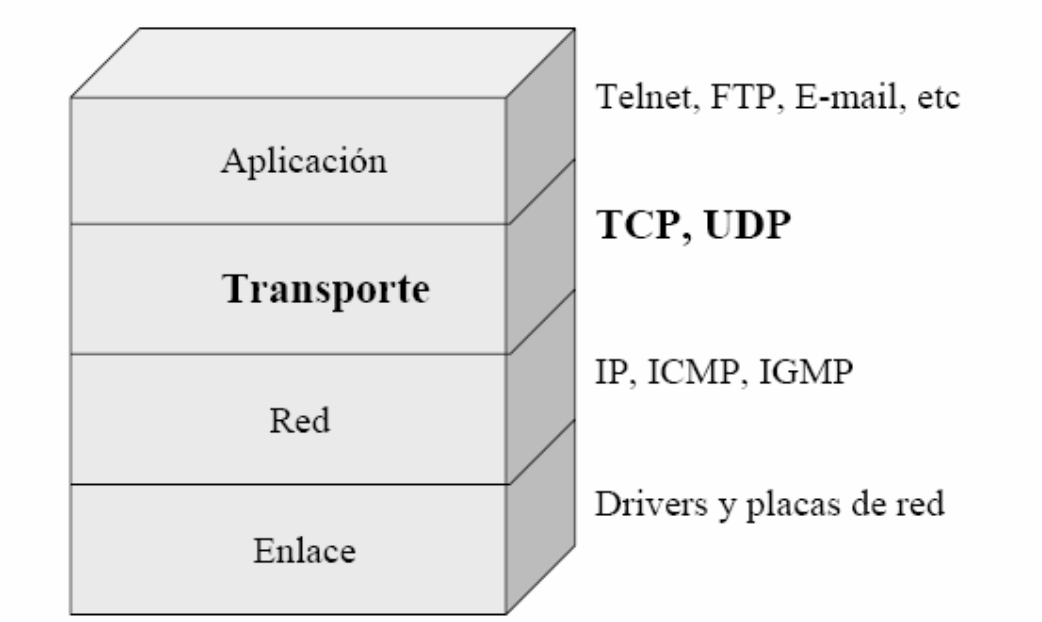

Fig. 1.1. Modelo de referencia TCP/IP y ejemplos de protocolos.

Cuando una aplicación desea enviar una serie de datos a otro nodo, estos deben seguir un proceso de encapsulado mediante el cual, cada capa inferior debe añadir una cabecera propia mediante la cual aplicar sus funcionalidades El proceso de encapsulado se aprecia en la figura 1.2.

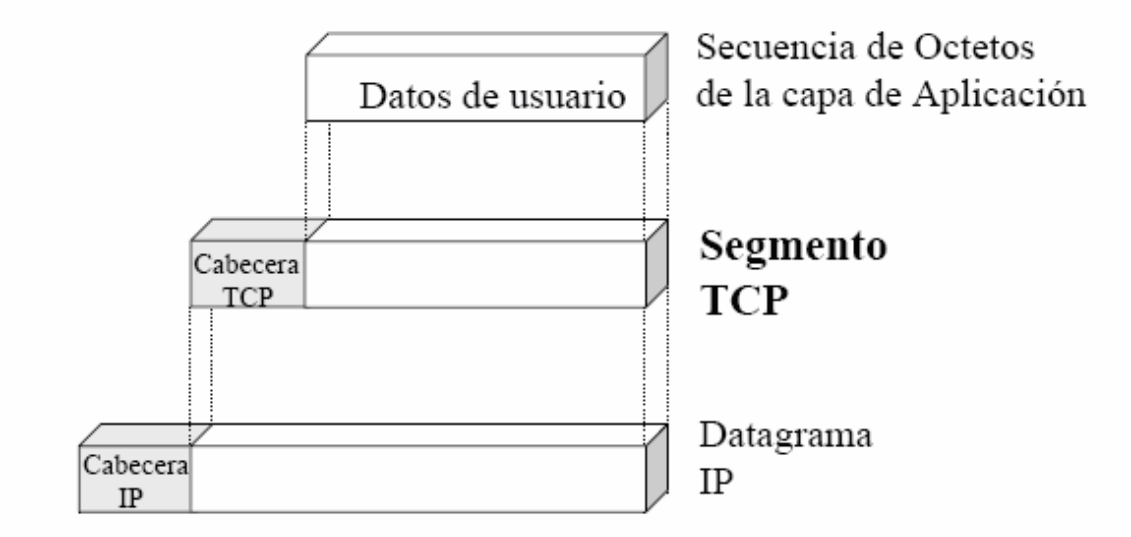

Fig. 1.2. Encapsulado de un paquete en el modelo de referencia TCP/IP.

1.1. Características de TCP

Las características principales de TCP son las siguientes [1]:

- Orientado a conexión: establece la conexión mediante el *three-way handstake*, una vez establecida transmite y por último, cierra la conexión.

- Orientado a *byte*: sus transmisiones se dan en cantidades múltiplos de *byte*.

- Su transferencia se hace recopilando datos en un buffer, de manera que guarda los datos a transmitir hasta llegar al MSS (Máxima longitud de segmento) y después lo envía.

- Los segmentos siguen guardados una vez enviados hasta recibir la confirmación de recepción. Dispone de un temporizador (RTO) utilizado para retransmitir el segmento de datos sino es confirmado antes de que este contador expire.

- La conexión es *full duplex*: transmisión simultanea en ambos sentidos.

- Aplica control de flujo que se trata mediante el uso de una ventana deslizante de tamaño variable que indica el número de *bytes* que se pueden recibir mas allá de la última información confirmada.

- Aplica control de congestión. Lo hace gracias a los mecanismos de *Slow Start* y *Congestion Avoidance* mediante los cuales ajusta la ventana de transmisión para que la cantidad de segmentos transmitidos se ajuste a lo que puede absorber la red en la que se transmiten.

- No disminuye el flujo de datos drásticamente. Esto se consigue mediante el mecanismo de *Fast Retransmit* que después de un número concreto de confirmaciones duplicadas hace una retransmisión sin esperar a que expire el temporizador de un segmento (RTO). Junto a este mecanismo se ejecuta otro llamado *Fast Recovery* que provoca que la ventana de transmisión se mantenga abierta después de cada confirmación duplicada recibida para no parar el flujo de datos y esperar a la confirmación de los nuevos datos enviados.

1.2. Versiones de TCP

La primera estandarización que se hizo fue en septiembre de 1981 en el RFC 793 y no implementaba control de congestión.

Para solventar los colapsos que se creaban fueron desarrollando diferentes versiones de TCP [6] [7].

1.2.1. TCP Tahoe

La primera se conoce como *Tahoe,* surgió en agosto de 1988 y se estandariza en el RFC 1122.

Implementaba los algoritmos *slow start* y *congestion avoidance*.

El sistema busca llegar a un estado de equilibrio basado en el principio de "conservación de los paquetes" en los enlaces, donde se establece que se envía un paquete a la red cada vez que verifica que un paquete ha abandonado la red.

La mayor desventaja de esta implementación, es que cada vez que se pierde un segmento, se inicia nuevamente la etapa de *slow start*, lo cual retrasa el momento en que las tasas de transmisión son aceptables.

1.2.2. TCP Reno

La segunda versión fue TCP Reno, surgió en abril de 1990 y su RFC es el 2581. Se trata de la versión más utilizada en nuestros días.

La diferencia fundamental existente entre esta versión y la anterior es que cuando se detecta congestión, *Tahoe* modifica la variable ventana de congestión (cwnd) llevándola a un segmento, lo que implica ir a la fase de *slow start*. Mientras, Reno la configura con el valor de *ssthresh* (Umbral), quedando en la etapa de *congestion avoidance* implementando así un algoritmo de *fast recovery*. Esto permite que se recupere más rápidamente de congestión y errores en la red.

La mejora realizada en el TCP Reno, implica mejoras en pérdidas de segmentos aislados, pero no tiene ventajas en caso de pérdidas en sucesivos segmentos. Estas pérdidas son comunes en enlaces de alta velocidad, donde la pérdida en general contiene varios segmentos.

La figura 1.4. muestra la pérdida de tres segmentos consecutivos y sus consecuencias; como puede verse, se acaba iniciando la etapa de *slow start*:

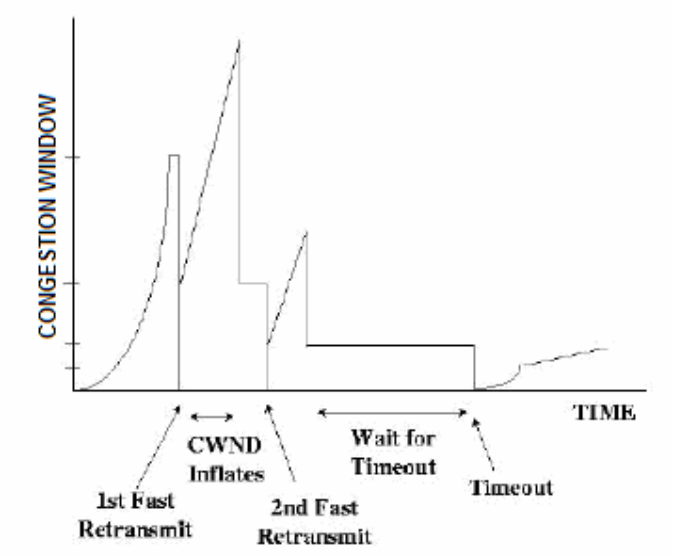

Fig. 1.4. Tratamiento de varias pérdidas en TCP Reno

1.2.3. TCP New Reno

TCP *New Reno* surgió en abril de 1990 junto a *TCP Reno* y se estandariza en el RFC 2582.

Implementa una modificación del algoritmo de *fast recovery*, que comienza al recibirse 3 *ACK* duplicados, almacenando en una variable el número de

secuencia más alto transmitido. En caso de recibirse *ACK* de todos los datos enviados por el transmisor, incluso durante la fase de *fast recovery,* o de darse un *timeout* en la retransmisión, pasa a la fase de *congestion avoidance*.

Para comenzar una fase *fast recovery* no debe estarse en una fase *fast recovery*. Tras la recepción de los tres *ACK* duplicados *ssthresh* toma el valor dado por la ecuación 1.1.

$$ssthresh = \max (FS / 2, 2 \times SMSS) \quad \textbf{(1.1)}$$

FS: Datos enviados pero no reconocidos.
SMSS: tamaño de segmento.
ssthresh: umbral.

Luego retransmite el segmento perdido y *cwnd* toma el valor dado por la ecuación 1.2.

$$cwnd = ssthresh + 3 \times SMSS \quad \textbf{(1.2)}$$

Para cada *ACK* duplicado recibido seguirá un criterio similar y transmitirá un nuevo segmento. Ante la llegada de un *ACK*, puede suceder que sea un *ACK* de todos los datos pendientes de confirmar, lo que implica un reconocimiento de todos los segmentos enviados entre el que se perdió y la recepción del tercer *ACK* duplicado. Aquí *cwnd* toma el valor:

$$cwnd = \min (ssthresh + SMSS) \quad \textbf{(1.3)}$$

A continuación se sale de la fase de *fast recovery*.

Si el *ACK* no cubre todos los datos, se trata de un *partial ACK*, por lo que se retransmite el primer segmento no reconocido (uno solo) y el transmisor sigue en la fase de *fast recovery*.

Por lo tanto, cuando múltiples paquetes son perdidos en una ventana de datos, *New Reno* retransmite un paquete perdido por RTT hasta que todos los paquetes perdidos de esa ventana de datos han sido retransmitidos. *New Reno* permanece en *Fast Recovery* hasta que todos los datos pendientes, cuando *Fast Recovery* se inició, son confirmados.

1.2.4. Otros TCPs

Existen otras versiones de TCP que enumeramos y explicamos brevemente, ya que no las utilizaremos en nuestro trabajo. El motivo es que son las versiones menos utilizadas y *Network Simulator* no las implementa con tanto detalle como las anteriores:

- *SACK 's* (Selective *ACK*): Surge en octubre de 1996 y su RFC es el 2018. Brinda un mecanismo para proporcionar más información sobre los segmentos que han sido recibidos correctamente y el transmisor puede conocer cuales son los segmentos que se han perdido.

- *Net Reno*: No está estandarizado en ningún RFC ni implementado en *network simulator*. Es una mejora del protocolo TCP *New Reno*. Ofrece un conjunto de optimizaciones que lo hace más sensible a la red, mejorando su funcionamiento, reduce los *timeouts* por retransmisiones.

- *Vegas*: No está estandarizado en ningún RFC. Tiene grandes cambios en comparación con las primeras versiones de TCP. Utiliza técnicas para mejorar el *throughput* y tratar de evitar pérdidas de segmentos. Básicamente la mejora es en el algoritmo de estimación del *round-trip time*, a partir de registrar el momento de envío de cada segmento y el momento de recibir cada *ACK*.

CAPÍTULO 2. INTRODUCCIÓN AL MEDIO WIRELESS

En los últimos años de evolución en este tipo de tecnologías se ha depositado mucho interés en probar medios inalámbricos como medios transmisores, debido a su gran comodidad a la hora de realizar las instalaciones y la posible movilidad de los equipos; aunque también aporta ciertas desventajas a la comunicación, como pueden ser velocidades de transmisión menores, más perdidas de paquetes, etc. Hoy en día, la tecnología *wireless* tiene mucha importancia, ya que se han desarrollado diferentes redes *wireless* para diferentes usos, como pueden ser *GSM*, *GPRS* o *802.11*. Todas ellas muy conocidas y en continuo desarrollo.

El hecho por el cual surge un problema aplicando *TCP* sobre el medio *wireless* es que *TCP* fue creado para transmisiones en medios cableados (con tasas de error menores que *wireless*) y fue definido mucho antes de aparecer la tecnología *wireless* en redes donde se tuviera que aplicar el modelo de referencia TCP/IP, como por ejemplo 802.11 o GPRS.

2.1. Introducción a GPRS

En nuestro trabajo con *network simulator* simulamos una red móvil GPRS [17].

GPRS es un servicio de comunicación de telefonía móvil basado en la transmisión de paquetes. Es una tecnología que evoluciona del sistema GSM y de la cual evoluciona la tecnología UMTS.

Las características más importantes de la tecnología GPRS son las que enumeramos:

- Reutiliza los recursos del sistema GSM, con un mínimo de modificaciones.

- La novedad respecto GSM, que solo soportaba tráfico de voz, es que también soporta tráfico de datos. Este tráfico es principalmente asimétrico y a ráfagas.

- Ofrece diferentes velocidades y calidades de servicio para el transporte de datos.

La arquitectura a nivel muy básico de GPRS se puede explicar a partir de la figura 2.1. Cabe decir que en ella no se entra en detalle en los elementos que forman dicha arquitectura, pero si se observa de manera básica su forma y se justifica los escenarios de las simulaciones hechas que explicaremos en puntos siguientes.

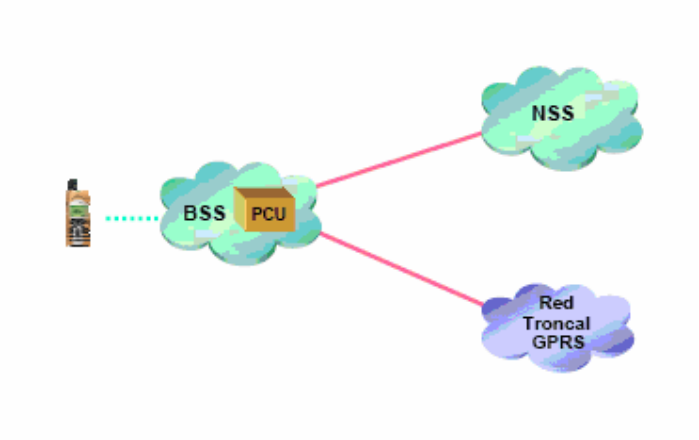

Fig. 2.1. Arquitectura de GPRS

En la figura 2.1. El teléfono móvil se conecta a la estación base (BSS) mediante el medio *wireless*. Desde esta estación base hasta el primer *router* de la red troncal GPRS el enlace es cableado. Este *router* es el emisor del tráfico hasta el teléfono móvil. Por otra parte la estación base también queda conectada con la NSS (Red de conmutación de circuitos GSM).

2.2. Problemas de TCP sobre wireless

A continuación remarcamos los principales problemas que surgen al implementar estas dos tecnologías juntas [2]:

- Enlace radio poco fiable.
- Variaciones bruscas del retardo.
- Efectos de las retransmisiones innecesarias.

A continuación se explican los problemas enumerados.

2.2.1. Enlace radio poco fiable

Este tipo de medio es mucho más sensible a cualquier tipo de interferencia, por lo que presenta una alta probabilidad de error. Además cabe destacar, que si se trata un dispositivo móvil, a la hora de hacer un traspaso de un área de cobertura a otra, se podría perder la conexión física.

2.2.2. Variaciones bruscas del retardo

Existen variaciones bruscas del retardo en relación con la poca fiabilidad del medio. El protocolo de transporte tiene un algoritmo que se encarga de medir el *RTO*, pero este algoritmo está pensado para cambios suaves, así que el cálculo del *RTO* puede no ajustarse a las necesidades de la transmisión. También afecta en este problema las posibles transmisiones innecesarias de segmentos que llegan con retraso.

2.2.3. Efectos de las retransmisiones innecesarias

Cuando a *TCP* le expira el temporizador de un segmento, lo reenvía. Al reenviarlo aplica ciertos mecanismos de control de congestión que deben solucionar el problema. Utilizando medio *wireless* es probable que esta retransmisión sea debida al alto retardo y no a la congestión. Si se activan los mecanismos de control de congestión, además producirá una retransmisión de los segmentos no reconocidos hasta el momento, lo que desencadenará más *ACKs* duplicados y culminará con la activación del mecanismo *Fast Retransmit* después del tercer *ACK* duplicado. En definitiva, si se pierde un paquete o se retrasa demasiado, *TCP* asume que se ha perdido por culpa de la congestión y limita el caudal de la transmisión.

2.3. Posibles soluciones

En este punto se describen brevemente algunas soluciones propuestas para mejorar el funcionamiento de TCP sobre redes *wireless* [2] [3] [4].

2.3.1. Compresión de cabeceras

Esta solución es aplicable a los enlaces *wireless* de baja velocidad, como por ejemplo GPRS

Se basa en el hecho de que la mayoría de la información que viaja en la cabecera se repite en los paquetes de una misma conexión. El objetivo es eliminar la información redundante. Es capaz de comprimir la cabeceras TCP/IP de 40 *bytes* hasta entre 3 y 5 *bytes*.

El proceso de compresión consiste en:

- Los campos que se pueden calcular en el destino se calculan.
- Los campos que no varían se introducen en el destino.

- En los campos que varían poco se envían solo las diferencias.

Con la compresión se consiguen disminuir los parámetros siguientes:

- Tiempo de transmisión.
- Probabilidad de error.
- Posibilidad de fragmentación de un paquete.

2.3.5. Protocolos de nivel de enlace

Las dos técnicas más utilizadas en cuanto a protocolos de nivel de enlace se refiere consisten en la corrección de errores (utilizando técnicas como el FEC) y la repetición automática de paquetes como respuesta a los mensajes ARQ.

La ventaja de estos mecanismos es que encajan perfectamente dentro de la estructura de protocolos de la red, ya que operan independientemente de los protocolos de más alto nivel.

El problema es que el uso de estos mecanismos puede afectar gravemente al funcionamiento del protocolo de la capa de transporte, en este caso, TCP, ya que puede interferir en las retransmisiones de segmentos en la semántica extremo-a-extremo del protocolo de transporte.

2.3.3. Protocolo TCP Indirecto

TCP Indirecto es uno de los primeros protocolos que partía la conexión extremos a extremo en dos. Consiste en independizar cada conexión entre un origen y un destino del resto. Es decir, una conexión entre el origen y la estación base, y otra entre la estación base y el destino.

Esta opción intenta separar la recuperación de las pérdidas sobre el enlace *wireless* de las pérdidas del enlace cableado.

Sin embargo, experimentos de la universidad de Berkeley (California) [5] demuestran que esta opción crea muchos problemas de funcionamiento de TCP ya que no está preparado para un enlace con pérdidas no debidas a congestión.

Además, los tiempos de proceso se doblan ya que la estación base debe procesarlas, lo que crea una gran ineficiencia.

Otra desventaja de este método es la ruptura de la semántica de reconocimientos TCP extremo-a-extremo, ya que los paquetes pueden ser confirmados por la estación base antes de llegar al verdadero destino.

La última desventaja de este mecanismo es la gran información que tiene que manejar la estación base, lo que complica mucho el proceso, y lo más importante, queda muy ralentizado.

2.3.4. Confirmaciones selectivas

Usando confirmaciones acumulativas con TCP, normalmente no se tiene la suficiente información para recuperar de forma rápida una pérdida entre múltiples paquetes en una sola ventana de transmisión. Los estudios demuestran que con confirmaciones selectivas la situación descrita presenta un mejor funcionamiento [4].

Las dos propuestas más interesantes de confirmaciones selectivas son TCP *SACK 's (Selective ACK 's)* y *SMART.*

- *SACK's*: Propone que cada confirmación lleve información sobre más de tres paquetes de datos aunque no hayan sido recibidos de forma sucesiva. Cada paquete de datos se describe por el final de su número de secuencia, aunque en realidad se ha demostrado que es mejor informar solamente al origen del último paquete de datos recibido [4].
- *SMART*: Las confirmaciones llevan dos tipos de información. El número de *ACK* acumulativo confirmado y el paquete de datos que provocó la generación de esta confirmación. Este mecanismo tiende a no ordenar correctamente las confirmaciones e incluso a perderlas.

2.3.5. Snoop

Snoop[3] es un elemento que trabaja a nivel de enlace y no rompe la semántica extremo-a-extremo de TCP. Su función es evitar que el emisor aplique algoritmos de control de congestión y reduzca así su ventana de congestión.

La figura 2.2. muestra el funcionamiento del *snoop*.

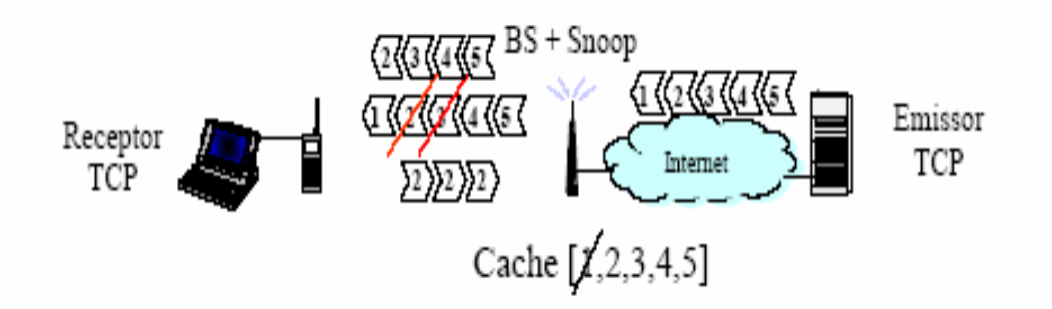

Fig. 2.2. Funcionamiento *snoop*

La función de *snoop* es monitorizar todos los paquetes en los que el terminal móvil es origen o destino. Para cada conexión mantiene un *buffer* en el que guarda los paquetes y los reconocimientos. Cada reconocimiento que recibe provoca que se eliminen los datos confirmados por este *ACK*.

Por otra parte, *snoop* gestiona un temporizador local de retransmisiones regido por la llegada de *ACK*s.

Snoop filtra los *ACKs* duplicados para que no lleguen al emisor, evitando de esta forma los mecanismos que control de congestión que el mismo emisor pondría en marcha.

Además, si *snoop* detecta *ACKs* duplicados o la expiración de su temporizador, realiza retransmisiones locales. De esta forma se pretende recuperar la comunicación cuando se producen las incidencias habituales del canal inalámbrico, como cortes o traspasos en la comunicación sin que el emisor tenga constancia.

Sin embargo, está demostrado [3] que la aplicación de *snoop* a un entorno real *wireless* de baja velocidad, por ejemplo GPRS, puede provocar una degradación aún mayor que sin presencia de este proxy. El motivo es que el *proxy* está adecuado a redes de alta velocidad, donde el retardo es pequeño y equiparable a una red fija.

En un entorno de baja velocidad, el retardo alto provoca, la mayoría de las veces, que la utilización de *snoop* no evite las retrasmisiones del emisor, lo que provocará la activación de los mecanismos de control de congestión.

Además, al eliminar los *ACKs* duplicados, las retransmisiones se darán siempre a causa de la expiración del *timer*, lo que provocará el inicio con *slow start*. En cambio, sin el *snoop* y con la llegada de los *ACKs* duplicados se hubieran utilizado los mecanismos de *fast retransmit* y *fast recovery* que hubieran mejorado la duración de la comunicación.

2.3.5.1 Snoop adaptado

Debido a los problemas que provoca la utilización del *snoop* en un entorno como el introducido en el punto 2.1. se propone la utilización de un *snoop* adaptado. Parte de este trabajo se centrará en desarrollar algunos cambios en el módulo *snoop* del simulador *Network Simulator.*

Snoop adaptado conserva las funciones originales y añade una funcionalidad específica para actuar de forma eficiente en enlaces de baja velocidad. Se trata de que conjuntamente a cada envío de un *ACK* hacia el emisor, se añade un cierre de ventana o una apertura.

Cuando el *ACK* que se está enviando es duplicado, se añade el cierre de ventana. De esta forma, el emisor no enviará más paquetes, por lo que no recibirá más *ACKs* duplicados. *Snoop* se encarga de continuar la comunicación

con el receptor hasta que recibe un *ACK* de nuevos datos. En este caso lo reenviará hacia el emisor junto a una apertura de ventana.

De esta forma evita que expire el temporizador de retransmisión del emisor y neutraliza el efecto negativo de los mecanismos de control de congestión.

En la figura 2.3. se puede ver el funcionamiento básico de la funcionalidad añadida.

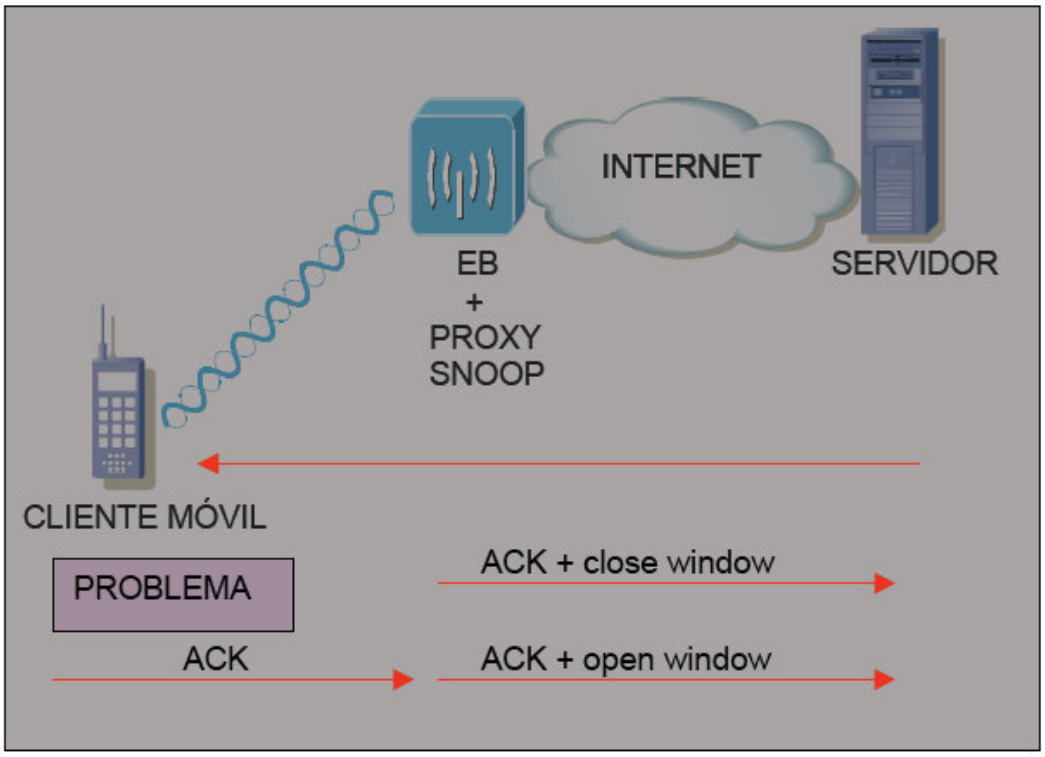

Fig. 2.3. Funcionamiento *snoop* adaptado

Además de dicha funcionalidad, se adaptará también el temporizador de retransmisiones de snoop. Un temporizador demasiado pequeño comportará retransmisiones innecesarias por parte del proxy. Por otra parte, este temporizador tampoco debe ser mayor que el del emisor, ya que sino no evitaría la aplicación de los algoritmos de control de congestión.

Existe la posibilidad de que el temporizador del snoop sea fijo o adaptativo (calculado).

En caso de que sea fijo, los recursos se minimizan y en caso de cortes largos la situación se recuperará más fácilmente tal y como está explicado en la referencia [3].

En cambio, si se utiliza un temporizador adaptativo, aumenta la flexibilidad ante situaciones como traspasos, degradaciones del canal, canal compartido por varios usuarios, etc.

En nuestro caso, dado que las condiciones de nuestra simulación y el tamaño de nuestros segmentos son estables, nos decidimos por un temporizador fijo.

CAPÍTULO 3. *NETWORK SIMULATOR*

Network Simulator [8] es una herramienta informática cuyos inicios se remontan a 1989, cuando comenzó como una variante al, ya existente, *REAL Network Simulator*. Debido al gran interés que mostraron en el programa diversas instituciones interesadas en la investigación de redes, como LBL, Xerox, UCB, etc. *Network Simulator* ha evolucionado sustancialmente. Prueba de ello son las numerosas versiones que ofrece el simulador.

3.1. Introducción

Para la realización del proyecto, utilizaremos la herramienta *Network Simulator* (NS en adelante). NS es un software simulador de eventos discretos diseñado y creado para la ayuda a la investigación sobre redes telemáticas. NS ofrece un gran soporte para la simulación de multitud de protocolos de la capa de aplicación como pueden ser http, ftp, cbr o telnet; de la capa de transporte donde encontramos TCP, UDP, RTP o SRM y de enrutamiento, *unicast* o *multicast*. Además, se puede crear cualquier topología de red con un gran número de generadores de tráfico. Estas características son aplicables tanto en redes cableadas como en redes *wireless* e incluso en redes híbridas, aunque éste último aspecto no se ha desarrollado en su totalidad como veremos en puntos posteriores. El simulador también contempla mecanismos referentes a la capa de enlace de datos en redes locales, como el protocolo MAC (Control de Acceso al Medio) del tipo CSMA/CD. Así mismo, incluye diferentes algoritmos para la planificación de colas como FQ (*Fairness Queuing*) o FIFO (*First In First Out*).

3.2. Funciones y herramientas

Para poder llevar a cabo las funcionalidades descritas en la introducción, NS dispone de diferentes funciones que permiten al usuario crear escenarios con condiciones muy particulares e incluso realizar modificaciones en protocolos de transporte. Además, NS posee herramientas externas al simulador en sí, para facilitar la lectura de resultados al usuario.

3.2.1. NAM

Una de las herramientas más conocidas que complementa el funcionamiento de NS es la llamada *Network Animator* (NAM), cuya utilidad es la de representar gráficamente la red que previamente hayamos definido mediante comandos escritos en un lenguaje de programación llamado Tcl y posteriormente compilados por NS. Así pues, NAM permite visualizar dinámicamente una simulación en la que NS ha generado un fichero de resultados, a partir de un *script* que se ha programado previamente con la topología de la red, los protocolos, los parámetros y demás actividad posible en la red, en lenguaje Tcl.

3.2.2. Lenguajes de programación

NS se trata de un simulador orientado a eventos, que utiliza dos lenguajes de programación bien diferenciados como son C++ y OTcl. El primero es el lenguaje mediante el cual NS trabaja 'internamente' y OTcl (lenguaje de los *script*s) es el lenguaje de fachada al usuario. El simulador soporta dos jerarquías de clases, una de ellas escrita en C++ [16] a la que denominan jerarquía compilada y la otra escrita en OTcl que se conoce como jerarquía interpretada. Ambas jerarquías tienen una estrecha relación. Desde el punto de vista del usuario, hay una relación equivalente (1:1) entre una clase de la jerarquía interpretada y una clase de la jerarquía compilada. Cuando se crea un nuevo objeto simulador en un *script* Tcl, este objeto se crea duplicado una vez en cada jerarquía.

Según las necesidades que tenga el usuario, deberá hacer cambios en un lenguaje de programación o en otro. Los cambios que se efectúan en clases programadas en C++ requieren compilar y linkar todo el código de la clase, por lo que requiere mayor tiempo de iteración; en cambio, los cambios que se efectúan en el *script,* son cambios que no precisan compilado ni linkado. Por ejemplo, si queremos hacer un cambio en el protocolo de transporte, modificaremos la clase correspondiente en C++ y después, compilaremos y linkaremos el código; sin embargo, si queremos cambiar la velocidad de un enlace o la posición de un nodo, lo haremos en el *script*, y el cambio se aplicará en la siguiente ejecución.

3.2.3. Simulaciones

Mediante el programa se pueden realizar simulaciones de todo tipo. Se pueden crear topologías utilizando el medio *wireless*, cableado o híbrido; variar el número de nodos; tener diferentes generadores de tráfico; movilidad en los nodos, etc. Todos ellos podríamos decir que se tratan de cambios 'físicos' y estas descripciones las crea el usuario cuando programa su *script* mediante el lenguaje Tcl. La simplicidad del lenguaje Tcl facilita la creación de escenarios concretos mediante *scripts*.

3.2.3.1 Trazas

Al finalizar una simulación realizada con NS, se genera un archivo de resultados en el que se guardan una serie de parámetros para cada evento generado durante la simulación. A continuación se observa un ejemplo de traza sobre una simulación TCP en el escenario de la figura 3.1. y el significado de los parámetros para su estudio.

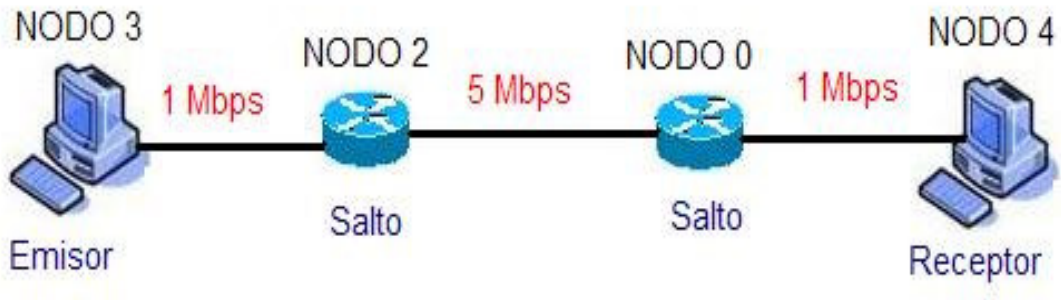

Fig. 3.1. Escenario de la simulación para el estudio de la traza.

r 22.744625 4 0 *ACK* 40 ------- 0 4.0 3.0 7 15
Para interpretar la traza:

r - Es el identificador del tipo de Evento. (r en caso de una recepción, h en caso de un salto de nodo, + en caso de inserción del segmento en cola y - en caso de expulsión del segmento de la cola).
22.744625 - Es el tiempo en el que se produce el evento. (Segundos)
4 - Es el último nodo que envía el segmento.
0 - Es el nodo siguiente que recibe el segmento.
ACK - Es el tipo de segmento. (Si es de datos - tcp, si es de reconocimiento - *ACK*).
40 - Es el tamaño del segmento. (Bytes)
-------- - Son los flags TCP que se pueden determinar en algunos casos.
0 - Es el identificador de flujo.
4.0 - Dirección del nodo origen del Paquete.
3.0 - Dirección del nodo destino del paquete.
7 - Número de secuencia del segmento.
15 - Identificador único de evento. Cada vez que se genera un evento durante la simulación, NS le asigna un identificador único.

Así pues, de esta traza podemos concluir que se trata de un *ACK* que ha generado el nodo 4 (con dirección 4.0) que en el instante 22.744625 lo ha recibido el nodo 0 (que es el nodo contiguo al 4); este *ACK* confirma la recepción de datos hasta el segmento de datos con número de secuencia 7 y va destinado al nodo con dirección 3.0 (nodo 3).

3.2.3.2 Definición de escenarios: Scripts.

Para crear un *script* Tcl en primer lugar hay que tener claro el escenario a implementar (topología, velocidad de enlaces, protocolo/s a utilizar, generadores de tráfico, formato de la traza de resultados, etc.).
A continuación se explica paso a paso como definir un escenario sencillo para simular, utilizando Tcl [9]. El escenario se corresponde con la figura 3.2.

Fig. 3.2. Escenario para ejemplo de *script* tcl

En primer lugar se debe crear un nuevo objeto simulador (que, como se ha descrito antes se duplicará en las dos jerarquías de clases que utiliza NS) mediante el comando

```
set NS [new Simulator]
```

Una vez se tiene el objeto simulador creado, se definen los archivos que se utilizarán para exponer los resultados, que en la mayoría de los casos son dos: uno de trazas con extensión .tr y uno para la visualización dinámica mediante NAM con extensión .nam. A los dos archivos se les da permiso de escritura añadiendo una "w" al final del comando.

```
set tr_f [open nombre_archivo.tr w]
set nam_f [open nombre_archivo.nam w]
$ns trace-all $tr_f
$ns namtrace-all $nam_f
```

El siguiente paso corresponde a la descripción de la topología la red, definiendo nodos, enlaces, posiciones y movimientos de nodos, si es oportuno. En el ejemplo se crean dos nodos fijos con un enlace directo cableado dúplex con velocidad de 1 Mbps y un retardo de 5 ms, además se aplica una disciplina de cola de descarte.

```
set nodo1 [$ns node]
set nodo2 [$ns node]
```

```
$ns duplex-link $nodo1 $nodo2 1Mb 5ms DropTail
```

A continuación, se debe adjuntar a cada nodo un 'Agente'. Los Agentes en NS se utilizan para aplicar funcionalidades de la capa de transporte en un nodo creado en Tcl basándose en un Módulo programado en C++. Es decir, si se pretende que el emisor implemente TCP *New Reno*, se adjunta como agente el Módulo correspondiente como se muestra a continuación

```
set tcp [new Agent/TCP/Newreno]
$ns attach-agent $nodo1 $tcp
```

En este caso el nodo receptor no requiere un agente específico que implemente muchos métodos ya que solo debe encargarse de generar los *ACKs* correspondientes según los datos recibidos correctamente. Para esto hay un agente definido en NS con el nombre de Sink; así que, adjuntamos el agente sink al nodo 2 y creamos el enlace a nivel de transporte (TCP) entre ambos nodos de la siguiente manera

```
set sink [new Agent/TCPSink]
$ns attach-agent $nodo2 $sink
$ns connect $tcp $sink
```

Por último queda crear el generador de tráfico y adjuntárselo al nodo emisor y dar la orden de comienzo de simulación. Como ejemplo, se propone un generador de tráfico FTP.

```
set ftp [new Application/FTP]
$ftp attach-agent $tcp

$ns at 10.0 "$ftp start"
$ns at 50.0 "stop"

$ns run
```

La simulación tiene una durada de 50 segundos, y en el segundo 10, empieza a generar tráfico el nodo 1 hacia el nodo 2.

3.2.3.3 Módulos

Como se ha visto en el punto anterior, en NS, cuando queremos que un nodo posea un protocolo de transporte, como TCP o UDP, descrito en clases de la jerarquía compilada (C++), se le debe adjuntar un Agente. Estos agentes que se definen en el *script* se relacionan con la jerarquía compilada y generan una vinculación entre el nodo y una clase programada en C++ a la que, en NS, se le llama Módulo. NS posee un gran número de módulos que implementan diferentes protocolos. Los módulos son los encargados de implementar las funciones que debe realizar un nodo cuando recibe un evento. Resumiendo, mediante un *script* Tcl, NS es capaz de generar una serie de eventos, que los

tratará un módulo escrito en C++ y al finalizar la simulación creará un archivo de resultados mediante el cual estudiar la simulación.

En NS hay definidos muchos módulos, que representan multitud de protocolos; para este proyecto, los módulos más interesantes a tratar son los módulos que implementan el protocolo TCP y el módulo que implementa *Snoop*. Los módulos pueden ser modificados o creados por el usuario programando en C++, así que el potencial que ofrece este simulador es muy grande. Para crear un módulo en NS se debe programar un archivo .cc y uno .h si es necesario, y mediante el compilador de linux, generar un archivo .o (el compilador usado es gcc).

En este estudio se han revisado de manera específica los siguientes módulos:

tcp.cc - tcp.h
tcp-newreno.cc - tcp-newreno.h
tcp-full.cc - tcp-full.h
snoop.cc - *snoop*.h
tcp-sink.cc

Para adjudicar los agentes (que llaman a los módulos en C++) en los nodos de nuestro escenario, debemos tener en cuenta que hay varios módulos que implementan características de diferentes TCP.

En NS, todos los módulos que implementan variantes de TCP heredan de un módulo jerárquicamente superior, llamado tcp (con sus correspondientes archivos .cc y .h), que almacena todas las características comunes para toda la gama de TCPs; como por ejemplo, almacena la estructura que define la cabecera TCP, las funciones que generan paquetes TCP, temporizadores y estimaciones de RTT, etc.

Para aclarar la clasificación de estos módulos, según sus características, se puede observar la figura 3.2.

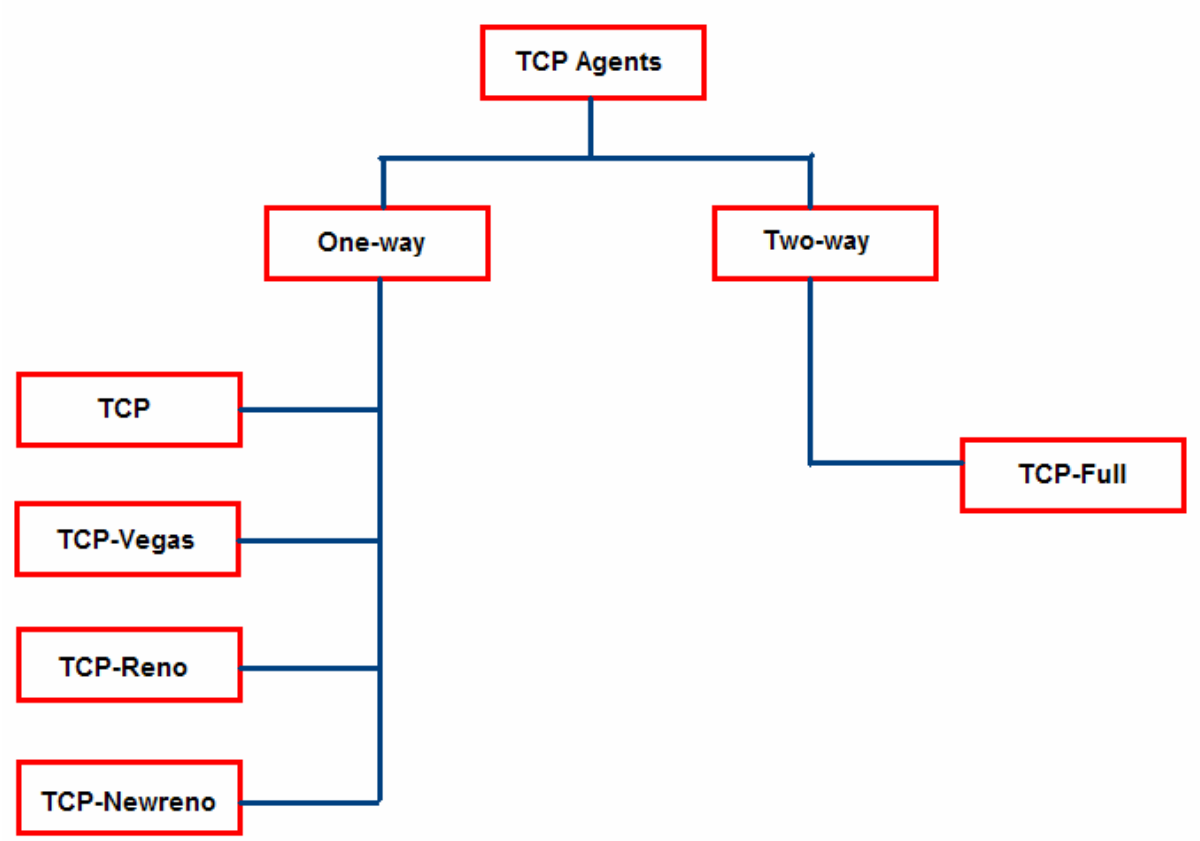

Fig. 3.3. Módulos TCP *One-way* y módulo *TCP Two-way*.

Los agentes *TCP One-way* son agentes emisores TCP, es decir, sólo se implementan en los nodos que actúen como fuente de datos; también decir, que los agentes TCP no generan tráfico, para eso se define un generador de tráfico que luego se adjunta al agente como se explica en el siguiente capítulo. Estos módulos se diferencian entre ellos en que cada uno utiliza diferentes TCPs. Sin embargo, el agente *TCP Two-way* actúa en ambos lados de la transmisión, tanto en la fuente como en el receptor. Este agente de doble sentido está aún bajo desarrollo por lo que posee características que pueden limitar el uso de NS. En la tabla 3.1 se diferencian los agentes de un único sentido de los de doble sentido.

	TCP One-way	TCP Two-way
SYN/FIN	No	Si
Tx datos	Unidireccional	Bidireccional
Nº seq.	Segmentos	Bytes

Tabla 3.1 Diferencias Agentes TCP One-way y Two-way.

La principal diferencia se observa que es la capacidad del agente Two-way de actuar tanto como emisor como receptor. En consecuencia, los nodos One-way necesitan un agente exclusivo que se dedique a reconocer todos los paquetes recibidos. Este agente se llama TCP Sink y tiene varias funcionalidades que se describen a continuación:

- Básica

  Devuelve un *ACK* por cada paquete recibido y el tamaño del *ACK* se puede configurar en el módulo.

- Delayed-Ack

  Con esta función se reconocen varios paquetes con un mismo *ACK*, ya que sería capaz de reconocer todos los segmentos recibidos mientras espera una retransmisión de un paquete anterior. Esta función también permite configurar el tiempo entre *ACKs*.

- SACK

  Permite utilizar S*ACK* (Selective Ack), explicado en el capítulo 1. Se puede modificar la cantidad de información (en bloques) en un *ACK*, que, por defecto, son 3 bloques.

CAPÍTULO 4. MODIFICACIONES Y CARACTERÍSTICAS CRÍTICAS DE NS

Para realizar el estudio del proyecto, se deben crear unas adaptaciones, a nivel de programación, en el simulador. En éste capítulo se describen las modificaciones efectuadas en el simulador y los problemas que, éste mismo, presenta al efectuarlas.

4.1. Modificaciones en NS

Un módulo se forma siempre como mínimo por 2 archivos: el primero, el módulo en sí, con su constructor y sus métodos; segundo, el archivo .o que se genera al compilar el módulo creado (previa modificación del *Makefile* para agregarlo) y, en la mayoría de los casos, también un archivo de librería (.h) que se incluye en el módulo (.cc). El nombre del archivo .cc y .o deben ser idénticos.

En el caso de la creación de un nuevo módulo hay que tener en cuenta diversos aspectos. El primero surge cuando se introducen nuevas variables.Al no estar definidas en ninguna librería (.h) del simulador, NS obliga la generación de un archivo de librería .h para definir las nuevas variables e incluido en el archivo .cc. Esto bien se podría evitar utilizando variables ya definidas en las librerías del simulador, y adjuntando la librería en el nuevo módulo.

Básicamente, cada módulo está compuesto por un método *TclClass*, que es el que se llama desde el *script* Tcl, y a partir de ahí se crea un objeto en C++ con su método constructor y los demás métodos que se activan por eventos. Hay que tener en cuenta que, después de cada cambio efectuado en algún módulo de NS, se tiene que compilar y linkar el módulo que haya sufrido cambios.

La creación de los *script*s resulta bastante más sencilla, ya explicada en el capítulo 3. Cuando se crea un archivo tcl se debe ubicar en la carpeta raíz de NS (en el caso de esta versión, el directorio es "ns-allinone-2.29/ns-2.29", mientras que los módulos están clasificados por categorías (por ejemplo, el directorio "ns-allinone-2.29/ns-2.29/tcp" contiene todos los módulos que implementan el protocolo TCP con sus diferentes versiones como el *New Reno*, *Tahoe, Vegas* y demás). En la figura 4.1 se observa parte de la distribución de los archivos más importantes en NS.

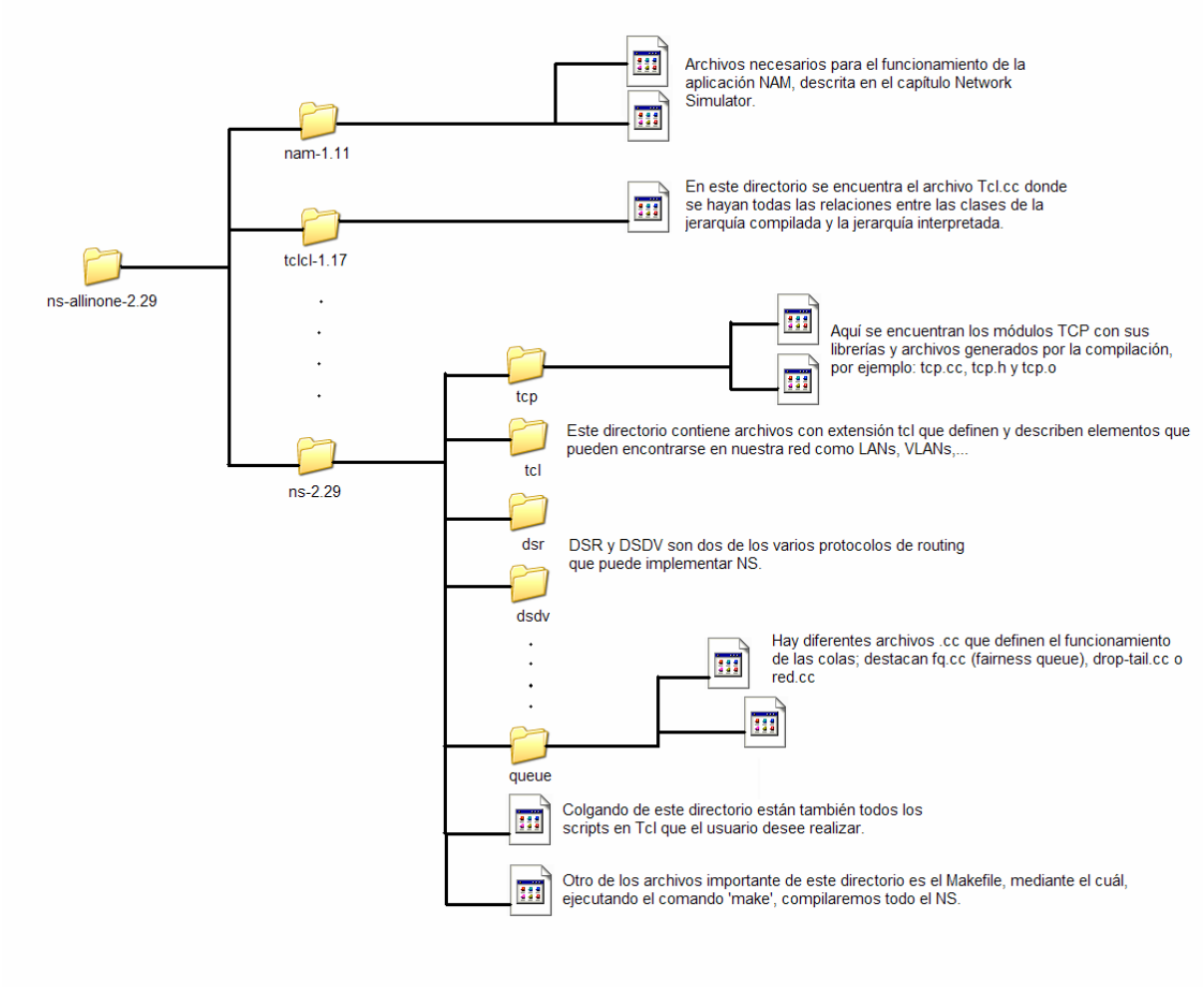

Fig. 4.1. Estructura de directorios y archivos en NS

Antes de empezar a explicar las modificaciones efectuadas en los módulos de NS, debe quedar claro cual es el objetivo, cuál es realmente la simulación que queremos llevar a cabo y con qué valores de parámetros. En principio, el objetivo era plantear un escenario con 3 nodos como el que se presenta en la figura 4.2.

Fig. 4.2. Escenario ideal para realizar el estudio.

Se observa que se trata de una red híbrida con un nodo intermedio snoop, que debe intentar mejorar las condiciones que ofrecen las comunicaciones TCP sobre *wireless*. Desgraciadamente, en NS existe una característica importante, se trata de un problema de protocolos de *routing*. El simulador posee varios protocolos de *routing*, pero al intentar hacer una red híbrida, los protocolos no son compatibles, ya que NS intenta aplicar el protocolo a toda la red, y al ser híbrida hay una parte que difiere. Muchos foros en Internet [12] comentan la problemática de este asunto y la posibilidad de encontrar soluciones en futuras versiones del simulador.

La solución que se ha adoptado para este contratiempo es la de simular un enlace *wireless* con un enlace cableado, pero introduciéndole un modelo de errores con una probabilidad de error que no se aleje de un caso extremo en cuanto a condiciones desfavorables pero real, aprox. 0.05 (5%).

Las modificaciones que se han realizado durante el transcurso del proyecto en los módulos NS son:

- Cabecera TCP
- Snoop
- TCP New Reno

4.1.1. Cabecera TCP

Como se aprecia en la figura 1.2. es necesario aplicar una cabecera TCP para poder ofrecer sus funcionalidades. La cabecera real de TCP se compone de los campos que se observan en la figura 4.3.

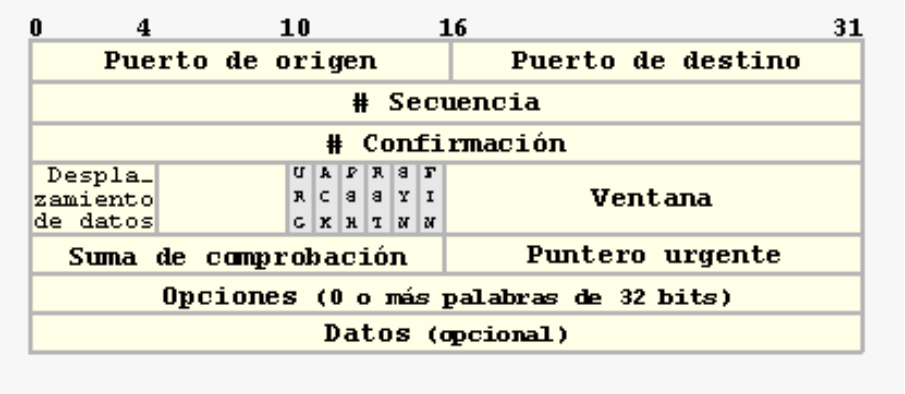

Fig. 4.3. Cabecera TCP

La modificación más importante, aunque sencilla, es la inserción de un nuevo campo en la cabecera TCP del paquete definido en NS. Este campo toma el nombre wnd_. La agregación éste campo en la cabecera TCP surge de la necesidad de avisar al emisor para que cierre o abra la ventana. Así pues,

snoop insertará un valor u otro en éste campo (siempre en una cabecera correspondiente a un *ACK*) para que el nodo emisor compruebe el valor del mismo al recibir cualquier *ACK*, y actuar en consecuencia.

En NS, la cabecera TCP se traduce a C++ como una estructura de datos llamada hdr_tcp, que se encuentra en el archivo de librería tcp.h (véase anexo 3). La estructura tiene la forma del código 4.1.

```
struct hdr_tcp {
      #define NSA 3
      double ts_;       /* time packet generated (at source) */
      double ts_echo_;  /* the echoed timestamp (originally  */
                  /*sent by the peer) */
     int seqno_;              /* sequence number */
     int reason_;             /* reason for a retransmit */
     int sack_area_[nsA+1][2];    /* sACK blocks: start, end */
                              /* of block */
     int sa_length_;          /* Indicate the number of SACK sin*/
                 /* this packet. Adds 2+sack_length*8 bytes */
      int ackno_;              /* ACK number for FullTcp */
      int hlen_;               /* header len (bytes) for FullTcp */
      int tcp_flags_;          /* TCP flags for FullTcp */
      int last_rtt_;           /* more recent RTT measurement */
                               /* in ms, */
                                       /*   for statistics only */
      int wnd_;             /* Para SnoopAdaptado */

      static int offset_;      // offset for this header
      inline static int& offset() { return offset_; }
      inline static hdr_tcp* access(Packet* p) {
            return (hdr_tcp*) p->access(offset_);
      }

      /* per-field member functions */
      double& ts() { return (ts_); }
      double& ts_echo() { return (ts_echo_); }
      int& seqno() { return (seqno_); }
      int& reason() { return (reason_); }
      int& sa_left(int n) { return (sack_area_[n][0]); }
      int& sa_right(int n) { return (sack_area_[n][1]); }
      int& sa_length() { return (sa_length_); }
      int& hlen() { return (hlen_); }
      int& ackno() { return (ackno_); }
      int& flags() { return (tcp_flags_); }
      int& last_rtt() { return (last_rtt_); }
      int& wnd() { return (wnd_); }
};
```

Cód. 4.1. Estructura hdr_tcp.

En el trozo de código mostrado, observamos como insertamos el campo wnd_ al final de la cabecera TCP definiéndola como una variable int, ya que siempre recibirá valores enteros, que le asignará snoop. La segunda sentencia que incluye la nueva variable, se utiliza para permitir el acceso a ésta y su modificación desde otro módulo como es el snoop.

4.1.2. Snoop

Durante el transcurso del proyecto, el módulo que sufre los cambios más significativos es el módulo snoop. El módulo esta pensado para actuar en redes híbridas con dos partes bien diferenciadas, una parte *wireless* y una parte cableada. La diferencia más destacable respecto los módulos tcp se produce a la hora de duplicar los objetos en la jerarquía compilada, porque, en el caso de snoop, se crean dos objetos dentro de esta jerarquía, uno llamado Snoop y otro LLSnoop. Esto se produce para que snoop pueda trabajar en la capa la LLC (LLSnoop) y, a la vez, aplicar funcionalidades que no pertenecen estrictamente a esta capa (Snoop). Ésta característica se produce para poder diferenciar la parte *wireless* de la parte cableada.

Cuando NS genera un evento en forma de paquete que se recibe en un nodo snoop, el primero en recibirlo es el objeto LLSnoop, ya que se sitúa una capa por debajo del objeto Snoop. LLSnoop, se encarga de comprobar la procedencia del paquete. Si llega desde el lado *wireless*, llama a un método llamado Snoop::handle(), el cual, llama al método snoop_ack() o al método snoop_wless_data(), según sea un paquete de datos o de reconocimiento.

Mientras que, si el paquete llega desde el lado cableado, se llama al método Snoop::recv(), que hace las mismas funciones que Snoop::handle(), diferenciando ahora en, snoop_data y snoop_wired_ack, con los mismos condicionantes que en el anterior caso. El tratamiento de un evento en snoop sigue el recorrido mostrado en la figura 4.4.

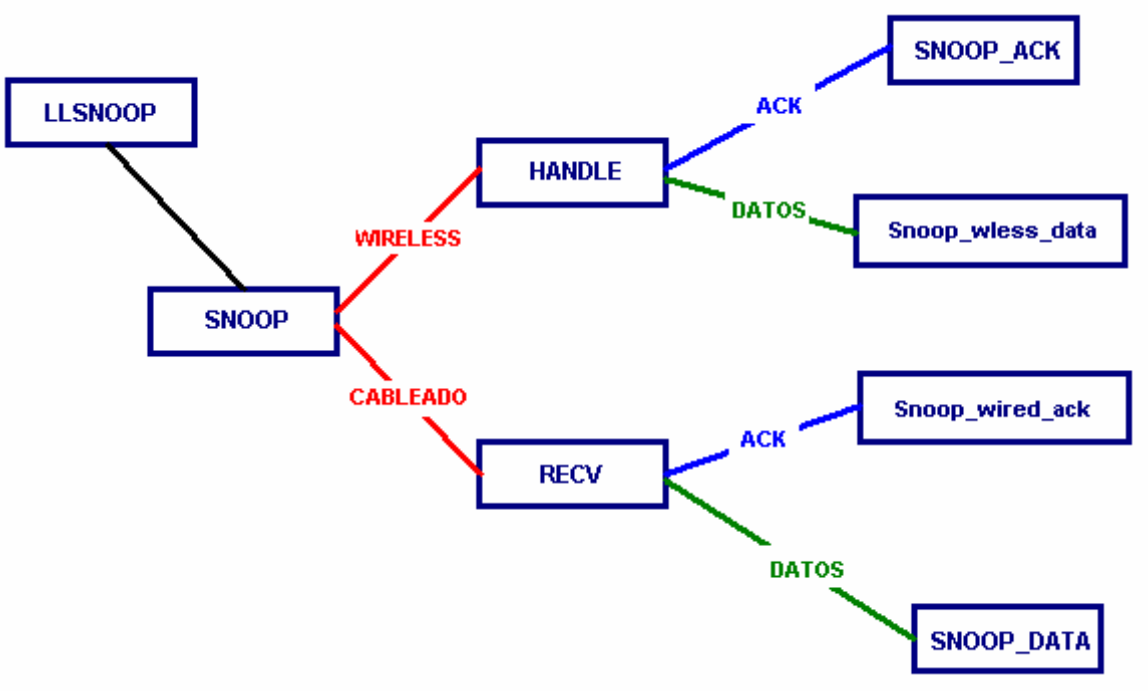

Fig. 4.4. Tratamiento de un evento en Snoop.cc

Como se ha comentado, antes de empezar a hacer modificaciones en un módulo, debe haberse realizado un estudio completo de éste, con el objetivo de clarificar los cambios a realizar y su coste temporal aproximado.

4.1.2.1. Adaptación cierre y apertura ventana emisor

En la figura 4.4 observamos como las clases más importantes en relación con el estudio son snoop_ack y snoop_data ya que los paquetes de datos, en el escenario de nuestra red, llegan al snoop desde el lado cableado y los reconocimientos llegan desde el lado *wireless*. En las figuras 4.5. y 4.6. se muestran las acciones que realizan las clases snoop_ack y snoop_data al recibir un *ACK* y un paquete respectivamente.

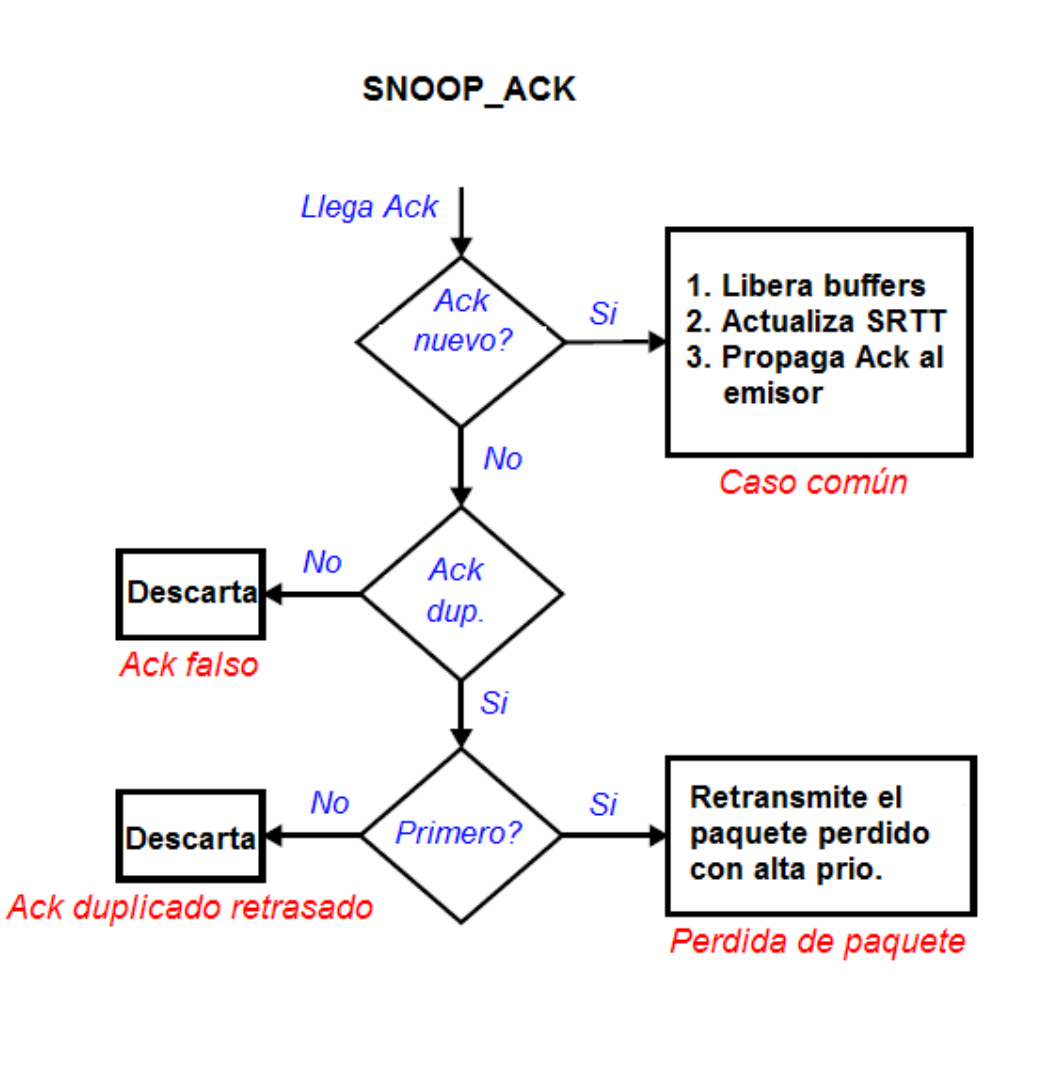

Fig. 4.5. Snoop_ack()

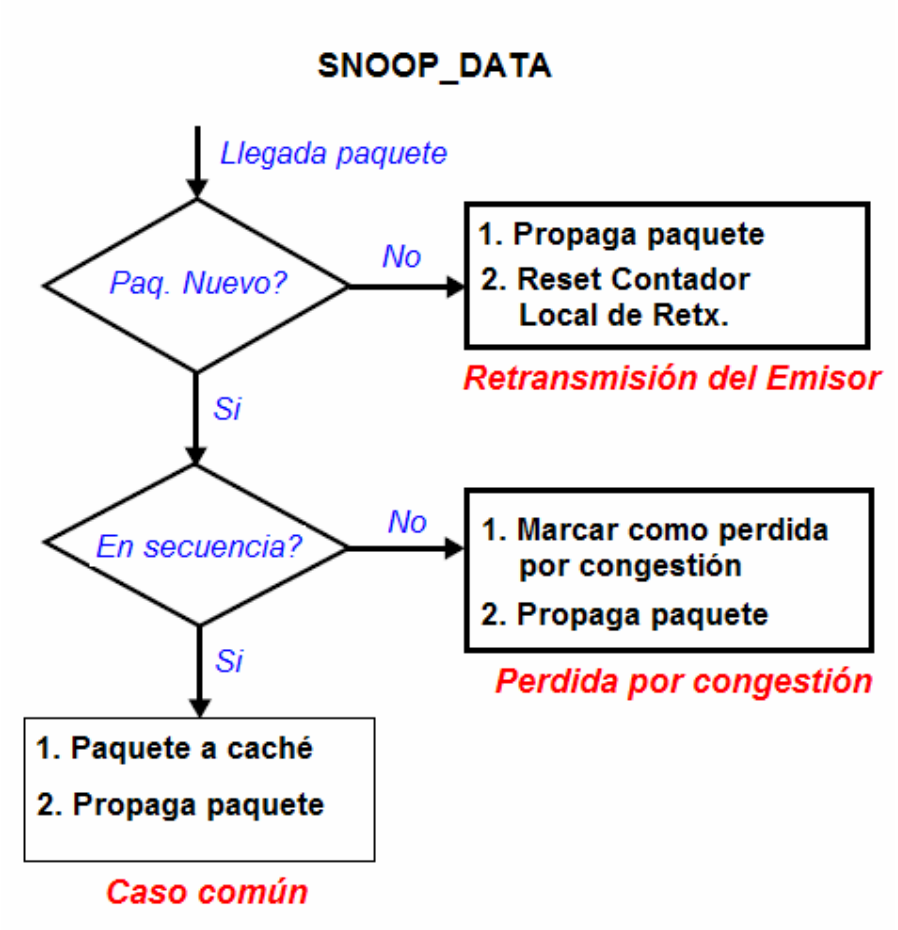

Fig. 4.6. Snoop_data()

Así pues, sabiendo que clases se deben modificar, se tiene que plantear el código a modificar. Sabiendo que la modificación de Snoop (véase anexo 2) debe notificar al emisor los cambios en la ventana mediante un campo de la cabecera TCP de un paquete de reconocimiento, debemos agregar en la función *snoop_ack* el código 4.2.

```
if ((oc == 1) && (lastAck_ >= ack) && (maxseq_ < ack) {
                hdr_tcp::access(p)->wnd()=2;
                oc = 2;
               // printf("SNOOP. CLOSE\n");
  }

else if ((oc == 2) && (lastAck_ < ack)) {
                hdr_tcp::access(p)->wnd()=1;
                oc = 1;
                //printf("SNOOP. OPEN\n");
  }
```

Cód 4.2. Código agregado en snoop_ack.

En concreto, este código se ejecutará cuando llegue un *ACK* de la parte *wireless*.

El código del primer *if* se ejecuta para cambiar el valor del campo wnd_ y avisar al emisor de que debe cerrar la ventana de congestión. La variable oc es una variable auxiliar, creada para evitar dos cierres de ventana consecutivos, o dos aperturas, *lastAck* _ hace referencia al último *ACK* recibido correctamente en secuencia y la variable *ack* se refiere al *ACK* que acaba de llegar. Entonces, entrará en cerrar ventana si no es la segunda vez consecutiva que se debe cerrar y si el *ACK* que se acaba de recibir es duplicado. Le dará al campo wnd_ un valor de 2. En cambio, mandará un abrir ventana cuando hayamos cerrado previamente la ventana y volvamos a recibir un *ACK* en secuencia correctamente, insertará un 1 en el campo wnd_ de la cabecera TCP.

Se utilizan los valores 1 y 2 para el campo wnd_ ya que por defecto, cuando se crea un paquete, el valor de este campo es 0. Para no tener problemas si se comprueba este campo en un paquete en el que no se han hecho cambios (no se ha cerrado ni abierto la ventana) utilizamos los valores 1 y 2.

4.1.2.2. Adaptación timer retransmisión

Según se ha visto en el punto 2.3.5. también se tiene que adaptar el temporizador de retransmisiones de snoop. Por defecto, el valor no está pensado para transmisiones por enlaces de baja velocidad por lo que requiere una modificación.

El retardo sobre un enlace del tipo GPRS varía según el tamaño de los datos transmitidos. En el proyecto se utilizan paquetes de 536 bytes de datos, por lo que, respetando el gráfico de la figura 4.7, se prevé un RTO estimado de 1.5 segundos. Se utiliza un RTO fijo ya que el escenario de pruebas es único y cambian pocas características a la hora de hacer las transmisiones. En otro caso (ver punto 2.3.5.) sería conveniente programar un cálculo de RTO adaptado a las muestras de RTT calculadas en el simulador por el snoop.

En la figura 4.7. se observan los retardos adecuados para configurar el temporizador de retransmisión de snoop en función de la longitud del segmento [3].

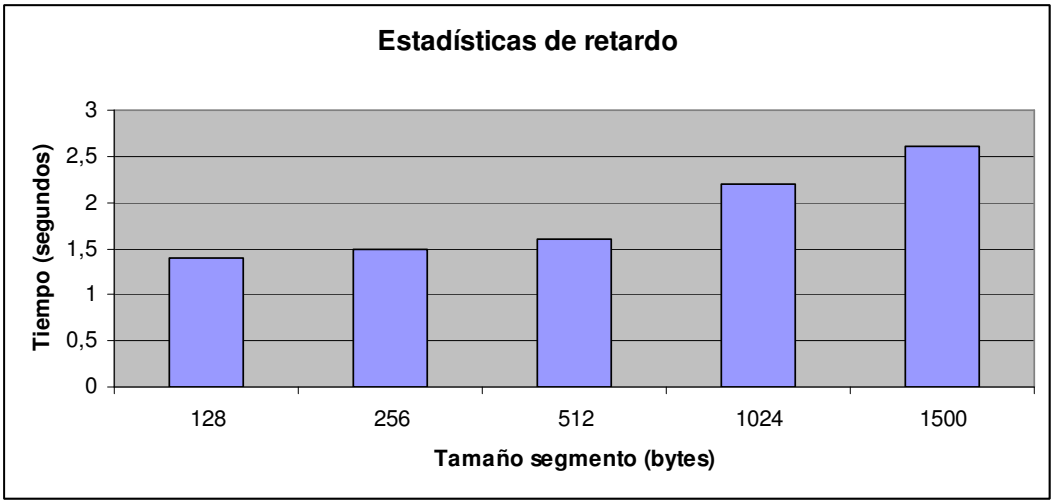

Fig. 4.7. Estadísticas de retardo sobre GPRS

4.1.3. TCP New Reno

Ya está creado el campo de la cabecera que se utilizará para indicar el cierre o apertura de la ventana, también está modificado el snoop para que realice las inserciones de valor en ese campo correctamente y por último queda modificar el módulo TCP para que lea este campo y actuar en consecuencia abriendo y cerrando la ventana. Debido a una característica de NS, se utiliza el módulo tcp-newreno.cc (véase anexo 1), en una primera vista, parece mejor utilizar el módulo tcp-full.cc, más adelante se explica el motivo de la elección.

El módulo TCP tiene un funcionamiento similar a snoop, en cuanto al cambio que queremos hacer se refiere. *TCP New Reno* tiene una función recv() que es a la que se llama cuando el módulo recibe un paquete; en el caso del escenario planteado, al emisor sólo le llegan paquetes *ACK*, por lo tanto, dentro de la función recv() insertaremos un trozo de código que compruebe el valor del campo wnd_. El código insertado es el siguiente:

```
if (tcph->wnd() == 2) {
      cwnd_ = 0.0;
      //printf("VENTANA REDUCIDA\n");
}

if (tcph->wnd() == 1) {
      cwnd_ = wnd_;
      //printf("VENTANA AMPLIADA\n");
}
```

Cód 4.3. Lectura del campo wnd_ en TCP *New Reno*.

Esta modificación permite que TCP abra o cierre la ventana de congestión según el valor que haya insertado snoop en el campo wnd_ de la cabecera TCP. Los valores de apertura y cierre poseen una motivación: para el cierre de

la ventana se escoge valor 0 para evitar que el emisor siga generando tráfico en la red durante las retransmisiones, y, para la apertura, se escoge el valor de wnd_ para que la transmisión se reinicie con el valor de la ventana inicial.

4.2. Características críticas de NS

NS es un programa que ofrece un gran potencial en cuanto a libertad y uso de sus herramientas; aunque también cabe remarcar que ciertos aspectos del simulador están aún bajo desarrollo por lo que, de alguna manera, restringen funcionalidades en algunos escenarios concretos. Durante el transcurso del estudio se han detectado diversas características a nivel del simulador que han impedido realizar rigurosamente la simulación del escenario planteado como objetivo.

Sin embargo, debido al gran potencial que tiene NS, permite salvar las dificultades aplicando otras alternativas.

Un ejemplo de ello es un aviso que genera el simulador con la frase *"code omitted because of length"*. Exactamente no conocemos el alcance de este aviso, ya que aparentemente no afecta a la simulación. Se deduce que es un aviso conforme el simulador ejecuta una gran cantidad de órdenes que pueden ocasionarle cierta saturación. La variable que controla la cantidad de código se denomina *MAX_CODE_TO_DUMP*, que se define como un entero de valor (8*1024). Aumentando el valor de la variable, el mensaje de aviso deja de aparecer en pantalla.

4.2.1. Implementación red híbrida

En NS, a la hora de intentar aplicar el escenario ideal para realizar las simulaciones, presenta un punto crítico cuando se define una red híbrida. Éste es uno de los aspectos que están en periodo de desarrollo en NS.

Concretamente el problema surge al definir el protocolo de encaminamiento para la parte móvil, ya que sólo se puede definir para toda la red. Quizás la agregación de una estación base que diferenciase las dos partes de la red podría solventar la situación. Sin embargo, la solución adoptada ha sido la inserción de un modelo de errores en un enlace cableado que simule fidedignamente un enlace *wireless*.

El modelo de errores aplicado para es el mostrado en el código 4.4.

```
set loss_module [new ErrorModel]
$loss_module set rate_ 0.05
$ns lossmodel $loss_module $n0 $d
```

Cód. 4.4. Modelo de errores para simular enlace *wireless*.

Se ha elegido un 5% de BER, que en la realidad sería un caso extremo (muchos errores) para poder diferenciar aún más el comportamiento de snoop.

Se puede observar la situación del modelo de errores dentro del *script* en el Capítulo 5 donde se muestra el código entero del archivo Tcl. Así pues, debido a esta limitación, el escenario sufre la primera modificación como se muestra en la figura 4.8:

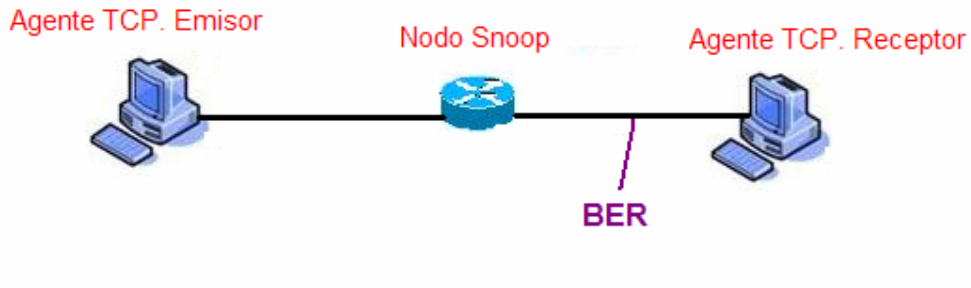

Fig. 4.8. Primera modificación del escenario a simular.

4.2.2. Generación de snoop

Sabemos que *Snoop* actúa en la capa LLC por lo que, la capa, posee diferencias a las capas pertenecientes al mismo nivel del resto de nodos. Por este motivo es necesario la utilización del elemento LAN para introducir el *Snoop*, ya que sino, únicamente podríamos definir una capa del nivel LLC para todos los elementos de la red.

En consecuencia a la utilización de la LAN, también debemos utilizar un nodo auxiliar que se sitúe entre ésta y el nodo receptor.

Los pasos para crear la topología de red son los siguientes:

- Antes de empezar se deben incluir en el código los archivos que definen LANs para poder crearla:

```
Source /root/Desktop/ns-allinone-2.29/ns-
2.29/tcl/lan/vlan.tcl
source /root/Desktop/ns-allinone-2.29/ns-2.29/tcl/lan/ns-
mac.tcl
```

- Como hemos dicho antes, se define la capa LLC (entre otras opciones que se comentan en el capítulo siguiente) de toda la red y nodos que se creen sobre ella:

```
set opt(ll) LL
```

- Se define un nodo que servirá de salto:

```
set n0 [$ns node]
lappend nodelist $n0
```

- Acto seguido, se conecta este nodo directamente a una LAN que creamos con el siguiente código:

```
set lan [$ns make-lan $nodelist $opt(bw) $opt(delay) $opt(ll) $opt(ifq) $opt(mac) $opt(chan)]
```

- Ahora se inserta el nodo snoop en la LAN cambiando antes su capa LLC de la siguiente manera:

```
set opt(ll) LL/LLSnoop
set opt(ifq) Queue/DropTail
$opt(ifq) set limit_ 100
set NSnoop [$ns node]
$lan addNode [list $nsnoop] $opt(bw) $opt(delay) $opt(ll) $opt(ifq) $opt(mac)
```

  Se observa como además de definir la capa sobre la que trabajará el nodo creado (en este caso LLSnoop), se definen también otros parámetros como el tipo de cola que implementará el interfaz del nodo y su capacidad.

- La LAN interconecta el nodo 0 (salto) y el nodo NSnoop (snoop), ahora queda introducir el emisor y receptor; estos irán fuera de la LAN, con enlaces directos a los nodos ya creados (los valores se modifican según las necesidades de la simulación, se tratará en el capítulo 5, donde se especifican las condiciones de trabajo de toda la red). Se crea de la siguiente forma:

```
set s [$ns node]
set d [$ns node]

$ns duplex-link $s $nsnoop 1Mb 100ms DropTail
$ns queue-limit $s $nsnoop 100000
$ns duplex-link-op $s $nsnoop orient right

$ns duplex-link $n0 $d 0.024Mb 10ms DropTail
$ns queue-limit $n0 $d 100000
$ns duplex-link-op $n0 $d orient left
```

Haciendo esto, se tienen en cuenta diferentes aspectos que deben permitir hacer la simulación del escenario con la mayor realidad posible. Se debe conseguir que la LAN creada, ya que en un principio no debería porque definirse, no limite en ningún aspecto el funcionamiento del escenario y la simulación, por lo que se especifican los parámetros pensando en ello.

El escenario con la segunda modificación queda como se ve en la figura 4.9.

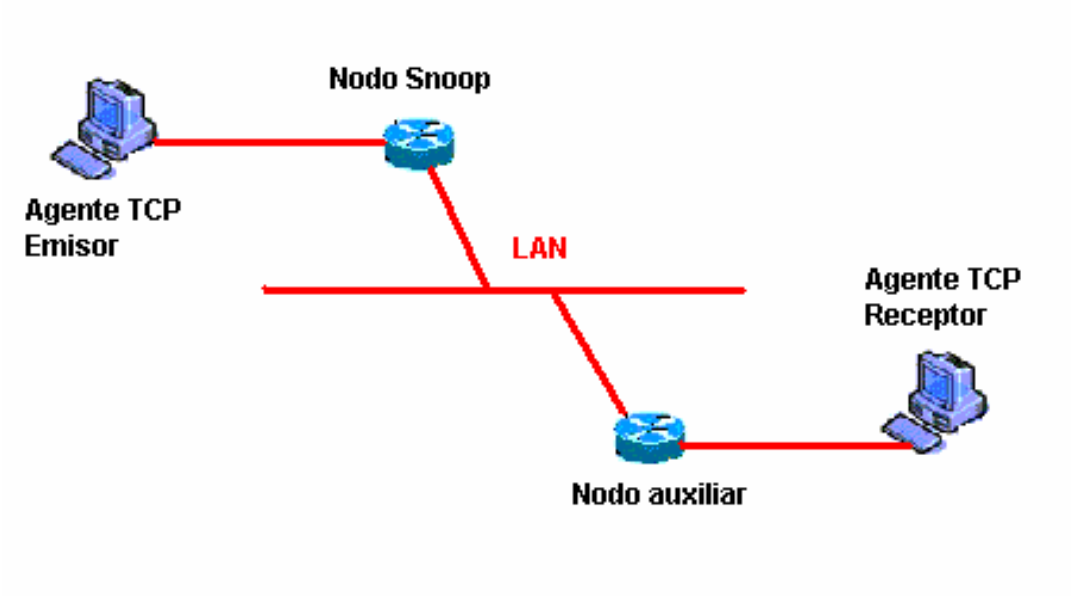

Fig. 4.9. Escenario con segunda modificación.

4.2.3. Elección del Agente TCP.

En el momento de decidir el módulo que usaremos en el escenario, el más adecuado para hacer la simulación resulta el módulo de dos sentidos *FullTCP*, aunque aún está bajo desarrollo y cada cierto tiempo se resuelven *bugs* y se realizan mejoras. La elección de *TCP Full* viene motivada por el aumento del número de secuencia y de *ACK* en *bytes* y por la realización de SYN/FIN al inicio de la conexión. Sin embargo, NS presenta problemas cuando en la comunicación de *FullTCP* interviene un nodo *snoop*, ya que el módulo Snoop, a diferencia del módulo *FullTCP*, trabaja aumentando los números de secuencia y de *ACK* en segmentos.

Recordando el funcionamiento de Snoop, éste almacena los paquetes que le llegan del emisor en un *búffer* para hacer futuras retransmisiones. Según el número de *ACK* que le llegue del receptor, hará una retransmisión o propagara el *ACK* hacia el emisor, por lo que una no concordancia entre los números de secuencia que utilizan el agente TCP y el agente Snoop tiene consecuencias negativas para la simulación. Una posible solución sería la adaptación de snoop para que aumentara números de secuencia y de *ACK* en *bytes*. La solución es compleja, aunque, según algunos comentarios en foros sobre el simulador, prevén una mejora sustancial a medio plazo en futuras versiones [12].

El módulo definitivo es TCP *New Reno* (tcp-newreno.cc), con los cambios comentados anteriormente, que implementa las funcionalidades descritas en el capítulo 1, introductorio a TCP.

CAPÍTULO 5. ESCENARIOS

5.1. Explicación de los escenarios

En los escenarios simulados el objetivo es comprobar las diferencias entre la utilización de *snoop*, *snoop* adaptado y no utilizar *snoop* para un escenario tipo GPRS.

Las explicaciones que se hacen en este punto tienen como referencia la figura 5.1.

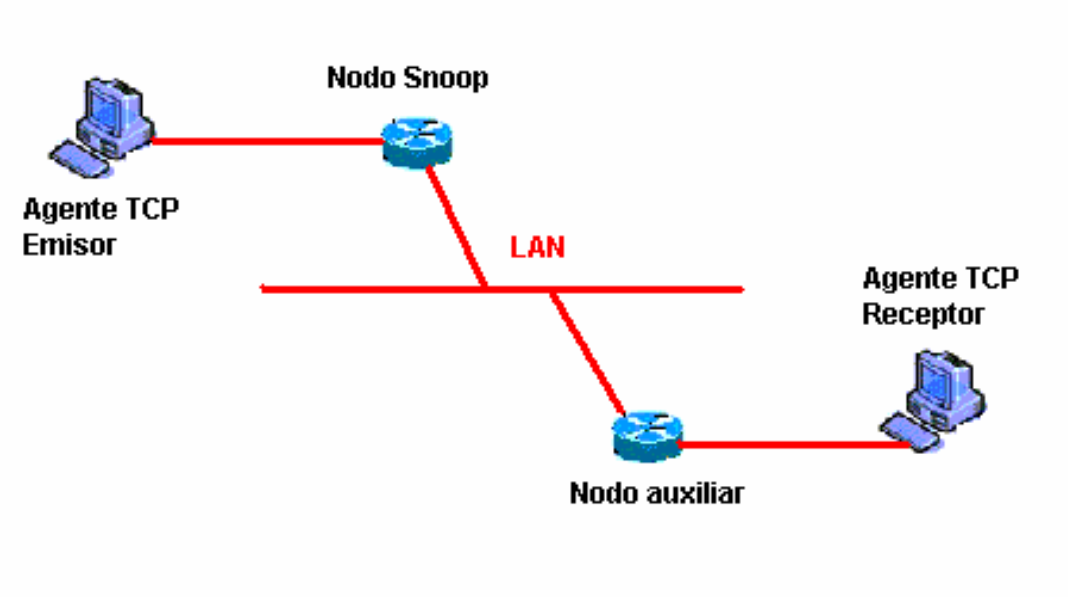

Fig. 5.1. Configuración virtual de los escenarios

En concreto, se simula un nodo (Receptor) en uno de los extremos del escenario. Este nodo se conectará mediante un enlace *wireless* con un nodo auxiliar.

El enlace *wireless* se simula mediante un enlace cableado incluyéndole una función de errores cuya tasa de error podremos elegir desde el *script* tcl que presentaremos en un punto posterior.

El nodo snoop antes nombrado se conecta, mediante un enlace cableado, esta vez sin incluir ningún tipo de error, con un nodo final (Emisor).

El nodo snoop y auxiliar forman parte de la LAN intermedia.

En una red real esto no es exactamente así, ya que el nodo *snoop* se añade a la estación base de GPRS, explicada anteriormente en el punto 2.1. En este caso *Network Simulator* necesita para simular la inclusión del nodo *snoop*, la creación de dos nodos unidos por una LAN según se ha visto en el punto 4.2.3. Esta LAN en ningún caso limitará las prestaciones del escenario ya que los parámetros de velocidad y retardo con la que la configuraremos no serán limitadores respecto al resto de enlaces del escenario.

5.1.1. Escenario con snoop

En el escenario con snoop simulamos un receptor (Sink) conectado con el nodo auxiliar mediante un enlace *wireless* simulado (cableado con una función de pérdidas). El nodo snoop tiene añadido el módulo *snoop* aún sin modificar, es decir, sin añadirle la funcionalidad de abrir y cerrar la ventana ni la adaptación del temporizador. El nodo auxiliar se conectará mediante la LAN antes explicada con el nodo snoop. Por último, el nodo snoop se conecta con un nodo emisor (TCP New Reno).

5.1.2. Escenario con snoop adaptado

El escenario con snoop adaptado es exactamente igual que el anterior en cuando a distribución de los elementos. El cambio que realizamos se trata de añadir la funcionalidad de abrir y cerrar la ventana del emisor en *snoop* y la adaptación del temporizador, antes explicada en el punto 2.2.5.

5.1.3. Escenario sin snoop

El escenario sin snoop también es virtualmente igual que los anteriores en cuanto a los elementos. Su diferencia es que el nodo snoop no tendrá añadido ningún tipo de *snoop* trabajando a nivel de enlace, es decir, simplemente será un nodo más auxiliar.

5.2. Parámetros variables para los escenarios

En NS cada escenario se define mediante un *script* como ha quedado explicado en el punto 3.2.3.2. En este *script* se pueden varias algunos parámetros de la simulación que se realizará mediante dicho escenario.

En la tabla 5.1. se aprecian los diferentes parámetros que se pueden variar en cada módulo de TCP de NS.

Algunos de los parámetros variables de tcp-full no están explicados puesto que no hemos trabajado en profundidad con él y no han sido utilizados.

	TAHOE	SACK'S	VEGAS	RENO	NEW RENO	FULL
seqno: número de secuencia	X	X	X	X	X	X
rtt: tiempo ida y vuelta	X	X	X	X	X	X
srtt: rtt estimado	X	X	X	X	X	X
rttvar: varianza del rtt	X	X	X	X	X	X
backoff: valor de backoff	X	X	X	X	X	X
dupacks: ack's duplicados	X	X	X	X	X	X
wnd_: ventana ofrecida	X	X	X	X	X	X
cwnd: ventana de congestión	X	X	X	X	X	X
packetsize: tamaño del paquete	X	X	X	X	X	
ack: número de ack	X	X	X	X	X	X
ssthresh: umbral	X	X	X	X	X	X
maxseq: último número de secuencia recibido	X	X	X	X	X	X
ndatapack: cantidad paquetes recibida	X	X	X	X	X	X
ndatabytes: cantidad bytes recibida	X	X	X	X	X	X
nrexmit: número de retransmisiones	X	X	X	X	X	X
nrexmitpack: cantidad de paquetes diferentes retransmitidos	X	X	X	X	X	X
nrexmitbytes: cantidad bytes retransmitidos	X	X	X	X	X	X
necn: número de veces que flag ecn (explicit congestion notification) ha sido activado	X	X	X	X	X	X
partial_ack: número de *ACK* parcial recibido		X				
exit_recovery_fix: desactivación de fast recovery					X	
alpha			X			
beta			X			
gamma			X			
segsize: tamaño del segmento						X
segsperack: número de segmentos a recibir para generar un ack						X
tcprexmtthresh: umbral para la activación de fast recovery y fast retransmit						X
iss: número de secuencia inicial						X
data_on_syn: activación de carga de datos en el syn						X
dupseg_fix						X
dupack_reset						X
close_on_empty						X
signal_on_empty						X
interval						X
ts_option_size						X
reno_fastrecov						X
pipectrl						X
open_cwnd_on_pack						X
halfclose						X
nopredict						X
spa_thresh						X

Tabla 5.1. Parámetros variables en los escenarios

5.3. Definición del script de los escenarios

El *script* que hemos utilizado para realizar las simulaciones es el siguiente.

Sobre él explicamos en color rojo los comandos utilizados así como las diferencias en su ejecución para realizar cada una de las simulaciones:

```
# El comando “puts” presenta por pantalla la texto que se pone a continuación
entre comillas. El comando source utiliza los archivos de configuración cuya
ruta le pasamos donde están configurados algunos parámetros para que snoop
funcione correctamente.

puts "sourcing ../../lan/vlan.tcl..."
source /root/Desktop/ns-allinone-2.29/ns-2.29/tcl/lan/vlan.tcl
source /root/Desktop/ns-allinone-2.29/ns-2.29/tcl/lan/ns-mac.tcl
source /root/Desktop/ns-allinone-2.29/ns-2.29/tcl/lan/ns-ll.tcl

# Variables de nuestra red simulada, necesarias para caracterizar nuestra LAN
set opt(bw) 10Mb
set opt(delay) 0,01ms
set opt(ll) LL
set opt(ifq) Queue/DropTail
set opt(mac) Mac/802_3
set opt(chan) Channel


# Iniciamos un nuevo objeto simulador
set NS [new Simulator]

# Hace que la cabecera TCP sea visible a nivel tcl y poderla mostrar en las
trazas
Trace set show_tcp_hdr_ 1

# Creamos los archivos donde se guardaran las trazas y resultados de la
simulación
set tr_f        [open tresnodos.tr w]
set nam_f       [open tresnodos.nam w]
set record_f    [open tresnodos.data w]

# Indica para que se utlizaran los archivos que acabamos de crear
anteriormente
$ns trace-all $tr_f
$ns namtrace-all $nam_f

# Creamos un nodo simple que actuará como salto y lo añadimos a una lista de
nodos para poder crear nuestra LAN a partir de dicha lista
set n0 [$ns node]
lappend nodelist $n0
```

```
# Creamos nuestra LAN, utilizando las variables de red y la lista de nodos creados
set lan [$ns make-lan $nodelist $opt(bw) $opt(delay) $opt(ll) $opt(ifq) $opt(mac) $opt(chan)]

# Creamos el nodo snoop, variamos la capa de enlace para que funcione con snoop y añadimos el nodo a nuestra LAN. Los nodos snoop y 0 quedan unidos mediante la LAN. En caso de hacer la simulación sin el nodo snoop la parte en la que se añade el snoop a la capa de enlace no la utilizaremos.
set opt(ll) LL/LLSnoop
set opt(ifq) Queue/DropTail
$opt(ifq) set limit_ 100
set NSnoop [$ns node]

# Añade a nuestra LAN el nodo snoop
$lan addNode [list $nsnoop] $opt(bw) $opt(delay) $opt(ll) $opt(ifq) $opt(mac)

#Crea los nodos origen y destino
set s [$ns node]
set d [$ns node]

# Creamos el enlace que une los nodos de nuestra red mediante el proxy. El valor de los parámetros en este caso son 1 Mbps de velocidad y 5 ms de retardo. Son configurables.
$ns duplex-link $s $nsnoop 1Mb 10ms DropTail
$ns queue-limit $s $nsnoop 100000
$ns duplex-link-op $s $nsnoop orient right

#Creamos el enlace mediante el que se une el destino con el nodo 0
$ns duplex-link $n0 $d 0,024Mb 100ms DropTail
$ns queue-limit $n0 $d 100000
$ns duplex-link-op $n0 $d orient left


# Modelo de errores. Se añade entre el snoop y el destino para simular un enlace wireless. En este caso tiene un valor de 0.05, pero es totalmente configurable
set loss_module [new ErrorModel]
$loss_module set rate_ 0.05
$ns lossmodel $loss_module $n0 $d

# Creamos los agentes tcp1 y tcp2 para nuestros nodos simples. En el caso de tcp se trata del nodo emisor, y los parámetros configurables que utilizamos son la ventana ofrecida, el tamaño del paquete, la ventana de congestión inicial, el umbral entre slow start y congestion avoidance, y el número de dupACK snecesario para iniciar fast retransmit y fast recovery.
set tcp1 [new Agent/TCP/Newreno]

#Este es el comando necesario si quisiéramos implementar Full Tcp, pero no lo haremos por las causas explicadas en el punto anterior.
```

```
#set tcp1 [new Agent/TCP/FullTcp]

$ns attach-agent $s $tcp1
$tcp1 set window_ 20
$tcp1 set packetSize_ 536
$tcp1 set cwnd_ 10
$tcp1 set ssthresh_ 5
$tcp1 set tcprexmtthresh_ 3

set tcp2 [new Agent/TCPSink]

#Este es el comando necesario si quisieramos implementar Full Tcp, pero no lo
haremos por las causas explicadas en el punto anterior.
#set tcp2 [new Agent/TCP/FullTcp]

$ns attach-agent $d $tcp2

#Este comando también es necesario para poner el nodo es estado LISTEN
para que la conexión se realice con éxito para Full Tcp.
#$tcp2 listen



# Creamos la aplicación que generará el tráfico que usaremos para la
simulación. Es nuestro caso utilizaremos FTP, CBR o Telnet.
#set ftp [new Application/Traffic/CBR]
set ftp [new Application/FTP]
#set ftp [new Application/Telnet]
$ftp attach-agent $tcp1

# Creamos la conexión entre los nodos
$ns connect $tcp1 $tcp2

# Función que se inicia desde el tiempo 0.0 y que guarda los valores de las
variables del sistema definidas cada 0.1 segundos
proc record {} {
   global NS tcp1 tcpr record_f
   set now [$ns now]

   set time 0.1

   set cwin     [$tcp1 set cwnd_]
   set ssthresh [$tcp1 set ssthresh_]
   set win      [$tcp1 set window_]
   set bwe 0
   set seq      [$tcp1 set t_seqno_]
   set rtt      [$tcp1 set rtt_]
   set srtt     [$tcp1 set srtt_]
   set rttvar   [$tcp1 set rttvar_]
   set bACK off  [$tcp1 set bACK off_]
```

```
    set dupACK   [$tcp1 set dupACK s_]
    set ACK      [$tcp1 set ACK _]

    puts $record_f "$now [expr $cwin*1] $ssthresh $win $bwe $seq $rtt $srtt
$rttvar $bACK off $dupACK $ACK "

    $ns at [expr $now+$time] "record "
}

# Le indicamos a la simulación cuando deben empezar los eventos
$ns at 0.0 "record"
$ns at 10.0 "$ftp start"
$ns at 50.0 "stop"
$ns at 50.01 "puts \"ns EXITING...\" ; $ns halt"

# Función que al finalizar la simulación, crea gráficas y ficheros de trazas con
las variables definidas en stop.
proc stop {} {
    global NS tr_f record_f nam_f
    $ns flush-trace
    close $tr_f
    close $record_f
    close $nam_f

    exec awk { { print $1, $2 } } tresnodos.data > temp.cwnd
    exec awk { { print $1, $3 } } tresnodos.data > temp.sst
    exec awk { { print $1, $4 } } tresnodos.data > temp.wnd
    exec awk { { print $1, $5 } } tresnodos.data > temp.bwe
    exec awk { { print $1, $6 } } tresnodos.data > temp.seq
    exec awk { { print $1, $7 } } tresnodos.data > temp.rtt
    exec awk { { print $1, $8 } } tresnodos.data > temp.srtt
    exec awk { { print $1, $9 } } tresnodos.data > temp.rttvar
    exec awk { { print $1, $10 } } tresnodos.data > temp.bACK off
    exec awk { { print $1, $11 } } tresnodos.data > temp.dupACK
    exec awk { { print $1, $12 } } tresnodos.data > temp.ACK


    exec xgraph temp.cwnd temp.sst -m -x tiempo -y valorTCP1 -geometry
600x200 &
    exec xgraph temp.wnd -m -x tiempo -y valorTCP1 -geometry 600x200 &
    exec xgraph temp.seq -m -x tiempo -y valorTCP1 -geometry 600x200 &
    exec xgraph temp.rtt -m -x tiempo -y valorTCP1 -geometry 600x200 &
    exec xgraph temp.srtt -m -x tiempo -y valorTCP1 -geometry 600x200 &
    exec xgraph temp.rttvar -m -x tiempo -y valorTCP1 -geometry 600x200 &
    exec xgraph temp.bACK off -m -x tiempo -y valorTCP1 -geometry 600x200 &
    exec xgraph temp.dupACK -m -x tiempo -y valorTCP1 -geometry 600x200 &
    exec xgraph temp.ACK -m -x tiempo -y valorTCP1 -geometry 600x200 &
    exit 0
}
```

```
puts "EMPEZANDO SIMULACION"

# Empieza la simulación
$ns run
```

5.4. Archivo .data y gráficos Xgraph

Una vez realizadas las simulaciones, se debe tener en cuenta el archivo de resultados. Una pequeña modificación que se añade en este estudio es la creación de un segundo archivo de resultados, con extensión .data. En este archivo se almacenan periódicamente valores de parámetros que implementan los agentes de los nodos, como por ejemplo, en un agente TCP podemos monitorizar cada 0.1 segundos el valor de la ventana de congestión, el umbral, el número de *ACKs* duplicados, etc. Para agregar esta nueva configuración, se debe indicar en el script Tcl de la siguiente manera, justo después de crear los archivos de trazas y de NAM comentados en el capítulo 3:

```
set record_f    [open tresnodos.data w]
```

De ésta manera se crea el archivo, pero también se debe indicar cuándo se realiza el comienzo de escritura de datos del archivo y cuáles son los parámetros que deben aparecer en él. Para ello, se crea una función con nombre *"record"* que contiene el siguiente código Tcl:

```
proc record {} {
    global NS tcp1 record_f
    set now [$ns now]

    set time 0.1

    set cwin     [$tcp1 set cwnd_]
    set ssthresh [$tcp1 set ssthresh_]
    set win      [$tcp1 set window_]
    set bwe 0
    set seq      [$tcp1 set t_seqno_]
    set rtt      [$tcp1 set rtt_]
    set srtt     [$tcp1 set srtt_]
    set rttvar   [$tcp1 set rttvar_]
    set backoff  [$tcp1 set backoff_]
    set dupACK   [$tcp1 set dupacks_]
    set ACK      [$tcp1 set ack_]

    puts $record_f "$now [expr $cwin*1] $ssthresh $win $bwe $seq $rtt
$srtt $rttvar $backoff $dupACK $ack"

    $ns at [expr $now+$time] "record "
}
```

Al principio de la función se utilizan tres variables globales ya definidas antes en el *script* del escenario: NS es la variable que se refiere al nuevo objeto

simulador, *tcp1* hace referencia al agente TCP del nodo 1 y *record_f* es el nombre que recibe el archivo nuevo de resultados creado al inicio del *script*.

Seguidamente se le debe indicar cuándo debe empezar la escritura de datos y cada cuánto tiempo debe iterar. En este caso, empieza cuando se llama a la función *record* y cada 0.1 segundos escribirá en el archivo creado una línea indicando el tiempo y los valores de los parámetros especificados en orden.

El valor del periodo de iteración se ha estimado después de realizar diversas pruebas con valores diferentes intentando llegar a una solución de compromiso entre tiempo de simulación y resolución de los resultados.

El siguiente paso es crear variables de todos los parámetros que se quieran imprimir en el archivo de resultados. Como podemos ver hay parámetros de toda índole (ventana de congestión, umbral, ventana ofrecida, número de secuencia, RTT, RTT estimado, varianza RTT, tiempo de *backoff*, número de *ACKs* duplicados y número de Acks).

La penúltima sentencia de la función es la que ordena el proceso de escritura en el archivo y el orden de impresión de los valores de cada parámetro. Y la última, permite que el proceso sea iterativo desde que se llama la función, cada cierto valor *time.*

Existe la posibilidad de, mediante el archivo .data creado, generar gráficas a través de dos herramientas internas de NS llamadas *awk* y *xgraph*. Para utilizar esta herramienta se necesita otra función para realizar la extracción de gráficos una vez finalizada la simulación. En este estudio, se utiliza una función Stop que termina la simulación y crea los gráficos de todas las variables que se han almacenado durante el transcurso de la simulación en el archivo .data (aquí vemos la importancia del valor del tiempo para realizar las iteraciones, puesto que, cuanto más reducido sea el tiempo, más precisos serán los gráficos, aunque también se necesita más tiempo para realizar la simulación). El proceso para generar los gráficos es sencillo; primero, se generan archivos temporales que almacenan únicamente el valor de un parámetro en concreto, por ejemplo:

```
exec awk { { print $1, $4 } } tresnodos.data > temp.wnd
```

En esta sentencia se llama a la función awk para que copie todos los valores de wnd_ (Ventana ofrecida) del archivo tresnodos.data al temp.wnd. $1 se refiere al valor temporal y $4 se refiere al valor del parámetro wnd_ (ya que hay que seguir el orden propuesto en la función *record*).
El siguiente paso es para crear un gráfico con el archivo temp.XXX que se haya generado, se hace de la siguiente forma:

```
exec xgraph temp.wnd -m -x tiempo -y valorTCP1 -geometry 600x200
&
```

Por orden de ejecución: llama a la herramienta xgraph para que cree un gráfico del archivo temp.wnd, dónde el eje de coordenadas **X** representará el tiempo y

el eje de coordenadas **Y** indicará el valor del campo. *–geometry* se refiere al tamaño del gráfico.
Véase la figura 4.8.

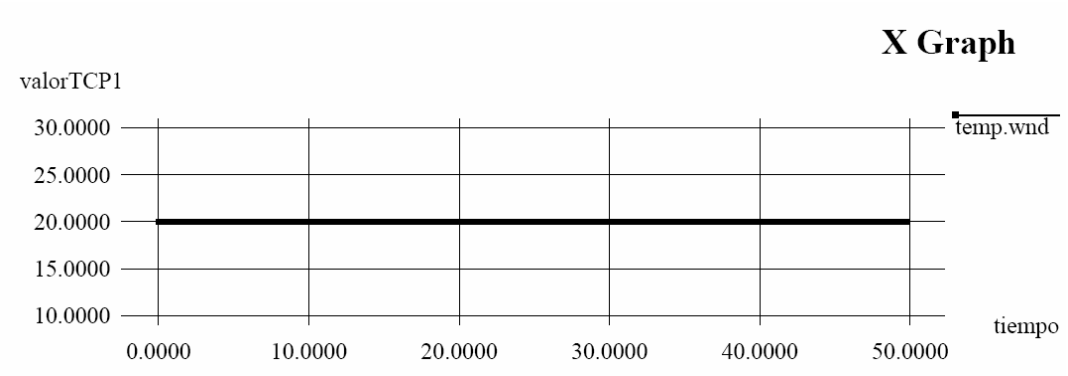

Fig. 4.8. Gráfico de ejemplo. WND_

CAPÍTULO 6. RESULTADOS DE LAS SIMULACIONES

Las condiciones de trabajo que se han simulado en los enlaces que componen el escenario son las siguientes:

- El enlace cableado será de una velocidad que variaremos en 1 Mbps y 0,1 Mbps. El retardo en este enlace será siempre de 10 ms.

- La LAN será de una velocidad de 10 Mbps y un retardo muy pequeño, del orden de 0,01 ms. Claramente ninguno de estos dos parámetros limitará el funcionamiento de nuestro escenario ya que los otros dos enlaces son más lentos y llevan asociado un retardo mayor.

- El enlace *wireless* será de 24 Kbps de velocidad con el objetivo de simular el enlace inalámbrico de una conexión GPRS. Su retardo será de 100 ms. La función de pérdidas asociada a este enlace tiene una porcentaje de pérdidas que hemos decidido fijarlo en el 5%, es un porcentaje alto de error y para elegirlo nos hemos basado en que de esta forma se podría probar el funcionamiento de snoop en unas condiciones poco favorables.

Los valores de los parámetros en el nodo emisor son los siguientes:

- La aplicación que el nodo emisor utilizará como fuente en la transmisión de datos la variaremos entre FTP y CBR.

- Tamaño del segmento: 536 *bytes* de datos + 20 *bytes* de cabecera IP + 20 *bytes* de cabecera TCP.

- El número de confirmaciones duplicadas para activar los mecanismos de *fast recovery* y *fast retransmit* es de 3.

- La ventana ofrecida inicial la fijamos en 20 segmentos.

Gracias al archivo .data explicado en el punto 4.1.4. de este proyecto se pueden visualizar todos los parámetros del sistema que se requieran, como por ejemplo: número de secuencia, de ack, backoff, rtt, srtt, varianza de rtt, umbral, ventana de congestión, ventana ofrecida, número de dupacks, etc.

En nuestro caso los valores que se van a mostrar en los resultados son las retransmisiones de segmentos hechas por el nodo emisor, por el nodo snoop y el *throughput* el cual se consigue en la transmisión de datos y la evolución del número de secuencia.

Los resultados obtenidos son los que se muestran en las tablas y figuras siguientes.

6.3. Simulación sin snoop

Aplicación	Velocidad enlace emisor – nodo intermedio 1	Retransmisiones emisor por timer	Retransmisiones emisor por dupacks
FTP	1 Mbps	1	10
FTP	0,1 Mbps	1	10
CBR	1 Mbps	1	10
CBR	0,1 Mbps	1	10

Tabla 6.1. Resultados obtenidos sin snoop

Los resultados obtenidos para la simulación con la aplicación FTP y la velocidad entre el nodo emisor y el snoop de velocidad 1 Mbps son los observados a continuación.

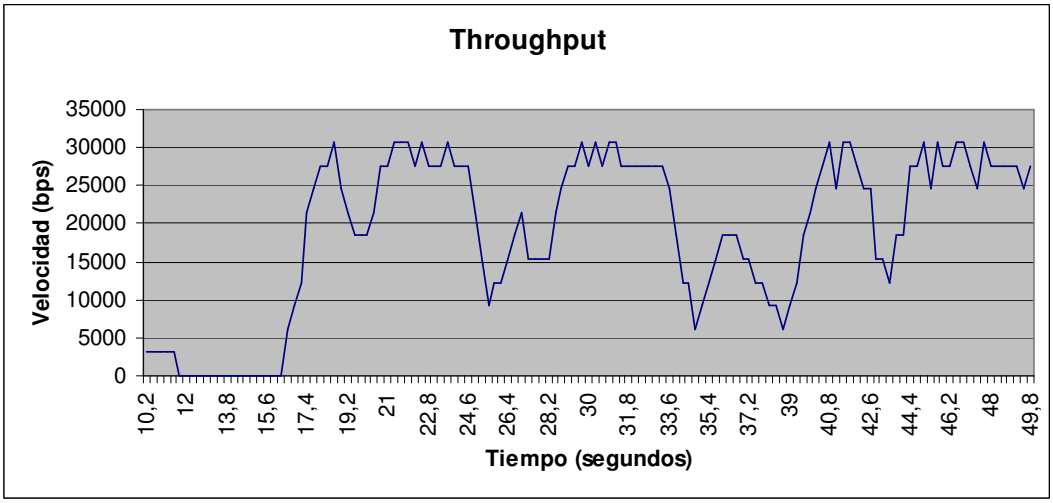

Fig. 6.1. *Throughput* sin snoop, FTP, 1Mbps emisor- snoop.

Fig. 6.2. Número de secuencia sin snoop, FTP, 1 Mbps emisor - snoop.

Los resultados obtenidos para la simulación con la aplicación FTP y la velocidad entre el nodo emisor y el snoop de velocidad 0,1 Mbps son los observados a continuación.

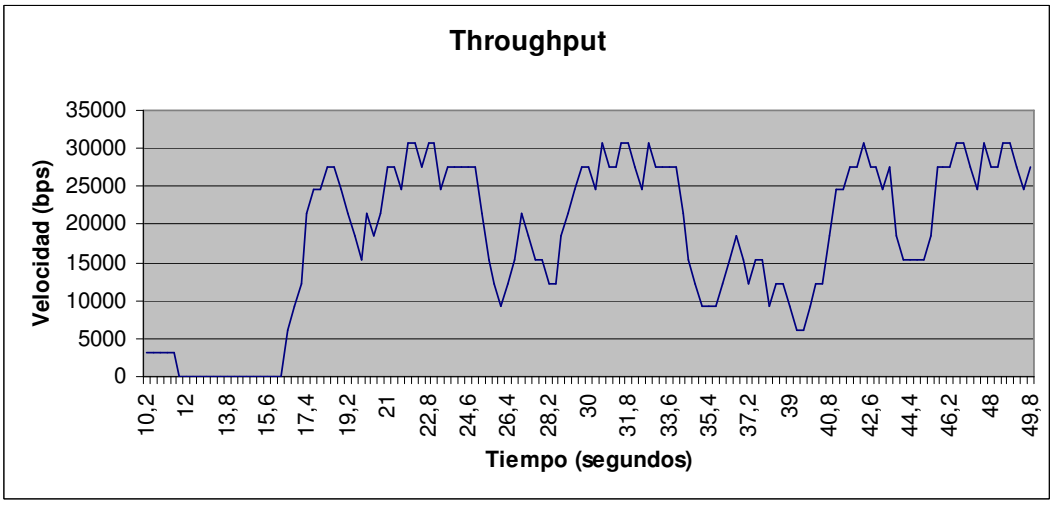

Fig. 6.3. *Throughput* sin snoop, FTP, 0.1Mbps emisor- snoop.

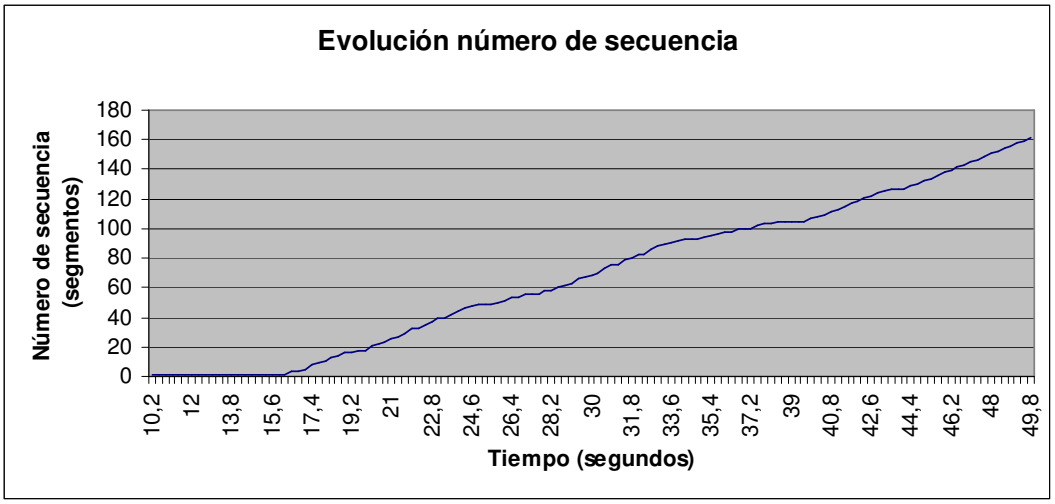

Fig. 6.4. Número de secuencia sin snoop, FTP, 0.1 Mbps emisor - snoop.

Los resultados obtenidos para la simulación con la aplicación CBR y la velocidad entre el nodo emisor y el snoop de velocidad 1 Mbps son los observados a continuación.

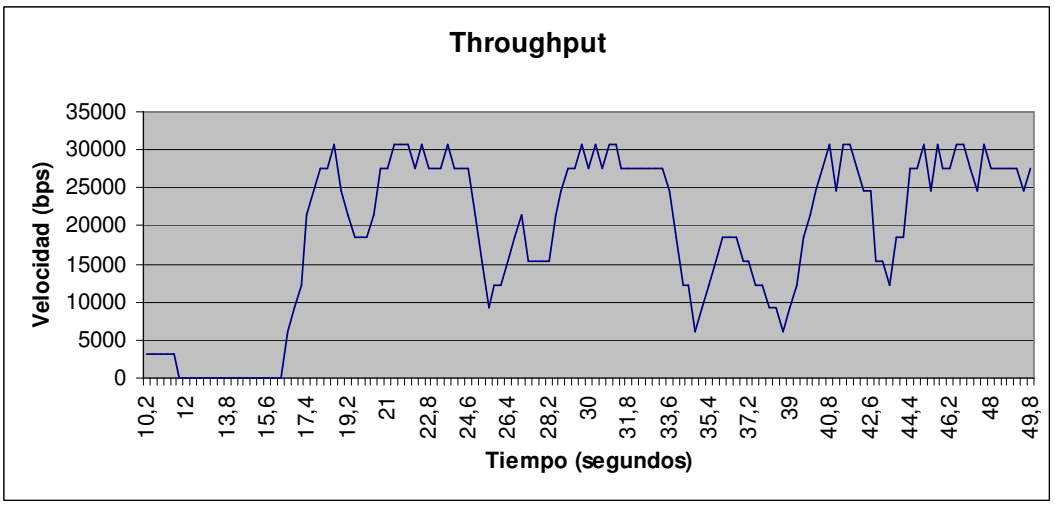

Fig. 6.5. *Throughput* sin snoop, CBR, 1Mbps emisor- snoop.

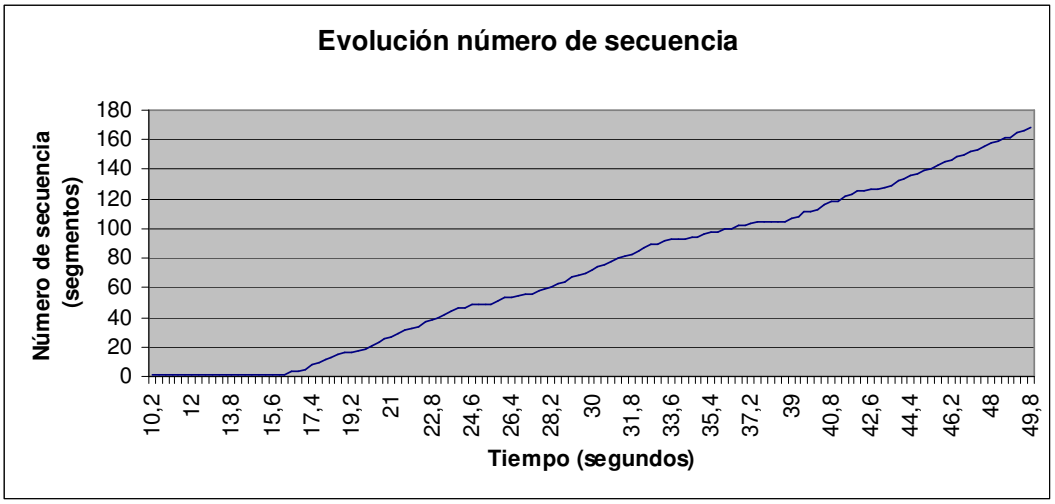

Fig. 6.6. Número de secuencia sin snoop, CBR, 1 Mbps emisor snoop.

Los resultados obtenidos para la simulación con la aplicación CBR la velocidad entre el nodo emisor y el snoop de velocidad 0,1 Mbps son los observados a continuación.

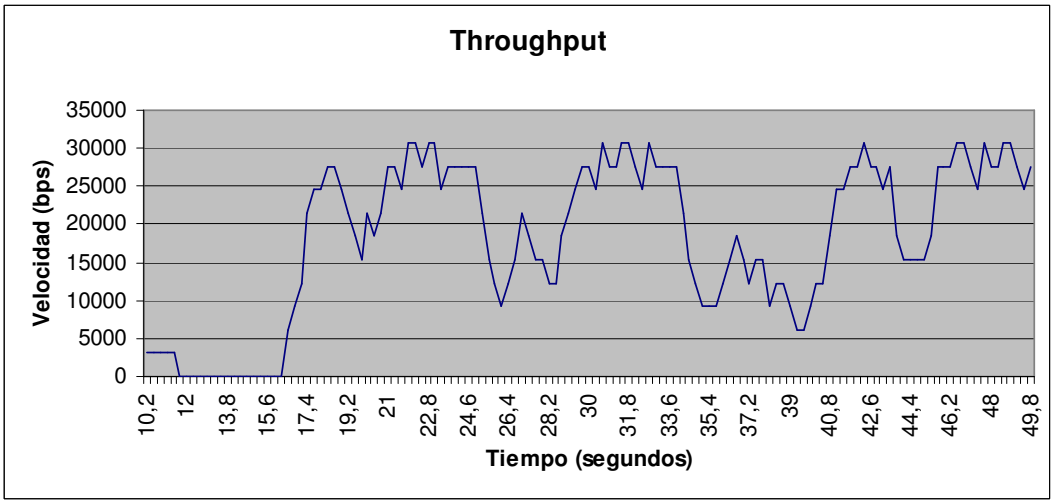

Fig. 6.7. *Throughput* sin snoop, CBR, 0.1Mbps emisor- snoop.

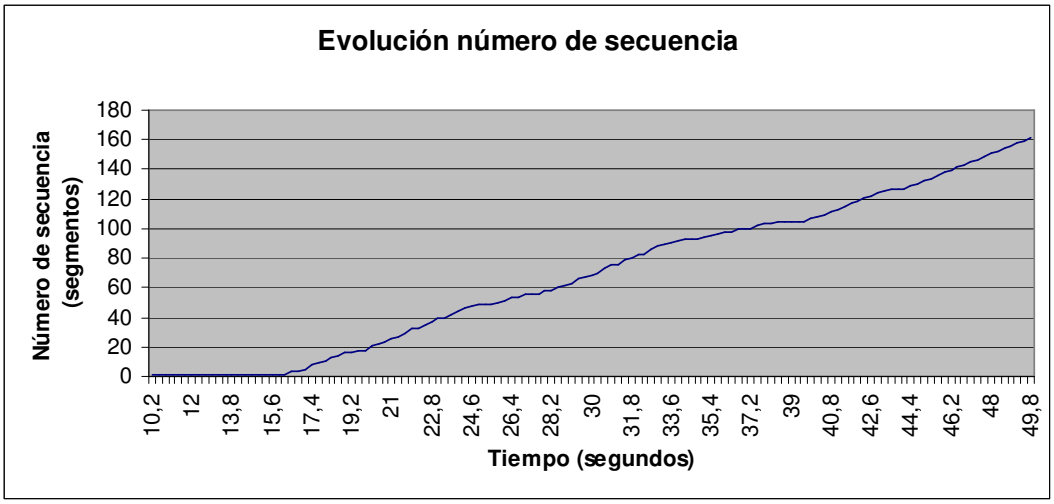

Fig. 6.8. Número de secuencia sin snoop, CBR, 0.1 Mbps emisor - snoop.

6.2. Simulación con snoop originario de NS

Aplicación	Velocidad enlace emisor - nodo intermedio 1	Retransmisiones emisor por timer	Retransmisiones emisor por dupacks	Retransmisiones totales desde nodo intermedio 1
FTP	1 Mbps	2	10	95
FTP	0,1 Mbps	1	7	60
CBR	1 Mbps	2	10	95
CBR	0,1 Mbps	1	7	60

Tabla 6.2. Resultados obtenidos con snoop sin modificar

Los resultados obtenidos para la simulación con la aplicación FTP y la velocidad entre el nodo emisor y el snoop de velocidad 1 Mbps son los observados a continuación.

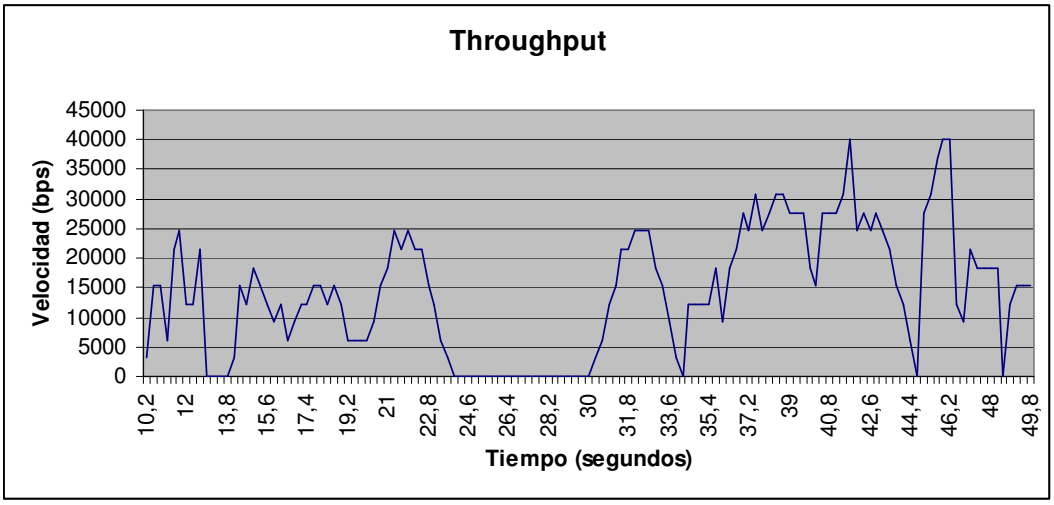

Fig. 6.9. *Throughput* con snoop, FTP, 1Mbps emisor-snoop.

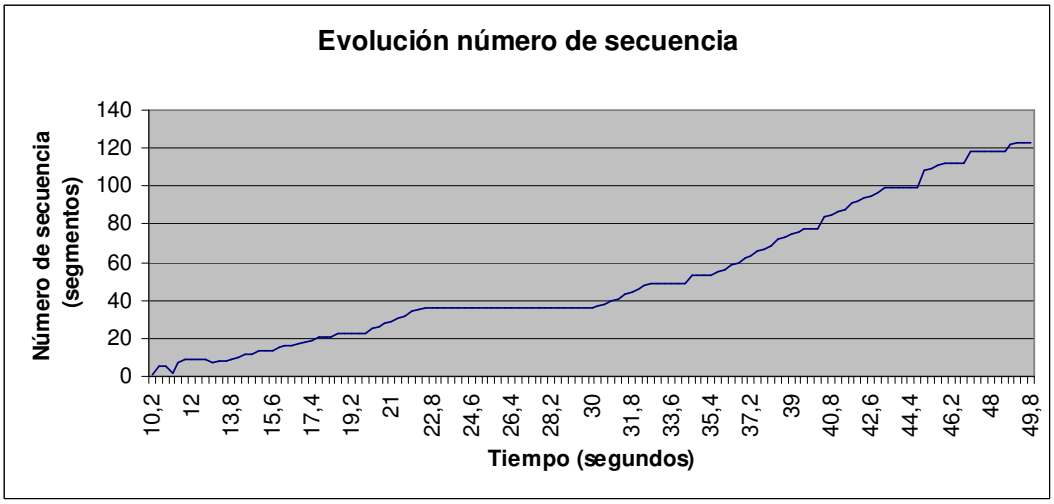

Fig. 6.10. Número de secuencia con snoop, FTP, 1 Mbps emisor - snoop.

Los resultados obtenidos para la simulación con la aplicación FTP y la velocidad entre el nodo emisor y el snoop de velocidad 0,1 Mbps son los observados a continuación.

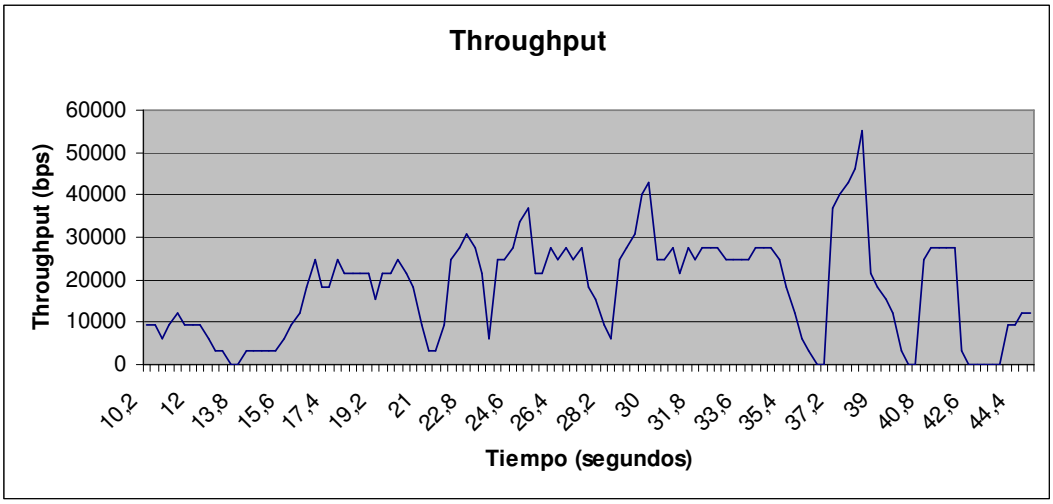

Fig. 6.11. *Throughput* con snoop, FTP, 0.1Mbps emisor- snoop.

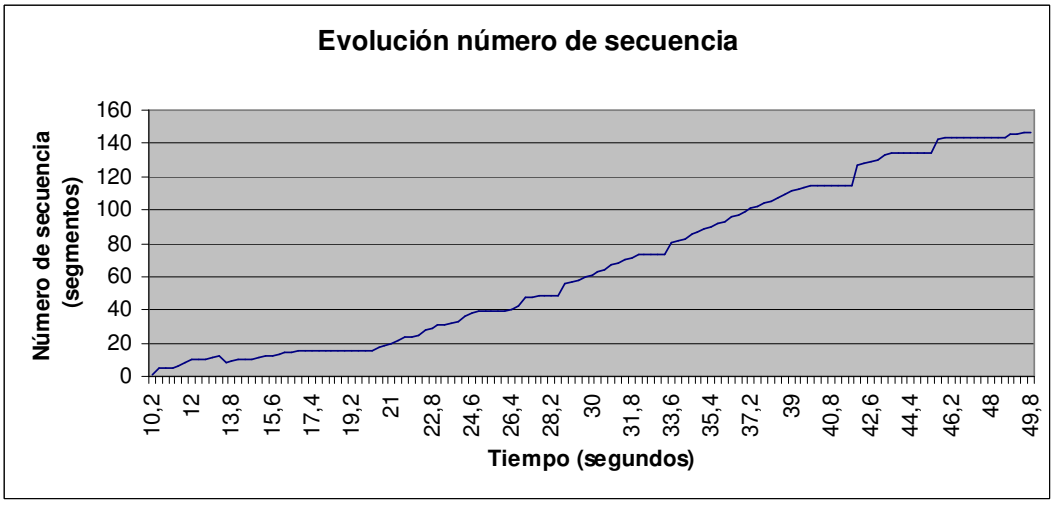

Fig. 6.12. Número de secuencia con snoop, FTP, 0.1 Mbps emisor - snoop.

Los resultados obtenidos para la simulación con la aplicación CBR y la velocidad entre el nodo emisor y el snoop de velocidad 1 Mbps son los observados a continuación.

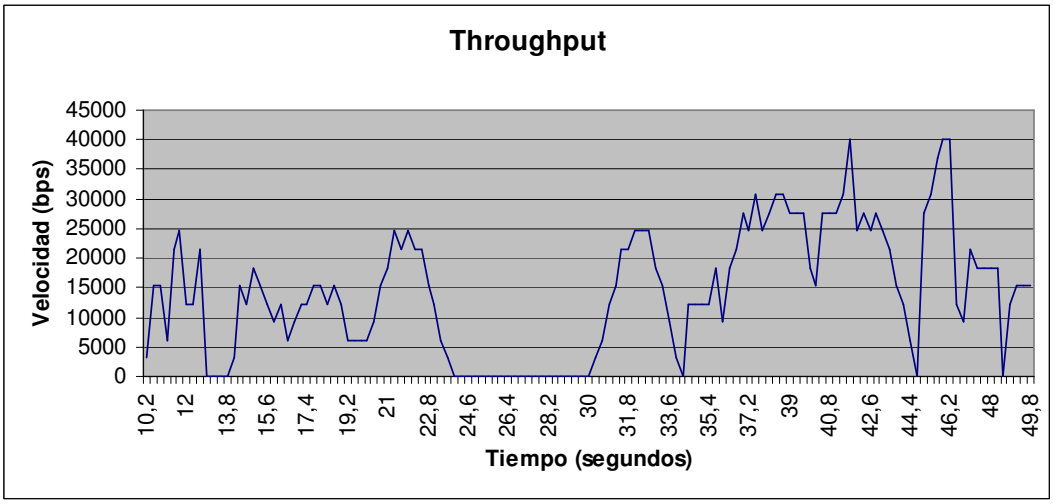

Fig. 6.13. *Throughput* con snoop, CBR, 1Mbps emisor- snoop.

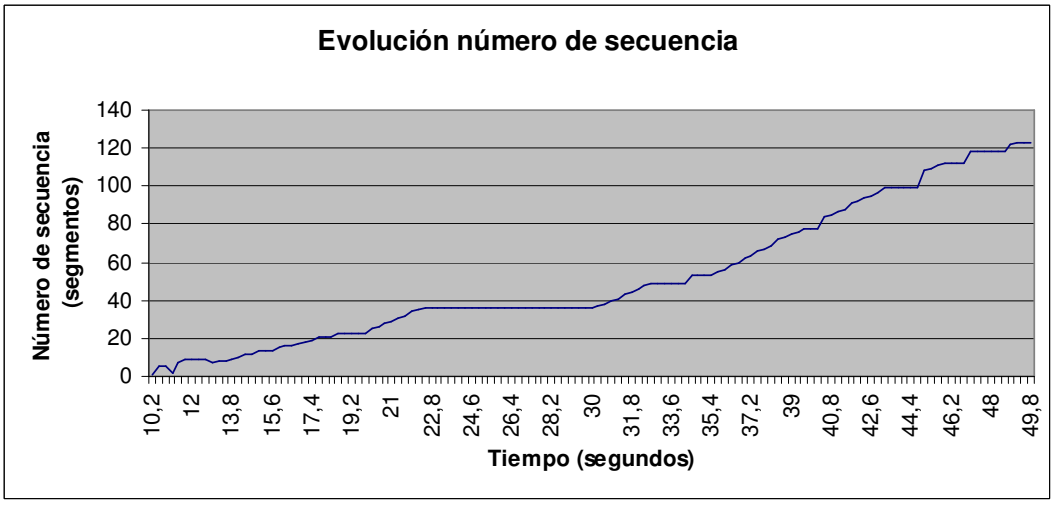

Fig. 6.14. Número de secuencia con snoop, CBR, 1 Mbps emisor - snoop.

Los resultados obtenidos para la simulación con la aplicación CBR la velocidad entre el nodo emisor y el snoop de velocidad 0,1 Mbps son los observados a continuación.

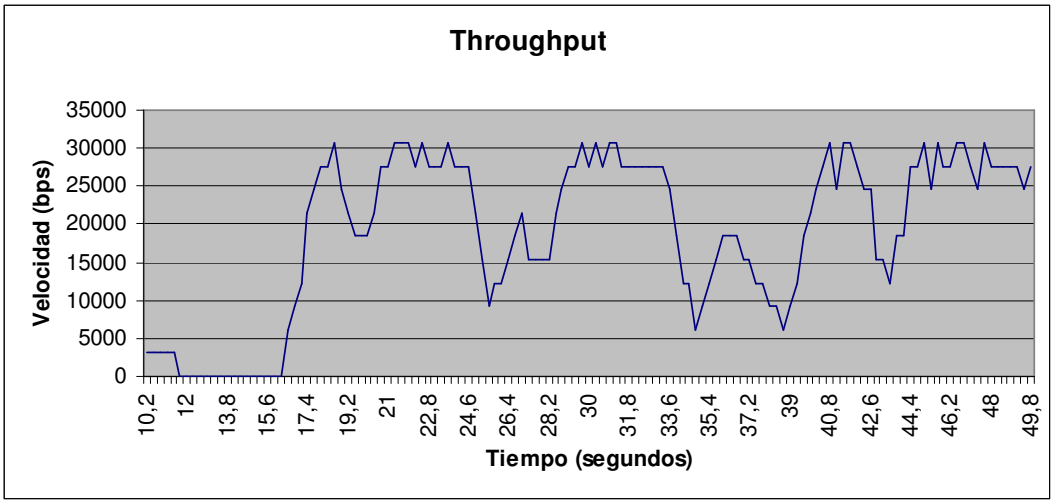

Fig. 6.15. *Throughput* con snoop, CBR, 0.1Mbps emisor- snoop.

Fig. 6.16. Número de secuencia con snoop, CBR, 0.1 Mbps emisor - snoop.

6.3. Simulación con snoop adaptado

Una vez realizados los cambios comentados en el punto 4.1. Modificaciones en snoop, hemos realizado las simulaciones.

La justificación de que los cambios funcionan los encontramos en la figura 6.9.

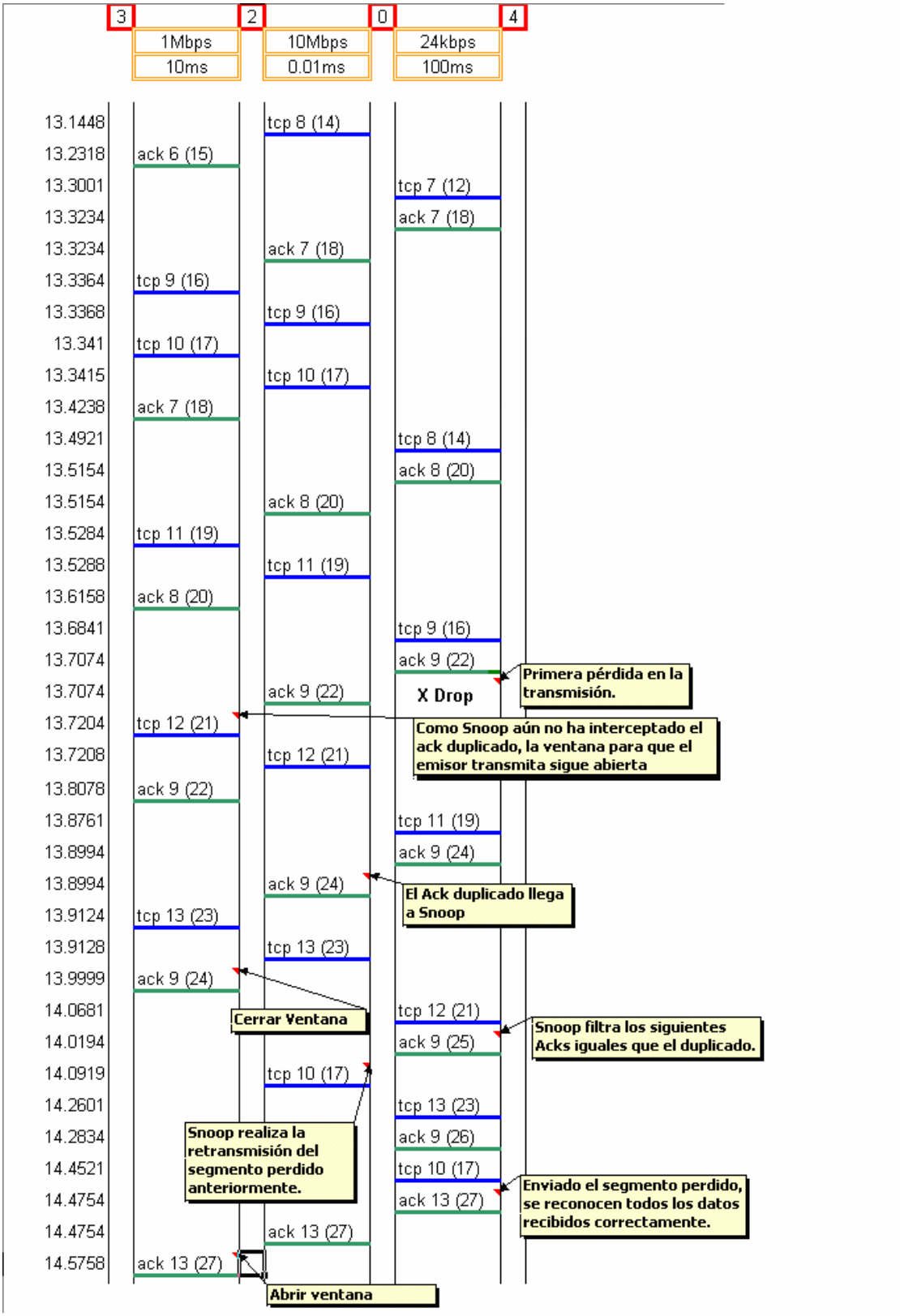

Fig. 6.17. Justificación cambios hechos en snoop correctos

Con los cambios hechos, los resultados obtenidos son los siguientes:

Aplicación	Velocidad enlace emisor - nodo intermedio 1	Retransmisiones emisor por timer	Retransmisiones emisor por dupacks	Retransmisiones totales desde nodo intermedio 1
FTP	1 Mbps	0	0	15
FTP	0,1 Mbps	0	0	15
CBR	1 Mbps	0	0	15
CBR	0,1 Mbps	0	0	15

Tabla 6.3. Resultados obtenidos con snoop adaptado

Los resultados obtenidos para la simulación con la aplicación FTP y la velocidad entre el nodo emisor y el snoop adaptado de velocidad 1 Mbps son los observados a continuación.

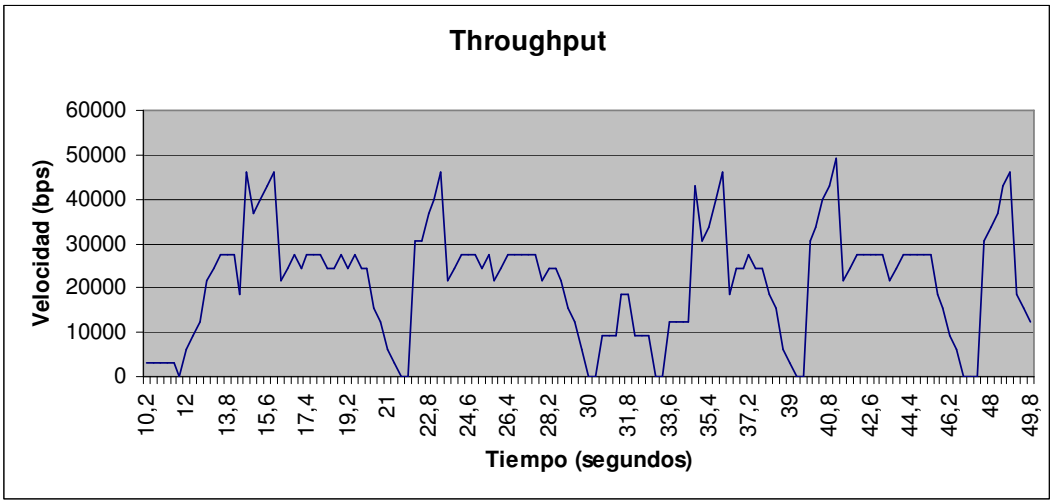

Fig. 6.18. *Throughput* con snoop adaptado, FTP, 1Mbps emisor- snoop adaptado.

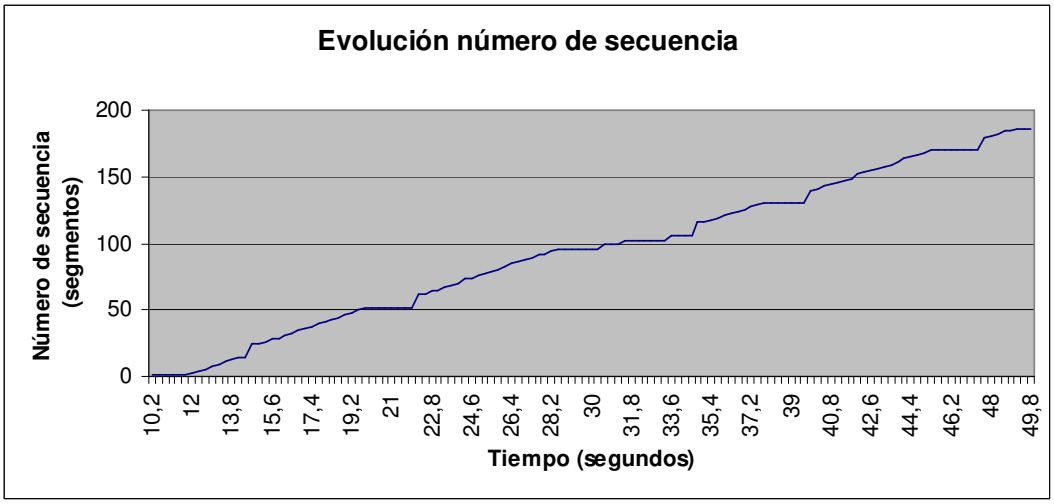

Fig. 6.19. Número de secuencia con snoop adaptado, FTP, 1 Mbps emisor - snoop adaptado.

Los resultados obtenidos para la simulación con la aplicación FTP y la velocidad entre el nodo emisor y el snoop adaptado de velocidad 0,1 Mbps son los observados a continuación.

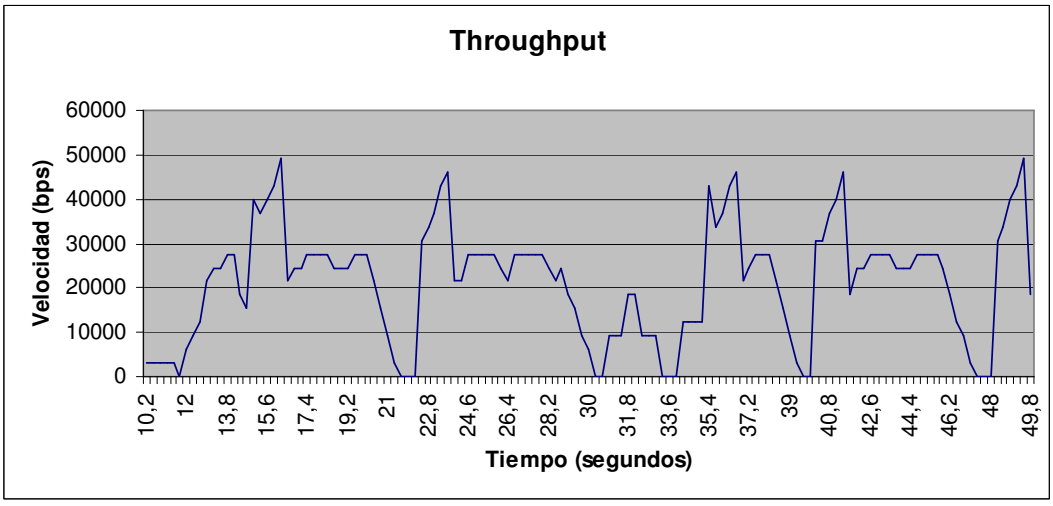

Fig. 6.20. *Throughput* con snoop adaptado, FTP, 0.1Mbps emisor- snoop adaptado.

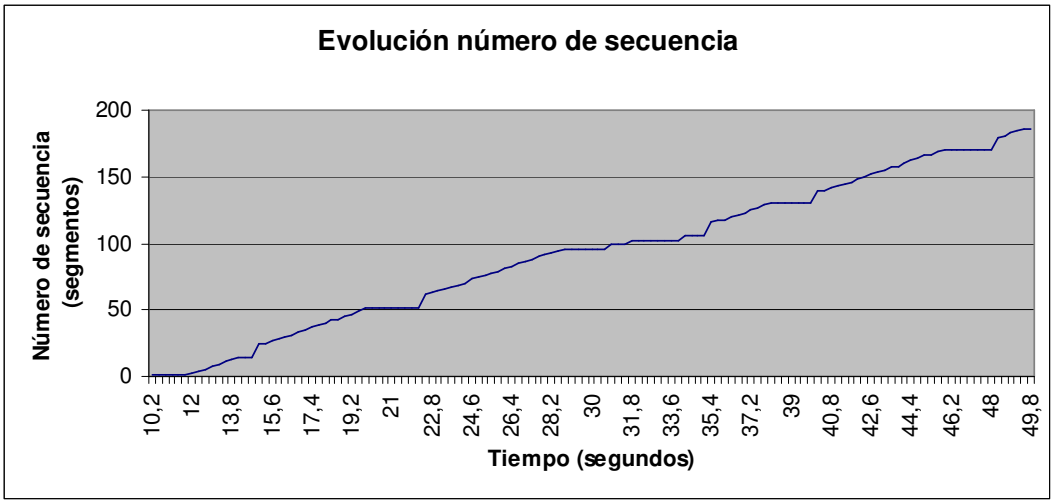

Fig. 6.21. Número de secuencia con snoop adaptado, FTP, 0.1 Mbps emisor - snoop adaptado.

Los resultados obtenidos para la simulación con la aplicación CBR y la velocidad entre el nodo emisor y el nodo intermedio 1 de velocidad 1 Mbps son los observados a continuación.

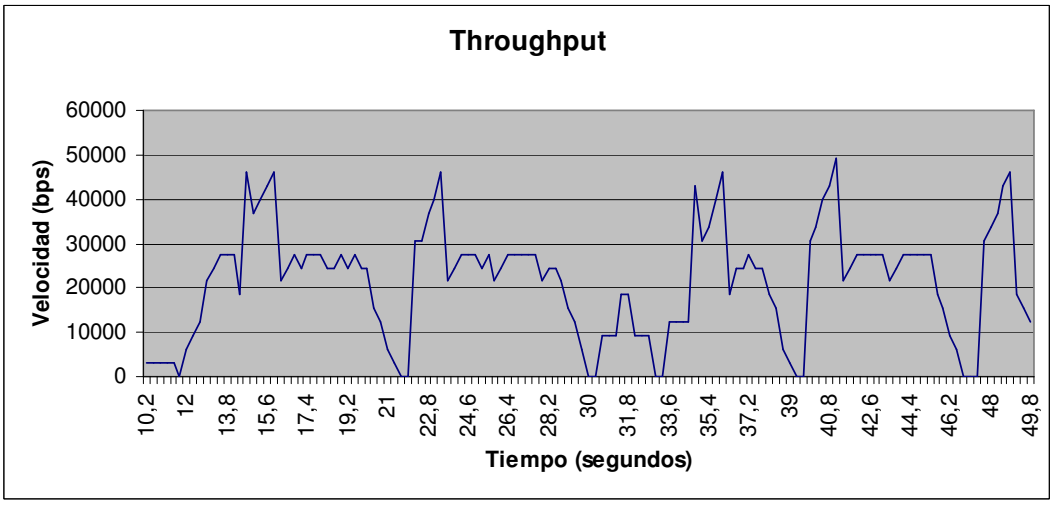

Fig. 6.22. *Throughput* con snoop adaptado, CBR, 1Mbps emisor- snoop adaptado.

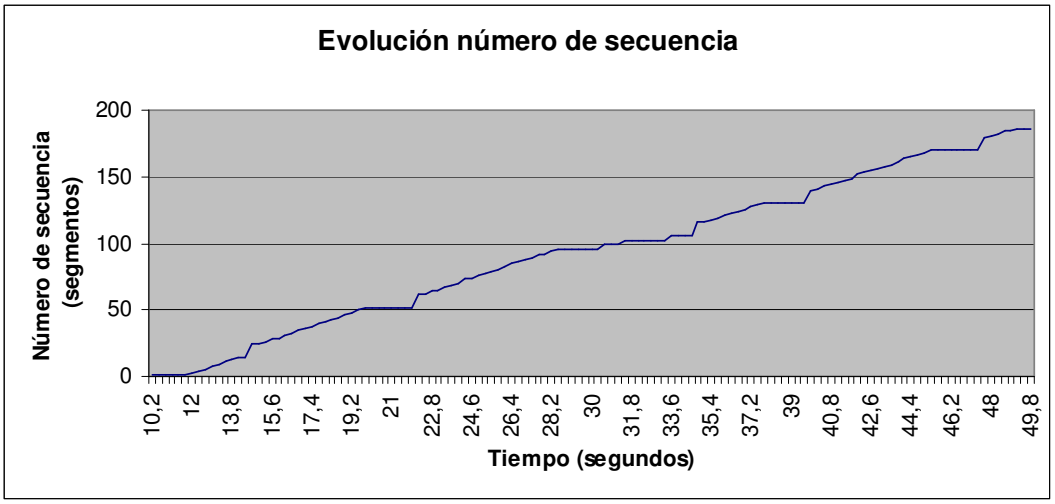

Fig. 6.23. Número de secuencia con snoop adaptado, CBR, 1 Mbps emisor - snoop adaptado.

Los resultados obtenidos para la simulación con la aplicación CBR la velocidad entre el nodo emisor y el snoop adaptado de velocidad 0,1 Mbps son los observados a continuación.

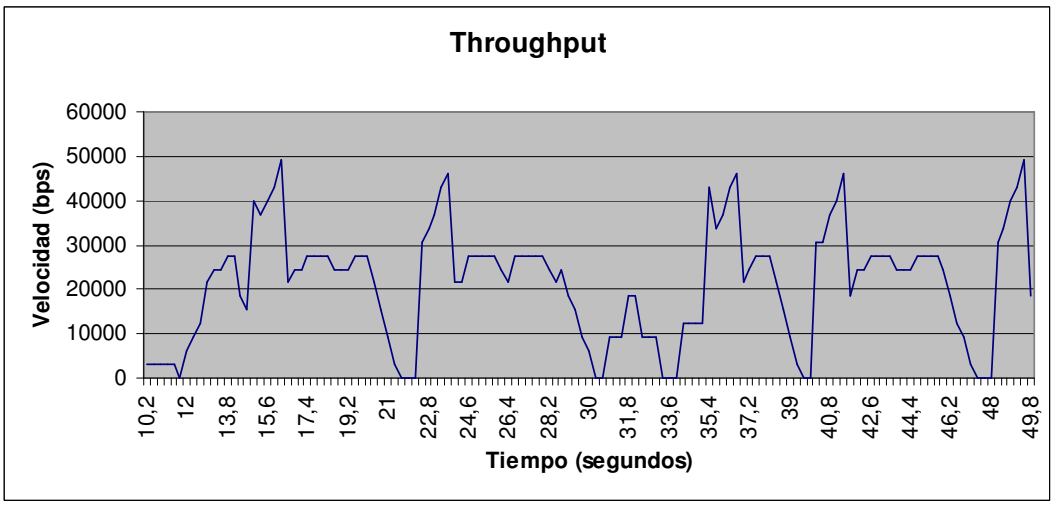

Fig. 6.24. *Throughput* con snoop adaptado, CBR, 0.1Mbps emisor- snoop adaptado.

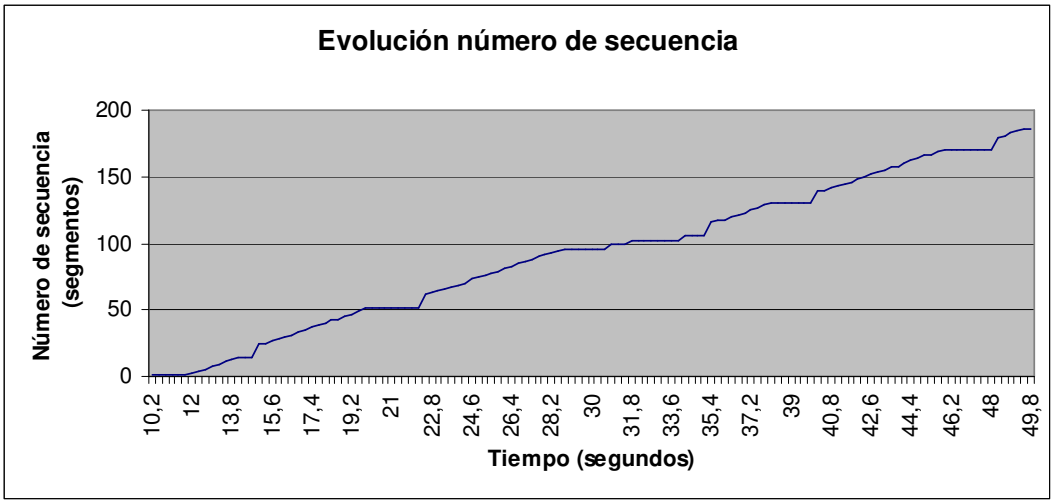

Fig. 6.25. Número de secuencia con snoop adaptado, CBR, 0.1 Mbps emisor - snoop adaptado.

CAPÍTULO 7. CONCLUSIONES

En este proyecto se ha presentado una evaluación del funcionamiento del proxy snoop adaptado en un entorno GPRS. A continuación se muestran las conclusiones obtenidas.

En la tabla 7.1. presentamos los *throughput*s medios obtenidos, que también utilizamos para alcanzar las conclusiones.

		Velocidad enlace emisor - snoop (Mbps)	***Throughput* medio (bps)**
FTP	Sin snoop	1	19432,72727
		0,1	18618,18182
	Snoop original	1	19432,72727
		0,1	18618,18182
	Snoop adaptado	1	14196,3636
		0,1	16989,0909
CBR	Sin snoop	1	14196,3636
		0,1	16989,0909
	Snoop original	1	21527,2727
		0,1	21527,2727
	Snoop adaptado	1	21527,2727
		0,1	21527,2727

Tabla 7.1. *Throughput*s medios obtenidos

La versión original de snoop no es una solución recomendable para las redes GPRS en las que el retardo es importante (en nuestro escenario 100ms), ya que causa retransmisiones innecesarias del emisor, degradando el rendimiento de la comunicación; por lo tanto, se justifica la necesidad de un proxy que se adapte mejor a este tipo de entornos.

La degradación se deduce en los resultados obtenidos de la simulación con snoop original por los efectos siguientes:

- El número de secuencia que se alcanza es menor incluso que si no se utilizará snoop.

- Según se aprecia en sus gráficas de *throughput* en algunos momentos la transmisión queda estancada durante periodos considerables de tiempo.

- Cabe destacar que en este caso el *throughput* medio es menor al obtenido con snoop adaptado y sin snoop.

- El número de retransmisiones por parte del nodo emisor oscila entre 8 y 12 dependiendo de la simulación, lo que demuestra que este tipo de snoop no consigue mejorar las condiciones de trabajo de dicho emisor.

En los resultados de las simulaciones sin snoop se estima que el funcionamiento es mejor que con snoop original ya que los efectos obtenidos son más positivos:

- El número de secuencia alcanzado es mayor. Esto es debido al estancamiento que snoop original provoca antes comentado.

- El *throughput* sin snoop alcanza valores mayores comparando con snoop original, pero menores que los que se consiguen con snoop adaptado, dado que la comunicación no se beneficia de sus mejoras.

Con snoop adaptado los resultados demuestran una mejoría en la comunicación debido a los siguientes puntos:

- Tanto el número de secuencia alcanzado como el valor de *throughput* son mayores que en los casos anteriores.

- No hay retransmisiones desde el nodo emisor ni por expiración del *timer* ni por la recepción de *ACKs* duplicados como consecuencia de la acción de snoop adaptado.

Concluimos que aunque la utilización del proxy snoop adaptado cause un aumento del retardo total, dado el tiempo de proceso que supone su utilización, los beneficios que produce a la comunicación compensan este tiempo añadido, ya que aumenta el *throughput* y, por lo tanto, disminuiría el tiempo de transmisión en una comunicación.

7.1. Estudio del coste medioambiental

El objeto de estudio de este proyecto es la utilización del proxy snoop adaptado para las comunicaciones mediante GPRS.

No tendría ningún efecto negativo sobre el medioambiente debido a que la ubicación de este proxy sería las estaciones base de telefonía ya instaladas.

Por lo tanto, concluimos con que el efecto medioambiental no sería peor del ya sufrido debido a dichas estaciones base.

BIBLIOGRAFÍA

1. Documentación de la asignatura Sistemas y aplicaciones. *Departamento ENTEL.* EPSC septiembre 2004.

2. Rafael Vidal Ferré. Bloque de redes telemáticas. *Unidad 16. TCP-IP sobre radio.* EPSC noviembre 2005.

3. Anna Calveras Augé, María Luisa Catalán Cid, Carles Montenegro, Josep Paradells Aspas, Lluís Casals Ibáñez. *Mejora del acceso a Internet a traves de GPRS mediante la utilización de un proxy snoop.* Telecom I + D 2003.

4. Hari Balakrishnan, V. Padmanabhan, S. Seshan, andR. Katz. *A Comparison of Mechanisms for ImprovingTCP Performance over Wireless Links*. ACM/IEEE Transactions on Networking, Vol. 5,No. 6:756–769, 1997.

5. Hari Balakrishnan, Srinivasan Seshan, Elan Amir and Randy H. Katz, Computer Science Division University of California at Berkeley, *Improving TCP/IP Performance over Wireless Networks,* 1-10 (1995).

6. Federico Rodríguez Teja, Leonardo Vidal, Leonardo Alves. *TCP sobre enlaces wireless, Problemas y algunas posibles soluciones existentes.* Instituto de Ingeniería Eléctrica, Facultad de Ingeniería, Universidad de la República, Marzo 2004.

7. Miguel Barreto, Maximiliano Belino, Marcelo San Martín. *TCP SACK* . Instituto de Ingeniería Eléctrica, Facultad de Ingeniería, Universidad de la República.

8. Pagina Web del simulador, http://www.isi.edu/nsnam/ns/

9. Manual de Marc Greis, http://www.isi.edu/nsnam/ns/tutorial/

10. NS para wireless TCP, http://wine.icu.ac.kr/wine2005/data/ns_simulations_workshop_mo_aug_2005.pdf

11. Estructura de clases de la jerarquía compilada de NS, http://www.grid.unina.it/%7Evollero/resources/doc_ns2.26/html/

12. Foro de usuarios de NS, http://mailman.isi.edu/pipermail/ns-users/

13. Información sobre TCP, http://webepcc.unex.es/jlgs/Docen/Transpas-TCP.pdf

14. Convertidor imágenes de NS a extensión pdf, http://www.ps2pdf.com/convert/convert.htm

15. Preguntas y errores frecuentes al simular con NS/NAM, http://www.it.uc3m.es/~rcalzada/ns/faq.html#top

16. Manual del leguaje de programación C++, http://www.cplusplus.com/doc/tutorial/

17. Rafael Vidal Ferré. Bloque de redes telemáticas. *Unidad 14. Xarxes cel·lulars 2,5G.* EPSC noviembre 2005.

Escola Politècnica Superior de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

ANEXOS

TÍTULO DEL TFC: Diseño de un entorno para el estudio de los parámetros de funcionamiento del protocolo TCP

TITULACIÓN: Ingeniería Técnica de Telecomunicaciones, especialidad Telemática

AUTORES: Sergio Higón Fernández
Alejandro Feliu Argila

DIRECTOR: Lluís Casals Ibáñez

FECHA: 25 de julio de 2006

ANEXOS

En esta parte del documento describimos los códigos de los módulos de NS más relevantes que hemos utilizado para realizar el trabajo final de carrera.

Anexo 1. TCP New Reno

```
/* -*-      Mode:C++; c-basic-offset:8; tab-width:8; indent-tabs-mode:t
-*- */
/*
 * Copyright (c) 1990, 1997 Regents of the University of California.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms are permitted
 * provided that the above copyright notice and this paragraph are
 * duplicated in all such forms and that any documentation,
 * advertising materials, and other materials related to such
 * distribution and use acknowledge that the software was developed
 * by the University of California, Lawrence Berkeley Laboratory,
 * Berkeley, CA.  The name of the University may not be used to
 * endorse or promote products derived from this software without
 * specific prior written permission.
 * THIS SOFTWARE IS PROVIDED ``AS IS'' AND WITHOUT ANY EXPRESS OR
 * IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED
 * WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
 */

#ifndef lint
static const char rcsid[] =
    "@(#) $Header: /nfs/jade/vint/CVSROOT/ns-2/tcp/tcp-newreno.cc,v
1.56 2004/10/26 22:59:42 sfloyd Exp $ (LBL)";
#endif

//
// newreno-tcp: a revised reno TCP source, without sack
//

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>

#include "packet.h"
#include "ip.h"
#include "tcp.h"
#include "flags.h"


static class NewRenoTcpClass : public TclClass {
public:
      NewRenoTcpClass() : TclClass("Agent/TCP/Newreno") {}
      TclObject* create(int, const char*const*) {
            return (new NewRenoTcpAgent());
      }
} class_newreno;
```

```
NewRenoTcpAgent::NewRenoTcpAgent() : newreno_changes_(0),
  newreno_changes1_(0), acked_(0), firstpartial_(0),
  partial_window_deflation_(0), exit_recovery_fix_(0)
{
      bind("newreno_changes_", &newreno_changes_);
      bind("newreno_changes1_", &newreno_changes1_);
      bind("exit_recovery_fix_", &exit_recovery_fix_);
      bind("partial_window_deflation_", &partial_window_deflation_);
}

/*
 * Process a packet that acks previously unacknowleges data, but
 * does not take us out of Fast Retransmit.
 */


void NewRenoTcpAgent::partialnewack(Packet* pkt)
{
      hdr_tcp *tcph = hdr_tcp::access(pkt);
      if (partial_window_deflation_) {
            // Do partial window deflation before resetting last_ack_
            unsigned int deflate = 0; // Should initialize it?? -
haoboy
            if (tcph->seqno() > last_ack_) // assertion
                  deflate = tcph->seqno() - last_ack_;
            else
                  printf("False call to partialnewack:  deflate %u \
last_ack_ %d\n", deflate, last_ack_);
            if (dupwnd_ > deflate)
                  dupwnd_ -= (deflate - 1);
            else {
                  cwnd_ -= (deflate - dupwnd_);
                  // Leave dupwnd_ > 0 to flag "fast recovery" phase
                  dupwnd_ = 1;
            }
            if (cwnd_ < 1) {cwnd_ = 1;}
      }
      last_ack_ = tcph->seqno();
      highest_ack_ = last_ack_;
      if (t_seqno_ < last_ack_ + 1)
            t_seqno_ = last_ack_ + 1;
      if (rtt_active_ && tcph->seqno() >= rtt_seq_) {
            rtt_active_ = 0;
            t_backoff_ = 1;
      }
}

void NewRenoTcpAgent::partialnewack_helper(Packet* pkt)
{
      if (!newreno_changes1_ || firstpartial_ == 0) {
            firstpartial_ = 1;
            /* For newreno_changes1_,
             * only reset the retransmit timer for the first
             * partial ACK, so that, in the worst case, we
             * don't have to wait for one packet retransmitted
             * per RTT.
             */
            newtimer(pkt);
      }
      partialnewack(pkt);
      output(last_ack_ + 1, 0);
```

```
}

int
NewRenoTcpAgent::allow_fast_retransmit(int /* last_cwnd_action_*/)
{
      return 0;
}

void
NewRenoTcpAgent::dupack_action()
{
        int recovered = (highest_ack_ > recover_);
      int recovered1 = (highest_ack_ == recover_);
        int allowFastRetransmit =
allow_fast_retransmit(last_cwnd_action_);
        if (recovered || (!bug_fix_ && !ecn_) || allowFastRetransmit)
{
                goto reno_action;
        }
      if (bug_fix_ && less_careful_ && recovered1) {
            /*
             * For the Less Careful variant, allow a Fast Retransmit
             *  if highest_ack_ == recover.
             * RFC 2582 recommends the Careful variant, not the
             *  Less Careful one.
             */
                goto reno_action;
      }

        if (ecn_ && last_cwnd_action_ == CWND_ACTION_ECN) {
                last_cwnd_action_ = CWND_ACTION_DUPACK;
                /*
                 * What if there is a DUPACK action followed closely
by ECN
                 * followed closely by a DUPACK action?
                 * The optimal thing to do would be to remember all
                 * congestion actions from the most recent window
                 * of data.  Otherwise "bugfix" might not prevent
                 * all unnecessary Fast Retransmits.
                 */
                reset_rtx_timer(1,0);
                output(last_ack_ + 1, TCP_REASON_DUPACK);
            dupwnd_ = numdupacks_;
                return;
        }

        if (bug_fix_) {
            if (bugfix_ts_ && tss[highest_ack_ % tss_size_] ==
ts_echo_)
                 goto reno_action;
            else if (bugfix_ack_ && cwnd_ > 1 && highest_ack_ -
prev_highest_ack_ <= numdupacks_)
                 goto reno_action;
            else
                /*
                 * The line below, for "bug_fix_" true, avoids
                 * problems with multiple fast retransmits in one
                 * window of data.
                 */
                 return;
        }
```

```
reno_action:
        recover_ = maxseq_;
        reset_rtx_timer(1,0);
        if (!lossQuickStart()) {
                trace_event("NEWRENO_FAST_RETX");
            last_cwnd_action_ = CWND_ACTION_DUPACK;
            slowdown(CLOSE_SSTHRESH_HALF|CLOSE_CWND_HALF);
            output(last_ack_ + 1, TCP_REASON_DUPACK);       // from top
            dupwnd_ = numdupacks_;
      }
        return;
}


void NewRenoTcpAgent::recv(Packet *pkt, Handler*)
{
      hdr_tcp *tcph = hdr_tcp::access(pkt);
      int valid_ack = 0;

      int c = nrexmit_;
      int p = nrexmitpack_;

      printf("TCP Retx: TIMEOUT: %-d PACK: %-d \n", c, p);

      //COMPROBADO: LEE EL CAMPO WND_ DE LA CABECERA CORRECTAMENTE
      if (tcph->wnd() == 2) {
            cwnd_ = 0.0;
            //printf("VENTANA REDUCIDA\n");
      }

      if (tcph->wnd() == 1) {
            cwnd_ = double(ssthresh_ +0.1 );
                //printf("VENTANA AMPLIADA\n");
        }



      /* Use first packet to calculate the RTT  --contributed by
Allman */

        if (qs_approved_ == 1 && tcph->seqno() > last_ack_)
            endQuickStart();
        if (qs_requested_ == 1)
                processQuickStart(pkt);
      if (++acked_ == 1)
            basertt_ = Scheduler::instance().clock() - firstsent_;

      /* Estimate ssthresh based on the calculated RTT and the
estimated
         bandwidth (using ACKs 2 and 3).  */

      else if (acked_ == 2)
            ack2_ = Scheduler::instance().clock();
      else if (acked_ == 3) {
            ack3_ = Scheduler::instance().clock();
            new_ssthresh_ = int((basertt_ * (size_ / (ack3_ - ack2_)))
/ size_);
            if (newreno_changes_ > 0 && new_ssthresh_ < ssthresh_)
                  ssthresh_ = new_ssthresh_;
      }
```

```
#ifdef notdef
      if (pkt->type_ != PT_ACK) {
            fprintf(stderr,
                  "ns: confiuration error: tcp received non-ack\n");
            exit(1);
      }
#endif
        /* W.N.: check if this is from a previous incarnation */
        if (tcph->ts() < lastreset_) {
                // Remove packet and do nothing
                Packet::free(pkt);
                return;
        }
      ++nackpack_;
      ts_peer_ = tcph->ts();

      if (hdr_flags::access(pkt)->ecnecho() && ecn_)
            ecn(tcph->seqno());
      recv_helper(pkt);
      recv_frto_helper(pkt);
      if (tcph->seqno() > last_ack_) {
            if (tcph->seqno() >= recover_
                || (last_cwnd_action_ != CWND_ACTION_DUPACK)) {
                  if (dupwnd_ > 0) {
                       dupwnd_ = 0;
                       if (last_cwnd_action_ == CWND_ACTION_DUPACK)
                        last_cwnd_action_ = CWND_ACTION_EXITED;
                       if (exit_recovery_fix_) {
                        int outstanding = maxseq_ - tcph->seqno() + 1;
                        if (ssthresh_ < outstanding)
                                        cwnd_ = ssthresh_;
                                else
                                        cwnd_ = outstanding;
                           }
                  }
                  firstpartial_ = 0;
                  recv_newack_helper(pkt);
                  if (last_ack_ == 0 && delay_growth_) {
                        cwnd_ = initial_window();
                  }
            } else {
                  /* received new ack for a packet sent during Fast
                   *  Recovery, but sender stays in Fast Recovery */
                  if (partial_window_deflation_ == 0)
                        dupwnd_ = 0;
                  partialnewack_helper(pkt);
            }
      } else if (tcph->seqno() == last_ack_) {
            if (hdr_flags::access(pkt)->eln_ && eln_) {
                  tcp_eln(pkt);
                  return;
            }
            if (++dupacks_ == numdupacks_) {
                  dupack_action();
                        if (!exitFastRetrans_)
                                dupwnd_ = numdupacks_;
            } else if (dupacks_ > numdupacks_ && (!exitFastRetrans_
                  || last_cwnd_action_ == CWND_ACTION_DUPACK)) {
                  trace_event("NEWRENO_FAST_RECOVERY");
                  ++dupwnd_;  // fast recovery
```

```
                  /* For every two duplicate ACKs we receive (in the
                   * "fast retransmit phase"), send one entirely new
                   * data packet "to keep the flywheel going".  --
Allman
                   */
                  if (newreno_changes_ > 0 && (dupacks_ % 2) == 1)
                        output (t_seqno_++,0);
            } else if (dupacks_ < numdupacks_ && singledup_ ) {
                        send_one();
                  }
      }
        if (tcph->seqno() >= last_ack_)
                  // Check if ACK is valid.  Suggestion by Mark Allman.
                  valid_ack = 1;
      Packet::free(pkt);
#ifdef notyet
      if (trace_)
            plot();
#endif

      /*
       * Try to send more data
       */

        if (valid_ack || aggressive_maxburst_)
            if (dupacks_ == 0)
                  /*
                   * Maxburst is really only needed for the first
                   *  window of data on exiting Fast Recovery.
                   */
                  send_much(0, 0, maxburst_);
            else if (dupacks_ > numdupacks_ - 1 && newreno_changes_ ==
0)
                  send_much(0, 0, 2);
}
```

Anexo 2. Snoop adaptado

```
/* -*-      Mode:C++; c-basic-offset:8; tab-width:8; indent-tabs-mode:t
-*- */
/*
 * Copyright (c) 1997 Regents of the University of California.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above
copyright
 *    notice, this list of conditions and the following disclaimer in
the
 *    documentation and/or other materials provided with the
distribution.
 * 3. All advertising materials mentioning features or use of this
software
 *    must display the following acknowledgement:
 *    This product includes software developed by the Daedalus
Research
 *    Group at the University of California Berkeley.
 * 4. Neither the name of the University nor of the Research Group may
be
 *    used to endorse or promote products derived from this software
without
 *    specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS''
AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE
LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
OF
 * SUCH DAMAGE.
 */

#ifndef lint
static const char rcsid[] =
    "@(#) $Header: /nfs/jade/vint/CVSROOT/ns-2/tcp/snoop.cc,v 1.25
2003/01/28 23:31:03 sfloyd Exp $ (UCB)";
#endif
```

```
#include "snoop.h"

int hdr_snoop::offset_;

class SnoopHeaderClass : public PacketHeaderClass {
public:
        SnoopHeaderClass() : PacketHeaderClass("PacketHeader/Snoop",
                                    sizeof(hdr_snoop)) {
            bind_offset(&hdr_snoop::offset_);
      }
} class_snoophdr;

static class LLSnoopClass : public TclClass {
public:
      LLSnoopClass() : TclClass("LL/LLSnoop") {}
      TclObject* create(int, const char*const*) {
            return (new LLSnoop());
            printf("SNOOP\n");
      }
} llsnoop_class;
int a;
int oc;

static class SnoopClass : public TclClass {
public:
      SnoopClass() : TclClass("Snoop") {}
      TclObject* create(int, const char*const*) {
            return (new Snoop());
            printf("SNOOP\n");
      }
} snoop_class;

Snoop::Snoop() : NsObject(),
      fstate_(0), lastSeen_(-1), lastAck_(-1),
      expNextAck_(0), expDupacks_(0), bufhead_(0),
      toutPending_(0), buftail_(0),
      wl_state_(SNOOP_WLEMPTY), wl_lastSeen_(-1), wl_lastAck_(-1),
      wl_bufhead_(0), wl_buftail_(0)
{
      bind("snoopDisable_", &snoopDisable_);
      bind_time("srtt_", &srtt_);
      bind_time("rttvar_", &rttvar_);
      bind("maxbufs_", &maxbufs_);
      bind("snoopTick_", &snoopTick_);
      bind("g_", &g_);
      bind("tailTime_", &tailTime_);
      bind("rxmitStatus_", &rxmitStatus_);
      bind("lru_", &lru_);

      printf("SNOOP\n");

      rxmitHandler_ = new SnoopRxmitHandler(this);
      oc = 1;
      a=0;
      int i;
      for (i = 0; i < SNOOP_MAXWIND; i++) /* data from wired->wireless
*/
            pkts_[i] = 0;
      for (i = 0; i < SNOOP_WLSEQS; i++) {/* data from wireless->wired
*/
```

```
            wlseqs_[i] = (hdr_seq *) malloc(sizeof(hdr_seq));
            wlseqs_[i]->seq = wlseqs_[i]->num = 0;
      }

      if (maxbufs_ == 0)
            maxbufs_ = SNOOP_MAXWIND;
}


void
Snoop::reset()
{
//    printf("%x resetting\n", this);
      fstate_ = 0;
      lastSeen_ = -1;
      lastAck_ = -1;
      expNextAck_ = 0;
      expDupacks_ = 0;
      bufhead_ = buftail_ = 0;
      if (toutPending_) {
            Scheduler::instance().cancel(toutPending_);
            // xxx: I think that toutPending_ doesn't need to be freed
because snoop didn't allocate it (but I'm not sure).
            toutPending_ = 0;
      };
      for (int i = 0; i < SNOOP_MAXWIND; i++) {
            if (pkts_[i]) {
                  Packet::free(pkts_[i]);
                  pkts_[i] = 0;
            }
      }
}

void
Snoop::wlreset()
{
      wl_state_ = SNOOP_WLEMPTY;
      wl_bufhead_ = wl_buftail_ = 0;
      for (int i = 0; i < SNOOP_WLSEQS; i++) {
            wlseqs_[i]->seq = wlseqs_[i]->num = 0;
      }
}


int
Snoop::command(int argc, const char*const* argv)
{
      //Tcl& tcl = Tcl::instance();

      if (argc == 3) {
            if (strcmp(argv[1], "llsnoop") == 0) {
                  parent_ = (LLSnoop *) TclObject::lookup(argv[2]);
                  if (parent_)
                        recvtarget_ = parent_->uptarget();
                  return (TCL_OK);
            }

            if (strcmp(argv[1], "check-rxmit") == 0) {
                  if (empty_()) {
                        rxmitStatus_ = SNOOP_PROPAGATE;
                        return (TCL_OK);
```

```
                }

                Packet *p = pkts_[buftail_];
                hdr_snoop *sh = hdr_snoop::access(p);

                if (sh->sndTime()!=-1 && sh->sndTime()<atoi(argv[2])
&&
                    sh->numRxmit() == 0)
                      /* candidate for retransmission */
                      rxmitStatus_ = snoop_rxmit(p);
                else
                      rxmitStatus_ = SNOOP_PROPAGATE;
                return (TCL_OK);
          }
      }
      return NsObject::command(argc, argv);
}

void LLSnoop::recv(Packet *p, Handler *h)
{
      Tcl &tcl = Tcl::instance();
      hdr_ip *iph = hdr_ip::access(p);

      /* get-snoop creates a snoop object if none currently exists */
      hdr_cmn *ch = HDR_CMN(p);
      if(ch->direction() == hdr_cmn::UP)
            /* get-snoop creates a snoop object if none currently
exists */
            /* In ns, addresses have ports embedded in them. */
            tcl.evalf("%s get-snoop %d %d", name(), iph->daddr(),
                  iph->saddr());

      else
            tcl.evalf("%s get-snoop %d %d", name(), iph->saddr(),
                  iph->daddr());

      Snoop *snoop = (Snoop *) TclObject::lookup(tcl.result());

      snoop->recv(p, h);

      if (integrate_)
            tcl.evalf("%s integrate %d %d", name(), iph->saddr(),
                  iph->daddr());
      if (h)                  /* resume higher layer (queue) */
            Scheduler::instance().schedule(h, &intr_, 0.000001);
      return;
}

/*
 * Receive a packet from higher layer or from the network.
 * Call snoop_data() if TCP packet and forward it on if it's an ack.
 */
void
Snoop::recv(Packet* p, Handler* h )
{

      hdr_cmn *ch = HDR_CMN(p);
      if(ch->direction() == hdr_cmn::UP) {
            handle((Event *) p);
            return;
      }
```

```
      packet_t type = hdr_cmn::access(p)->ptype();
      /* Put packet (if not ack) in cache after checking, and send it
on */

      if (type == PT_TCP)
            snoop_data(p);

      else if (type == PT_ACK)
            snoop_wired_ack(p);

      ch->direction() = hdr_cmn::DOWN;  // Ben added
      parent_->sendDown(p);   /* vector to LLSnoop's sendto() */
}

/*
 * Handle a packet received from peer across wireless link.  Check
first
 * for packet errors, then call snoop_ack() or pass it up as
necessary.
 */
void
Snoop::handle(Event *e)
{

      Packet *p = (Packet *) e;

      packet_t type = hdr_cmn::access(p)->ptype();
      //int seq = hdr_tcp::access(p)->seqno();
      int prop = SNOOP_PROPAGATE; // by default;  propagate ack or
packet
      Scheduler& s = Scheduler::instance();

      //hdr_ll *llh = hdr_ll::access(p);
      if (hdr_cmn::access(p)->error()) {
            parent_->drop(p); // drop packet if it's been corrupted
            return;
      }

      if (type == PT_ACK)
            prop = snoop_ack(p);

      else if (type == PT_TCP) /* XXX what about TELNET? */
            snoop_wless_data(p);

      if (prop == SNOOP_PROPAGATE)
            s.schedule(recvtarget_, e, parent_->delay());
      else {                  // suppress ack
            /*          printf("---- %f suppressing ack %d\n",
s.clock(), seq);*/
            Packet::free(p);
      }
}

/*
 * Data packet processing.  p is guaranteed to be of type PT_TCP when
 * this function is called.
 */
void
Snoop::snoop_data(Packet *p)
{
```

```
	Scheduler &s = Scheduler::instance();
	int seq = hdr_tcp::access(p)->seqno();
	int resetPending = 0;

	//    printf("%x snoop_data: %f sending packet %d\n", this,
s.clock(), seq);
	if (fstate_ & SNOOP_ALIVE && seq == 0)
		reset();
	fstate_ |= SNOOP_ALIVE;
	if ((fstate_ & SNOOP_FULL) && !lru_) {
//		printf("snoop full, fwd'ing\n t %d h %d", buftail_,
bufhead_);
		if (seq > lastSeen_)
			lastSeen_ = seq;
		return;
	}
	/*
	 * Only if the ifq is NOT full do we insert, since otherwise we
want
	 * congestion control to kick in.
	 */

	if (parent_->ifq()->length() < parent_->ifq()->limit()-1)
		resetPending = snoop_insert(p);
	if (toutPending_ && resetPending == SNOOP_TAIL) {
		s.cancel(toutPending_);
		// xxx: I think that toutPending_ doesn't need to be freed
because snoop didn't allocate it (but I'm not sure).
		toutPending_ = 0;
	}
	if (!toutPending_ && !empty_()) {
		toutPending_ = (Event *) (pkts_[buftail_]);
		s.schedule(rxmitHandler_, toutPending_, timeout());
	}
	return;
}

/*
 * snoop_insert() does all the hard work for snoop_data(). It
traverses the
 * snoop cache and looks for the right place to insert this packet (or
 * determines if its already been cached). It then decides whether
 * this is a packet in the normal increasing sequence, whether it
 * is a sender-rexmitted-but-lost-due-to-congestion (or network
 * out-of-order) packet, or if it is a sender-rexmitted packet that
 * was buffered by us before.
 */
int
Snoop::snoop_insert(Packet *p)
{



	int i, seq = hdr_tcp::access(p)->seqno(), retval=0;

	if (seq <= lastAck_)
		return retval;

	if (fstate_ & SNOOP_FULL) {
		/* free tail and go on */
		printf("snoop full, making room\n");
```

```
            Packet::free(pkts_[buftail_]);
            pkts_[buftail_] = 0;
            buftail_ = next(buftail_);
            fstate_ |= ~SNOOP_FULL;
      }

      if (seq > lastSeen_ || pkts_[buftail_] == 0) { // in-seq or
empty cache
            i = bufhead_;
            bufhead_ = next(bufhead_);
      } else if (seq < hdr_snoop::access(pkts_[buftail_])->seqno()) {
            buftail_ = prev(buftail_);
            i = buftail_;
      } else {
            for (i = buftail_; i != bufhead_; i = next(i)) {
                  hdr_snoop *sh = hdr_snoop::access(pkts_[i]);
                  if (sh->seqno() == seq) {  // cached before

                        sh->numRxmit() = 0;
                        sh->senderRxmit() = 1; //must be a sender retr
                        sh->sndTime() = Scheduler::instance().clock();
                        return SNOOP_TAIL;
                  } else if (sh->seqno() > seq) {

                        //not cached before, should insert in the
middle
                        // find the position it should be: prev(i)

                        Packet *temp = pkts_[prev(buftail_)];
                        for (int j = buftail_; j != i; j = next(j))
                              pkts_[prev(j)] = pkts_[j];
                        i = prev(i);
                        pkts_[i] = temp;   // seems not necessary. Ben
comments
                        buftail_ = prev(buftail_);
                        break;
                  }
            }

            // This should not happen, since seq must be > lastSeen,
which is
            // handled before in the first if.   Ben comments
            if (i == bufhead_)
                  bufhead_ = next(bufhead_);
      }

      // save in the buffer
      savepkt_(p, seq, i);

      if (bufhead_ == buftail_)
            fstate_ |= SNOOP_FULL;
      /*
       * If we have one of the following packets:
       * 1. a network-out-of-order packet, or
       * 2. a fast rxmit packet, or 3. a sender retransmission
       * AND it hasn't already been buffered,
       * then seq will be < lastSeen_.
       * We mark this packet as having been due to a sender rexmit
       * and use this information in snoop_ack(). We let the dupacks
       * for this packet go through according to expDupacks_.
       */
```

```
	if (seq < lastSeen_) { /* not in-order -- XXX should it be <= ?
*/
		if (buftail_ == i) {
			hdr_snoop *sh = hdr_snoop::access(pkts_[i]);
			sh->senderRxmit() = 1;
			sh->numRxmit() = 0;
		}
		expNextAck_ = buftail_;
		retval = SNOOP_TAIL;
	} else
		lastSeen_ = seq;

	return retval;
}

void
Snoop::savepkt_(Packet *p, int seq, int i)
{
	pkts_[i] = p->copy();
	Packet *pkt = pkts_[i];
	hdr_snoop *sh = hdr_snoop::access(pkt);
	sh->seqno() = seq;
	sh->numRxmit() = 0;
	sh->senderRxmit() = 0;
	sh->sndTime() = Scheduler::instance().clock();
}

/*
 * Ack processing in snoop protocol.  We know for sure that this is an
ack.
 * Return SNOOP_SUPPRESS if ack is to be suppressed and
SNOOP_PROPAGATE o.w.
 */
int
Snoop::snoop_ack(Packet *p)
{
	Packet *pkt;

	int ack = hdr_tcp::access(p)->seqno();

	if ((oc == 1) && (lastAck_ >= ack) && (maxseq_ < ack)) {
			hdr_tcp::access(p)->wnd()=2;
			oc = 2;
		 //printf("SNOOP.                                CLOSE\n");
	  }

	else if ((oc == 2) && (lastAck_ < ack)) {
			hdr_tcp::access(p)->wnd()=1;
			oc = 1;
	    //printf("SNOOP.                                 OPEN\n");
	  }

	/*
	 * There are 3 cases:
	 * 1. lastAck_ > ack.  In this case what has happened is
	 *    that the acks have come out of order, so we don't
	 *    do any local processing but forward it on.
	 * 2. lastAck_ == ack.  This is a duplicate ack. If we have
	 *    the packet we resend it, and drop the dupack.
	 *    Otherwise we never got it from the fixed host, so we
	 *    need to let the dupack get through.
```

```
       *    Set expDupacks_ to number of packets already sent
       *    This is the number of dup acks to ignore.
       * 3. lastAck_ < ack.  Set lastAck_ = ack, and update
       *    the head of the buffer queue. Also clean up ack'd packets.
       */
      if (fstate_ & SNOOP_CLOSED || lastAck_ > ack)
            return SNOOP_PROPAGATE; // send ack onward

      if (lastAck_ == ack) {
            /* A duplicate ack; pure window updates don't occur in ns.
*/

            pkt = pkts_[buftail_];

            if (pkt == 0)
                  return SNOOP_PROPAGATE;

            hdr_snoop *sh = hdr_snoop::access(pkt);

            if (pkt == 0 || sh->seqno() > ack + 1)
                  /* don't have packet, letting thru' */
                    return SNOOP_PROPAGATE;

            /*
             * We have the packet: one of 3 possibilities:
             * 1. We are not expecting any dupacks (expDupacks_ == 0)
             * 2. We are expecting dupacks (expDupacks_ > 0)
             * 3. We are in an inconsistent state (expDupacks_ == -1)
             */


            if (expDupacks_ == 0) { // not expecting it
#define RTX_THRESH 1

                  static int thresh = 0;
                  if (thresh++ < RTX_THRESH)
                        /* no action if under RTX_THRESH */
                        return SNOOP_PROPAGATE;

                  thresh = 0;

                  // if the packet is a sender retransmission, pass on
                  if (sh->senderRxmit())
                        return SNOOP_PROPAGATE;

                  /*
                   * Otherwise, not triggered by sender.  If this is
                   * the first dupack recd., we must determine how many
                   * dupacks will arrive that must be ignored, and also
                   * rexmit the desired packet.  Note that expDupacks_
                   * will be -1 if we miscount for some reason.
                   */


                  expDupacks_ = bufhead_ - expNextAck_;
                  if (expDupacks_ < 0)
                        expDupacks_ += SNOOP_MAXWIND;
                  expDupacks_ -= RTX_THRESH + 1;
                  expNextAck_ = next(buftail_);

                  if (sh->numRxmit() == 0)
```

```
                        return snoop_rxmit(pkt);
                } else if (expDupacks_ > 0) {
                        expDupacks_--;
                        return SNOOP_SUPPRESS;
                } else if (expDupacks_ == -1) {
                        if (sh->numRxmit() < 2) {
                                return snoop_rxmit(pkt);
                        }
                } else            // let sender deal with it
                        return SNOOP_PROPAGATE;
        } else {          // a new ack

                fstate_ &= ~SNOOP_NOACK; // have seen at least 1 new ack

                /* free buffers */
                double sndTime = snoop_cleanbufs_(ack);

                if (sndTime != -1)
                        snoop_rtt(sndTime);

                expDupacks_ = 0;
                expNextAck_ = buftail_;
                lastAck_ = ack;
        }
        return SNOOP_PROPAGATE;
}

/*
 * Handle data packets that arrive from a wireless link, and we're not
 * the end recipient.  See if there are any holes in the transmission,
and
 * if there are, mark them as candidates for wireless loss.  Then,
when
 * (dup)acks troop back for this loss, set the ELN bit in their
header, to
 * help the sender (or a snoop agent downstream) retransmit.
 */
void
Snoop::snoop_wless_data(Packet *p)
{
        hdr_tcp *th = hdr_tcp::access(p);
        int i, seq = th->seqno();

        if (wl_state_ & SNOOP_WLALIVE && seq == 0)
                wlreset();
        wl_state_ |= SNOOP_WLALIVE;

        if (wl_state_ & SNOOP_WLEMPTY && seq >= wl_lastAck_) {
                wlseqs_[wl_bufhead_]->seq = seq;
                wlseqs_[wl_bufhead_]->num = 1;
                wl_buftail_ = wl_bufhead_;
                wl_bufhead_ = wl_next(wl_bufhead_);
                wl_lastSeen_ = seq;
                wl_state_ &= ~SNOOP_WLEMPTY;
                return;
        }
        /* WL data list definitely not empty at this point. */
        if (seq >= wl_lastSeen_) {
                wl_lastSeen_ = seq;
                i = wl_prev(wl_bufhead_);
                if (wlseqs_[i]->seq + wlseqs_[i]->num == seq) {
```

```
                wlseqs_[i]->num++;
                return;
            }
            i = wl_bufhead_;
            wl_bufhead_ = wl_next(wl_bufhead_);
      } else if (seq == wlseqs_[i = wl_buftail_]->seq - 1) {
      } else
            return;

      wlseqs_[i]->seq = seq;
      wlseqs_[i]->num++;

      /* Ignore network out-of-ordering and retransmissions for now */
      return;
}

/*
 * Ack from wired side (for sender on "other" side of wireless link.
 */
void
Snoop::snoop_wired_ack(Packet *p)
{
      hdr_tcp *th = hdr_tcp::access(p);
      int ack = th->seqno();
      int i;

      if (ack == wl_lastAck_ && snoop_wlessloss(ack)) {
            hdr_flags::access(p)->eln_ = 1;
      } else if (ack > wl_lastAck_) {
            /* update info about unack'd data */
            for (i = wl_buftail_; i != wl_bufhead_; i = wl_next(i)) {
                  hdr_seq *t = wlseqs_[i];
                  if (t->seq + t->num - 1 <= ack) {
                        t->seq = t->num = 0;
                  } else if (ack < t->seq) {
                        break;
                  } else if (ack < t->seq + t->num - 1) {
                        /* ack for part of a block */
                        t->num -= ack - t->seq +1;
                        t->seq = ack + 1;
                        break;
                  }
            }
            wl_buftail_ = i;
            if (wl_buftail_ == wl_bufhead_)
                  wl_state_ |= SNOOP_WLEMPTY;
            wl_lastAck_ = ack;
            /* Even a new ack could cause an ELN to be set. */
            if (wl_bufhead_ != wl_buftail_ && snoop_wlessloss(ack))
                  hdr_flags::access(p)->eln_ = 1;
      }
}

/*
 * Return 1 if we think this packet loss was not congestion-related,
and
 * 0 otherwise.  This function simply implements the lookup into the
table
 * that maintains this info; most of the hard work is done in
 * snoop_wless_data() and snoop_wired_ack().
 */
```

```
int
Snoop::snoop_wlessloss(int ack)
{
      if ((wl_bufhead_ == wl_buftail_) || wlseqs_[wl_buftail_]->seq >
ack+1)
            return 1;
      return 0;
}

/*
 * clean snoop cache of packets that have been acked.
 */
double
Snoop::snoop_cleanbufs_(int ack)
{
      Scheduler &s = Scheduler::instance();
      double sndTime = -1;

      if (toutPending_) {
            s.cancel(toutPending_);
            // xxx: I think that toutPending_ doesn't need to be freed
because snoop didn't allocate it (but I'm not sure).
            toutPending_ = 0;
      };

      if (empty_())
            return sndTime;

      int i = buftail_;
      do {
            hdr_snoop *sh = hdr_snoop::access(pkts_[i]);
            int seq = hdr_tcp::access(pkts_[i])->seqno();

            if (seq <= ack) {
                  sndTime = sh->sndTime();
                  Packet::free(pkts_[i]);
                  pkts_[i] = 0;
                  fstate_ &= ~SNOOP_FULL; /* XXX redundant? */
            } else if (seq > ack)
                  break;
            i = next(i);
      } while (i != bufhead_);

      if ((i != buftail_) || (bufhead_ != buftail_)) {
            fstate_ &= ~SNOOP_FULL;
            buftail_ = i;
      }
      if (!empty_()) {
            toutPending_ = (Event *) (pkts_[buftail_]);
            s.schedule(rxmitHandler_, toutPending_, timeout());
            hdr_snoop *sh = hdr_snoop::access(pkts_[buftail_]);
            tailTime_ = sh->sndTime();
      }

      return sndTime;
}

/*
 * Calculate smoothed rtt estimate and linear deviation.
 */
void
```

```
Snoop::snoop_rtt(double sndTime)
{
      double rtt = Scheduler::instance().clock() - sndTime;

      if (parent_->integrate()) {
            parent_->snoop_rtt(sndTime);
            return;
      }

      if (rtt > 0) {
            srtt_ = g_*srtt_ + (1-g_)*rtt;
            double delta = rtt - srtt_;
            if (delta < 0)
                  delta = -delta;
            if (rttvar_ != 0)
                  rttvar_ = g_*delta + (1-g_)*rttvar_;
            else
                  rttvar_ = delta;
      }
}


/*
 * Calculate smoothed rtt estimate and linear deviation.
 */
void
LLSnoop::snoop_rtt(double sndTime)
{
      double rtt = Scheduler::instance().clock() - sndTime;
      if (rtt > 0) {
            srtt_ = g_*srtt_ + (1-g_)*rtt;
            double delta = rtt - srtt_;
            if (delta < 0)
                  delta = -delta;
            if (rttvar_ != 0)
                  rttvar_ = g_*delta + (1-g_)*rttvar_;
            else
                  rttvar_ = delta;
      }
}

/*
 * Returns 1 if recent queue length is <= half the maximum and 0
otherwise.
 */
int
Snoop::snoop_qlong()
{
      /* For now only instantaneous lengths */
      //    if (parent_->ifq()->length() <= 3*parent_->ifq()-
>limit()/4)

      return 1;
            //    return 0;
}

/*
 * Ideally, would like to schedule snoop retransmissions at higher
priority.
 */
int
```

```
Snoop::snoop_rxmit(Packet *pkt)
{
      Scheduler& s = Scheduler::instance();
      if (pkt != 0) {
            hdr_snoop *sh = hdr_snoop::access(pkt);
            if (sh->numRxmit() < SNOOP_MAX_RXMIT && snoop_qlong()) {
                  /*                && sh->seqno() == lastAck_+1)  */


                  printf("%f Rxmitting packet %d\n", s.clock(),
                         hdr_tcp::access(pkt)->seqno());


                  // need to specify direction, in this case, down
                  hdr_cmn *ch = HDR_CMN(pkt);
                  ch->direction() = hdr_cmn::DOWN;  // Ben added

                  sh->sndTime() = s.clock();
                  sh->numRxmit() = sh->numRxmit() + 1;
                  Packet *p = pkt->copy();
                  parent_->sendDown(p);
            } else
                  return SNOOP_PROPAGATE;
      }
      /* Reset timeout for later time. */
      if (toutPending_) {
            s.cancel(toutPending_);
            // xxx: I think that toutPending_ doesn't need to be freed
because snoop didn't allocate it (but I'm not sure).
      };
      toutPending_ = (Event *)pkt;
      s.schedule(rxmitHandler_, toutPending_, timeout());
      return SNOOP_SUPPRESS;
}

void
Snoop::snoop_cleanup()
{
}

void
SnoopRxmitHandler::handle(Event *)
{
      Packet *p = snoop_->pkts_[snoop_->buftail_];
      snoop_->toutPending_ = 0;
      if (p == 0)
            return;
      hdr_snoop *sh = hdr_snoop::access(p);
      if (sh->seqno() != snoop_->lastAck_ + 1)
            return;
      if ((snoop_->bufhead_ != snoop_->buftail_) ||
          (snoop_->fstate_ & SNOOP_FULL)) {
                        a++;
                        printf("%f Snoop timeout %-d\n",
Scheduler::instance().clock(), a);
            if (snoop_->snoop_rxmit(p) == SNOOP_SUPPRESS)
                  snoop_->expNextAck_ = snoop_->next(snoop_->buftail_);
      }
}
```

Anexo 3. tcp.h

```
/* -*-      Mode:C++; c-basic-offset:8; tab-width:8; indent-tabs-mode:t
-*- */ /*
 * Copyright (c) 1991-1997 Regents of the University of California.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *  2.  Redistributions  in  binary  form  must  reproduce  the  above
copyright
 *     notice, this list of conditions and the following disclaimer in
the
 *        documentation  and/or  other  materials  provided  with  the
distribution.
 * 3. All advertising materials mentioning features or use of this
software
 *    must display the following acknowledgement:
 *    This product includes software developed by the Computer Systems
 *    Engineering Group at Lawrence Berkeley Laboratory.
 * 4. Neither the name of the University nor of the Laboratory may be
used
 *      to endorse or promote products derived from this software
without
 *    specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS''
AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE
LIABLE
 *  FOR  ANY  DIRECT,  INDIRECT,  INCIDENTAL,  SPECIAL,  EXEMPLARY,  OR
CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
GOODS
 *  OR  SERVICES;  LOSS  OF  USE,  DATA,  OR  PROFITS;  OR  BUSINESS
INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
OF
 * SUCH DAMAGE.
 *
 *   @(#)   $Header:   /nfs/jade/vint/CVSROOT/ns-2/tcp/tcp.h,v   1.121
2005/06/20 16:30:30 sfloyd Exp $ (LBL)
 */
#ifndef ns_tcp_h
#define ns_tcp_h

#include "agent.h"
#include "packet.h"
```

```
//class EventTrace;

struct hdr_tcp {
#define NSA 3
      double ts_;             /* time packet generated (at source) */
      double ts_echo_;         /* the echoed timestamp (originally sent
by
                                  the peer) */
      int seqno_;             /* sequence number */
      int reason_;            /* reason for a retransmit */
      int sack_area_[NSA+1][2];     /* sack blocks: start, end of block
*/
      int sa_length_;          /* Indicate the number of SACKs in this
*
                                * packet.   Adds  2+sack_length*8  bytes
*/
      int ackno_;             /* ACK number for FullTcp */
      int hlen_;              /* header len (bytes) for FullTcp */
      int tcp_flags_;         /* TCP flags for FullTcp */
      int last_rtt_;          /* more recent RTT measurement in ms, */
                        /*   for statistics only */
      double wnd_;            /*  Para  SnoopAdaptado  y  FullTCP  ->
FullTcp debe comprobar este campo. Ventana ofrecida */

      static int offset_;     // offset for this header
      inline static int& offset() { return offset_; }
      inline static hdr_tcp* access(Packet* p) {
            return (hdr_tcp*) p->access(offset_);
      }

      /* per-field member functions */
      double& ts() { return (ts_); }
      double& ts_echo() { return (ts_echo_); }
      int& seqno() { return (seqno_); }
      int& reason() { return (reason_); }
      int& sa_left(int n) { return (sack_area_[n][0]); }
      int& sa_right(int n) { return (sack_area_[n][1]); }
      int& sa_length() { return (sa_length_); }
      int& hlen() { return (hlen_); }
      int& ackno() { return (ackno_); }
      int& flags() { return (tcp_flags_); }
      int& last_rtt() { return (last_rtt_); }
      double& wnd() { return (wnd_); }
};

/* these are used to mark packets as to why we xmitted them */
#define TCP_REASON_TIMEOUT    0x01
#define     TCP_REASON_DUPACK 0x02
#define     TCP_REASON_RBP          0x03   // used only in tcp-rbp.cc
#define TCP_REASON_PARTIALACK   0x04

/* these are reasons we adjusted our congestion window */

#define     CWND_ACTION_DUPACK      1     // dup acks/fast retransmit
#define     CWND_ACTION_TIMEOUT     2     // retransmission timeout
#define     CWND_ACTION_ECN         3     //  ECN  bit  [src  quench  if
supported]
#define CWND_ACTION_EXITED        4         // congestion recovery has
ended
                              // (when previously CWND_ACTION_DUPACK)
```

```
/* these are bits for how to change the cwnd and ssthresh values */

#define     CLOSE_SSTHRESH_HALF     0x00000001
#define     CLOSE_CWND_HALF         0x00000002
#define     CLOSE_CWND_RESTART      0x00000004
#define     CLOSE_CWND_INIT         0x00000008
#define     CLOSE_CWND_ONE          0x00000010
#define CLOSE_SSTHRESH_HALVE  0x00000020
#define CLOSE_CWND_HALVE      0x00000040
#define THREE_QUARTER_SSTHRESH  0x00000080
#define CLOSE_CWND_HALF_WAY   0x00000100
#define CWND_HALF_WITH_MIN    0x00000200
#define TCP_IDLE        0x00000400
#define NO_OUTSTANDING_DATA     0x00000800

/*
 * tcp_tick_:
 * default 0.1,
 * 0.3 for 4.3 BSD,
 * 0.01 for new window algorithms,
 */

#define NUMDUPACKS 3          /* This is no longer used.  The variable
*/
                        /* numdupacks_ is used instead. */
#define TCP_MAXSEQ 1073741824    /* Number that curseq_ is set to for
*/
                        /* "infinite send" (2^30)            */

#define TCP_TIMER_RTX         0
#define TCP_TIMER_DELSND      1
#define TCP_TIMER_BURSTSND    2
#define TCP_TIMER_DELACK      3
#define TCP_TIMER_Q         4
#define TCP_TIMER_RESET         5

class TcpAgent;

class RtxTimer : public TimerHandler {
public:
      RtxTimer(TcpAgent *a) : TimerHandler() { a_ = a; }
protected:
      virtual void expire(Event *e);
      TcpAgent *a_;
};

class DelSndTimer : public TimerHandler {
public:
      DelSndTimer(TcpAgent *a) : TimerHandler() { a_ = a; }
protected:
      virtual void expire(Event *e);
      TcpAgent *a_;
};

class BurstSndTimer : public TimerHandler {
public:
      BurstSndTimer(TcpAgent *a) : TimerHandler() { a_ = a; }
protected:
      virtual void expire(Event *e);
      TcpAgent *a_;
```

```
};

/*
 * Variables for HighSpeed TCP.
 */
//int *hs_win_;         // array of cwnd values
//int *hs_increase_;    // array of increase values
//double *hs_decrease_; // array of decrease values
struct hstcp {
      double low_p;  // low_p
      double dec1;      // for computing the decrease parameter
      double dec2;    // for computing the decrease parameter
      double p1;  // for computing p
        double p2;      // for computing p
      /* The next three parameters are for CPU overhead, for computing
*/
      /*      the  HighSpeed  parameters  less  frequently.    A  better
solution */
      /*   might be just to have a look-up array.  */
      double cwnd_last_;      /* last cwnd for computed parameters */
        double increase_last_;     /* increase param for cwnd_last_ */
      hstcp() : low_p(0.0), dec1(0.0), dec2(0.0), p1(0.0), p2(0.0),
          cwnd_last_(0.0), increase_last_(0.0) { }
};

class TcpAgent : public Agent {
public:
      TcpAgent();
      ~TcpAgent() {free(tss);}
        virtual void recv(Packet*, Handler*);
      virtual void timeout(int tno);
      virtual void timeout_nonrtx(int tno);
      int command(int argc, const char*const* argv);
      virtual void sendmsg(int nbytes, const char *flags = 0);


      void trace(TracedVar* v);
      virtual void advanceby(int delta);
protected:
      virtual int window();
      virtual double windowd();
      void print_if_needed(double memb_time);
      void traceAll();
      virtual void traceVar(TracedVar* v);
      virtual int headersize();   // a tcp header

      virtual void delay_bind_init_all();
      virtual int delay_bind_dispatch(const char *varName, const char
*localName, TclObject *tracer);

      TracedInt t_seqno_;     /* sequence number */
      /*
       * State encompassing the round-trip-time estimate.
       * srtt and rttvar are stored as fixed point;
       * srtt has 3 bits to the right of the binary point, rttvar has
2.
       */
      TracedInt t_rtt_;       /* round trip time */
      TracedInt t_srtt_;      /* smoothed round-trip time */
      TracedInt t_rttvar_;    /* variance in round-trip time */
```

```
      TracedInt t_backoff_;   /* current multiplier, 1 if not backed
off */
      #define T_RTT_BITS 0
      int  T_SRTT_BITS;            /* exponent of weight for updating
t_srtt_ */
      int srtt_init_;         /* initial value for computing t_srtt_ */
      int  T_RTTVAR_BITS;          /* exponent of weight for updating
t_rttvar_ */
      int rttvar_exp_;        /* exponent of multiple for t_rtxcur_ */
      int rttvar_init_;        /* initial value for computing t_rttvar_
*/
      double t_rtxcur_; /* current retransmit value */
      double rtxcur_init_;    /* initial value for t_rtxcur_ */
      virtual void rtt_init();
      virtual double rtt_timeout(); /*   provide   RTO   based   on   RTT
estimates */
      virtual void rtt_update(double tao);     /* update RTT estimate
*/
      virtual void rtt_backoff();        /* double multiplier */
      /* End of state for the round-trip-time estimate. */

      /* Timestamps. */
      double ts_peer_;          /* the most recent timestamp the peer
sent */
      double ts_echo_;          /* the most recent timestamp the peer
echoed */
      int ts_option_size_;    // header bytes in a ts option
        double *tss;               // To store sent timestamps, with
bugfix_ts_
        int tss_size_;          // Current capacity of tss
      int ts_option_;         /* use RFC1323-like timestamps? */
      /* End of timestamps. */

      /* connection and packet dynamics */
      virtual void output(int seqno, int reason = 0);
      virtual void send_much(int force, int reason, int maxburst = 0);
      virtual void newtimer(Packet*);
      virtual void dupack_action();       /* do this on dupacks */
      virtual void send_one();            /* do this on 1-2 dupacks */
      double  linear(double  x,  double  x_1,  double  y_1,  double  x_2,
double y_2);
      /* the "linear" function is for experimental highspeed TCP */
      void opencwnd();

      void slowdown(int how);             /* reduce cwnd/ssthresh */
      void ecn(int seqno);          /* react to quench */
      virtual void set_initial_window();  /* set IW */
      double initial_window();            /* what is IW? */
      void reset();
      void newack(Packet*);
      void tcp_eln(Packet *pkt); /* reaction to ELN (usually wireless)
*/
      void finish(); /* called when the connection is terminated */
      void reset_qoption();   /* for QOption with EnblRTTCtr_ */
      void rtt_counting();    /* for QOption with EnblRTTCtr_ */
      int network_limited();  /* Sending limited by network? */
      double  limited_slow_start(double  cwnd,  double  max_ssthresh,
double increment);
                        /* Limited slow-start for high windows */
      virtual int numdupacks(double cwnd);     /*      for      getting
numdupacks_ */
```

```
	virtual void processQuickStart(Packet *pkt);
	virtual void endQuickStart();
	int lossQuickStart();

	/* Helper functions. Currently used by tcp-asym */
	virtual void output_helper(Packet*) { return; }
	virtual void send_helper(int) { return; }
	virtual void send_idle_helper() { return; }
	virtual void recv_helper(Packet*) { return; }
	virtual void recv_frto_helper(Packet*);
	virtual void recv_newack_helper(Packet*);
	virtual void partialnewack_helper(Packet*) {};
	/* End of helper functions. */

	int force_wnd(int num);
	void spurious_timeout();

	/* Timers */
	RtxTimer rtx_timer_;
	DelSndTimer delsnd_timer_;
	BurstSndTimer burstsnd_timer_;
	virtual void cancel_timers() {
	      rtx_timer_.force_cancel();
	      burstsnd_timer_.force_cancel();
	      delsnd_timer_.force_cancel();
	}
	virtual void cancel_rtx_timer() {
	      rtx_timer_.force_cancel();
	}
	virtual void set_rtx_timer();
	void reset_rtx_timer(int mild, int backoff = 1);
	int timerfix_;          /* set to true to update timer *after* */
	                /* update the RTT, instead of before   */
	int rfc2988_;           /* Use updated RFC 2988 timers */
	/* End of timers. */

	double boot_time_;      /* where between 'ticks' this sytem came
up */
	double overhead_;
	double wnd_;
	double wnd_const_;
	double wnd_th_;         /* window "threshold" */
	double wnd_init_;
	double wnd_restart_;
	double tcp_tick_; /* clock granularity */
	int wnd_option_;
	int wnd_init_option_;   /* 1 for using wnd_init_ */
	                /* 2 for using large initial windows */
	double decrease_num_;   /* factor for multiplicative decrease */
	double increase_num_;   /* factor for additive increase */
	int tcpip_base_hdr_size_;  /* size of base TCP/IP header */
	int ts_resetRTO_; /* Un-backoff RTO after any valid RTT, */
	                /*   including from a retransmitted pkt?  */
	                /* The old version was "false". */
	                /* But "true" gives better performance, and */
	                        /* seems conformant with RFC 2988. */
	int maxcwnd_;           /* max # cwnd can ever be */
	  int numdupacks_;      /* dup ACKs before fast retransmit */
	int numdupacksFrac_;    /* for a larger numdupacks_ with large */
	                /* windows */
	double maxrto_;         /* max value of an RTO */
```

```
      double minrto_;         /* min value of an RTO */

      /* For modeling SYN and SYN/ACK packets. */
      int syn_;         /* 1 for modeling SYN/ACK exchange */
      int delay_growth_;      /*  delay  opening  cwnd  until  1st  data
recv'd */
      /* End of modeling SYN and SYN/ACK packets. */

      /* F-RTO */
      int frto_enabled_;      /* != 0 to enable F-RTO */
      int sfrto_enabled_;     /* != 0 to enabled SACK-based F-RTO */
      int spurious_response_; /* Response variant to spurious RTO */
      /* End of R-RTO */

      /* Parameters for backwards compatility with old code. */
      int bug_fix_;           /* 1 for multiple-fast-retransmit fix */
      int less_careful_;      /*  1  for  Less  Careful  variant  of
bug_fix_, */
                        /*  for illustration only  */
      int exitFastRetrans_;   /* True to clean exits of Fast Retransmit
*/
                        /* False for buggy old behavior */
      int bugfix_ack_;        // 1 to enable ACK heuristic, to allow
                        //  multiple-fast-retransmits in special cases.
                        // From Andrei Gurtov
      int bugfix_ts_;          // 1 to enable timestamp heuristic, to
allow
                        //  multiple-fast-retransmits in special cases.
                        // From Andrei Gurtov
                        // Not implemented yet.
      int old_ecn_;           /* For backwards compatibility with the
                         * old ECN implementation, which never
                         * reduced the congestion window below
                         * one packet. */
      /* End of parameters for backwards compatility. */

      /* Parameters for alternate congestion control mechanisms. */
      double k_parameter_;     /* k parameter in binomial controls */
      double l_parameter_;     /* l parameter in binomial controls */
      int precision_reduce_;  /* non-integer reduction of cwnd */
      int maxburst_;          /* max # packets can send back-2-back */
      int aggressive_maxburst_;     /* Send on a non-valid ack? */
      /*   End   of   parameters   for   alternate   congestion   control
mechanisms. */

      FILE *plotfile_;
      /*
       * Dynamic state.
       */
      TracedInt dupacks_;     /* number of duplicate acks */
      TracedInt curseq_;      /* highest seqno "produced by app" */
      TracedInt highest_ack_; /* not frozen during Fast Recovery */
      TracedDouble cwnd_;     /* current window */
      TracedInt ssthresh_;    /* slow start threshold */
      TracedInt maxseq_;      /* used for Karn algorithm */
                        /* highest seqno sent so far */
      int last_ack_;          /* largest consecutive ACK, frozen during
                         *          Fast Recovery */
      //TracedDouble cong_wnd;
      int recover_;           /* highest pkt sent before dup acks, */
                        /*   timeout, or source quench/ecn */
```

```
      int last_cwnd_action_;  /* CWND_ACTION_{TIMEOUT,DUPACK,ECN} */
      int count_;       /* used in window increment algorithms */
      int rtt_active_;  /* 1 if a rtt sample is pending */
      int rtt_seq_;           /* seq # of timed seg if rtt_active_ is 1
*/
      double rtt_ts_;         /* time at which rtt_seq_ was sent */
      double firstsent_;      /* When first packet was sent  --Allman
*/
      double lastreset_;      /* W.N. Last time connection was reset -
for */
                      /* detecting pkts from previous incarnations */
      int closed_;            /* whether this connection has closed */

      /*  Dynamic  state  used  for  alternate  congestion  control
mechanisms */
      double awnd_;           /* averaged window */
      int first_decrease_;    /* First decrease of congestion window.
*/
                      /* Used for decrease_num_ != 0.5. */
      double fcnt_;           /* used in window increment algorithms */
      double base_cwnd_;      /*   base   window   (for   experimental
purposes) */
      /* End of state for alternate congestion control mechanisms */

      /* Dynamic state only used for monitoring */
      int trace_all_oneline_; /* TCP tracing vars all in one line or
not? */
      int nam_tracevar_;       /* Output nam's variable trace or just
plain
                          text variable trace? */
        TracedInt ndatapack_;   /* number of data packets sent */
        TracedInt ndatabytes_;  /* number of data bytes sent */
        TracedInt nackpack_;    /* number of ack packets received */
        TracedInt nrexmit_;     /* number of retransmit timeouts
                          when there was data outstanding */
        TracedInt nrexmitpack_; /* number of retransmited packets */
        TracedInt nrexmitbytes_; /* number of retransmited bytes */
        TracedInt necnresponses_; /* number of times cwnd was reduced
                          in response to an ecn packet -- sylvia */
        TracedInt ncwndcuts_;      /* number of times cwnd was reduced
                          for any reason -- sylvia */
        TracedInt ncwndcuts1_;     /* number of times cwnd was reduced
                                    due  to  congestion  (as  opposed  to
idle
                                    periods */
      /* end of dynamic state for monitoring */

      /* Specifying variants in TCP algorithms.  */
      int   slow_start_restart_;   /*   boolean:   re-init   cwnd   after
connection
                           goes idle.  On by default. */
      int restart_bugfix_;    /* ssthresh is cut down because of
                          timeouts during a connection's idle period.
                          Setting this boolean fixes this problem.
                          For now, it is off by default. */
        TracedInt singledup_;   /* Send on a single dup ack.  */
      int LimTransmitFix_;    /* To fix a bug in Limited Transmit. */
      int noFastRetrans_;     /* No Fast Retransmit option.  */
      int oldCode_;           /* Use old code. */
      int useHeaders_;  /* boolean: Add TCP/IP header sizes */
      /* end of specifying variants */
```

```
      /* Used for ECN */
      int ecn_;         /* Explicit Congestion Notification */
      int cong_action_; /* Congestion Action.  True to indicate
                           that the sender responded to congestion. */
        int ecn_burst_;       /* True when the previous ACK packet
                         *  carried ECN-Echo. */
      int ecn_backoff_; /* True when retransmit timer should begin
                           to be backed off.  */
      int ect_;         /* turn on ect bit now? */
      int SetCWRonRetransmit_;  /* True to allow setting CWR on */
                        /*  retransmitted packets.   Affects */
                        /*  performance for Reno with ECN.  */
      /* end of ECN */

      /* used for Explicit Loss Notification */
        int eln_;                       /* Explicit Loss Notification
(wireless) */
        int  eln_rxmit_thresh_;     /*  Threshold  for  ELN-triggered
rxmissions */
        int eln_last_rxmit_;      /* Last packet rxmitted due to ELN
info */
      /* end of Explicit Loss Notification */

      /* for experimental high-speed TCP */
      /* These four parameters define the HighSpeed response function.
*/
      int low_window_;  /* window for turning on high-speed TCP */
      int high_window_; /* target window for new response function */
      double high_p_;         /*  target  drop  rate  for  new  response
function */
      double high_decrease_;  /* decrease rate at target window */
      /* The next parameter is for Limited Slow-Start. */
      int max_ssthresh_;      /* max value for ssthresh_ */

      /* These two functions are just an easy structuring of the code.
*/
      double increase_param();  /* get increase parameter for current
cwnd */
      double decrease_param();  /* get decrease parameter for current
cwnd */
      int cwnd_range_;  /* for determining when to recompute params. */
      hstcp hstcp_;           /* HighSpeed TCP variables */
        /* end of section for experimental high-speed TCP */

      /* for Quick-Start, experimental. */
      int rate_request_;       /* Rate request in Kbps, for QuickStart.
*/
      int qs_enabled_;        /* to enable QuickStart. */
      int qs_requested_;
      int qs_approved_;
        int  qs_window_;    /*  >0:  there  are  outstanding  non-acked
segments
                            from QS window */
        int qs_cwnd_; /* Initial window for Quick-Start */
        int tcp_qs_recovery_; /* != 0 if we apply slow start on packet
                                 losses during QS window */
      int  qs_request_mode_;     /*  1  =  Try  to  avoid  unnecessary  QS
requests
                           for short flows. Use qs_rtt_ as the RTT
                           used in window calculation.
```

```
                        2 = Always request 'rate_request_' bytes,
                        regardless of flow size */
      int qs_thresh_;          /* Do not use QS if there are less data
to send
                        than this. Applies only if
                        qs_request_mode_ == 1 */
      int qs_rtt_;             /* QS needs some assumption of the RTT
in
                        in order to be able to determine how much
                        it needs for rate request with given amount
                        of data to send. milliseconds. */
      int ttl_diff_;
        /* end of section for Quick-Start. */

      /* F-RTO: !=0 when F-RTO recovery is underway, N:th round-trip
       * since RTO. Can have values between 0-2 */
      int frto_;
      int pipe_prev_; /* window size when timeout last occurred */

        /* support for event-tracing */
        //EventTrace *et_;
        void trace_event(char *eventtype);

      /* these function are now obsolete, see other above */
      void closecwnd(int how);
      void quench(int how);

      /* TCP quiescence, reducing cwnd after an idle period */
      void process_qoption_after_send() ;
      void process_qoption_after_ack(int seqno) ;
      int QOption_ ; /* TCP quiescence option */
      int EnblRTTCtr_ ; /* are we using a corase grained timer? */
      int T_full ; /* last time the window was full */
      int T_last ;
      int T_prev ;
      int T_start ;
      int RTT_count ;
      int RTT_prev ;
      int RTT_goodcount ;
      int F_counting ;
      int W_used ;
      int W_timed ;
      int F_full ;
      int Backoffs ;
      int  control_increase_  ;  /*  If  true,  don't  increase  cwnd  if
sender */
                        /*  is not window-limited.  */
      int prev_highest_ack_ ; /* Used to determine if sender is */
                        /*  window-limited.  */
      /* end of TCP quiescence */
};

/* TCP Reno */
class RenoTcpAgent : public virtual TcpAgent {
 public:
      RenoTcpAgent();
      virtual int window();
      virtual double windowd();
      virtual void recv(Packet *pkt, Handler*);
      virtual void timeout(int tno);
      virtual void dupack_action();
```

```
 protected:
      int allow_fast_retransmit(int last_cwnd_action_);
      unsigned int dupwnd_;
};

/* TCP New Reno */
class NewRenoTcpAgent : public virtual RenoTcpAgent {
 public:
      NewRenoTcpAgent();
      virtual void recv(Packet *pkt, Handler*);
      virtual void partialnewack_helper(Packet* pkt);
      virtual void dupack_action();
 protected:
      int newreno_changes_;   /*   0   for   fixing   unnecessary   fast
retransmits */
                        /* 1 for additional code from Allman, */
                        /* to implement other algorithms from */
                        /* Hoe's paper, including sending a new */
                        /* packet for every two duplicate ACKs. */
                        /* The default is set to 0. */
      int newreno_changes1_;  /* Newreno_changes1_ set to 0 gives the
*/
                        /* Slow-but-Steady variant of NewReno from */
                        /* RFC 2582, with the retransmit timer reset */
                        /* after each partial new ack. */
                        /* Newreno_changes1_ set to 1 gives the */
                        /* Impatient variant of NewReno from */
                        /* RFC 2582, with the retransmit timer reset */
                        /* only for the first partial new ack. */
                        /* The default is set to 0 */
      void partialnewack(Packet *pkt);
      int allow_fast_retransmit(int last_cwnd_action_);
      int acked_, new_ssthresh_;  /* used if newreno_changes_ == 1 */
      double ack2_, ack3_, basertt_; /* used if newreno_changes_ == 1
*/
      int firstpartial_;      /* For the first partial ACK. */
      int partial_window_deflation_; /* 0 if set cwnd to ssthresh upon
*/
                               /* partial new ack (default) */
                               /* 1 if deflate (cwnd + dupwnd) by */
                               /* amount of data acked */
                               /* "Partial window deflation" is */
                               /* discussed in RFC 2582. */
      int exit_recovery_fix_;  /* 0 for setting cwnd to ssthresh upon
*/
                         /* leaving fast recovery (default) */
                         /* 1 for setting cwnd to min(ssthresh, */
                         /* amnt. of data in network) when leaving */
};

/* TCP vegas (VegasTcpAgent) */
class VegasTcpAgent : public virtual TcpAgent {
 public:
      VegasTcpAgent();
      ~VegasTcpAgent();
      virtual void recv(Packet *pkt, Handler *);
      virtual void timeout(int tno);
protected:
      double vegastime() {
            return(Scheduler::instance().clock() - firstsent_);
      }
```

```
      virtual void output(int seqno, int reason = 0);
      virtual void recv_newack_helper(Packet*);
      int vegas_expire(Packet*);
      void reset();
      void vegas_inflate_cwnd(int win, double current_time);

      virtual void delay_bind_init_all();
      virtual int delay_bind_dispatch(const char *varName, const char
*localName, TclObject *tracer);

      double t_cwnd_changed_; // last time cwnd changed
      double firstrecv_;      // time recv the 1st ack

      int    v_alpha_;        // vegas thruput thresholds in pkts
      int    v_beta_;

      int    v_gamma_;        // threshold to change from slow-start to
                        // congestion avoidance, in pkts

      int     v_slowstart_;     // # of pkts to send after slow-start,
deflt(2)
      int     v_worried_;       // # of pkts to chk after dup ack (1 or
2)

      double v_timeout_;      // based on fine-grained timer
      double v_rtt_;
      double v_sa_;
      double v_sd_;

      int    v_cntRTT_;       // # of rtt measured within one rtt
      double v_sumRTT_;       // sum of rtt measured within one rtt

      double v_begtime_;      // tagged pkt sent
      int    v_begseq_; // tagged pkt seqno

      double* v_sendtime_;    //   each   unacked   pkt's   sendtime   is
recorded.
      int*   v_transmits_;    // # of retx for an unacked pkt

      int    v_maxwnd_; // maxwnd size for v_sendtime_[]
      double v_newcwnd_;      // record un-inflated cwnd

      double v_baseRTT_;      // min of all rtt

      double v_incr_;         // amount cwnd is increased in the next
rtt
      int    v_inc_flag_;     // if cwnd is allowed to incr for this
rtt

      double v_actual_; // actual send rate (pkt/s; needed for tcp-rbp)

      int ns_vegas_fix_level_;   // see comment at end of tcp-vegas.cc
for details of fixes
};

// Local Variables:
// mode:c++
// c-basic-offset: 8
// End:

#endif
```