Universidad Politécnica de Cartagena

Escuela Técnica Superior de Ingeniería de Telecomunicación

PRÁCTICAS DE REDES DE ORDENADORES

Propuesta del Trabajo de Prácticas:

Simulación de los mecanismos de control de congestión en TCP/IP

Profesores:

Esteban Egea López
Juan José Alcaraz Espín
Joan García Haro

Índice.

1 Consideraciones generales.

1.1 Objetivos

En este trabajo de prácticas los alumnos realizarán por parejas un simulador de fuentes de tráfico TCP/IP con mecanismos control de congestión, concretamente fuentes TCP RENO. Se validarán mediante la obtención de gráficas que muestren la evolución de diversos parámetros cuyo comportamiento es conocido. Con estas fuentes se evaluará una topología sencilla, basada en varias fuentes TCP/IP conectadas a un Router a través de enlaces de una determinada velocidad. Este Router estará conectado a un nodo destino mediante un enlace de salida que podrá tener una velocidad inferior a la suma de las tasas de pico entrantes. El Router tendrá también un *buffer* de capacidad finita. Las fuentes TCP/IP tendrán un gran volumen de información a transmitir (*Bulk Data Flow*), de forma que siempre estén transmitiendo. En este entorno se comprobará la utilidad de una herramienta de simulación para evaluar las prestaciones de una red y el dimensionado de sus nodos y enlaces.

1.2 Introducción

1.2.1 Qué deben hacer los alumnos.

Los alumnos deben implementar por parejas un módulo simple en C++ y NED llamado TCPIP. Lo que debe hacer este módulo se detalla en esta propuesta. Este módulo deberá ser incorporado en un entorno de simulación del que se proporcionan el resto de los módulos. Para ello los alumnos también deberán declarar la red a simular mediante un módulo compuesto en NED.

Una vez que el módulo esté implementado, integrado en la red y depurado se podrán realizar medidas de ciertos parámetros. Estas medidas también deberán ser programadas por el alumno y vienen especificadas en este documento.

Se deberá entregar una memoria del trabajo realizado y realizará un examen que consistirá principalmente en una exposición por parejas del trabajo seguidas de un turno de preguntas. El contenido de la memoria, el formato del examen y los criterios de evaluación también están descritos en esta propuesta.

1.2.2 Como está organizada esta propuesta.

A lo largo de esta propuesta se detalla exactamente lo que debe hacer el módulo simple TCPIP, que debe implementar, como se ya se ha indicado, una fuente TCP RENO. Los algoritmos que constituyen este tipo de TCP están detallados en el apartado 2. En el apartado 1.3 se representa un diagrama de estados que servirá de punto de partida y de referencia a lo largo de esta descripción.

En el apartado 3 se dan indicaciones que serán requisitos indispensables del simulador a realizar, como son la estructura en módulos del mismo y su implementación en NED, y los requisitos mínimos necesarios de la implementación del módulo TCPIP.

En el apartado 4 se dan sugerencias para la implementación. No son indispensables, como las del apartado anterior, pero pueden servir de ayuda para conseguir un código mejor y en menor tiempo.

En el apartado 5 se describen las medidas a realizar y los escenarios de referencia.

En el apartado 6 se describe el formato de la memoria a entregar, cómo será el examen que realizarán los alumnos y los criterios que se tendrán en cuenta a la hora de evaluar.

El apartado 7 es de bibliografía relacionada.

El Anexo 1 y el Anexo 2 describen los módulos simples Router y Sink que se proporcionan al alumno. Le serán útiles sobre todo para acceder al código de alguno de ellos, o de ambos en caso de que se implementen las medidas propuestas.

1.3 Diagrama de Estados

El módulo TCPIP a implementar representa una fuente de tráfico TCP RENO. Esta fuente se caracteriza por los algoritmos de control de congestión que posee. Las fuentes a implementar siempre estarán transmitiendo, pero en función de qué algoritmo de control de congestión se esté aplicando podremos identificar tres posibles estados de la fuente: Slow Start, Congestion Avoidance y Fast Retransmit / Fast Recovery. El control del flujo de la fuente se basa en un algoritmo de ventana deslizante que se describe en 2.1. La ventana se va desplazando hacia la derecha con la llegada de ACK's del otro extremo reconociendo paquetes que se han recibido, permitiendo a la fuente enviar nuevos paquetes. El tamaño de la ventana gobierna el flujo de paquetes instantáneo que puede transmitir la fuente y puede variar cuando se producen los siguientes eventos: Llegada de un ACK, vencimiento del Timeout de retransmisión, o llegada de varios ACK's duplicados. Los estados mencionados gobiernan el algoritmo con el que se varía el tamaño de dicha ventana cuando se produce uno de estos eventos. El paso de un estado a otro también está determinado por ciertos eventos, que se pueden observar en la figura, y que se explicarán en detalle en el apartado 2. Slow Start no es en sí mismo un mecanismo de control de congestión, sino un forma de adaptar la tasa de flujo de la fuente TCP lo más rápidamente posible al throughput máximo que le permite la red, empezando desde flujos muy pequeños e incrementando la ventana de transmisión y por tanto la tasa de transmisión a un ritmo rápido.

Congestion Avoidance es un mecanismo para adaptar la tasa de la fuente al estado de la red cuando la ventana ha superado un determinado umbral o tras recuperarse de una pérdida de algún paquete, sin incrementos bruscos y por tanto aumentando la ventana de transmisión más lentamente que en el caso anterior.

Fast Retransmitit / Fast Recovery es un algoritmo para recuperarse de pérdidas puntuales de paquetes sin reducir drásticamente la tasa de transmisión. Es el algoritmo que caracteriza a RENO frente a otras fuentes.

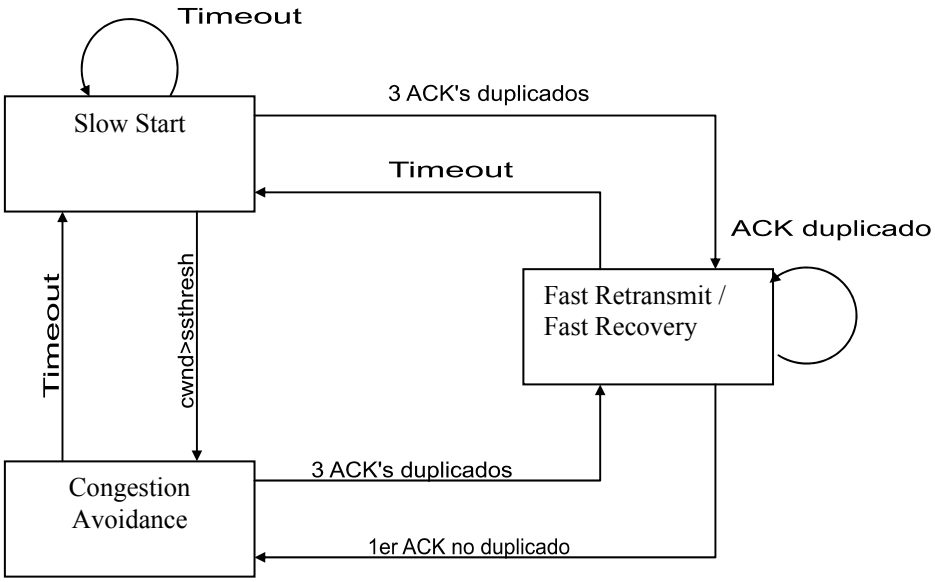

Figura 1. Estados relativos al control de congestión.

2 Algoritmos a implementar

2.1 Algoritmo de ventana deslizante

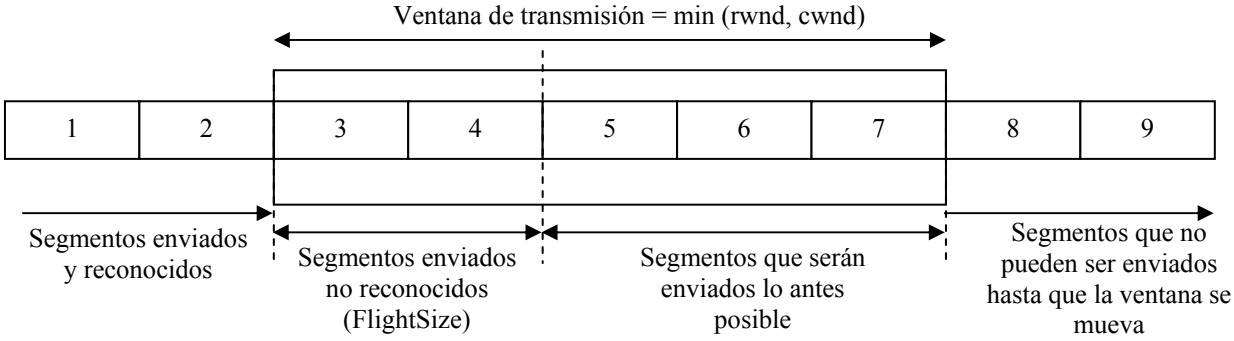

Figura 2.

El algoritmo de ventana deslizante es una de las técnicas que emplea TCP para el control de flujo del transmisor. El tamaño de la ventana regula cuántos segmentos puede enviar consecutivamente el transmisor antes de recibir un reconocimiento del receptor (ACK's), es decir, cuántos paquetes puede enviar "de golpe". El control de flujo debe combatir dos situaciones: la saturación del receptor y la congestión de la red. Por tanto el tamaño de la ventana deberá variar en función de estos dos problemas. Para prevenir la saturación del receptor, éste le indica al transmisor un tamaño máximo para la ventana, llamado ventana de recepción (rwnd), que estará en función del estado del buffer de recepción. Para controlar la congestión de la red será el propio transmisor el que regule el tamaño máximo de su ventana, estableciendo un tamaño máximo llamado ventana de congestión (cwnd) que irá variando en función del estado de la red. El transmisor se regula a sí mismo en función del comportamiento de la red. El tamaño de la ventana definitivo será el menor entre rwnd y cwnd.

Como TCP es un protocolo orientado a conexión, es necesario que el receptor informe al transmisor de los segmentos que le van llegando. Esto lo hace emitiendo ACK's (reconocimientos) que indican el número del siguiente byte que el receptor espera recibir (número de reconocimiento). Cuando un transmisor recibe un ACK, el número de reconocimiento le indica que todos los bytes por debajo de ese número de reconocimiento han llegado correctamente. El transmisor no necesita un ACK por cada segmento que envía ya que un ACK puede reconocer de golpe varios segmentos (ACK's acumulativos). Por este motivo si el receptor recibe un segmento fuera de secuencia, no puede emitir un ACK hasta que no se hayan completado los huecos que haya dejado el paquete fuera de secuencia. En la figura 3 se ilustra este hecho con el siguiente ejemplo: El transmisor transmite 5 segmentos; el receptor emite un ACK indicando el byte siguiente al último byte del segmento 2 (ACK 2), es decir, reconoce los segmentos 1 y 2; el segmento 4 se pierde y el 5 se recibe. Al recibir el 5 se vuelve a enviar un ACK 3, ya que no se puede enviar un ACK 5.

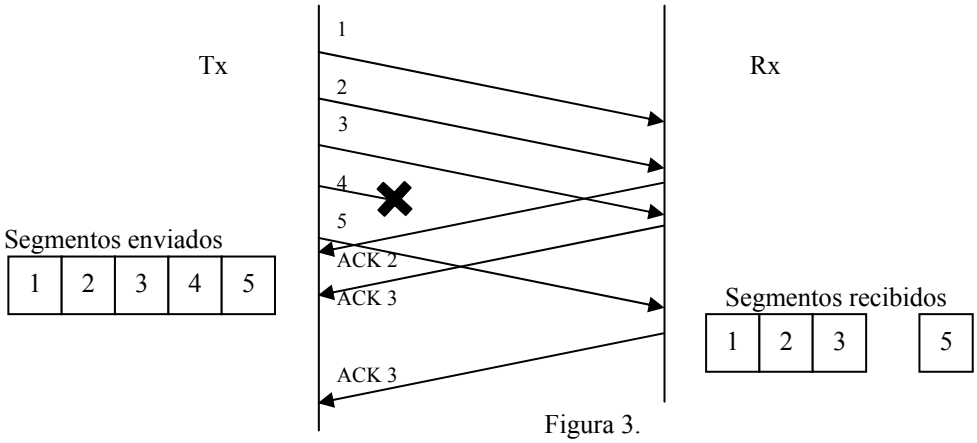

Figura 3.

Cada vez que se recibe un ACK se desplaza la ventana de transmisión a la derecha hasta la posición indicada por el ACK. En cada ACK el receptor indica al transmisor el valor de rwnd. Suponiendo que rwnd es 5 en todos los casos el movimiento de la ventana de transmisión en el ejemplo anterior es el siguiente

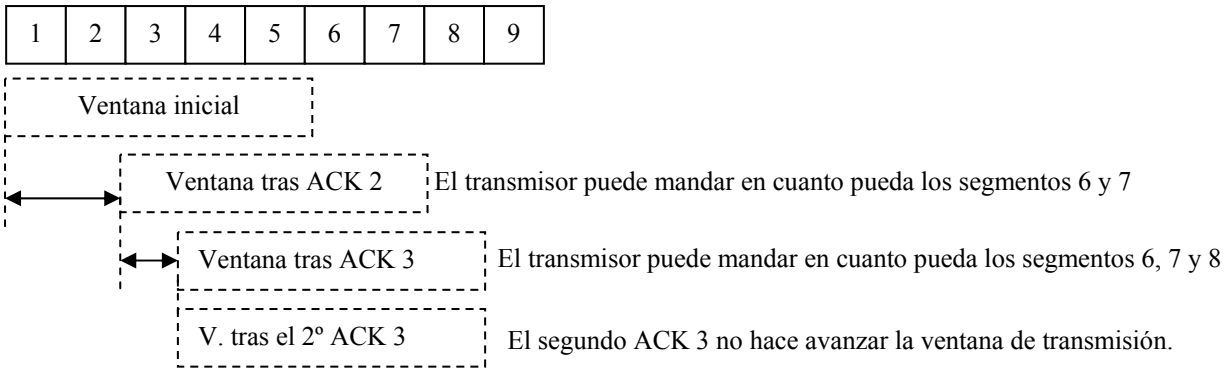

Figura 4.

2.2 Retransmisión

TCP asegura el envío de todos los datos de las capas superiores, por tanto debe ser capaz de detectar las pérdidas de datos y retransmitirlos. Una forma de detectarlo es mediante un temporizador, llamado *RTO Timer*. Este temporizador se activa cuando se envía un segmento (excepto si ya se activó en un segmento anterior). Si se cumple el tiempo máximo (RTO) asignado al temporizador, sin que se haya recibido un ACK que reconozca el segmento, se retransmite el primero de los segmentos enviados pendiente de reconocimiento. El valor de RTO lo va actualizando el transmisor permanentemente en función de la medida del *Round Trip Time* (RTT). En cualquier caso, cuando se produce un *Timeout* se aplica un algoritmo de retroceso exponencial (*exponential backoff)* para establecer el RTO del siguiente reenvío de ese segmento. Este algoritmo consiste simplemente en duplicar el valor del RTO para el siguiente reenvío. Una vez que se ha retransmitido el paquete se deben llevar a cabo las acciones indicadas en 2.4 y 2.5.

El algoritmo recomendado para el manejo del temporizador de retransmisión (*RTO Timer*) está descrito en [3] y es el siguiente:

1. Cada vez que se envía un paquete con datos (incluidas las retransmisiones), se activa el temporizador si no está activado previamente, para que expire en RTO segundos.
2. Si llega un ACK que reconoce nuevos segmentos, se reinicia el temporizador, que expirará en RTO segundos.[1] El valor de RTO se recalculará, si el ACK reconoce el SN del segmento en que se inició la medida y si éste no es un segmento retransmitido. El algoritmo para el cálculo del RTO se detalla en el apartado2.3.

En caso de que expire el temporizador se llevan a cabo las siguientes acciones:

3. Retransmitir el primer segmento pendiente de ACK.
4. Aplicar el algoritmo de Backoff, de forma que RTO = 2*RTO. Hasta alcanzar un valor máximo de 60 segundos.
5. Reiniciar el temporizador, asignándole un *Timeout* de RTO segundos (con el valor de RTO obtenido en el punto anterior).

Debe tenerse en cuenta que existirá un único temporizador por conexión, y no uno por paquete enviado. Por tanto, cuando expire el *RTO Timer*, la fuente TCPIP debe asumir que se ha perdido todo el FlightSize, y por tanto deberá actualizar el valor de esa variable y del SN después de retransmitir.

[1] Esto es lo que recomienda la RFC 2988, pero le resultará más sencillo reiniciar el temporizador cuando el ACK reconozca el SN del segmento en el que se reinició el temporizador por última vez, y hacer la medida del RTT al mismo tiempo. Consulte la Figura 6 para entender este punto.

2.3 Medida del RTT

El valor del RTO es un parámetro que se determina en función del RTT, que a su vez es un parámetro variable que depende, entre otras cosas del nivel de ocupación de la red. El transmisor TCP toma medidas del RTT y, a partir de estas medidas, calcula el valor del RTO. El algoritmo de cálculo del RTO es el siguiente.
1º Parámetros que intervienen en el cálculo del RTO:

- M: medida de RTT
- A: estimador de la media de RTT
- D: estimador de la desviación típica de RTT
- Err: diferencia entre la última medida de RTT y el valor del estimador de la media, A.
- G: granularidad del reloj de medida: se fija a 100 ms
- g: ganancia para el cálculo del estimador de la media, A. Se fija a 0.125.
- h: ganancia para el cálculo del estimador de la varianza, D. Se fija a 0.25.

2º Inicialización de RTO, según RFC 2988

$$RTO = 3$$

3º Cuando se envía el primer segmento se pone en marcha el un temporizador para la medida de RTT. Cuando se recibe un ACK reconociendo ese segmento, si no es un segmento retransmitido, se para dicho temporizador, obteniendo la medida del RTT, M. La medida M debe obtenerse como un múltiplo entero de G. Tras esta medida se inicializan por segunda vez los estimadores:

$$A = M + 0.5$$
$$D = A/2$$
$$RTO = A + \max(4D, G)$$

4º A partir del segundo ACK reconociendo un segmento no reenviado, ya se emplean las fórmulas definitivas para mantener actualizados los parámetros:

$$Err = M - A$$
$$A = A + gErr$$
$$D = D + h(|Err| - D)$$
$$RTO = A + \max(4D, G)$$

5º Tanto para el paso 3º como el 4º, si el RTO obtenido es inferior a 1s, se redondeará RTO a 1s.

6º Si se produce un *Timeout,* se aplican las instrucciones de 2.2 : el valor de RTO se duplica y el valor obtenido es el que se emplea para el temporizador RTO del paquete retransmitido.

7º Algoritmo de **Karn**: Se aplica siempre que se ha producido una retransmisión, por ejemplo, cuando se produce un *Timeout.* Por este algoritmo no se actualizan los estimadores cuando llega el ACK que reconoce un segmento retransmitido, ya que no puede saberse si corresponde a la primera o la segunda transmisión. El último valor calculado de RTO se mantendrá hasta que el algoritmo de Karn deje de aplicarse y se recalcule otro RTO.
En el siguiente diagrama se observa como se realizan las medidas de RTT M_1, M_2 y M_3.

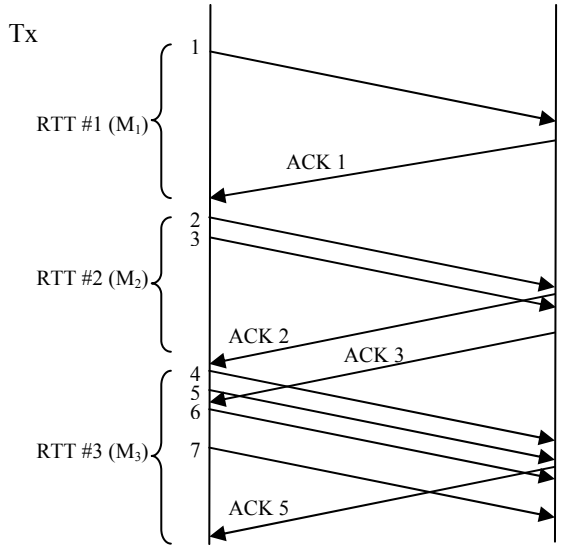

Figura 6.

Cuando hay un temporizador activo para la medida de RTT no se pone en marcha un segundo.

El temporizador iniciado con el segmento 2 se para al recibir ACK 2. El ACK 3 no debe parar la medida del RTT #2 ya que no reconoce el segmento en el que se inició ésta, el segmento 4.

El temporizador iniciado con el segmento 4 se para al recibir ACK 5, dando lugar a la medida de M_3 del RTT #3.

2.4 Slow Start

Slow Start es un algoritmo diseñado para que el transmisor adapte su tasa de transmisión a la tasa que la red puede aceptar, y se basa en el principio de que la tasa a la que el transmisor puede enviar paquetes debe ser la tasa a la que le llegan los reconocimientos del otro extremo. El algoritmo de Slow Start es el siguiente:

1º Parámetros que intervienen en Slow Start:

- cwnd: ventana de congestión
- ssthresh: umbral de *Slow Start*
- FlightSize: segmentos enviados y aún no reconocidos
- IW: valor inicial de la ventana de congestión
- LW: valor de la ventana de congestión tras producirse un *Timeout*.
- SMSS: Tamaño máximo del segmento.

2º Inicialización:

cwnd = IW = 1 ó 2 segmentos
ssthresh = rwnd

3º Siempre que cwnd sea **menor o igual** que ssthresh estaremos en Slow Start. Esto implica que por cada segmento que el receptor reconozca, cwnd se incrementa en un segmento.

4º Si se produce un *Timeout* se aplicaría la modificación de parámetros impuesta por Congestion Avoidance:

cwnd = LW = 1 segmento.
ssthresh = max (FlightSize/2 , 2 segmentos)

Tras realizar estas correcciones la fuente continúa en SlowStart.

Comentario sobre la generación de reconocimientos:

El receptor no debe retrasar excesivamente la generación de ACK's, por ese motivo la RFC 2581 indica que se debería generar un ACK cada 2 segmentos completos recibidos, y en cualquier caso se debe generar un ACK antes de que transcurran 500 ms desde la llegada del primer segmento en secuencia sin reconocer (no duplicado).

En el siguiente diagrama vemos un ejemplo de Slow Start.

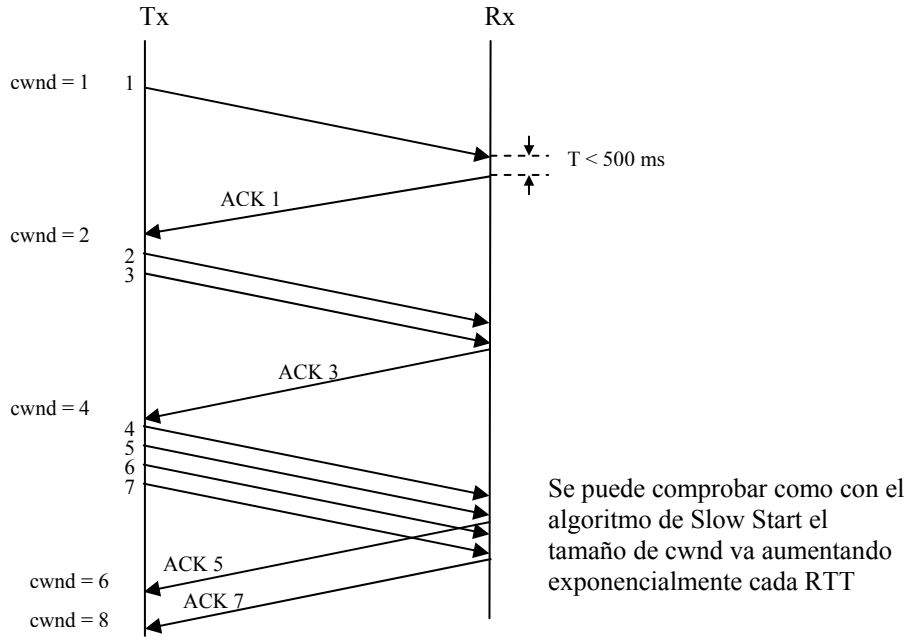

Figura 7.

2.5 Congestion Avoidance

Congestion Avoidance es un algoritmo muy relacionado con Slow Start como se ha visto anteriormente y su objetivo es adaptar la tasa de la fuente TCP a la situación de la red: ausencia de congestión, en cuyo caso va aumentando la tasa lentamente ó congestión, percibida como pérdida de paquetes (*Timeout*), en cuyo caso tomo las medidas vistas en el punto 4º del apartado anterior, que implican una reducción del ritmo de incremento de la ventana de transmisión, como se verá.

1º Parámetros que intervienen en Congestion Avoidance: Los mismos que en Slow Start, la inicialización de los mismos se realiza en esa etapa.

2º Siempre que cwnd sea **mayor** que ssthresh estaremos en Congestion Avoidance. Esto implica que por cada segmento que el receptor reconozca, cwnd se incrementa de acuerdo a la siguiente fórmula:

$$cwnd = cwnd + 1/cwnd, \text{ si cwnd se implementa como número de segmentos.}$$

ó

$$cwnd = cwnd + (SMSS)^2/cwnd, \text{ si cwnd se implementa como número de bytes}$$

3º Si se produce un *Timeout* se aplicaría la corrección de parámetros ya conocida:

cwnd = LW = 1 segmento.

ssthresh = max (FlightSize/2 , 2 segmentos)

Tras realizar estas correcciones la fuente pasaría a SlowStart ya que cwnd < ssthresh

2.6 Fast Retransmit

Fast Retransmit, es un algoritmo destinado a resolver una situación de pérdida de paquetes sin esperar a que se cumpla el *Timeout*, contando con la colaboración del receptor. El algoritmo se basa en el hecho de que la recepción tres o más ACKs idénticos es un indicador de que un segmento se ha perdido, y al mismo tiempo la congestión de la red no es tan grave como para tomar medidas drásticas como en el caso de un timeout. El hecho de que se sigan recibiendo ACKs es un indicador de que hay paquetes que han abandonando la red llegando a su destino.

El algoritmo consiste en tres acciones, una llevada a cabo en el transmisor y dos en el receptor:

Receptor:

- En cuanto se recibe un segmento fuera de orden, es decir que exista un hueco la secuencia de segmentos recibidos entes que él, se debe enviar inmediatamente un ACK duplicado, que reconozca el último segmento recibido correctamente.
- En cuanto se reciba el segmento que cubre el hueco dejado, se debe enviar inmediatamente un ACK reconociendo hasta el último segmento recibido correctamente.

Transmisor: Estando en CA ó SS, si se reciben 3 ACKs replicados, es decir que reconocen un mismo segmento ya confirmado, se retransmite inmediatamente el segmento siguiente al que reconocen los ACKs replicados.

Normalmente Fast Retransmit se implementa conjuntamente con Fast Recovery, dando lugar a la implementación **Reno** de TCP. En ese caso el transmisor entra en Fast Retransmit / Fast Recovery, continuando con el algoritmo descrito en el siguiente apartado.

2.7 Fast Recovery

Al entrar en Fast Retransmit / Fast Recovery, con la recepción de tres ACKs, se retransmite el paquete perdido, como indica Fast Retransmit, y se llevan a cabo las siguientes acciones:

1º Se asigna a ssthresh el valor determinado por Congestion Avoidance:

ssthresh = max (FlightSize/2 , 2 segmentos)

2º Tras retransmitir el paquete (Fast Retransmit), se asigna a cwnd el siguiente valor:

cwnd = ssthresh + 3 * SMSS

3º Cada vez que se recibe otro ACK duplicado, se incrementa cwnd un segmento, y si el valor de cwnd lo permite, se transmite un nuevo segmento. Tenga en cuenta que cada vez que llegue un ACK duplicado deberá decrementar el valor de FlightSize.

4º En el momento en que llegue el primer ACK no duplicado se asigna a cwnd el valor de ssthresh obtenido en el punto 1º. En este momento habremos entrado en Congestion Avoidance, ya que estamos reduciendo a la mitad la tasa a la que estábamos transmitiendo en el momento en que se produjo la pérdida del paquete.
En el siguiente diagrama vemos un ejemplo de Fast Retransmit / Fast Recovery

Nota: Recordar que el algoritmo de Karn se aplica siempre que hay una retransmisión.

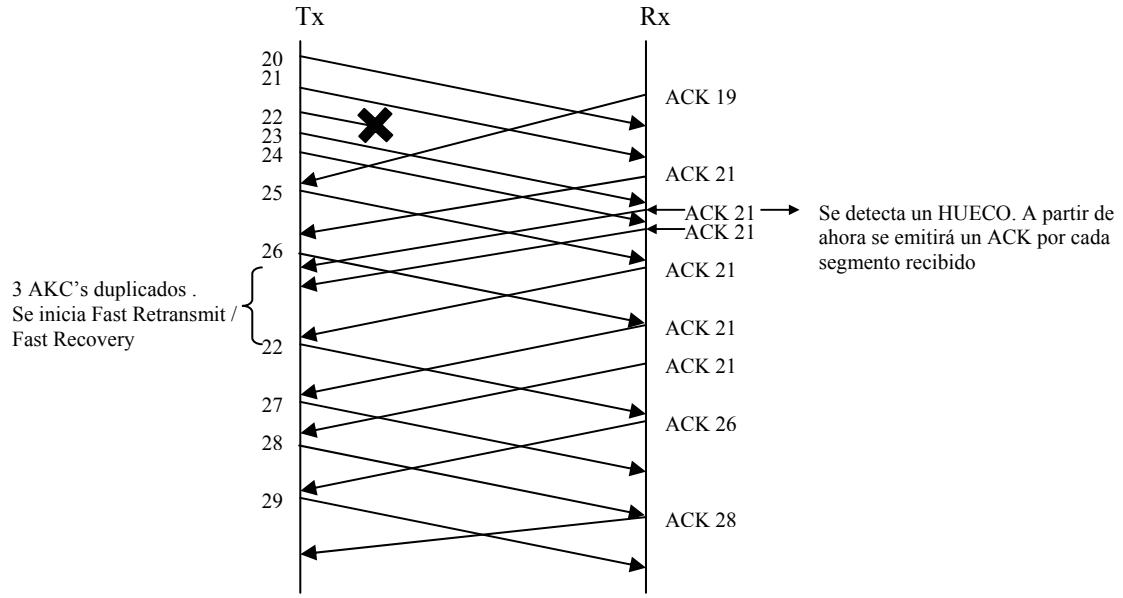

Figura 8.

3 Requerimientos del simulador a realizar

3.1 Elementos que componen el simulador

El simulador estará compuesto por los siguiente módulos:

TCPIP: Módulo simple que actúa como fuente TCP/IP. Se explicará con más detalle en el siguiente apartado. Es el único módulo simple que deberá ser programado por el alumno.

Router: Módulo simple que actúa como multiplexor de diversas fuentes TCP/IP. Dispone de un vector de puertas de entrada y un vector de puertas de salida, con las que se conecta a cada módulo simple TCPIP, y una puerta de salida y otra de entrada que se conectarán a través de sendos enlaces al módulo NodoDestino. Sus funciones son: 1) Recibir los paquetes TCP/IP procedentes de los módulos TCPIP, almacenarlos en una cola finita FIFO, y reenviarlos por el enlace de salida hacia el módulo NodoDestino. La cola ó *buffer* del Router tendrá una capacidad limitada en bytes, no en paquetes, ya que el tamaño de éstos puede variar. 2) En el otro sentido de la comunicación actúa como demultiplexor: recibe los ACK's procedentes de NodoDestino y los encamina por el enlace adecuado, en función de la información de cabecera del paquete ACK. Ver Anexo 1.

NodoDestino: Módulo compuesto que consiste en un módulo Router conectado a tantos módulos simples Sink como módulos TCPIP tenga el sistema. El Router actúa como demultiplexor de la información procedente de las fuentes TCPIP y como multiplexor de los ACK's generados en cada módulo Sink.

Sink: Módulo simple cuyo cometido es generar los ACK's para los paquetes que recibe del otro extremo, un módulo TCPIP. Debe haber un módulo TCPIP por cada módulo Sink. Ver Anexo 2.

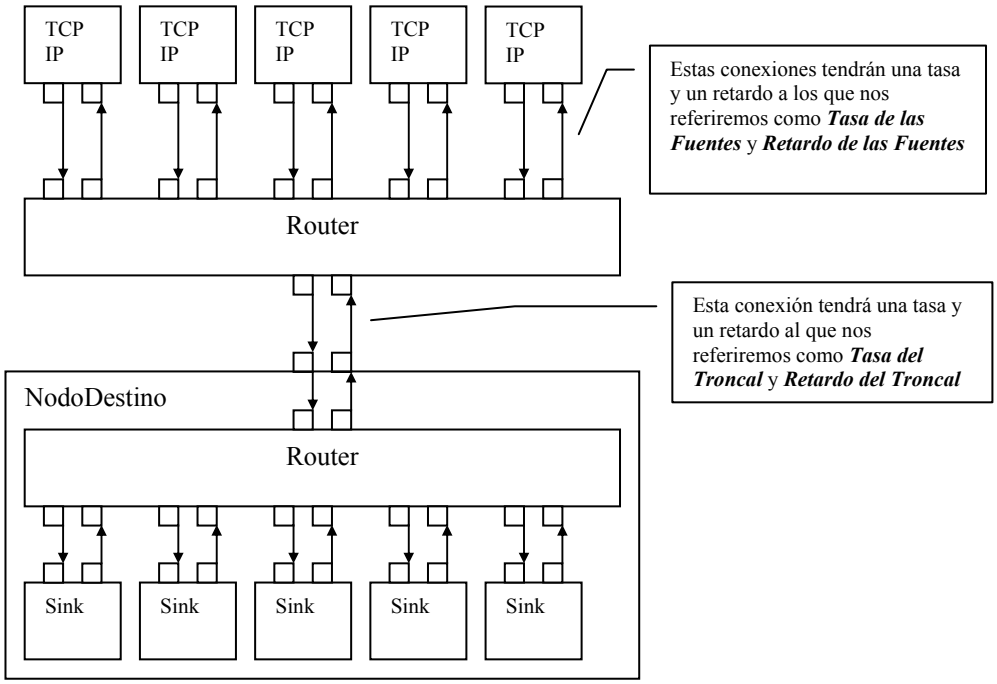

Figura 9.

3.2 Módulo TCPIP

La implementación del módulo TCPIP es el objetivo principal del presente trabajo de prácticas. En este apartado se describirán los requisitos mínimos necesarios que deberá cumplir su implementación. En el apartado 4 se dan sugerencias que pueden servir de referencia.

1. El primer requisito es que el módulo implemente los algoritmos de control descritos en el apartado 2: Envío de paquetes con el algoritmo ventana deslizante en las tres situaciones posibles: Slow Start, Congestion Avoidance y Fast Retransmit – Fast Recovery, retransmisión de paquetes por timeout y ACK's replicados, medida del RTT y cálculo del RTO.
2. No se debe implementar el control de flujo, ya que de hecho el módulo Sink, que es el módulo destino, no señaliza la ventana de congestión. Tampoco se deberán implementar otros algoritmos de TCP como el establecimiento de la conexión.
3. Deben existir, al menos los tres estados indicados en la Figura 1.
4. Los módulos TCPIP siempre tendrán datos que transmitir, por lo que no será necesario emplear funciones de generación de números aleatorios (el sistema es determinista)
5. El módulo TCPIP deberá interoperar con su módulo Sink asociado, para ello debe cumplirse lo siguiente:
    a. Los paquetes generados y los ACK's recibidos serán de la clase Packet descrita en el apartado 3.5, derivada de la clase base cMessage..
    b. Los paquetes ACK enviados por el módulo Sink son de tipo 1. (Como recordará en omnet++ se puede asignar un tipo a los mensajes con setKind(int kind), y acceder a al tipo con kind( ))
    c. El número de secuencia reconocido por el módulo Sink en cada paquete ACK, corresponde al último segmento que el módulo sink ha recibido en secuencia, no al que espera recibir. Se puede consultar el Anexo 2: Descripción del módulo simple Sink, para más información.
6. En la definición NED del módulo TCPIP deberán existir al menos los siguientes parámetros:
    a. ID: será un número entero que identificará al módulo y será igual al ID del módulo Sink con el que estará conectado. Realmente se trata del identificador de la conexión, y los paquetes generados deberán ser marcados con el ID del módulo.
    b. SMSS: Tamaño del segmento a transmitir. La longitud en bytes de los paquetes generados será la suma del tamaño en bytes del SMSS con la longitud de la cabecera IP y la cabecera TCP, de 20 bytes cada una.

En la implementación en C++ deberá configurar la longitud del paquete en bits, ya que velocidad del enlace se debe dar en bps.

7. Para homogeneizar el código generado, se deberán asignar nombres predefinidos a ciertas variables necesarias en la implementación. Los nombres son los siguientes, y están basados en la nomenclatura empleada en la bibliografía recomendada:
   - M: Medida de RTT
   - A: Estimador de la Media de RTT
   - D: Desviación Típica de RTT
   - RTO: *Retransmission Timeout*
   - SMSS: (parámetro del módulo) Tamaño del paquete.
   - cwnd: Ventana de congestión
   - ssthresh: Umbral del *SlowStart*
   - FlightSize: Número de segmentos transmitidos no reconocidos
   - ID: (parámetro del módulo) Identificador del módulo. Representa la pareja puerto TCP - dirección IP, que identifica unívocamente a un host TCP. El módulo Sink tendrá el mismo ID.
   - STATE: Estado del módulo {SS, CA, FRFR} , cuyos valores numéricos son {1, 2, 3}
   - RTT_Start_Time: Instante de tiempo en que se inició el timer
   - Duplicate_ACK: Contador de ACKs duplicados
   - SN: Número de secuencia del último segmento transmitido.
   - Last_ACK_SN: SN del último segmento reconocido

   Del mismo modo si se emplean las siguientes constantes, se les deberá asignar estos nombres:
   - G = 0,1 : Granularidad del reloj en segundos, ver 2.3
   - g = 0,125 : ver 2.3
   - h = 0,25 : ver 2.3
   - rwnd = 65535 : ventana de recepción (en bytes)
   - IPheader = 20 : bytes de cabecera del paquete IP
   - TCPheader = 20 : bytes de cabecera del paquete TCP
   - MaxRTO = 60 s: máxima duración en segundos del RTO.

Durante la implementación el alumno decidirá qué otras variables auxiliares pueden ser necesarias, y deberá describirlas claramente en la documentación y en el código.

3.3 Definición del la red en NED

La red deberá ser definida en lenguaje ned por el alumno. Como es sabido, la red debe constar de un módulo compuesto en el que se especifican todos los elementos que la componen y cómo se interconectan (módulos, conexiones, parámetros, etc). En la Figura 9 se muestra un diagrama del módulo que deberá definir y al que llamaremos *SimuladorTCP*. En dicho diagrama el número de módulos es tan sólo un ejemplo ya que la cantidad de entidades TCPIP será un valor variable.
Se proporciona la definición en NED del módulo simple Router y del Módulo compuesto NodoDestino descritos en el apartado 3.1, que deberá incorporar a su módulo *SimuladorTCP*.
El módulo *SimuladorTCP* deberá tener como parámetros, configurables desde omnetpp.ini ó en ejecución, las siguientes variables:

- Tamaño del segmento: Este parámetro servirá para configurar el parámetro equivalente en el módulo TCPIP. Puede definirlo en bytes ó bits, y deberá tener presente las unidades en la implementación en C++.
- Número de entidades TCPIP: Este parámetro determinará el número de módulos TCPIP que tendrá la simulación y también el número de módulos Sink de que constará el módulo compuesto NodoDestino.
- Retardo del enlace entre los módulos TCP y el Router: Este parámetro se empleará cuando se definan las conexiones del módulo compuesto *SimuladorTCP* entre los módulos TCPIP y el Router. El valor será el mismo para todos los módulos TCP.
- Retardo del enlace entre el Router y el NodoDestino: Este parámetro se empleará cuando se defina la conexión del módulo compuesto *SimuladorTCP* entre el Router y el NodoDestino.
- Tasa en bps del enlace entre los módulos TCP y el Router: Este parámetro se empleará cuando se definan las conexiones del módulo compuesto *SimuladorTCP* entre los módulos TCPIP y el Router. El valor será el mismo para todos los módulos TCP.

- Tasa en bps del enlace entre el Router y el NodoDestino: Este parámetro se empleará cuando se defina la conexión del módulo compuesto *SimuladorTCP* entre el Router y el NodoDestino.
- Tamaño del Buffer del Router: capacidad en bits del buffer del router.

3.4 Enlaces

Como se ha explicado en el apartado anterior, la capacidad en bps, así como el retardo de propagación de cada enlace será un parámetro del módulo compuesto (*SimuladorTCP*) y por tanto configurable en omnetpp.ini. La asignación de estas capacidades a los enlaces se llevará a cabo en la definición en NED de *SimuladorTCP*. En el apartado 4.5.7 del manual de Omnet++ ver. 2.3 puede consultar la sintaxis que deberá emplear.

3.5 Mensajes

Se ha definido una clase derivada de cMessage llamada Packet, que deberá ser empleada por el módulo TCPIP cuando genere paquetes de datos. También es la clase con la que el módulo Sink genera los ACK's por lo que deberá ser tenido en cuenta en la implementación, siendo necesario en ocasiones realizar un *downcast* a (*Packet) a los punteros a cMessage que manejan los métodos de cModule.

Para definir la clase Packet se ha empleado el método descrito en el apartado 6.2 del manual de Omnet++ ver. 2.3. Este método consiste únicamente en declarar los campos que va a contener el mensaje, indicando el tipo de dato que se va a almacenar en dicho campo (int, double…) en un fichero con extensión .msg. Omnet++ ver. 2.3 incorpora un script en perl llamado opp_msgc (*message subclassing compiler*) que genera el código C++ de la clase derivada a partir del fichero de declaración (*.msg*). Las clases derivadas proporcionan los métodos *set* y *get* para modificar y leer los campos declarados.

En nuestro caso el fichero Packet.msg contien la siguiente declaración:

```
message Packet
{
fields:
        int ID;
        int SN;
}
```

Siendo ID el campo que identifica la conexión, y que los módulos TCPIP y Sink lo igualarán a su parámetro ID cada vez que envíen un mensaje. El campo SN corresponde al número de secuencia del segmento que se envía (si lo genera módulo TCP) o que se reconoce (si lo envía el módulo Sink).

4 Sugerencias para la implementación

4.1 Activity ó handleMessage

Las ventajas de emplear una u otra solución están claramente desarrolladas en el apartado 5.4 del manual de omnet++ versión 2.3. Dadas las características del simulador a realizar cualquier opción es válida. HandleMessage aportaría la ventaja de que se podría emplear el módulo creado en simulaciones de una gran cantidad de módulos. En caso de que se decida emplear handleMessage es necesario recordar los siguientes detalles:

- Las variables del módulo deben definirse necesariamente como atributos del módulo, a no ser que su valor se reinicie en cada evento (por ejemplo iteradores de bucles).
- Con handleMessage no se puede hacer uso de las funciones receive( ) y wait( ). Los timeouts deberán implementarse como automensajes, con ScheduleAt( ).
- Cuando se hace uso de handleMessage es necesario inicializar las variables en initialize( ), donde además se deberá enviar el primer paquete.
- Con handleMesage es habitual tener como atributos del módulo punteros a mensajes que se inicializan una sola vez y que se van reutilizando, generalmente automensajes de timeout.

4.2 Ventana de congestión

La ventana de congestión cwnd puede implementarse como número de segmentos o número de bytes. Si se implementa como número de segmentos deberá tomar valores con decimales debido al algoritmo *congestion avoidance*, por tanto deberá ser de tipo double. Si elige implementar cwnd como número de segmentos, la variable ssthresh también deber definida de esa forma, debiendo ser de tipo entero.

En ocasiones puede ser necesario realizar operaciones aritméticas entre variables de tipo double y variables de tipo entero, por lo que deberá tener la precaución de realizar los *castings* adecuados cuando esto ocurra.

Relacionado con lo anterior pueden serle de utilidad los siguientes métodos:
De la librería math.h:

- *floor(double n)*, devuelve un valor double igual al mayor entero por debajo del argumento n.
- *ceil (double n),* devuelve un valor double igual al menor entero por encima de n.

De omnet++:

- *max(double x, double y)* : devuelve el mayor de los dos argumentos x ó y.
- *min(double x, double y)* : devuelve el menor de los dos argumentos x ó y.

4.3 Otras variables de utilidad

Aparte de las variables presentadas en 3.2 que son requisito en la implementación, debe considerar la necesidad de definir otras variables (atributos del módulo si se implementa con HandleMessage()) como por ejemplo:

Número de secuencia del RTO. Número entero igual al SN del paquete con el que se activó el temporizador y se inicializó la medida del RTT (RTT_start_time). Sólo cuando llegue el primer ACK que reconozca este número se parará el temporizador y se medirá el RTT. Revise la figura 4 para entender este hecho.

Medida del RTT activa o inactiva. El algoritmo se Karn obliga a no realizar la medida del RTT cuando llegan ACK's que reconocen segmentos retransmitidos. Esto puede resolverse con una variable booleana que "desactive" la medida del RTT cuando hay alguna retransmisión y no se vuelva a activar hasta que los segmentos retransmitidos estén reconocidos y se envíe un segmento nuevo.

Máximo SN transmitido hasta el momento. Relacionado con lo anterior, cuando se produce un Timeout y se vuelven a retransmitir los paquetes empezando por el siguiente al último reconocido (Last_ACK_SN +1), el valor de SN se retrasa hasta ese valor, pero el módulo debe recordar cual fue el último SN no duplicado que envió para saber cuando volver a activar la medida de RTT.

4.4 Utilidades de Omnet++

Con los conocimientos de Omnet++ adquiridos hasta el momento, puede implementarse el módulo TCPIP planteado casi en su totalidad. En este apartado se mencionan funciones potencialmente útiles para este trabajo, en su mayoría ya conocidas por el alumno. Para conocer más sobre ellas se deberá consultar el manual y el API de Omnet++.

Par (…)
scheduleAt (…)
cancelEvent (…)
simTime( )
transmissionFinishes ( ), es un método de cGate
setDisplayString ( ), este método puede ser útil para depurar el módulo, ya que permite cambiar el color del mismo en su representación gráfica durante la ejecución, pudiendo relacionar un color a cada estado.

4.5 Consejos para la depuración, verificación y validación del módulo

En este caso, al tratarse de un modelo determinista, las tareas de depuración, verificación y validación están unidas, ya que si se implementan sin errores los algoritmos descritos se habrá reproducido exactamente una fuente TCP con

transmisión constante. A partir de ahora nos referiremos a estas tres tareas simplemente con el término depuración, al ser el más habitual en la elaboración de software.

Uso de variables de depuración: Consiste en definir ciertas variables booleanas en el módulo y cuyo valor se puede ser definido por el usuario, o cambiarse en cada compilación. Estas variables servirán de condición inicial para que se ejecuten líneas de código destinadas únicamente a la depuración. Por ejemplo:
Se define la variable (flag) debug = true, en la inicialización del módulo.
A lo largo del código se insertan líneas del tipo:
if (debug) ev << "El módulo TCP "<< ID <<" ha entrado en FRFR"<<endl;
if (debug) ev << "Al módulo TCP "<< ID << " le ha vencido el Timeout"<<endl;
etc, de esta forma se pueden activar y desactivar los mensajes que interesen en función de lo que se esté depurando, sin tener que borrar líneas de código cada vez y sin que se sature la interfaz gráfica con mensajes que ya no interesan.

Presentar en pantalla y en tiempo de ejecución la evolución de determinadas variables del sistema, como por ejemplo cwnd, ssthresh, RTO … Para ello deberán definirse objetos cOutVector (uno por cada variable que se desee monitorizar) y cada vez que se actualice el valor de la variable en cuestión, se llamará al método record(…) de dichos objetos. Consulte el manual y el API para más información. El entorno gráfico puede presentar en pantalla la evolución de la gráfica generada por cada objeto cOutVector definido en el módulo.

4.6 Consejos para una mayor claridad en el código.

Cuanto más claro sea su código, más sencillo le será depurarlo y añadirle funcionalidades posteriormente. En general el desarrollo será más rápido si sigue una serie de normas básicas:

1) Dedique el tiempo suficiente a comprender bien el funcionamiento de los algoritmos y a hacerse un idea general del funcionamiento del módulo que debe implementar. Antes de empezar a teclear código es conveniente que sepa perfectamente qué es lo que va a programar, para ello escriba en papel la estructura general de su programa y para cada una de sus secciones ó funciones escriba el pseudocódigo (o diagrama de flujo) antes de comenzar a programar. También le será de mucha ayuda dibujar diagramas del tipo de las figuras 3, 6, 7, 8.

2) Desglose el código en funciones, no lo desarrolle todo en una sola función activity( ) ó handleMessage( ). Identifique qué actividades pueden ser agrupadas en una función, y asigne a dicha función un nombre que indique su utilidad, por ejemplo *RTO_Meassure( )*, para registrar el valor de RTT y calcular el valor de RTO. Si determinadas líneas de código se repiten en distintas partes del código es posible que puedan implementarse como una sola función.

3) No espere a tener muchas funcionalidades implementadas para empezar a compilar su código. Cuanto antes lo empiece a depurar, mejor. De la misma forma empiece a ejecutarlo lo antes posible. Para ello puede fijarse una serie de hitos, por ejemplo: empezar sólo con *Slow Start* y *Congestion Avoidance*, ejecutar y depurar esa versión, posteriormente añadir las retransmisiones por timeout, después *Fast Retransmit – Fast Recovery*, y así hasta completar todas las funcionalidades.

4) Comente su código, sobre todo en las partes esenciales de los algoritmos es aconsejable indicar los pasos que realiza, no necesariamente línea a línea. Comentarios del tipo: "*inicializamos variables de cálculo de RTO*" "*actualizamos cwnd",* "*desactivamos el timer*" que pueden estar asociados a más de una línea de código son suficientes. También se pueden identificar con comentarios determinados momentos en el algoritmo, por ejemplo "*el ACK es replicado*". En cualquier caso el criterio para comentar el código debe definirlo el programador, pensando en qué indicaciones le harían falta si necesitase revisar ese código mucho tiempo después de haberlo escrito.

4.7 Gestión eficiente de la memoria

Es conveniente que se eliminen los mensajes recibidos en los módulos una vez que éstos no van a emplearse más, para evitar que estos se almacenen en la memoria innecesariamente. Sin embargo esta tarea es delicada ya que puede dar lugar a errores en la ejecución no detectables en compilación. Es aconsejable que antes de proceder a la inclusión de instrucciones para la eliminación de mensajes haya implementado y depurado completamente su código. Deje esa tarea para el final.

5 Medidas a Realizar

5.1 Variables del protocolo

Deberá definir un objeto del tipo tipo cOutVector, que le permita ver la evolución de los siguientes parámetros del protocolo en tiempo de simulación y representarlos con plove posteriormente:

- Cwnd
- Ssthresh
- RTO

5.2 Parámetros de rendimiento

1. Thoughput de cada fuente:
   - Throughput en bps instantáneo de cada fuente: Deberá elegir un intervalo de medida para promediar. Será necesario que defina un objeto de tipo cOutVector, que le permita ver la evolución del throughput en tiempo de simulación.
   - Throughput medio en bps por fuente.

2. Tasa de paquetes perdidos: (IPLR: *IP Packet Loss Ratio*) Se define como

$$\frac{\text{Número de paquetes perdidos}}{\text{Número de paquetes transmitidos}}$$

3. Retardo de los paquetes: (IPTD :*IP Packet Transfer Delay*): Se define como el tiempo transcurrido para cada paquete desde el momento en que comienza su transmisión hasta el momento que finaliza su recepción.
   - IPTD medio.
   - Estimación de la función de distribución de IPTD mediante la definición de un objeto cDoubleHistogram.

5.3 Escenarios de Referencia

Estos son los cinco escenarios de referencia que se proponen para la realización de medidas. Son escenarios sencillos, en los que, a excepción del escenario 1, se modelan situaciones de congestión. Los parámetros que definen cada escenario son parámetros que deben poder variarse desde el fichero omnetpp.ini como se explica en el apartado 3.3.
El tiempo de simulación para todos los escenarios será de 45 minutos.

Escenario	***1***	***2***	***3***	***4***	***5***
Nº de entidades TCP	2	2	2	2	2
Retardo Fuentes	5 ms	5 ms	50 ms	50 ms	5 ms
Retardo Troncal	50 ms	50 ms	500 ms	500 ms	50 ms
Tasa Fuentes	100000 bps	100000 bps	100000 bps	100000 bps	100000 bps
Tasa Troncal	200000 bps	100000 bps	100000 bps	100000 bps	100000 bps
Tamaño Segmento	1024 bytes	1024 bytes	1024 bytes	1024 bytes	1024 bytes
Tamaño Buffer	4 paquetes	20 paquetes	20 paquetes	4 paquetes	4 paquetes

5.4 Resultados a Presentar en la memoria

En caso de que implemente las medidas planteadas, deberá incluir en su memoria, al menos, los siguientes resultados

5.4.1 Valores

Los valores medios en cada escenario de los siguientes los parámetros:

- Throughput medio en bps por fuente.
- Tasa de paquetes perdidos
- IPTD medio

5.4.2 Gráficas

1. Evolución de cwnd, ssthresh y RTO para los 10 primeros minutos de simulación de los escenarios 4 y 5
2. Throughput en bps instantáneo de cada fuente en los 20 últimos minutos de la simulación para los escenarios 1 y 2.
3. Función de distribución de IPTD para los escenarios 1, 2 y 5.

6 Método y Criterios de Evaluación

Los alumnos deberán entregar una memoria del trabajo y realizarán un examen del mismo que consistirá en una exposición y un turno de preguntas. Se hará pública una convocatoria con la suficiente antelación en la que se anunciarán los días y los horarios disponibles para la realización del examen. Junto con la convocatoria se dejarán listas a disposición de los alumnos para que se apunten en el turno que elijan. La estructura de la memoria y el formato del examen se detallan a continuación.

6.1 Memoria

La memoria que deberá ser entregada el día del examen, constará de los siguientes apartados:

1) Introducción: En este apartado describirá la estructura general del programa realizado, y podrá apoyarse en un diagrama de bloques o una implementación en pseudocódigo muy general. Se indicarán también las funciones de que consta, y qué variables se han definido aparte de las indicadas en la presente propuesta.
2) Código del módulo TCPIP realizado. Deberá presentarlo función a función, *eliminando los comentarios entre las líneas de código así como las líneas de código que haya insertado para su depuración*. La explicación de cada función deberá darla al inicio ó al final del código de la misma, y si desea comentar una línea de código en concreto deberá referenciarla con una nota de este tipo: **[1] [2] [3]**, etc. Consulte los Anexos 1 y 2 para ver un ejemplo de esto.
3) Código NED implementado.
4) Resultados de las medidas y gráficas obtenidas si se han implementado. Interpretación de los resultados.

6.2 Exposición

La exposición se realizará por parejas y deberán intervenir los dos integrantes del grupo la misma cantidad de tiempo aproximadamente. La exposición durará un máximo de 15 minutos y en ella los alumnos podrán emplear, si lo desean, una presentación en Powerpoint. La exposición constará de los siguientes apartados:

1) Una introducción en la que se explicará la estructura general del programa realizado, indicando las funciones de que consta, y qué variables se han definido aparte de las indicadas en la presente propuesta.
2) Se deberá describir de manera breve y muy concisa cómo se han implementado los algoritmos de ventana deslizante (con SS, CA y FRFR), retransmisiones y medida del RTT, indicando en qué función ó funciones se realizan, sin extenderse en detalles.
3) Si se han realizado medidas se comentarán qué funciones y variables se han empleado y en qué puntos del código se han tomado las muestras.
4) Si los alumnos lo consideran oportuno podrán comentar qué problema ó problemas se han encontrado en la elaboración y depuración del código y cómo los han resuelto.
5) Se mostrará el funcionamiento del programa, ejecutándolo para alguna de las configuraciones elegida por los alumnos, comentando la evolución de las gráficas y los resultados obtenidos. Adicionalmente, los profesores podrán proponer una configuración distinta para la evaluación del programa en el examen.
6) Se pasará al turno de preguntas. Éstas podrán ir dirigidas directamente a uno de los integrantes del grupo.

6.3 Criterios de evaluación

En este apartado se enumeran los aspectos que se tendrán en cuenta en la evaluación del trabajo.

Se valorará especialmente la exposición de cada uno de los integrantes del grupo, en cuanto a claridad, rigor, brevedad y el conocimiento que demuestren de su propio código tanto en la exposición como en las respuestas a las cuestiones que se les planteen.

Se valorará que el simulador compile y funcione correctamente con las distintas configuraciones que se planteen.

Se valorará que se haya implementado y que funcione correctamente la toma de medidas. También se valorará que los alumnos sepan interpretar los resultados.

La memoria se evaluará de manera general en cuanto a su claridad y rigor. Si se han implementado medidas, están implementadas correctamente, y funcionan, se valorarán los resultados de las mismas presentados en la memoria y su interpretación.

Se valorará la corrección en la implementación de los distintos apartados:

- Algoritmos que se describen en esta propuesta.
- Implementación en NED del simulador.
- Medidas.

Se valorará la claridad del código. En general cuantas menos líneas de código se empleen (sin quitar funcionalidades) la implementación será mejor y más clara.

Se valorará que el módulo TCPIP no almacene paquetes innecesariamente (gestión óptima de memoria).

7 Bibliografía:

[1] TCP/IP Illustrated, Volume I
[2] RFC 2581. TCP Congestion Control.
[3] RFC 2988. Computing TCP's Retransmission Timer

Anexo 1: Descripción del módulo simple Router

Clase Router. Implementación de un multiplexor con buffer de longitud finita. Los paquetes recibidos por cualquiera de los enlaces de entrada se transmiten por el enlace de salida. Si está ocupado, se almacenan en el buffer. Si el tamaño (en bytes) del paquete es mayor que el espacio disponible en el buffer, se descarta el paquete. Consta de las funciones handleMessage y finish, y se han implementado dos funciones adicionales SendToLink y StoreInBuffer.

Declaración de la clase

```
#include <omnetpp.h>
#include <stdio.h>
#include <string.h>
#include <cqueue.h>
#include "constantes.h"
#include "Packet_m.h"

class Router : public cSimpleModule
{
private:
      int Bits_In_Buffer, BufferSize;
      cQueue TxBuffer;
protected:
      virtual void initialize();
public:
      Module_Class_Members(Router, cSimpleModule,0);
      virtual void handleMessage(cMessage *msg);
      virtual void finish();

      virtual void SendToLink(Packet *pckt);
      virtual void StoreInBuffer(Packet *pckt);
};

Define_Module(Router);
```

Función initialize

```
void Router::initialize()
{
TxBuffer.clear();
Bits_In_Buffer = 0;
BufferSize = (int)par("Buffersize");
}
```

Función handleMessage

```
void Router::handleMessage(cMessage *msg)
{
if (msg->isSelfMessage())[1]
{
      delete msg;
      if (!TxBuffer.empty())
      {
            Packet *pckt = (Packet *)TxBuffer.pop();
            Bits_In_Buffer = Bits_In_Buffer - pckt->length();
            SendToLink(pckt);
      }

}
```

```
else  [2]
      {
      Packet *packet = (Packet *)msg;
      if (msg->arrivedOn(gate("Link_Rx")->id()))[3]
            {
            send(packet,gate("outvec",packet->getID()-1)->id());[4]
            }
      else
      {
      if (gate("Link_Tx")->isBusy()||(!TxBuffer.empty()))[5]
            {
            StoreInBuffer(packet);
            }
      else  SendToLink(packet);
      }
}
}
```

[1] Final de transmisión propia. Continúo enviando mientras queden paquetes en el buffer
[2] No es un automensaje, es un paquete
[3] mensaje procedente del enlace troncal
[4] Se comprueba su ID y se envía por la puerta de índice correspondiente
[5] Se comprueba si "Link" esta transmitiendo

Función StoreInBuffer

Almacena paquetes en el buffer, mientras quede suficiente espacio libre

```
void Router::StoreInBuffer(Packet *pckt)
{
if ((Bits_In_Buffer + pckt->length())>BufferSize)
      {
      delete pckt;
      }
else
      {
      Bits_In_Buffer = Bits_In_Buffer + pckt->length();
      TxBuffer.insert(pckt);
      }
}
```

Función SendToLink

Envía mensajes al enlace troncal y planifica un automensaje de final de transmisión

```
void Router::SendToLink(Packet *pckt)
{
send(pckt,"Link_Tx");
cMessage *EOT = new cMessage("EOT",EndOfTransmission);

if (gate("Link_Tx")->isBusy()) [1]
      scheduleAt(gate("Link_Tx")->transmissionFinishes(), EOT);
else
      scheduleAt(simTime(), EOT);
}
```

[1] Esta comprobacion es necesaria porque si el retardo es 0, planificaría el mensaje en el pasado

Función finish

```
void Router::finish()
{
}
```

Anexo 2: Descripción del módulo simple Sink

Clase Sink. Implementa un receptor que reconoce los paquetes del extremo remoto, mediante el algoritmo Fast Restransmit/ Fast Recovery. Mantiene y actualiza una lista de paquetes perdidos. Consta de las funciones handleMessage y finish. Se han implementado dos funciones adicionales, RegisterLostSegments y SendACK.

Declaración de la clase

```
#include <omnetpp.h>
#include <stdio.h>
#include <string.h>
#include <cllist.h>
#include <list.h>
#include "constantes.h"
#include "Packet_m.h"

class Sink : public cSimpleModule
{
private:
      int Expected_SN, Highest_SN, HighestInSequence;
      list<int> LostSegments;
      cMessage* TimerPointer;
      bool NoLost, TimerOn;
      simtime_t ACK_Timeout;

protected:
      virtual void initialize();
      virtual void RegisterLostSegments(int init_Seq,int end_Seq);
      virtual void Send_ACK(int seq);
      int ID;

public:
      Module_Class_Members(Sink, cSimpleModule,0);
      virtual void handleMessage(cMessage *msg);
      virtual void finish();
};

Define_Module(Sink);
```

Función initialize

```
void Sink::initialize()
{
LostSegments.clear();
Expected_SN = 1;
Highest_SN = HighestInSequence = 0;
NoLost = true;
TimerOn = false;
TimerPointer = new cMessage("Timeout",Timeout);
ACK_Timeout = strToSimtime("500ms");
ID = (int)par("ID");

}
```

Función handleMessage

```
void Sink::handleMessage(cMessage *msg)
{
if (msg->isSelfMessage())[1]
{

      TimerPointer = msg;
      TimerOn = false;
      Send_ACK(HighestInSequence);
}
else [2]
 {
Packet *packet = (Packet *)msg;
int PacketSN = packet->getSN();
      if (!NoLost) [3]
      {
      if (PacketSN == Expected_SN)[4]
      {

            LostSegments.pop_front();

            if (LostSegments.empty())[5]
            {
                  NoLost = true;
                  HighestInSequence = Highest_SN;
                  Expected_SN = HighestInSequence + 1;
            }
            else[6]
            {

                  Expected_SN = (int) LostSegments.front();
                  HighestInSequence = Expected_SN - 1;
            }
      }

      else if (PacketSN == Highest_SN + 1)[7]
            Highest_SN += 1;


      else if (PacketSN > Highest_SN + 1)[8]
            RegisterLostSegments(Highest_SN, PacketSN );


      else if ((PacketSN < Highest_SN + 1)&&(PacketSN > Expected_SN))[9]
      {[10]
                  if (PacketSN == (int)(*iter))
                        {
                        LostSegments.erase(iter);
                        iter = LostSegments.end();
                        }

      }

      Send_ACK(HighestInSequence);[11]

      }
      else[12]
      {
            if (PacketSN == Expected_SN)
            {
                  HighestInSequence = Highest_SN = PacketSN;
```

```
                  Expected_SN = HighestInSequence + 1;
                  if (TimerOn)[13]
                  {
                        TimerPointer = cancelEvent(TimerPointer);
                        TimerOn = false;
                        Send_ACK(HighestInSequence);
                  }
                  else[14]
                  {
                  scheduleAt(simTime()+ACK_Timeout,TimerPointer);
                  TimerOn = true;
                  }
            }
            else if (PacketSN > Expected_SN)[15]
            {
                  NoLost=false;[16]

                  if (TimerOn)
                  {
                        TimerPointer = cancelEvent(TimerPointer);
                        TimerOn = false;
                  }
                  Send_ACK(HighestInSequence);

                  RegisterLostSegments(HighestInSequence,PacketSN);[17]
            }
      }
delete packet;
}
}
```

[1] Ha vencido el Timeout, por tanto se envía un ACK
[2] No es un automensaje, es un paquete
[3] Ya hay algún segmento perdido
[4] Elimina el primero de la lista de segmentos perdidos
[5] Lista vacía, volvemos a la situación normal
[6] La cola no queda vacía, el SN esperado es el que queda primero
[7] Llega el siguiente a Highest SN, se actualiza
[8] Nuevos segmentos perdidos (red muy congestionada)
[9] Se recibe un segmento mayor que el esperado pero menor que el máximo
[10] Se comprueba si es algún segmento perdido
[11] Siempre se envía un ACK
[12] No hay registrado ningún segmento perdido
[13] Desactiva TO y se envía un ACK
[14] Se activa TO
[15] Se recibe un SN mayor al esperado
[16] Se desactiva el TO y se envía un ACK
[17] Se registran los segmentos perdidos

Función RegisterLostSegments

Introduce los números de secuencia en la lista de segmentos perdidos

```
void Sink::RegisterLostSegments(int init_Seq,int end_Seq)
{
      for (int i = init_Seq + 1; i < end_Seq; i++)
      {
            LostSegments.push_back(i);
      }
      Highest_SN = end_Seq;
}
```

Función Send_ACK

Envía un ACK a su fuente origen

```
void Sink::Send_ACK(int seq)
{
      Packet *ACK_packet = new Packet("ACK",ACK);
      ACK_packet->setLength((IPheader+TCPheader)*8);
      ACK_packet->setID(ID);
      ACK_packet->setSN(seq);
      send(ACK_packet,"Out");
}
```

Función finish

```
void Sink::finish()
{
}
```