UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Desarrollo de alternativas al protocolo TCP para redes inalámbricas

Proyecto Final de Carrera

Ingeniería Informática

Autora: María Fernández Hernández

Director: Carlos Tavares Calafate

25 de septiembre de 2012

Índice de contenidos

1. Introducción

1.1. Visión general

El protocolo TCP es uno de los protocolos fundamentales de Internet. Desde que se creara, a principios de los años 70, se ha convertido, junto al protocolo IP (TCP/IP), en el estándar mundial de comunicaciones dentro de las redes informáticas. Da soporte a muchas de las aplicaciones más populares en Internet (navegadores, intercambio de ficheros, clientes FTP, etc.) y protocolos de aplicación HTTP, SMTP, SSH, FTP, etc. Y sobre él se han construido grandes estructuras de las que dependen económicamente grandes empresas y organismos de todo el mundo.

Las principales características de este protocolo son:

- Transferencia básica de datos, fiable, ordenada y segura, gracias al uso de reconocimientos (ACKs), números de secuencia y detección de errores.
- Comunicación orientada a conexión.
- Control del flujo para evitar la congestión en la red.
- Multiplexación gracias al uso de puertos.

A lo largo de los años, se ha investigado mucho y se han propuesto muchas mejoras para el protocolo TCP en redes tradicionales cableadas. Sin embargo, los crecientes avances en el mundo de las redes han permitido la creación de redes de alta velocidad inalámbricas de diferente índole (redes satélite, plataformas inalámbricas generales para la interconexión de equipos, redes móviles, redes ad hoc, etc.).

Con el incremento de la popularidad de las tecnologías inalámbricas, el protocolo TCP ha tenido que extender su capacidad para abarcar el aspecto inalámbrico, tanto en redes heterogéneas (con parte cableada y parte inalámbrica) como en redes completamente inalámbricas.

El problema es que el diseño y los mecanismos de control de congestión del protocolo TCP para redes cableadas, no proporcionan el mismo rendimiento para las redes inalámbricas, debido a la diferente naturaleza de éstas. La principal causa de esta reducción en las prestaciones es el mecanismo frente a pérdidas de paquetes del protocolo TCP: en las redes cableadas, la principal causa de pérdidas de paquetes es la congestión en la red, por lo que TCP se diseñó para que relacionase las pérdidas de paquetes con la presencia de congestión en la red, reduciendo la cantidad de paquetes inyectados en la red, para tratar de evitar esta congestión. Sin embargo, en las redes inalámbricas, gran cantidad de las pérdidas de paquetes producidas se deben a interferencias de la señal u otros factores momentáneos. En estos casos, al reducir la cantidad de paquetes que se transmiten, lo único que se consigue es una reducción del rendimiento. Por eso, el diseño de TCP para redes cableadas lo hace menos adecuado para redes inalámbricas.

Para evitar esta reducción en las prestaciones, se han llevado a cabo numerosas investigaciones, y se han presentado diversas mejoras y nuevas propuestas al protocolo TCP, con el objetivo de mantener las mismas ventajas que ofrece éste, pero atendiendo a las especificidades de las redes inalámbricas.

1.2. Objetivos del trabajo

Muchas de las investigaciones que se han llevado a cabo para mejorar el comportamiento del protocolo TCP en redes inalámbricas, consisten en modificaciones a dicho protocolo, mayormente centradas en cambios a su algoritmo de control de congestión.

El objetivo de este trabajo es la implementación de dos de estas propuestas, TCP Vegas y TCP Westwood, para la suite IP INET Framework, del simulador de redes OMNeT++, y su posterior análisis.

Dichas propuestas están basadas en la implementación base TCP Reno, y proponen un algoritmo del control de la congestión proactivo, frente al control de la congestión reactivo de Reno. Lo que plantean es un intento de ajustar la tasa de datos alrededor de un valor óptimo, de acuerdo a información, o feedback, del estado de la red en un momento determinado (y no sólo debido al feedback que proporciona el receptor, como en esquemas reactivos). De esta manera, el emisor puede reaccionar de manera inteligente a las condiciones de la red o las causas de la pérdida de paquetes, debidas tanto a congestión como a errores aleatorios.

La metodología que se seguirá es la siguiente: partiendo de la implementación de TCP Reno existente para el INET Framework, se implementarán los protocolos TCP Vegas y TCP Westwood, siguiendo las indicaciones de los autores de los protocolos en las respectivas propuestas.

Cabe señalar que el objetivo principal del trabajo es conseguir que las implementaciones de los protocolos sean lo más fieles posible a las propuestas presentadas por los autores, por lo que las pruebas y análisis posteriores estarán encaminados a verificar el comportamiento esperado de dichas implementaciones en base a sus propuestas.

1.3. Estructura del documento

La primera sección del presente documento introduce al lector en el tema tratado, la motivación, y los objetivos del trabajo que se ha realizado.

A continuación, en la sección dos, se presenta el protocolo TCP: conceptos básicos y sus características principales, los mecanismos y algoritmos que lo conforman, su funcionamiento, y un repaso por las principales implementaciones existentes. Además, se introduce y se detalla la problemática del protocolo en las redes inalámbricas, y se resume una serie de trabajos previos que han propuesto soluciones al problema, destacando los protocolos TCP Vegas y TCP Westwood, por ser los protocolos que se han implementado.

En la sección tres se presenta el simulador OMNeT++ brevemente, destacando la suite IP INET Framework. De este paquete se resumen las características principales de su arquitectura, y se describe, con algo más de detalle, el módulo TCP, para que el lector tenga una visión general de la estructura en la que se deben integrar los protocolos implementados.

La sección cuatro recoge todas las características de las implementaciones realizadas, su diseño y los detalles de implementación concretos de cada una de ellas.

En la quinta sección se presentan los resultados obtenidos de las simulaciones a las que se

han sometido las implementaciones para verificar su correcto funcionamiento.

Finalmente, la sexta sección recoge las conclusiones que se han obtenido a lo largo del desarrollo del trabajo.

2. El protocolo TCP

2.1. Concepto

El Protocolo de Control de Transmisión, o TCP, contemplado en el RFC 793 [1], es un protocolo de comunicación estándar, orientado a conexión y fiable del nivel de transporte que, en práctica, está incluido en toda implementación TCP/IP que no es usada exclusivamente para routing.

Su objetivo principal es proporcionar un servicio de conexión fiable y seguro entre pares de procesos (ver Figura 2.1). En la pila de protocolos TCP/IP, TCP es la capa intermedia entre el protocolo de internet (IP) y la aplicación. Dado que la capa IP aporta un servicio de datagramas no fiable (sin confirmación), es el protocolo TCP el que debe garantizar que la comunicación entre dos sistemas se efectúe libre de errores, sin pérdidas y con seguridad.

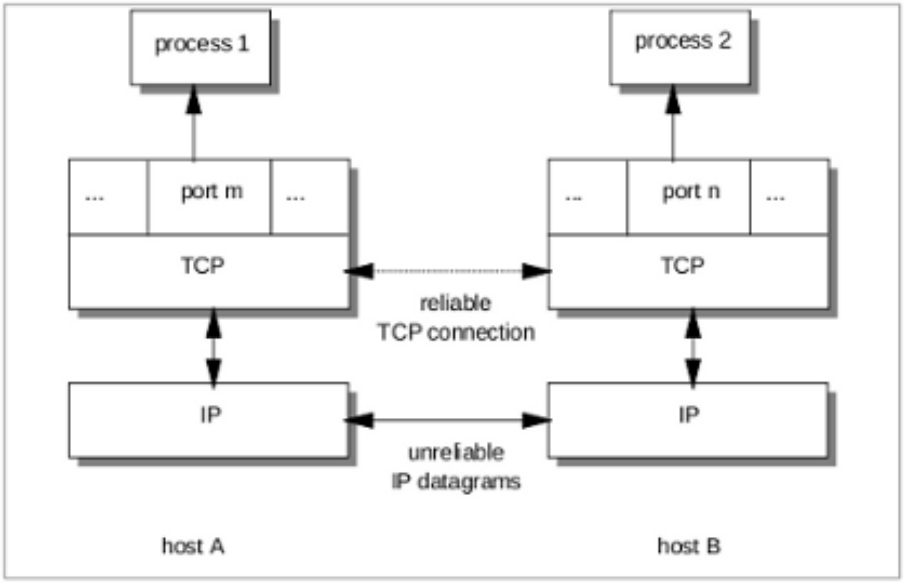

Figura 2.1: Conexión TCP entre procesos
Figura de [1]

Para proporcionar este servicio, TCP facilita los siguientes mecanismos a las aplicaciones que hacen uso de él:

- Transferencia básica de datos: desde el punto de vista de la aplicación, TCP transmite un flujo continuo de bytes a través del sistema de internet. La aplicación no tiene que preocuparse de dividir los datos en bloques básicos o datagramas. TCP realiza esta tarea agrupando los bytes en segmentos TCP, que son entregados al módulo IP para ser transmitidos al destino. Además, en general, TCP decide cómo segmentar los datos, cuándo bloquear y cuándo enviar los datos, según su propia conveniencia.

  Algunas veces, las aplicaciones necesitan estar seguras de que todos los datos que habían entregado al módulo TCP han sido transmitidos. Para este propósito se define una función

'push' (“enviar inmediatamente”). Para asegurar que los datos entregados al módulo de TCP son realmente transmitidos, la aplicación emisora debe indicarlo mediante dicha función. Un 'push' en un cierto instante causa que los módulos de TCP envíen y entreguen inmediatamente a la aplicación receptora los datos almacenados hasta ese instante. La función de cierre normal de conexión también envía inmediatamente los datos al destino.

- Fiabilidad: el módulo de TCP debe poder recuperar los datos que se corrompan, pierdan, dupliquen o se entreguen desordenados. Esto se consigue asignando un número de secuencia a cada byte transmitido, y exigiendo un ACK (del inglés 'acknowledgment', reconocimiento) del módulo de TCP receptor. Si no se recibe un ACK dentro de un cierto plazo de tiempo prefijado, los datos se retransmiten. En el receptor, se utilizan los números de secuencia para ordenar correctamente los segmentos que puedan haber llegado desordenados y para eliminar los duplicados. Además, como los datos se transmiten en bloques (segmentos TCP), sólo el número de secuencia del primer byte de datos en el segmento es enviado al receptor.

  La corrupción de datos se trata añadiendo un campo de suma de control ('checksum') a cada segmento transmitido, comprobándose en el receptor y descartando los segmentos dañados.

- Control de flujo: TCP proporciona al receptor un medio para controlar la cantidad de datos enviados por el emisor. Cuando el receptor envía un ACK al emisor, también indica el número de bytes que puede recibir (más allá del último segmento recibido con éxito) de acuerdo a su buffer interno. Se envía, con cada ACK, en forma de máximo número de secuencia que se puede recibir sin problemas. Este mecanismo es conocido como “mecanismo de ventana deslizante”, descrito en la sección 2.2.

- Multiplexing: para permitir que muchos procesos dentro de un único 'host' utilicen simultáneamente las posibilidades de comunicación de TCP, el módulo de TCP proporciona una serie de direcciones o puertos dentro de cada 'host'. Concatenadas con las direcciones de red y de 'host' de la capa de comunicación de internet forman lo que se denomina un 'socket' (o “dirección de conector”).

- Conexiones lógicas: la fiabilidad y los mecanismos de control de flujo descritos más arriba exigen que los módulos TCP inicialicen y mantengan una información de estado para cada flujo de datos. La combinación de esta información, incluyendo los 'sockets', los números de secuencia y los tamaños de las ventanas, se denomina una conexión lógica. Cada conexión queda especificada de forma única por un par de 'sockets' que corresponden a sus dos extremos, procesos emisor y receptor.

  Como las conexiones tienen que establecerse entre 'hosts' no fiables y sobre un sistema de comunicación IP no fiable, se utiliza un mecanismo que usa números de secuencia basados en tiempos de reloj para evitar una inicialización errónea de las conexiones.

- Full duplex: TCP proporciona servicio para el flujo de datos concurrente en ambas direcciones.

- Prioridad y seguridad: las aplicaciones de TCP pueden indicar el nivel de seguridad y prioridad de su comunicación. Se emplean valores por defecto cuando estas características no se necesitan.

2.2. El mecanismo de ventana deslizante

El mecanismo básico de la ventana que utiliza TCP se puede observar en la Figura 2.2. El emisor agrupa los paquetes que van a ser transmitidos, de acuerdo a las siguientes reglas:

- El emisor puede enviar todos los paquetes dentro de los límites de la ventana, antes de recibir un ACK, debiendo iniciar un contador de tiempo de espera ('timeout') para cada uno de ellos.
- El receptor debe enviar un reconocimiento (ACK) por cada paquete recibido, indicando el número de secuencia del último paquete correctamente recibido.
- El emisor desliza la ventana con cada ACK recibido.

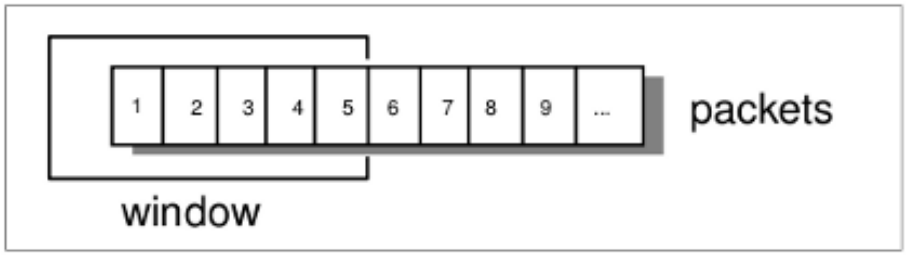

Figura 2.2: Ventana deslizante
Figura de [2]

Este mecanismo asegura:

- Una transmisión fiable de los datos.
- Mejor uso del ancho de banda de la red (mejor rendimiento).
- Control del flujo, ya que el receptor puede retrasar el envío del ACK de respuesta a un paquete, conociendo la capacidad de sus buffers y el tamaño de ventana de la comunicación.

Como se ha mencionado, el mecanismo descrito es muy básico. El protocolo TCP hace uso de él, con las siguientes diferencias:

- Puesto que TCP proporciona una conexión de flujo de datos, a cada byte en el flujo de datos se le asigna un número de secuencia. TCP divide este flujo de datos continuo en segmentos TCP para enviarlos. El principio de la ventana es utilizado a nivel de byte, es decir, los segmentos enviados y los ACKs recibidos contienen números de secuencia de bytes, y el tamaño de la ventana está expresado, igualmente, en número de bytes en lugar de en número de paquetes.

- El tamaño de la ventana lo decide el receptor en el establecimiento de la conexión, y puede variarse durante la transferencia de datos. Cada mensaje conteniendo un ACK incluirá el tamaño de ventana que el receptor puede aceptar en un instante determinado de tiempo.

Una vez explicado el principio de la ventana aplicado a TCP, el flujo de datos del emisor puede percibirse como muestra la Figura 2.3 en la siguiente página, donde:

A: bytes transmitidos y cuyo reconocimiento (ACK) ha llegado correctamente.
B: bytes que han sido enviados pero cuyo reconocimiento todavía no ha llegado.
C: bytes que se pueden enviar sin necesidad de esperar a un nuevo reconocimiento.
D: bytes que no pueden ser enviados todavía.

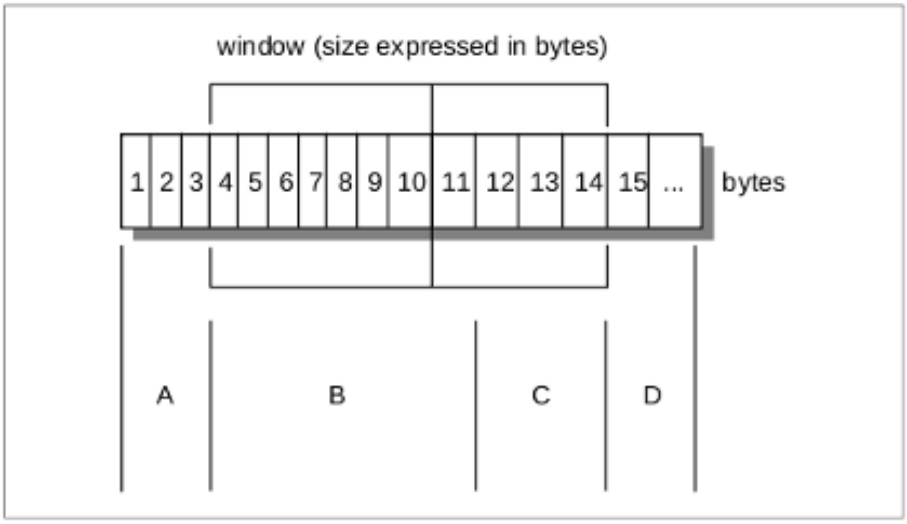

Figura 2.3: El principio de la ventana aplicado a TCP
Figura de [2]

2.3. Formato de los segmentos TCP

La Figura 2.4 muestra el formato de segmento TCP.

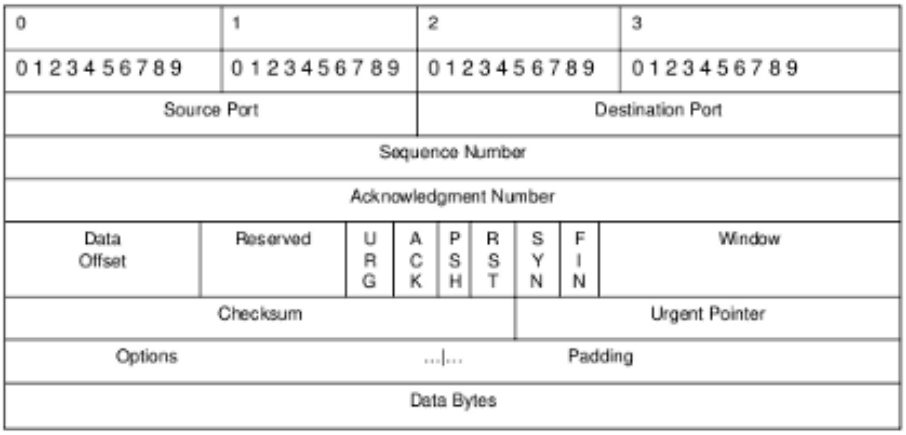

Figura 2.4: Formato de los segmentos TCP
Figura de [2]

Donde:

- Source Port (puerto de origen): número del puerto de origen, de 16 bits, usado por el receptor para responder.

- Destination Port (puerto de destino): número del puerto de destino, de 16 bits.

- Sequence number (número de secuencia): número de secuencia del primer byte de datos de este segmento, de 32 bits. Si el bit de control SYN está activo, es el número de secuencia inicial (ISN: 'initial sequence number'), y el primer byte de datos es ISN+1.
- Acknowledgment Number (número de reconocimiento): de 32 bits. Si el bit de control ACK está activo, este campo contiene el valor del siguiente número de secuencia que el receptor espera recibir.

- Data Offset: de 4 bits. Número de palabras de 32 bits en la cabecera TCP. Indica dónde comienzan los datos.

- Reserved (reservado): 6 bits reservados para uso futuro; debe valer 0.

- Bits de control: 6 bits (de izquierda a derecha):
  - URG: indica que el campo 'Urgent Pointer' (puntero urgente) es significativo en este segmento.
  - ACK: indica que el campo 'Acknowledgment Number' (número de reconocimiento) es significativo en este segmento.
  - PSH: Función 'push' (“entregar datos inmediatamente”).
  - RST: reiniciar la conexión.
  - SYN: sincronizar los números de secuencia.
  - FIN: últimos datos del emisor.

- Window (ventana): de 16 bits. Usado en segmentos ACK. Especifica el número de bytes de datos, empezando con el primero indicado en el campo 'Acknowledgment Number', que el receptor (el emisor de este segmento) está dispuesto a aceptar.

- Checksum (suma de control): de 16 bits. Es el complemento a uno de 16 bits de la suma de los complementos a uno de todas las palabras de 16 bits en la pseudo-cabecera, la cabecera TCP y los datos TCP. En el cálculo de la suma de control, el propio campo suma de control se considera formado por ceros.

La pseudo-cabecera es la misma que usa UDP para calcular la suma de control. Es una pseudo-cabecera IP, sólo usada para el cálculo de la suma de control, con el formato mostrado en la Figura 2.5.

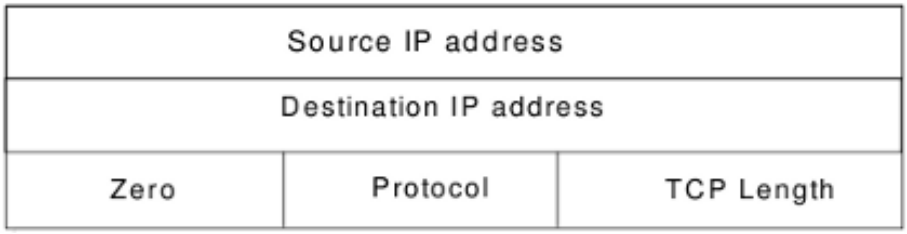

Figura 2.5: Pseudo-cabecera IP
Figura de [2]

- Urgent Pointer (puntero urgente): de 16 bits. Apunta al número de secuencia del octeto al que seguirán los datos urgentes. Sólo es significativo cuando el bit de control URG está activo.

- Options (opciones): los campos de opciones pueden ocupar cierto espacio al final de la cabecera de TCP, pero siempre de una longitud múltiplo de 8 bits.

  En el caso de datagramas IP, existen dos posibilidades para el formato de una opción:
  - Un byte único con el número de opción.
  - Una opción de longitud variable, con el formato mostrado en la Figura 2.6.

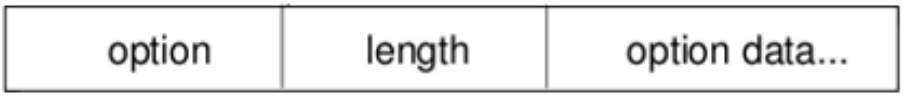

Figura 2.6: Formato de opción de datagrama IP, opción de longitud variable.
Figura de [2]

Las opciones definidas se muestran en la Tabla 2.1 y se describen a continuación.

Kind	Length	Meaning
0	-	End of option list
1	-	No operation
2	4	Maximum segment size
3	3	Window scale
4	2	Sack-permitted
5	X	Sack
8	10	Time stamps

Tabla 2.1: Opciones de datagrama IP
Tabla de [2]

- Maximum segment size option (opción de máximo tamaño de segmento): esta opción es usada únicamente durante el establecimiento de la conexión (el bit de control SYN activo) y es enviada para indicar el tamaño máximo de segmento que puede recibir el módulo de TCP que envía el segmento. Si la opción no es utilizada, se permite cualquier tamaño de segmento.
  La Figura 2.7 muestra el formato de esta opción.

Figura 2.7: Formato de Maximum segment size option
Figura de [2]

- Window scale option (opción de escala de ventana): esta opción no es obligatoria. Ambos módulos TCP deben enviar la opción de escala de ventana en sus segmentos SYN para habilitar el escalado de ventana en sus direcciones. La escala de ventana expande la definición de la ventana TCP a 32 bits. Define el tamaño de ventana de 32 bits usando un factor de escalado en el segmento SYN sobre el tamaño estándar de 16 bits. El receptor reconstruye el tamaño de ventana de 32 bits utilizando el tamaño de ventana de 16 bits y el factor de escalado.
  Esta opción se determina en la negociación en tres pasos inicial y no hay manera de cambiarla después de que la conexión ha sido establecida.
  La Figura 2.8 muestra el formato de esta opción.

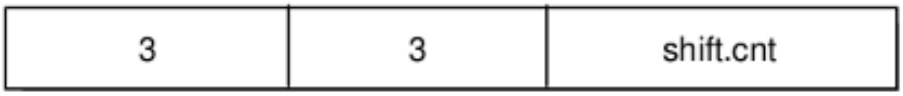

Figura 2.8: Formato de Window scale option
Figura de [2]

- SACK-permitted option (opción de SACK permitido): esta opción es activada cuando acuses de recibo selectivos ('selective acknowledgments') son usados en la conexión TCP.
  La Figura 2.9 muestra el formato de esta opción.

Figura 2.9: Formato de SACK-permitted option
Figura de [2]

- SACK option (opción SACK): la opción 'Selective Acknowledgment' SACK, “reconocimientos selectivos”, permite al receptor informar al emisor acerca de todos los segmentos que se han recibido satisfactoriamente. De esta manera, el emisor enviará únicamente los segmentos perdidos. Si el número de segmentos que se han perdido desde el último SACK fuera demasiado grande, la opción SACK sería también demasiado grande. Para evitarlo, el número de bloques que pueden ser reportados por dicha opción está limitado a cuatro. Para reducir esto, la opción SACK debe ser usada con los datos recibidos más recientes.

  La figura 2.10, en la siguiente página, muestra el formato de esta opción.

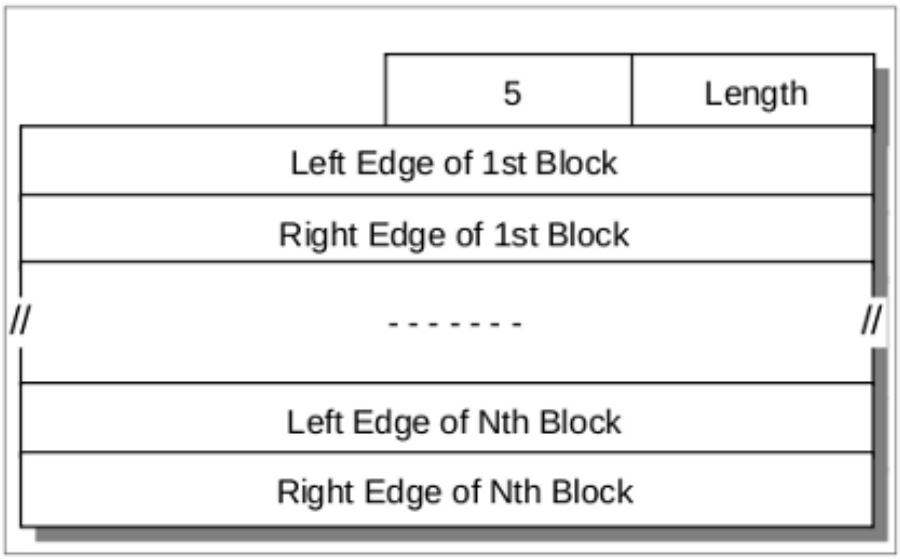

Figura 2.10: Formato de SACK option
Figura de [2]

- Timestamps options: en estas opciones se envía un 'timestamp', que representa el valor actual de tiempo del reloj del módulo TCP que envía la opción. El valor 'Timestamp Echo Value' sólo puede ser usado si el bit de control ACK está activo.
  La figura 2.11 muestra el formato de esta opción.

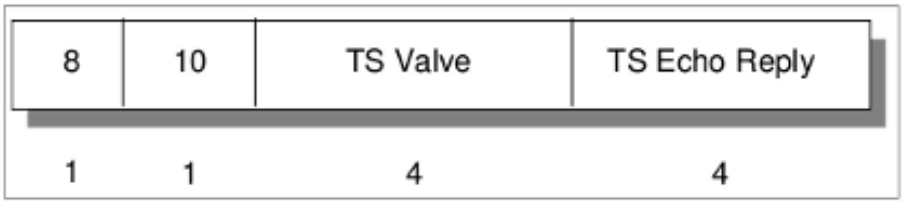

Figura 2.11: Formato de Timestamps option
Figura de [2]

- Relleno: está compuesto de ceros. Se utiliza para asegurar que la cabecera TCP finaliza, y que los datos comienzan en una posición múltiplo de 32 bits.

2.4. Funcionamiento básico de TCP

Las operaciones del protocolo TCP se pueden dividir en tres fases: establecimiento de la conexión, transferencia de datos y cierre de la conexión. Dichas conexiones son gestionadas por un sistema operativo, a través de una interfaz de programación que representa el extremo de la comunicación, el 'socket'. Durante la conexión TCP, cada extremo local progresa de acuerdo con una serie de estados durante su tiempo de vida [1].

- LISTEN (servidor): representa la espera de una solicitud de conexión proveniente de cualquier TCP y puerto remotos.

- SYN-SENT (cliente): representa la espera de una solicitud de conexión concordante tras haber enviado previamente una solicitud de conexión.

- SYN-RECEIVED (servidor): representa la espera del reconocimiento ('acknowledgment') confirmando la solicitud de conexión tras haber recibido y enviado una solicitud de conexión.

- ESTABLISHED (ambos cliente y servidor): representa una conexión abierta, los datos recibidos pueden ser entregados al usuario. Es el estado normal para la fase de transferencia de una conexión.

- FIN-WAIT-1 (ambos cliente y servidor): representa la espera de una solicitud de finalización de la conexión proveniente del TCP remoto, o del reconocimiento ('acknowledgment') de la solicitud de finalización previamente enviada.

- FIN-WAIT-2 (ambos cliente y servidor): representa la espera de una solicitud de finalización del TCP remoto.

- CLOSE-WAIT (ambos cliente y servidor): representa la espera de una solicitud de finalización de la conexión proveniente del usuario local.

- CLOSING (ambos cliente y servidor): representa la espera del paquete, proveniente del TCP remoto, con el reconocimiento de la solicitud de finalización.

- LAST-ACK (ambos cliente y servidor): representa la espera del reconocimiento ('acknowledgment') de la solicitud de finalización de la conexión previamente enviada al TCP remoto (lo que incluye el haber enviado el reconocimiento de la solicitud remota de finalización de la conexión).

- TIME-WAIT (cliente o servidor): representa la espera durante suficiente tiempo para asegurar que el TCP remoto recibió el reconocimiento ('acknowledgment') de su solicitud de finalización de la conexión.

- CLOSED (ambos cliente y servidor): representa un estado sin conexión en absoluto.

Los extremos de la conexión progresan de un estado a otro en respuesta a los siguientes eventos: llamadas de usuario, OPEN (abrir), SEND (enviar), RECEIVE (recibir), CLOSE (cerrar), ABORT (interrumpir) y STATUS (mostrar el estado); la llegada de segmentos, particularmente aquellos que contengan alguno de los bits de control SYN, ACK, RST y FIN activos, y la expiración de plazos de tiempo ('timeouts').

En la Figura 2.12, en la siguiente página, se puede observar un diagrama de estados simplificado que ilustra lo explicado.

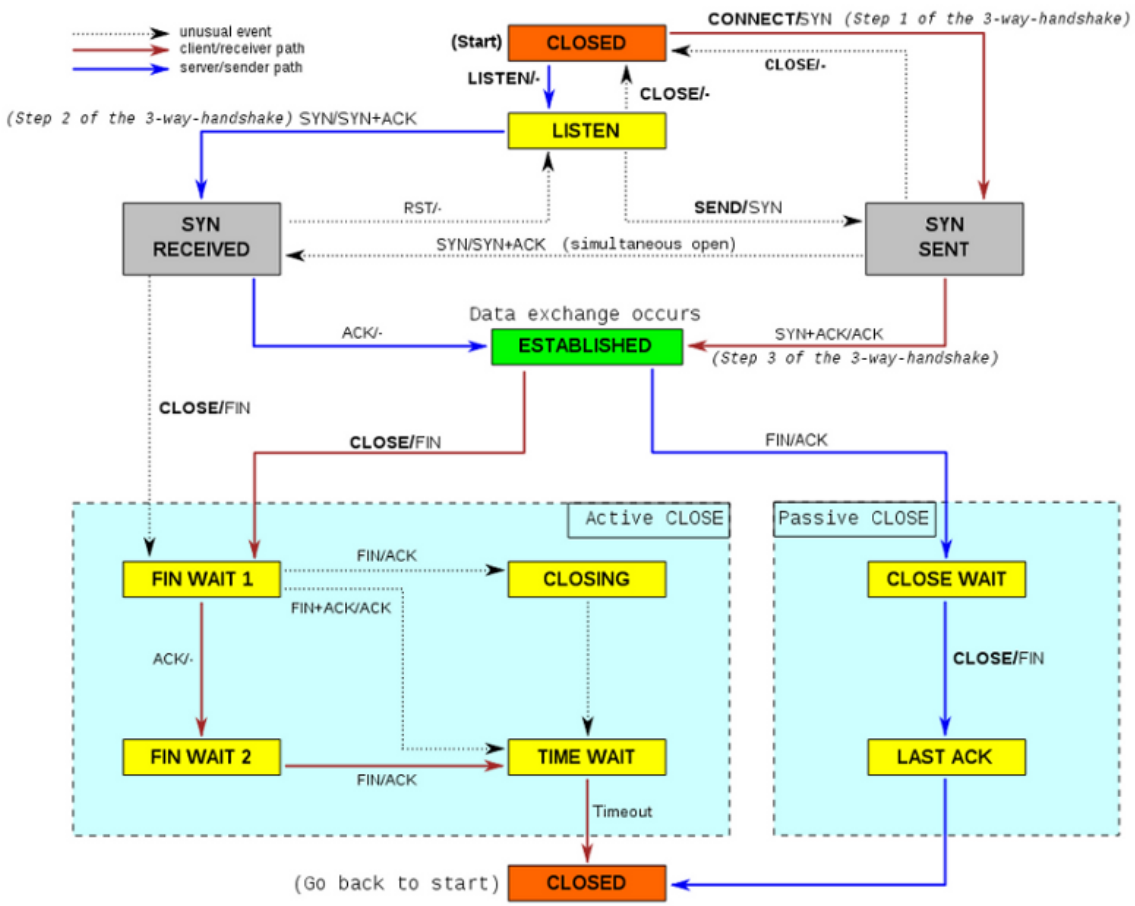

Figura 2.12: Diagrama de estados TCP simplificado
Figura de [3]

2.4.1. Establecimiento de la conexión

Para establecer la conexión, TCP utiliza el procedimiento conocido como negociación en tres pasos ('three-way handshake') (Observar Figura 2.13, en la siguiente página).

Antes de que el cliente intente conectar con el servidor, éste debe primero abrir un socket en un puerto TCP y quedar a la escucha de nuevas conexiones. Una vez que esta apertura pasiva ('passive open') queda establecida, el cliente puede iniciar una apertura activa ('active open').

Los tres pasos en los que se establece la conexión se definen a continuación:

1. SYN: la apertura activa la lleva a cabo el cliente enviando un segmento SYN al servidor. Es el cliente el que establece el número de secuencia del segmento a un valor aleatorio, x.

2. SYN-ACK: como respuesta, el servidor contesta con un segmento SYN-ACK. El número de reconocimiento ('acknowledgment') se establece a uno más que el número de secuencia recibido (x + 1), y el número de secuencia que el servidor elige para el paquete es otro número aleatorio, y.

3. ACK: finalmente, el cliente envía un segmento ACK de vuelta al servidor. El número de secuencia se establece al valor del reconocimiento recibido, i.e. (x + 1), y el número de reconocimiento a uno más que el número de secuencia recibido, i.e. (y + 1).

En este instante, tanto el cliente como el servidor han recibido un reconocimiento de la conexión, esta queda establecida y la fase de transferencia de datos puede comenzar [3].

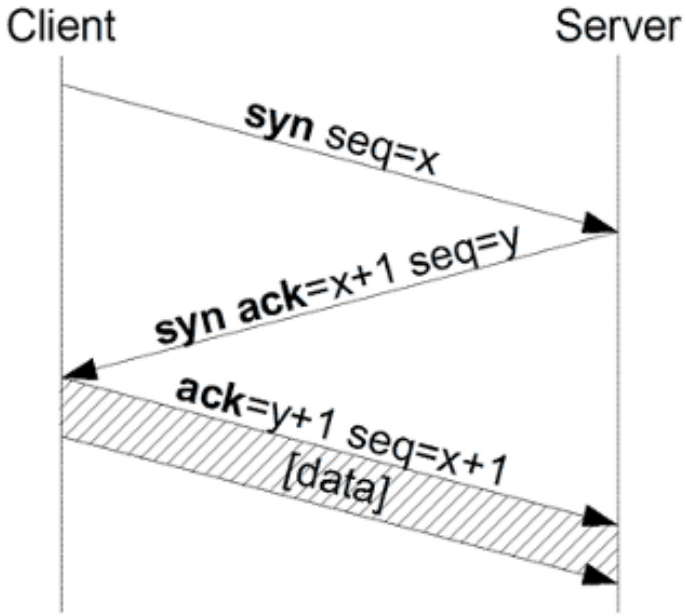

Figura 2.13: Three-way handshake (negociación en tres pasos)
Figura de [3]

2.4.2. Cierre de la conexión

Para terminar la conexión, TCP utiliza la negociación en cuatro pasos ('four-way handshake'), (observar Figura 2.14), procedimiento en el que cada lado de la conexión termina de manera independiente. Cuando uno de los extremos de la conexión quiere terminar su “mitad” de conexión, transmite un paquete FIN, al que el otro extremo responde con un reconocimiento, ACK. Por tanto, una desconexión típica requiere un par de segmentos FIN y ACK desde cada lado de la conexión. Después de que los dos intercambios FIN/ACK concluyan, el lado que envió el primer FIN espera cierto periodo de tiempo ('timeout') antes de cerrar la conexión, durante el que el puerto local no está disponible para nuevas conexiones; de esta manera se previene la confusión que puede surgir debida al envío de paquetes retrasados en conexiones posteriores.

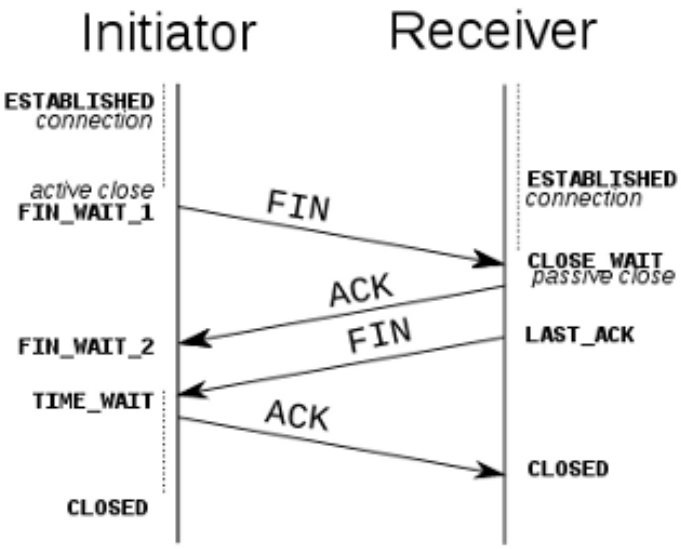

Figura 2.14: Four-way handshake (negociación en cuatro pasos)
Figura de [3]

Además, una conexión puede estar “medio abierta”, situación que se da cuando uno de los lados ha terminado, mientras que el otro no. El lado de la conexión que ha terminado no puede enviar ningún dato más, pero el otro sí. El lado que ya ha terminado debe continuar leyendo datos hasta que el otro lado termine también.

También es posible terminar la conexión mediante una negociación en tres pasos, cuando el 'host' A envía un segmento FIN, el 'host' B responde con un segmento FIN & ACK (combinando dos pasos en uno) y finalmente el 'host' A contesta con un segmento ACK [4].

2.5. Control del flujo

TCP utiliza un protocolo de control del flujo extremo-a-extremo ('end-to-end') para evitar que el emisor envíe datos demasiado rápido para la capacidad del receptor de recibirlos y procesarlos de manera fiable. Tener un mecanismo de control del flujo es esencial en escenarios, tan comunes en la actualidad, en los que dispositivos con diferentes velocidades de conexión se comunican. Por ejemplo, si un PC manda datos a un smartphone que es más lento procesando los datos recibidos, el smartphone debe regular el flujo de datos para no saturarse. [5]

Otra característica del protocolo de control de flujo de TCP es el uso de la ventana deslizante, concepto descrito en el apartado 2.2. En cada segmento TCP, el receptor especifica en el campo de “ventana del receptor” ('receive window') la cantidad adicional de datos (en bytes) que puede almacenar en su buffer en ese momento determinado de la conexión. El 'host' emisor puede mandar solamente hasta esa cantidad de datos antes de recibir un reconocimiento ('acknowledgment') y actualización de la ventana del 'host' receptor.

A continuación se describen los conceptos de 'persist timer' y 'silly window', relacionados con el control del flujo.

2.5.1. Persist timer

Cuando el receptor anuncia un tamaño de ventana con valor 0, el emisor para de enviar datos e inicia el 'persist timer' (temporizador de persistencia). El 'persist timer' es utilizado para proteger la conexión TCP de una situación de 'deadlock' (bloqueo mortal), que puede surgir si la próxima actualización de tamaño de ventana del receptor se pierde, y el emisor no puede enviar más datos hasta recibir una nueva actualización. Cuando el 'persist timer' expira, el emisor envía un pequeño paquete ('probe') para que el receptor responda enviando otro reconocimiento ('acknowledgment') que contenga el nuevo tamaño de ventana.

2.5.2. Síndrome de 'Silly Window'

Si el receptor está procesando datos entrantes en pequeños incrementos, podría, repetidamente, anunciar una ventana de receptor pequeña. Esta situación es conocida como síndrome de 'silly window' (SWS por sus siglas en inglés Silly Window Syndrom) (RFC 813, [6]), ya que es ineficiente enviar pocos bytes de datos en un segmento TCP, dada la sobrecarga relativamente grande de la cabecera TCP. Para evitar esta situación, los extremos TCP emplean un control del flujo lógico, conocido como algoritmo de Nagle, definido en el RFC 896 [7], consistente en retener por poco tiempo la transmisión de datos TCP cuando se advierte la situación descrita, a la espera de tener más bytes de datos que enviar.

2.6 Control de la congestión

Uno de los aspectos más importante del protocolo TCP es el control de la congestión. Dicho protocolo utiliza varios mecanismos para alcanzar un alto rendimiento y evitar un colapso debido a la congestión.

Ésta es una de las mayores diferencias entre los protocolos TCP y UDP [2]. Los algoritmos de control de la congestión previenen al emisor de saturar la capacidad de la red, ya que permiten adaptar el ratio del emisor a ésta, evitando así situaciones potenciales de congestión.

El control de la congestión TCP está definido en el RFC 2581 [8], en el que se contemplan sus cuatro algoritmos básicos: 'slow start', 'congestion avoidance', 'fast retransmit' y 'fast recovery'. Estos algoritmos fueron definidos inicialmente en [9] y [10], y su uso con TCP está estandarizado en [11]. Se describen, detalladamente, en las siguientes secciones.

2.6.1. Slow Start

Las implementaciones antiguas de TCP empezaban una conexión con el emisor inyectando múltiples segmentos en la red, hasta alcanzar el valor del tamaño de ventana anunciado por el receptor. Con este mecanismo es fácil que aparezcan problemas cuando entre ambos 'hosts' de la

conexión existen routers y links más lentos. Algunos de estos dispositivos intermedios pueden no ser capaces de manejar el flujo de segmentos que se está transmitiendo y, como consecuencia, algunos paquetes pueden comenzar a ser descartados, resultando en retransmisiones y degradación del rendimiento [2].

'Slow start' es el algoritmo que trata de evitar esta situación. Se basa en la idea de que el ratio al que nuevos paquetes deberían ser inyectados en la red es el ratio al cual los reconocimientos (ACKs) son devueltos por el otro extremo de la conexión. Para su propósito, 'slow start' añade otra ventana al emisor TCP: la ventana de congestión, *cwnd* ('congestion window'). Cuando una nueva conexión de establece, la ventana de congestión debe ser inicializada a un valor menor o igual a 2 segmentos (2*SMSS, 'Sender Maximum Segment Size', el tamaño del segmento más grande que puede transmitir el emisor, sin incluir la cabecera TCP/IP y las opciones) [8]. Normalmente suele ser inicializada a 1 segmento (tamaño de segmento anunciado por el otro extremo o el valor por defecto, típicamente 536 o 512) [12].

Adicionalmente, se puede inicializar la ventana de congestión entre 2 y 4 segmentos, con un límite superior de 4K, tal y como se explica en el RFC 3390 [13].

Una vez establecida la conexión, cada vez que se recibe un ACK (reconocimiento), la ventana de congestión se incrementa en un segmento. El emisor puede transmitir el menor valor entre la ventana de congestión (control del flujo impuesto por el emisor) y la ventana anunciada por el receptor ('advertised window'). La primera se basa en la percepción de la congestión de la red por parte del emisor y la segunda se relaciona con la cantidad de espacio disponible en el buffer del receptor para esta conexión.

En la Figura 2.15, en la página siguiente, se puede observar el algoritmo 'slow start' en acción (con una ventana de congestión inicial de 1 segmento): el emisor empieza enviando un segmento y esperando su ACK. Cuando lo recibe, la ventana de congestión es incrementada de uno a dos, y dos segmentos pueden ser enviados. Cuando estos dos segmentos son reconocidos (se reciben sus correspondientes segmentos ACK), la ventana de congestión es incrementada a cuatro. De esta manera, se observa un crecimiento exponencial, aunque no es exactamente exponencial, ya que el receptor podría retrasar el envío de ACKs, enviando un ACK por cada dos segmentos que recibe.
Siguiendo esta dinámica, en algún momento, la capacidad de la red IP se alcanzará, y algún router intermedio comenzará a descartar paquetes, indicando al emisor que la ventana de congestión ha alcanzado un valor demasiado elevado.

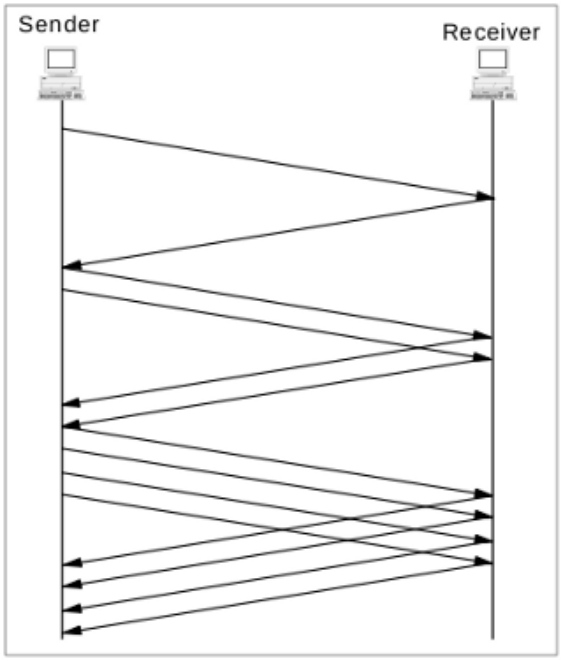

Figura 2.15: 'Slow start' en acción
Figura de [2]

2.6.2. Congestion Avoidance

'Congestion avoidance' es el otro algoritmo que el emisor TCP utiliza para controlar la cantidad de datos que están siendo inyectados en la red.

La suposición del algoritmo de que una pérdida de paquete es causada por daño es muy pequeña (muy inferior al 1%) [12]. De esta manera, la pérdida de paquete indica que hay congestión en algún punto de la conexión entre emisor y receptor.

Hay dos señales que indican una pérdida de paquete:

- Ocurre un 'timeout'.
- Se reciben ACK's duplicados.

'Congestion avoidance' y 'slow start' son algoritmos independientes con diferentes objetivos, pero cuando aparece congestión, TCP debe reducir el ratio de paquetes transmitidos a la red e invocar 'slow start' para normalizar la situación de nuevo. En práctica, ambos son implementados juntos.

Para su correcta implementación, requieren mantener dos variables para cada conexión:

- La ventana de congestión, *cwnd*, ya mencionada en el apartado anterior.
- El umbral de 'slow start', *ssthresh* ('slow start threshold').

El funcionamiento del algoritmo combinado es el siguiente:

1. En la inicialización de una conexión dada, se establecen *cwnd* a un segmento (2SMM como máximo [8]) y *ssthresh* a 65535 bytes.

2. La rutina de envío TCP nunca manda más bytes del mínimo valor entre *cwnd* y la ventana anunciada por el receptor.

3. Cuando ocurre la congestión (indicada por un 'timeout' o la recepción de ACK's duplicados), se guarda en sshthresh la mitad del tamaño de la ventana actual (mínimo valor entre *cwnd* y la ventana anunciada por el receptor). Adicionalmente, si la congestión es indicada por un 'timeout', se establece *cwnd* a un segmento (i.e., 'slow start').

4. Cuando nuevos datos son reconocidos (ACK) por el otro extremo de la conexión, se incrementa *cwnd*, y la manera de incrementar dicho valor depende de si TCP está en fase 'slow start' o 'congestion avoidance': si *cwnd* es menos o igual que *ssthresh*, TCP está en 'slow start'; en otro caso está en congestion avoidance.

   'Slow start' continúa hasta que *cwnd* está a la mitad de como estaba cuando la congestión ocurrió (ya que en el paso 2 se había guardado, en *ssthresh*, la mitad de la ventana que causó el problema), y entonces se entra en la fase de 'congestion avoidance'. Hay que recordar que, como se ha explicado en el apartado 2.6.1, 'slow start' ha iniciado *cwnd* a un segmento y la ha ido incrementando en un segmento cada vez que un ACK ha sido recibido (crecimiento exponencial de *cwnd*).

   Durante la fase de 'congestion avoidance', cada vez que se recibe un ACK, *cwnd* es incrementada en segsize*segsize/*cwnd* (donde segsize es el tamaño de segmento y *cwnd* está expresada en bytes). Esto supone un crecimiento lineal de *cwnd*, comparado con el crecimiento exponencial de 'slow start'. El incremento de *cwnd* debe ser, como mucho, un segmento cada 'round-trip time' (tiempo de ida y vuelta), aunque se reciban más ACK's durante dicho 'round-trip time', mientras que 'slow start' incrementa *cwnd* de acuerdo al número de ACK's recibidos.

En la Figura 2.16 se puede observar el comportamiento de 'slow start' y 'congestion avoidance' en acción.

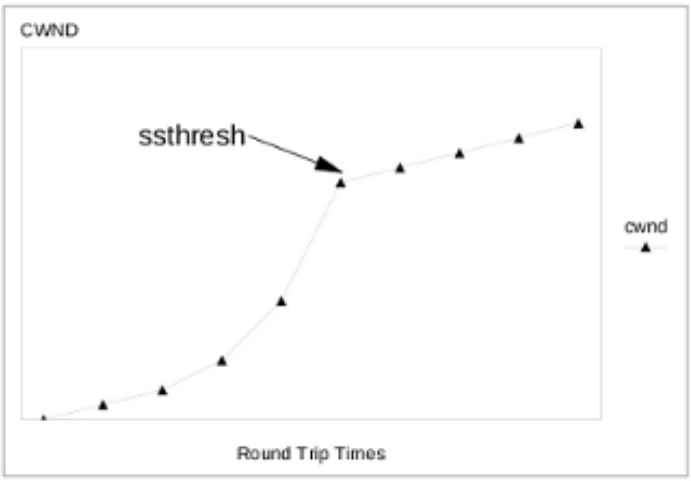

Figura 2.16: Comportamiento de 'slow start' y 'congestion avoidance' en acción
Figura de [2]

2.6.3. Fast Retransmit

El objetivo de 'fast retransmit' es evitar que TCP espere a un 'timeout' para reenviar segmentos perdidos [2].

Está descrito en las modificaciones al algoritmo de congestion avoidance propuestas en el RFC 2001 [12].

Para comprenderlo, es importante tener en cuenta que TCP puede generar reconocimientos duplicados inmediatos (ACKs duplicados) cuando es recibido un segmento fuera de orden. Este ACK duplicado no debe ser retrasado, siendo su objetivo hacer saber al otro extremo de la conexión que se ha recibido un segmento fuera de orden, y cuál es el número de secuencia que se espera.

Como TCP desconoce si un ACK duplicado ha sido causado por un segmento que se ha perdido o simplemente por reordenación de segmentos, espera a recibir un pequeño número de ACKs duplicados, asumiendo que si se trata simplemente de reordenación de segmentos, tan sólo habrá uno o dos ACKs duplicados antes de que el segmento que falta sea procesado, que generará un nuevo ACK. Así, el hecho de que tres ACKs duplicados (4 ACK's idénticos sin que llegue otro paquete, [8]) sean recibidos, se considera un indicativo de que un segmento se ha perdido. Dada esta situación, TCP retransmite lo que parece ser el segmento perdido, sin esperar a que el temporizador de retransmisión expire [12].

En la Figura 2.17 se puede observar un resumen del algoritmo en acción.

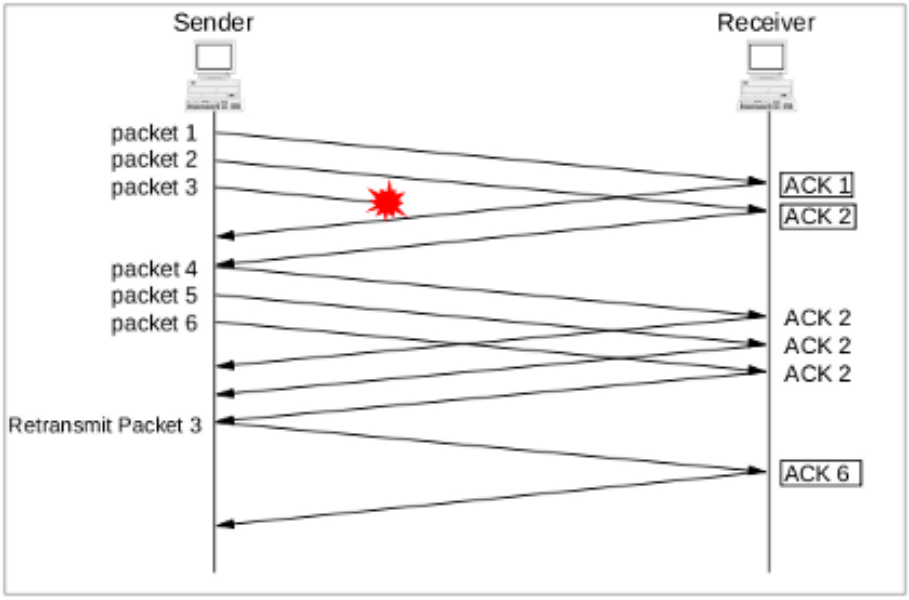

Figura 2.17: 'Fast retransmit' en acción
Figura de [2]

2.6.4. Fast Recovery

El algoritmo de 'fast recovery' consiste en el hecho de que después de que 'fast retransmit' envíe lo que parece ser el segmento perdido, se realiza 'congestion avoidance', en vez de 'slow start'. Se trata de una mejora que permite un alto rendimiento bajo una congestión moderada, especialmente con ventanas de gran tamaño [12].

La razón para no realizar 'slow start' en este caso es que la recepción de tres ACK's duplicados no sólo indica que un segmento se ha perdido. Como el receptor sólo puede generar un ACK duplicado cuando recibe otro segmento, dicho segmento ha abandonado la red y está en el buffer del receptor. Es decir, todavía hay un flujo de datos entre los dos extremos de la conexión, y TCP no debe reducir dicho flujo de manera abrupta entrando en fase de 'slow start'.

Normalmente, los algoritmos de 'fast retransmit' y 'fast recovery' son implementados juntos, de la siguiente manera:

1. Cuando se recibe el tercer ACK duplicado, *ssthresh* se establece a la mitad de la ventana de congestión actual, *cwnd*, pero a menos de dos segmentos. Se retransmite el segmento perdido. Se establece *cwnd* a *ssthresh* más 3 veces el tamaño de segmento. De esta manera se aumenta la ventana de congestión en el número de segmentos que han alcanzado el otro extremo de la conexión y han abandonado la red (3 segmentos).

2. Cada vez que otro ACK duplicado llega, se incrementa *cwnd* en el tamaño de un segmento. De nuevo, esto incrementa la ventana de congestión por el segmento adicional que ha abandonado la red. Se transmite un paquete, si el nuevo valor de *cwnd* lo permite.

3. Cuando llega el siguiente ACK que reconoce nuevos datos, se establece *cwnd* a *ssthresh* (el valor que se estableció en el paso 1). Este ACK debe ser el reconocimiento de la retransmisión del paso 1, un 'round-trip time' (tiempo de ida y vuelta) después de la retransmisión. Adicionalmente, este ACK debe reconocer todos los segmentos intermedios enviados entre el paquete perdido y el primer ACK duplicado. Este paso es considerado fase de 'congestion avoidance', ya que TCP se establece a la mitad del ratio en el que estaba cuando se detectó la pérdida del paquete.

2.7 Implementaciones TCP

Como se ha podido observar en los apartados anteriores, TCP es un protocolo complejo. A lo largo de los años se han propuesto e implementado muchas mejoras, pero las operaciones más básicas no han cambiado significativamente desde la primera especificación, el RFC 675 [14], en 1974, y la especificación v4 RFC 793 [1], publicada en Septiembre de 1981.

El RFC 1122 [11] clarifica diversos requerimientos para la implementación del protocolo. El RFC 2581 [8], que es uno de los RFCs más importantes relacionados con TCP de los últimos años, describe algoritmos actualizados para evitar la congestión. Y en 2001, el RFC 3168 [15] se escribió para describir el concepto de notificaciones explícitas de congestión ('explicit congestion notification', ECN) [3].

Se puede ver que el elemento principal del protocolo TCP es el algoritmo del control de la congestión, que supone la base primaria para el control de la congestión en Internet [16].

Las primeras implementaciones del protocolo surgieron en los años 1980 por parte de los sistemas BSD, que derivaban de los UNIX. Las implementaciones consideradas básicas son TCP Tahoe y TCP Reno, identificadas así por ser las implementaciones TCP de los sistemas 4.3BSD-Tahoe y 4.3BSD-Reno.

Un listado del nivel de implementación de las primeras versiones es [16]:

- 1983: 4.2BSD: primera versión ampliamente disponible de TCP/IP.
- 1986: 4.3BSD: mejor rendimiento de TCP.
- 1988: 4.3BSD-Tahoe: 'slow-start', 'congestion avoidance', 'fast retransmit'.
- 1990: 4.3BSD-Reno: 'fast recovery', predicción de cabeceras TCP, cabecera SLIP, compresión.
- 1993: 4.4BSD: multidifisuión, modificaciones para canales de alto rendimiento (RFC 1323 [17]).
- 1994: 4.4BSD-Lite: utiliza la implementación base TCP Reno.

Especialmente importante es la implementación de TCP de 4.4BSD-Lite, que usa la implementación TCP Reno, que es derivada de TCP Tahoe. Es la que se toma como referencia por haber sido la base de muchos sistemas actuales [16]. Por ello, se describen en detalle las características básicas de TCP Tahoe y Reno en el apartado 2.7.1.

Así pues, muchas de las implementaciones actuales se basan en mejoras sobre las dos implementaciones base indicadas. Entre ellas destacan New-Reno (descrita en el RFC 6582 [18], que es una versión de Reno, con un sistema mejorado de 'fast retransmit' , Reno Plus, popular en algunos sistemas Solaris, y SACK ('Selective Acknowledgements') que, como su nombre indica, utiliza reconocimientos selectivos, consistentes en que el receptor lista, explícitamente, qué paquetes, mensajes o segmentos han sido reconocidos (tanto negativamente como positivamente) [19]. Los reconocimientos selectivos positivos son una opción en TCP (RFC 2018 [20]).

Existen otras muchas más versiones, como TCP FACK [21], TCP Vegas [22], TCP Westwood [23], CUBIC TCP, TCP Peach, ATCP, TCP Hybla, etc.

2.7.1. TCP Tahoe y Reno

Como se ha explicado en las sección 2.6, TCP utiliza una estrategia de control de la congestión multi-faceta. En dicha sección se han descrito en detalle los cuatro algoritmos básicos: 'slow-start', 'congestion avoidance', 'fast retransmit' y 'fast recovery', por lo que no se describirán de nuevo en esta sección.

Es importante mencionar, sin embargo, que tanto TCP Tahoe, como TCP Reno, comparten el hecho de que hacen uso los tres primeros algoritmos ('slow start', 'congestion avoidance' y 'fast retransmit'). El comportamiento de ambos difiere en cómo detectan y reaccionan ante la pérdida de paquetes:

- Tahoe: tres ACKs duplicados son tratados de la misma manera que un 'timeout'. Tahoe ejecuta 'fast retransmit', reduce la ventana de congestion a 1MSS ('maximum segment size', tamaño máximo de segmento), y reinicia al estado de 'slow start' [24].

- Reno: cuando recibe tres ACKs duplicados, reduce a la mitad la ventana de congestión, ejecuta 'fast retransmit', y entra en fase de 'fast recovery'. Si un ACK genera un 'timeout', se entra en fase de 'slow start' como en Tahoe [24]. En el estado de 'fast recovery' Reno retransmite el paquete perdido que ha sido indicado por los tres ACKs duplicados, y espera a recibir un reconocimiento de la ventana de transmisión entera antes de volver a la fase de 'congestion avoidance'. Si no recibe dicho reconocimiento, se producirá un 'timeout', volviendo, como se ha dicho, a la fase de 'slow start'.

Ambos algoritmos, ante un evento de 'timeout' reducen la ventana de congestión a 1 MSS ('maximum segment size', tamaño máximo de segmento).

2.7.2. Implementaciones TCP sobre redes inalámbricas

A efectos de controlar la congestión, TCP plantea regular el tráfico generado en ambos extremos de la conexión. Para ello, como hemos observado en las implementaciones Tahoe y Reno, se utilizan como “sensores” del estado de la red la pérdida de uno o más segmentos y la recepción de ACKs duplicados, y se infiere a partir de ello la congestión en la red, disminuyendo la cantidad de segmentos inyectados en la misma.

Sin embargo, las características de las redes inalámbricas difieren de las caracterśiticas de las redes cableadas. Las principales diferencias que afectan al rendimiento de TCP en redes inalámbricas se listan a continuación [25][26]:

- Ancho de banda limitado: en redes cableadas es común encontrar desde 100Mbps hasta anchos de banda del orden de Gbps. Sin embargo, en redes inalámbricas el ancho de banda es mucho menor (el estándar IEEE 802.11b for Wireless LAN ofrece anchos de banda de hasta 11Mbps).

- RTT grandes: en general, los RTT (tiempo de ida y vuelta) en las redes inalámbricas son mayores que en las redes cableadas. Esto provoca que la ventana de congestión se incremente a un ritmo menor en los enlaces inalámbricos, limitando el throughout de TCP.

- Pérdidas aleatorias: mientras que en TCP las pérdidas se deben en más de un 99% a la congestión, y en menos de un 1% a problemas en el enlace, en las redes inalámbricas gran cantidad de las pérdidas de paquetes producidas se deben a interferencias de la señal u otros factores momentáneos.

- Usuarios móviles: Un usuario puede pasar de una red inalámbrica a otra, produciéndose una pérdida de conexión en el proceso.

- Flujos cortos: TCP es un protocolo orientado a conexión. Todo ese proceso de conexión lleva un tiempo que, en redes con alto retardo, degrada mucho el rendimiento, siendo este su efecto más perceptible en envíos de corta duración.

- Consumo de energía: Al ser la red inalámbrica un modo de conexión sin cables, es la opción más usada en dispositivos móviles que dependen de la vida de sus baterías, por lo que esta característica gana en importancia.

De estas diferencias entre las redes cableadas y las redes inalámbricas se desprende la necesidad de un nuevo protocolo o modificación del protocolo TCP para adecuarlo a las nuevas características de la red.

Se han propuesto diversas soluciones al problema, tanto centradas particularmente en enlaces inalámbricos, como enfocadas a redes mixtas (parte cableada, parte inalámbrica), particulares para redes móviles, y mejoras a las implementaciones TCP ya existentes. Algunas de ellas son las siguientes [27]:

- Soluciones a nivel de enlace: este tipo de soluciones se basan en la modificación del nivel de enlace del protocolo de red para incluir principalmente dos técnicas: corrección de errores, usando técnicas FEC de corrección de errores, y retransmisiones en respuesta a mensajes ARQ.

  Los protocolos del nivel de enlace para los sistemas digitales celulares en EEUU (CDMA [28] y TDMA [29]), utilizan técnicas ARQ.

  El protocolo AIRMAIL [30] combina técnicas FEC de corrección de errores y retransmisiones para recuperación de pérdidas de paquetes.

  El protocolo Snoop [31] introduce un módulo agente que monitoriza cada paquete, manteniendo una caché de los segmentos TCP que no han sido reconocidos por el receptor, realizando una retransmisión cuando se detecta alguna pérdida.

  El protocolo Tulip [32] se basa en la retransmisión automática de los paquetes al detectar una pérdida, introduciendo una aceleración MAC que le permite acelerar la recepción de los ACK sin renegociar el acceso al medio.

  Retrasar los ACKs duplicados es otra técnica, basada en retransmisiones a nivel de enlace y en el retraso de los envíos de los ACK duplicados para intentar no interferir en esas transmisiones.

  La mayor ventaja de emplear un protocolo de nivel de enlace para la recuperación de pérdidas es que se acopla naturalmente en la estructura de capas de los protocolos de red, opera de manera independiente, y no mantiene ningún estado explícito por conexión [27].

- Conexión dividida: se divide la conexión en dos conexiones distintas, una desde emisor a la estación base y la otra desde la estación base al receptor, usando TCP sobre la parte cableada y otro protocolo sobre la parte inalámbrica.

  Indirect TCP [33] usa TCP también sobre la parte inalámbrica.

  Mobile TCP crea una estructura de tres capas que está diseñada para trabajar en entornos donde es frecuente la pérdida de conexión. Los elementos de la capa superior se encargan del enrutamiento, control de flujo, reconexión con los dispositivos móviles, etc.

Wireless-TCP es un protocolo diseñado para trabajar sobre WWAN, que especifica un nuevo protocolo (WTCP) que cambia el control de flujo en base a ventanas por un control de flujo basado en la tasa de transferencia.

- Modificaciones de TCP: se modifica el protocolo TCP añadiendo elementos como los ACK selectivos o las notificaciones de pérdidas explícitas. En este apartado se destacan, y se describirán en profundidad en los apartados 2.7.2.1 y 2.7.2.2. Los protocolos TCP Vegas y TCP Westwood, ya que son los protocolos que se han implementado para este Proyecto.

  En TCP SACK [20] se informa al emisor de cuales son exactamente los paquetes que han llegado al receptor, en lugar de enviar sólo el ACK para el último paquete que se ha recibido correctamente cuando no hay pérdida de ningún paquete intermedio, permitiendo así que no se hagan retransmisiones innecesarias ni que se produzca una degradación del rendimiento por culpa de los temporizadores de ACK.

  TCP FACK [21] complementa al anterior, manteniendo información sobre el último paquete que ha llegado con éxito y la cantidad de información transmitida para estimar la ventana de congestión en base a esos parámetros.

  SMART [34] es una estrategia de retransmisión que combina el uso de GBN (Go-Back-N) y los ACK selectivos. Incluye en los ACK el número del paquete que se está reconociendo y el del último paquete que se recibió, lo que permite al receptor suponer que la diferencia entre los dos son los paquetes que se han perdido.

  Explicit Congestion Notification (ECN) [15] se basa en el uso del bit CE (Congestion Experienced) de la cabecera IP. Cuando el receptor recibe paquetes a los que los routers intermedios les han activado el bit CE (tienen los buffers muy llenos), responde con ACK con el bit CE activado. En caso contrario no activa el bit CE en sus ack. Cuando el emisor recibe tres o más ACK duplicados, si tienen el bit CE activo pasa al modo de control de congestión, y si no lo tienen activo solo se reduce la ventana de congestión posibilitando una recuperación rápida.

  TCP Vegas [22] emplea tres técnicas proactivas para aumentar el throughput y disminuir las pérdidas, tratando de mantener la cantidad “correcta” de datos extra que la conexión tiene en tránsito (datos que no se habrían enviado si el ancho de banda utilizado por la conexión fuera exactamente el ancho de banda disponible de la red). Utiliza un nuevo mecanismo de retransmisiones, una técnica para anticipar la congestión y ajustar el ratio de transmisión de acuerdo a ello, y una modificación al mecanismo TCP 'slow start' para evitar la pérdida de paquetes tratando de encontrar el ancho de banda disponible.

  TCP Veno [35] adopta una metodología similar a la de TCP Vegas para estimar la cantidad de datos extra que la conexión tiene en tránsito. Sugiere diferenciar la causa de pérdida de paquetes: si el número de paquetes extra en la conexión está bajo cierto umbral, la pérdida es considerada aleatoria (aumentando la ventana de congestión de manera conservativa), mientras que si no, se considera que es debida a congestión, adoptando el esquema estándar de Reno.

  TCP Westwood [23] es un protocolo extremo-a-extremo en el que el emisor estima el ancho de banda de la red de manera dinámica, midiendo el ratio de ACKs que se reciben.

TCP-Jersey [36] es otro esquema proactivo que adapta la tasa de envío de acuerdo a las condiciones de la red. Consiste en dos componentes clave: el algoritmo de estimación del ancho de banda (ABE), que es una modificación en el emisor TCP, y una configuración de los routers de la red para que alerten a las estaciones finales (CW, congestion warning), marcando todo los paquetes cuando hay signos de incipiente congestión (de esta manera, el extremo TCP podrá diferenciar pérdidas de paquetes causadas por congestión en la red de pérdidas aleatorias causadas por errores en enlaces inalámbricos). ECN tiene que ser soportada en los routers para poder implementar CW.

En TCP Santa-Cruz [37] se almacenan los tiempos de envío y recepción de todos los paquetes para poder calcular el tiempo entre llegadas y estimar, en base a este dato, la congestión del canal.

- Redes Ad Hoc: En las redes ad-hoc se producen fallos en el rendimiento producidos por el retardo derivado del tiempo de computación de reenrutamiento o del particionamiento de la red.

  En Feedback-Based Scheme for Improving TCP Performance (TCP-F) [38] se envía un paquete RFN cuando se produce un fallo en un punto de la ruta de encaminamiento del paquete, y un paquete RRN cuando ésta se reestablece. Esto permite al emisor detener el envío de paquetes y los temporizadores cuando la ruta se rompe, y reactivarlos cuando se recupera.

  Ad Hoc TCP (ATCP) [56], varía el estado del emisor entre “retransmitir” y “persistir” en función de si se detecta una ruta de comunicación válida o no.

2.7.2.1. TCP Vegas

El protocolo TCP Vegas [22] se basa en modificaciones a la implementación Reno. No implica cambios en la especificación TCP, y todas las modificaciones son realizadas en el lado emisor. Como se ha resumido en la sección anterior, Vegas emplea tres técnicas para incrementar el throughput y disminuir las pérdidas. A continuación se describen en detalle, comparándolas con Reno.

- **Nuevo mecanismo de retransmisiones**

Reno utiliza dos mecanismos para detectar y retransmitir segmentos perdidos, ya descritos en secciones anteriores. El mecanismo original de retransmisiones por 'timeout', que es parte de la especificación TCP, está basado en la estimación del RTT y la varianza, midiendo el tiempo desde que se envía un segmento y se recibe su ACK. En las implementaciones basadas en BSD se utiliza un reloj con ticks cada 500ms para calcular dicho RTT. Igualmente, la comprobación de si se debería producir un 'timeout' para un segmento ocurre con dicha frecuencia. El uso de este 'coarse-grained timer' implica que el intervalo de tiempo entre que se envía un segmento que se pierde hasta que se produce un timeout y se reenvía es, generalmente, bastante más largo de lo necesario.

Para solucionar este problema, Vegas utiliza una manera más refinada para estimar los RTT y extiende el mecanismo de retransmisiones de Reno de la siguiente manera: primero lee y registra

el reloj del sistema cada vez que un segmento es enviado. Cuando se recibe un ACK, Vegas lee de nuevo el reloj del sistema y calcula el RTT utilizando este tiempo y el 'timestamp' registrado cuando se envió el segmento. Con este cálculo más exacto del RTT, Vegas decide si retransmitir en las dos situaciones siguientes (en la figura 2.18, en la siguiente página, se muestra un ejemplo simple de funcionamiento):

- Cuando recibe un ACK duplicado, Vegas comprueba si la diferencia entre el tiempo actual y el 'timestamp' registrado cuando el segmento relevante se envió es mayor que un determinado valor de 'timeout'. Si es así, Vegas retransmite el segmento sin esperar a recibir n (3) ACK's duplicados. En algunas ocasiones, las pérdidas de paquetes son tan elevadas o la ventana es tan pequeña que el emisor nunca recibirá tres ACKs duplicados, con lo que Reno tendría que depender el 'coarse-grained timeout' mencionado antes.

- Cuando se recibe un nuevo ACK (no duplicado), si es el primero o el segundo después de una retransmisión, Vegas comprueba de nuevo si el intervalo de tiempo desde que fue enviado es mayor que el valor de timeout. Si es así, Vegas retransmite el segmento. Este mecanismo captura cualquier otro segmento que pueda haberse perdido previo a la retransmisión sin tener que esperar a recibir un ACK duplicado.

Por tanto, Vegas trata la recepción de ciertos ACKs como indicio para comprobar si un 'timeout' debería ocurrir. El objetivo del nuevo mecanismo de retransmisiones no es sólo reducir el tiempo para detectar paquetes perdidos desde el tercer ACK duplicado al primero o segundo, sino también detectar pérdidas de paquetes incluso cuando puede que no haya un segundo o tercer ACK duplicado. Además, Vegas todavía cuenta con el código del 'coarse-grain timeout' de Reno en caso de que el nuevo mecanismo de retransmisiones falle al reconocer un segmento que se ha perdido.

Un aspecto a destacar es que la ventana de congestión debería reducirse únicamente debido a pérdidas que han ocurrido durante la tasa de envío actual, y no antes, a tasas de envío más elevadas.

En Reno es posible reducir la ventana de congestión más de una vez por pérdidas ocurridas durante un RTT, problema ya destacado en [9]. Vegas, sin embargo reduce la ventana de congestión únicamente si el segmento retransmitido fue enviado después de la última reducción de ventana, ya que se asume que cualquier pérdida de paquete que sucediera antes de la última reducción de la ventana de congestión no implica que la red esté congestionada para el tamaño actual de la ventana. Este cambio es necesario debido a que Vegas detecta las pérdidas mucho antes que Reno.

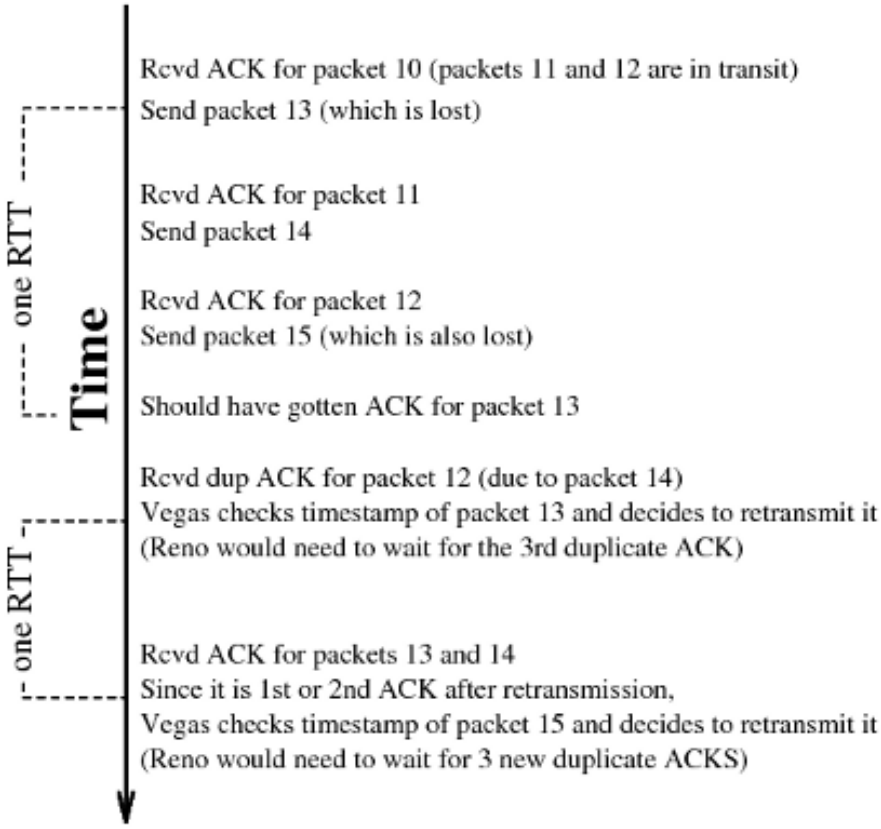

Figura 2.18: Ejemplo del mecanismo de retransmisiones en TCP Vegas
Figura de [22]

- **Mecanismo del control de la congestión**

TCP Vegas utiliza un mecanismo proactivo para el control de la congestión, en vez del mecanismo reactivo que utiliza Reno, que "necesita" que haya pérdidas de paquetes para detectar congestión y poder estimar el ancho de banda disponible en la conexión.

Se pueden encontrar diferentes modelos propuestos para la detección de la congestión proactiva, algunos de los cuales son:

- El algoritmo de Wang & Crowcroft's DUAL [39], basado en el incremento del RTT; cada 2 RTT el algoritmo comprueba si el RTT actual medido es mayor que la media de los RTT mínimo y el máximo medidos hasta el momento, reduciendo la ventana de congestión 1/8 si se da el caso.

- La aproximación de Jain's CARD (Congestión Avoidance using Round-trip Delay) [40] ajusta la ventana de congestión cada 2 RTT basado en el producto (*cwnd* actual – *cwnd* anterior) x (RTT actual – RTT anterior); si el resultado es positivo, se reduce la ventana de congestión en 1/8; si es negativo o cero, se incrementa la ventana de congestión 1 MSS (maximum segment size, tamaño máximo de segmento). De esta manera la ventana de congestión oscilará alrededor de un valor considerado óptimo.

- El esquema de Wang & Crowcroft's Tri-S [41] incrementa el tamaño de ventana en un segmento cada RTT, y después compara el throughput conseguido con el throughput que se tenía cuando la ventana era un segmento más pequeña; si la diferencia es menor que la mitad del throughput que se tenía cuando sólo había un segmento en tránsito (como era el caso al principio de la conexión, momento en el que se considera que no hay congestión), entonces se reduce la ventana en un segmento. Tri-S calcula el throughput dividiendo el número de bytes en tránsito que hay en la red entre el RTT.

Vegas es similar al enfoque presentado por el esquema Tri-S, en tanto que tiene en cuenta los cambios en el throughput, aunque el cálculo se realiza de manera diferente.

El objetivo de Vegas, como se ha mencionado, es mantener la cantidad "correcta" de datos extra en la red (datos que no se habrían enviado si el ancho de banda utilizado por la conexión fuera exactamente el ancho de banda disponible de la red). Evidentemente, si en una conexión se están enviando demasiados datos, se producirá congestión, mientras que si no se envían suficientes datos extra, no se está aprovechando de manera adecuada el ancho de banda disponible.

Las acciones del control de la congestión de TCP Vegas están basadas en cambios en la cantidad estimada de datos extra en la red, y no sólo en los segmentos descartados. El algoritmo, que no está activo durante la fase de slow-start, se describe a continuación:

1. Se define *BaseRTT* para una conexión dada, que es el RTT de un segmento, medido cuando la conexión no está congestionada; en la práctica se establece *BaseRTT* al mínimo de todos los RTT medidos, y suele ser el RTT del primer segmento enviado por la conexión, antes de que las colas de los routers se incrementen debido al tráfico generado. Así, asumiendo que no hay un exceso de tráfico en la conexión, el throughput esperado viene dado por la expresión:

$$Expected = WindowSize / BaseRTT$$

siendo *WindowSize* el tamaño de la ventana actual de congestión, que se asume es el número de bytes en tránsito.

2. Vegas calcula la tasa de envío actual real. Esto se realiza calculando cuántos bytes se transmiten desde que un determinado segmento es enviado hasta que su reconocimiento, ACK, es recibido. Se calcula el RTT de dicho segmento, y se obtiene el throughput real tal que:

$$Actual = BytesTransmitted / RTT$$

3. Vegas compara el throughput real con el esperado y ajusta la ventana de congestión de acuerdo a ello.

   Sea $Diff = Expected - Actual$. (Nótese que *Diff* es positivo o cero por definición, ya que $Actual > Expected$ implicaría que hay que modificar *BaseRTT* al último RTT medido). Además, se definen dos umbrales, $\alpha < \beta$, correspondientes a tener muy pocos y demasiados datos extra en la red, respectivamente. Cuando $Diff < \alpha$, Vegas incrementa linealmente la ventana de congestión durante el siguiente RTT, y cuando $Diff > \beta$, Vegas reduce la ventana de congestion linealmente durante el siguiente RTT. Cuando $\alpha < Diff < \beta$, Vegas no modifica la ventana de congestión.

Intuitivamente, cuando mayor es la diferencia entre el throughput real y el esperado, más congestión hay en la red, lo que implica que la tasa de envío debe reducirse (el umbral β detecta este caso). En la situación opuesta, cuando el throughput real está muy cerca del esperado, la conexión está en peligro de no estar utilizando el ancho de banda disponible (el umbral α detecta este caso). El objetivo es mantener entre α y β bytes extra en la red.

- **Mecanismo 'slow start' modificado**

El mecanismo 'slow start' de TCP Reno es muy “caro” en términos de pérdidas: puesto que dobla el tamaño de la ventana de congestión cada RTT mientras no se detectan pérdidas – lo que es equivalente a doblar la tasa de envío cada RTT – cuando finalmente se sobrepasa el ancho de banda de la conexión, se pueden esperar pérdidas del orden de la mitad de la ventana de congestión actual, si aparecen ráfagas de datos de otras conexiones.

Lo que se necesita en este caso es un modo de encontrar una tasa de envío que no incurra en este tipo de pérdidas. Para ello, Vegas incorpora a la fase de 'slow start' el mecanismo de detección de la congestión, con cambios menores. Para detectar la congestión y evitarla, Vegas permite el crecimiento exponencial durante 'slow start' solamente durante el siguiente RTT (es decir, cada dos RTT). Entre medias, la ventana de congestión permanece fija, de manera que se pueda realizar una comparación válida de los throughputs esperado y real. Cuando el throughput real cae por debajo del esperado una cierta cantidad (umbral γ, que suele ser 1), Vegas cambia de 'slow start' a fase de incremento/decremento lineal.

La razón por la cual se necesita medir el throughput real con una ventana de congestión fija es que se desea que el ratio real represente el ancho de banda del canal permitido por la conexión. De esta manera, sólo se pueden enviar tantos datos como se han reconocido en el ACK (Reno, en cambio, envía un segmento extra por cada ACK recibido durante la fase de 'slow start').

2.7.2.2. TCP Westwood

TCP Westwood [23] es una versión del protocolo TCP que mejora la performance de la ventana de congestión usando como feedback la medida extremo-a-extremo del ancho de banda disponible a través de la conexión TCP. El ancho de banda disponible se estima en el extremo TCP midiendo y aplicando un filtro pasa-bajo del ratio de ACKs recibidos. Después, dicha estimación del ancho de banda se utiliza para establecer la ventana de congestión y el umbral de 'slow start', *ssthresh*, después de un episodio de congestión, esto es, después de un 'timeout' o la recepción de 3 ACKs duplicados.

La base lógica de esta estrategia es la siguiente: TCP Westwood establece un umbral de 'slow start' y una ventana de congestión que son consistentes con la capacidad de la red medida cuando se experimenta congestión. En particular, TCP Westwood introduce un mecanismo llamado 'faster recovery' para evitar una reducción excesiva de la ventana de congestión después de un episodio de congestión.

La ventaja del mecanismo propuesto es que el emisor TCP se recupera más rápido (de ahí 'faster recovery') después de pérdidas, especialmente en conexiones con RTT elevados, o sobre

enlaces inalámbricos donde las pérdidas esporádicas son debidas a problemas en el enlace en vez de a congestión.

Además, se puede observar que las modificaciones propuestas por TCP Westwood siguen el principio de diseño extremo-a-extremo de TCP, requiriendo solamente ligeras modificaciones en el lado emisor y siendo compatibles con versiones anteriores del protocolo.

En los siguientes apartados se describen más profundamente las características del protocolo.

- **Estimación del ancho de banda extremo-a-extremo**

El protocolo TCP debe realizar un control de la congestión extremo-a-extremo, pues el diseño TCP asume que la red es una “caja negra”, lo que significa que los extremos no pueden recibir ningún feedback explícito de congestión de la red, y dependen de 'timeouts', ACKs duplicados y medidas de RTT.

TCP Westwood propone realizar una estimación del ancho de banda disponible extremo-a-extremo en la conexión TCP midiendo el ratio de ACKs recibidos cada RTT. De igual manera, los ACKs duplicados (DUPACKs) indicando la recepción de un segmento fuera de orden, también deben contar en la estimación del ancho de banda.

Después de un episodio de congestión, seguido de un timeout o n (3) ACKs duplicados, el ancho de banda usado por la conexión es exactamente el máximo disponible para esa conexión. Esto lo confirma el hecho de que se han descartado paquetes. Así pues, antes del episodio de congestión, el ancho de banda era menor o igual al disponible, ya que la fuente TCP estaba todavía probando la capacidad de la red.

TCP Westwood propone utilizar un filtro pasa-bajo para obtener los componentes de baja frecuencia del ancho de banda disponible. Se sabe que la congestión ocurre cuando la tasa de tráfico de baja frecuencia excede la capacidad del enlace [42]. Por ello, es adecuado tener en cuenta sólo los componentes de baja frecuencia del ancho de banda disponible. En el esquema de TCP Westwood se propone realizar la estimación del ancho de banda usando un filtro pasa-bajo, tal y como describe el siguiente pseudocódigo:

```
if (ACK is received)
     sample_BWE[k] = (acked*pkt_size*8) / (now – lastacktime);
     BWE[k] = (19/21)*BWE[k-1] + (1/21) * (sample_BWE[k] +
sample_BWE[k-1]);
endif
```

donde '`acked`' indica el número de segmentos reconocidos por el último ACK, '`pkt_size`' indica el tamaño de segmento en bytes, '`now`' indica el tiempo actual, '`lastacktime`' el tiempo en el que se recibió el ACK anterior, '*k*' y '`(k-1)`' indican los valores actual y previo de la variable, y `BWE` es la estimación del ancho de banda disponible, filtrada con el filtro pasa-bajo. Dicho filtro se obtiene discretizando un filtro pasa-bajo de primero orden, usando la regla trapezoidal (transformación bilinear o regla de Tustin [43], y asumiendo una constante de tiempo para el muestreo del ratio igual a 10.

La estimación del ancho de banda se utiliza, como se describe en la sección siguiente, para calcular la ventana de congestión apropiada.

- **Procedimiento tras un episodio de congestión**

La idea general es utilizar el ancho de banda estimado, BWE, para establecer tanto la ventana de congestión (*cwnd*) como el umbral de 'slow start' (*ssthresh*), después de un episodio de congestión.

Hay que destacar que la dinámica de la ventana de congestión durante las fases del 'slow start' y 'congestion avoidance' no se modifica, permaneciendo como en Reno (incremento exponencial y lineal, respectivamente).

Como ya se ha mencionado, se consideran episodios de congestión el hecho de recibir n (3) ACKs duplicados o la expiración de un 'timeout'. A continuación se describen los pasos a seguir en cada situación:

A. Después de recibir n ACKs duplicados
   El pseudocódigo del algoritmo es el siguiente:

```
if (n DUPACKs are received)
     if (cwnd > ssthresh) /* cong. Avoidance */
          ssthresh = f1(BWE*RTTmin);
          cwnd = ssthresh;
     endif
     if ( cwnd < ssthresh) /* slow-start */
          ssthresh = f2(BWE*RTTmin);
          if (cwnd > ssthresh)
               cwnd = ssthresh;
          endif
     endif
endif
```

La lógica que hay tras el algoritmo es la siguiente: durante la fase de 'congestion avoidance' se está “tanteando” el ancho de banda extra disponible. Así, cuando se reciben n (3) ACKs duplicados, significa que se ha superado la capacidad de la red (o en caso de enlaces inalámbricos, que uno o más segmentos han sido descartados debido a pérdidas esporádicas). En este momento, *ssthresh* y *cwnd* son establecidos a *BWE*RTTmin*, estando en fase de 'congestion avoidance', lo que provoca que se entre de nuevo en fase de “tanteo” del ancho de banda disponible. La función `f1` introduce un grado de libertad que se puede utilizar para 'tunear' el algoritmo.

Si TCP está en fase de 'slow start', todavía se está “tanteando” el ancho de banda disponible, *ssthresh* es establecido, igualmente, a *BWE*RTTmin* (la función `f2` introduce un grado más de libertad), pero *cwnd* es establecido a *ssthresh* únicamente si (*cwnd* > *ssthresh*). En otras palabras, durante 'slow start', la ventana de congestión todavía experimenta un crecimiento exponencial como en la implementación de TCP Reno.

B. Después de recibir la expiración de un 'timeout'
El pseudocódigo del algoritmo es el siguiente:

```
if (coarse timeout expires)
     if (cwnd > ssthresh) /* cong. Avoidance */
          ssthresh = f3(BWE*RTTmin);
          if (ssthresh < 2)
               ssthresh = 2;
               cwnd = 1;
          else
               cwnd = f4(BWE*RTTmin);
          endif
     endif
     if (cwnd < ssthresh) /* slow-start */
          ssthresh = f5(BWE*RTTmin);
          if (ssthresh < 2)
               ssthresh = 2;
               cwnd = 1;
          else
               cwnd = f6(BWE*RTTmin);
          endif
     endif
endif
```

La lógica del algoritmo de nuevo es simple. Tras un 'timeout', *cwnd* y *ssthresh* son establecidos de acuerdo a una de las funciones `fi`, `i`=3,6, dependiendo de la fase en la que el algoritmo se encuentra cuando se experimenta el 'timeout'. Nótese que las funciones `fi`, `i`=1-6, proporcionan seis grados de libertad para modificar el algoritmo.

3. OMNeT++

3.1. ¿Qué es OMNeT++?

OMNeT++ [45] es un simulador modular de eventos discretos de redes orientado a objetos, usado habitualmente para modelar el tráfico de redes de telecomunicaciones, protocolos, sistemas multiprocesadores y distribuidos, validación de arquitecturas hardware, evaluación del rendimiento de sistemas software y, en general, modelar cualquier sistema que pueda simularse con eventos discretos.

Comenzó a desarrollarse en 1992 en la Technical University of Budapest (Hungría), y actualmente está disponible tanto para sistemas basados en UNIX como para Windows y se distribuye bajo la Licencia Pública Académica. Su versión comercial, denominada OMNEST es desarrollada actualmente por Simulcraft Inc. [47].

3.2. Modelos en OMNeT++

Un modelo en OMNeT++ consiste en una serie de módulos que se comunican pasándose mensajes. Estos módulos activos se llaman módulos simples y están escritos en C++, usando la biblioteca de simulación. Además, pueden ser agrupados en módulos compuestos, que a su vez pueden volver a agruparse; el número de niveles en la jerarquía es ilimitado. El modelo completo se denomina 'network' y es, en sí mismo, un módulo compuesto. Los mensajes se pueden enviar vía conexiones que abarcan varios módulos o directamente entre módulos.

Enla Figura 3.1 se pueden observar los módulos simples (con el fondo gris) y los compuestos. Las flechas que conectan los módulos simples representan conexiones y puertas.

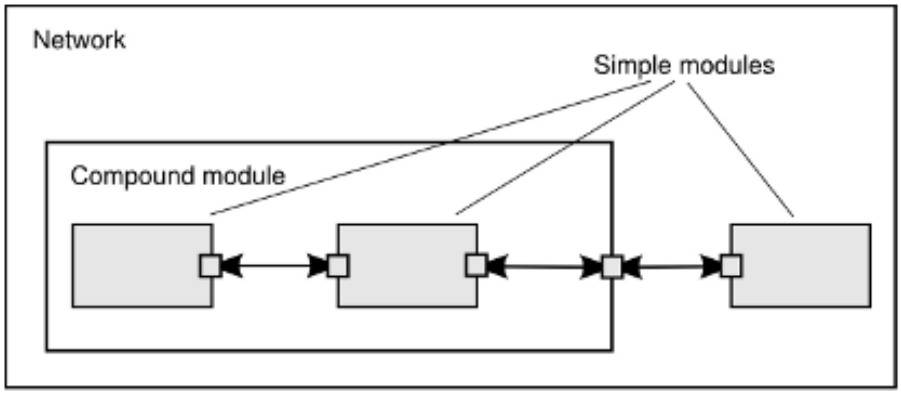

Figura 3.1: Módulos simples y compuestos en OMNeT++
Figura de [46]

Los mensajes, además de los atributos típicos, como los 'timestamps', pueden contener datos arbitrarios.

Los módulos simples normalmente se comunican vía puertas. Éstas son las interfaces de entrada y salida de los módulos y se enlazan mediante las conexiones. Estas conexiones contienen

tres parámetros que facilitan el modelado de redes de computadores, y que pueden ser utilizados también en otros modelos: el retardo de la propagación, el índice de error de bits y la tasa de transferencia de datos. Los módulos también pueden tener parámetros, y se pueden definir conexiones con propiedades específicas y reutilizarlas.

La estructura del modelo se define utilizando el lenguaje NED (NEtwork Definition), explicado en el apartado 3.2.1.

En resumen, OMNeT++ proporciona al usuario herramientas eficientes para describir la estructura del sistema real, siendo las características principales:

- Jerarquía de módulos anidados.
- Los módulos se comunican mediante mensajes a través de canales.
- Parámetros flexibles de módulos.
- Lenguaje de descripción de la topología.

Así pues, un modelo en OMNeT++ se compone de las siguientes partes:

- Lenguaje de descripción de la topología de la red, NED (archivos .ned), que describe la estructura de los módulos con parámetros, puertas, etc.
- Definición de los mensajes (archivos .msg). Es posible definir diferentes tipos de mensajes y añadirles campos de datos. OMNeT++ traducirá las definiciones de los mensajes a clases C++.
- Código fuente de los módulos simples. Se trata de los archivos C++ (.h/.cc).

Y la construcción de dicho modelo de simulación se puede resumir en los siguientes pasos:

- Descripción de la estructura (módulos y relaciones) del sistema mediante el lenguaje NED.
- Implementación de los módulos simples en C++.
- Generación del modelo. Se compilan los módulos y se enlazan con la biblioteca de simulación.
- Configuración de la simulación. Se especifican los parámetros adecuados para la ejecución de la simulación.

3.2.1. NED

Como ya se ha comentado, el lenguaje NED es utilizado para describir la topología, o estructura, de la simulación.

Hay tres elementos distintos que pueden describirse en un archivo NED:

- Definición de módulos: en estos bloques se pueden describir módulos simples y compuestos, y establecer sus parámetros. También se pueden definir las puertas que conectan los módulos.
- Definición de canales: describen los canales (enlaces) que conectan los módulos.

- Definición de la red: para conseguir que la simulación pueda ejecutarse, ha de establecerse un módulo principal, el que será el módulo de nivel superior para la red que se quiere simular. Este módulo de nivel superior se denomina módulo de sistema, y el resto de módulos son submódulos suyos (que a su vez pueden tener otros submódulos).

En la Figura 3.2 se puede observar un ejemplo de archivo .NED. En él se muestra la especificación de la red utilizada para las simulaciones de este Proyecto.

```
package tcpcomparison.simulations.tcpInetmanet;

import inet.linklayer.ethernet.EtherBus;
import inet.networklayer.autorouting.ipv4.FlatNetworkConfigurator;
import inet.nodes.ethernet.Eth100M;
import inet.nodes.ethernet.EtherHost;
import inet.nodes.inet.StandardHost;
import ned.DatarateChannel;

network TcpNet
{
    parameters:
        double retardo @unit(s) = default(replaceUnit(Eth100M.length / 2e8, "s"));
        double tasaDatos @unit(bps) = default(100Mbps);
    types:
        channel EthRetardada extends DatarateChannel {
            datarate = tasaDatos;
            delay = retardo;
        }
    submodules:
        server: StandardHost {
            @display("p=573,111");
        }
        client: StandardHost {
            @display("p=65,111");
        }
        flatNetworkConfigurator: FlatNetworkConfigurator {
            @display("p=313,33");
        }
    connections:
        server.ethg++ <--> EthRetardada <--> client.ethg++;
        client.ethg++ <--> EthRetardada <--> server.ethg++;
}
```

Figura 3.2: Ejemplo de lenguaje NED

Hay tres módulos simples, dos de los cuales son del tipo 'StandardHost' (módulo simple del INET Framework, descrito en la sección 3.3), y representan un nodo cliente y un nodo servidor. Están conectados mediante el canal 'EthRetardada', del tipo 'DatarateChannel', en el que, como se puede observar, se establecen los parámetros del retardo y la tasa de datos. Se define un valor por defecto para estos parámetros (con la etiqueta 'default'), que luego se podrá modificar en el archivo de inicialización (archivo .ini).

El tercer módulo es del tipo 'FlatNetworkConfigurator', también un módulo propio del INET Framework. Se encarga de configurar las direcciones IP y las rutas de los nodos IP de la red, en la inicialización.

Todos los elementos descritos conforman la red denominada 'TcpNet'. (Las etiquetas 'display' describen, únicamente, la manera en la que los módulos han de situarse en la representación visual de la simulación.

3.3. INET Framework

La suite IP que se ha utilizado en este Proyecto es el INET Framework [48], un paquete para simulaciones de redes de comunicaciones, de código abierto, para el entorno de simulación OMNeT++. Contiene implementaciones de los protocolos IPv4, IPv6, TCP, SCTP y UDP, y diversos modelos de aplicación. También incluye un modelo MPLS con RSVP-TE y señalización LDP. Los modelos de la capa de enlace son PPP, Ethernet y 802.11. Se puede utilizar routing estático utilizando autoconfiguradores de red, o se pueden utilizar implementaciones de protocolos de routing.

INET Framework soporta, también, simulaciones en redes inalámbricas y móviles. Dicho soporte ha sido derivado del paquete Mobility Framework [49].

3.3.1. Arquitectura de INET Framework

INET Framework se basa en OMNeT++ y utiliza el mismo concepto: módulos que se comunican pasándose mensajes (objetos `cMessage`).

Igualmente, los protocolos se representan con módulos simples. Así, la interfaz externa de un módulo (puertas o conectores y parámetros) se describe en un fichero NED, y su implementación está en una clase C++ con el mismo nombre. Ejemplos serían TCP, IPv4, UDP, etc.

Estos módulos pueden combinarse libremente para formar hosts y otros dispositivos de red con el lenguaje NED (no se necesita código C++ ni recompilar). Algunos modelos pre-ensamblados, como hosts, routers, switches, puntos de acceso, etc. se pueden encontrar en el subdirectorio Nodes/ (como por ejemplo 'StandardHost', mencionado en la sección 3.2.1), aunque el usuario puede crear sus propios nodos y configurarlos de manera libre para escenarios particulares de simulación.

Las interfaces de red (Ethernet, 802.11, etc.) son, normalmente, módulos compuestos (i.e. ensambladas a partir de módulos simples), y se componen de colas, MAC, y otros módulos simples. Un ejemplo es 'EthernetInterface'.

No todos los módulos implementan protocolos. Existen módulos que contienen datos (como por ejemplo 'RoutingTable'), otros tienen el objetivo de facilitar la comunicación de los módulos (como 'NotificationBoard'), realizar la autoconfiguración de la red (como 'FlatNetworkConfigurator', también mencionado en la sección 3.2.1), realizar tareas de gestión interna asociadas con canales de radio en simulaciones wireless (como 'ChannelControl'), etc.

Las cabeceras de los protocolos y los formatos de paquete se describen en archivos de definición de mensajes (archivos msg), que son traducidos a clases C++ por la herramienta de OMNeT++ *opp_msgc*. Las clases generadas son subclases de la clase `cMessage`, de OMNeT++.

3.3.2. El modelo TCP en INET Framework

3.3.2.1. El módulo TCP

El módulo simple `TCP` es la implementación principal del protocolo TCP en el INET Framework. Dicho módulo trabaja entre la capa de red y la capa de aplicación, con lo que tiene puertas para conectarse con IPv4 o IPv6 (ipIn/ipOut o ipv6In/ipv6Out), y con las aplicaciones (appIn[k], appOut[k]). Un módulo `TCP` puede servir a diferentes módulos de aplicación y a diferentes conexiones por aplicación. La aplicación '*k*' se conecta a los puertos `TCP appIn[k]` y `appOut[k]`.

El módulo TCP puede ser especificado por su interfaz de módulo (ITCP) en la definición NED de los hosts, con lo que puede reemplazarse por cualquier implementación que se comunique a través de las mismas puertas. El modelo TCP se basa en el envío y recepción de objetos `IPControlInfo` adjuntos como información de control a objetos que representan segmentos TCP.

Además, el módulo `TCP` maneja diferentes objetos `TCPConnection`, cada uno de los cuales lleva el estado de una conexión. Las conexiones se identifican por un identificador de conexión, que es elegido por la aplicación. Si la conexión se establece, puede ser identificada también por las direcciones y puertos locales y remotos. El módulo TCP despacha los comandos de aplicación entrantes y los paquetes a los correspondientes objetos.

- **Paquetes TCP**

  INET Framework modela las cabeceras TCP con la clase `TCPSegment`, que contiene los campos de un segmento TCP, excepto:

  - Data Offset: representado por `cMessage::length()`.
  - Reserved.
  - Checksum: modelado por `cMessage::hasBitError()`.
  - Options: sólo están permitidas EOL, NOP, MSS, WS, SACK_PERMITTED, SACK y TS.
  - Relleno.

  Los campos de datos, a su vez, pueden representarse de tres maneras diferentes:

  - Objetos C++ que representan paquetes encapsulados.
  - Raw's de bytes como instancias de `ByteArray`.
  - Cuenta de bytes (sólo el número de bytes),

que corresponden a los modos de transferencia OBJECT, BYTESTREAM y BYTECOUNT respectivamente.

- **Comandos TCP**

Las aplicaciones y el módulo TCP se comunican enviándose objetos `cMessage`. Estos mensajes están especificados en el archivo `TCPCommand.msg` [referencia].

Los tipos de mensaje que la aplicación puede enviar al módulo TCP son los siguientes:

- TCP_C_OPEN_ACTIVE: apertura activa.
- TCP_C_OPENP_PASSIVE: apertura pasiva.
- TCP_C_SEND: enviar datos.
- TCP_C_CLOSE: no más datos que enviar.
- TCP_C_ABORT: abortar conexión.
- TCP_C_STATUS: solicitud de información de estado de TCP.

Cada mensaje de comando debe tener adjunta información de control del tipo `TCPCommand`, con un campo `connId` que identifica la conexión de manera local, dentro de la aplicación.

Así mismo, los tipos de mensaje que TCP puede enviar a una aplicación se listan a continuación:

- TCP_I_ESTABLISHED: conexión establecida.
- TCP_I_CONNECTION_REFUSED: conexión denegada.
- TCP_I_CONNECTION_RESET: conexión reiniciada.
- TCP_I_TIME_OUT: timeout del temporizador de establecimiento de la conexión, o máximo número de retransmisiones alcanzado.
- TCP_I_DATA: paquete de datos.
- TCP_I_URGENT_DATA: paquete de datos urgente.
- TCP_I_PEER_CLOSED: FIN recibido de TCP remoto.
- TCP_I_CLOSED: conexión cerrada con normalidad.
- TCP_I_STATUS: información de estado.

Estos mensajes también tienen información de control adjunta con `TCPCommand` o algún tipo derivado (TCPConnectInfo, TCPStatusInfo, TCPErrorInfo).

- **Parámetros TCP**

El módulo TCP tiene los siguientes parámetros:

- `advertisedWindow: en bytes, corresponde a la máxima capacidad del buffer receptor (por defecto es 14*MSS).`

- delayedAcksEnabled: habilita/deshabilita el algoritmo de ACKs retrasados (RFC 1122 [11]).

- nagleEnabled: habilita/deshabilita el algoritmo de Nagle (RFC 896 [7]).

- limitedTransmitEnabled: habilita/deshabilita el algoritmo de limitar la conexión (RFC 3042 [44]).

- increasedIWEnabled: habilita/deshabilita la opción de ventana inicial incrementada (RFC 3390 [13]).

- sackSupport: soporte (opción de cabecera) para ACKs selectivos (RFC 2018, [20]). Se habilitará SACK para una conexión dada si ambos extremos de la conexión lo solicitan.

- windowScalingSupport: soporte (opción de cabecera) para el escalado de ventana (RFC 1323 [17]). Se habilitará para una conexión dada si ambos extremos de la conexión lo solicitan.

- timestampSupport: soporte (opción de cabecera) para 'timestamps' (RFC 1323 [17]). Se habilitará si ambos extremos de la conexión lo solicitan.

- mss: 'Maximum Segment Size', tamaño máximo de segmento (RFC 793 [1]). Es también una opción de cabecera y el valor por defecto es 536.

- tcpAlgorithmClass: nombre de la variante TCP que se usará. Posibles valores son “TCPReno” (por defecto), “TCPNewReno”, “TCPTahoe”, “TCPNoCongestionControl” y “DumpTCP”.
  TCPOpenCommand permite elegir tcpAlgorithmClass para cada conexión.

- recordStats: habilita/deshabilita que se escriban los vectores de recogida de estadísticas.

3.3.2.2. Conexiones TCP

La mayor parte de la especificación TCP está implementada en la clase TCPConnection: se encarga de la máquina de estados, guarda las variables de estado (TCB), envía/recibe los segmentos SYN, FIN, RST, ACKs, etc. TCPConnection implementa, ella misma, la “maquinaria” básica de TCP, mientras que los detalles del control de la congestión se tratan en las clases derivadas de TCPAlgorithm.

Hay dos objetos adicionales en los que TCPConnection se apoya internamente: instancias de las clases TCPSendQueue y TCPReceiveQueue. Estas clases polimórficas gestionan el flujo real de datos, de manera que TCPConnection solamente trabaja con variables que representan números de secuencia. Esto facilita la adecuación a las diferentes representaciones de los datos transferidos: flujo de bytes real, bytes “virtuales” (número de bytes únicamente), y secuencias de objetos cMessage (donde cada objeto es mapeado a un rango de números de secuencia TCP).

A continuación se describe el comportamiento del módulo TCP durante la conexión:

- **Inicio de la conexión**

Las aplicaciones pueden abrir un puerto local para conexiones entrantes enviando un mensaje TCP_C_PASSIVE_OPEN al módulo TCP. La información de control adjunta (TCPOpenCommand) contiene la dirección local y el puerto, y además la aplicación puede especificar si quiere manejar una conexión sólo o si aceptará múltiples conexiones. El campo dataTransferMode en TCPOpenCommand especifica si los datos de la aplicación son transmitidos como objetos C++, bytes reales o en número de bytes únicamente. El algoritmo para el control de la congestión puede ser también especificado en cada conexión, en el campo tcpAlgorithmClass.

Para abrir una conexión en un servidor remoto, la aplicación envía un mensaje TCP_C_OPEN_ACTIVE al módulo TCP. Éste crea un objeto TCPConnection y envía un segmento SYN. El número inicial de secuencia se establece de acuerdo al tiempo de simulación: 0 en el instante 0, e incrementándose en 1 cada 4μs. Si no hay respuesta al segmento SYN, se reintenta después de 3s, 9s, 21s y 45s. Después de 75s se produce un 'timeout' de establecimiento de la conexión (TCP_I_TIMEOUT), se reporta a la aplicación y se cierra la conexión.

Si la conexión se establece, TCP envía una notificación TCP_I_ESTABLISHED a la aplicación. Por el contrario, si la conexión es rechazada por el otro extremo, la aplicación recibe un mensaje TCP_I_CONNECTION_REFUSED.

- **Envío de datos**

La aplicación puede escribir datos en la conexión enviando un mensaje TCP_C_SEND al módulo TCP. La información de control adjunta debe ser del tipo TCPSendCommand.

TCP añadirá el mensaje a la cola de salida ('send queue'). Hay tres tipos de colas de salida, correspondientes a los tres modos de transferencia de datos (TCPMsgBaseSendQueue, TCPDataStreamSendQueue, TCPVirtualDataSendQueue). La cola se crea de acuerdo al valor del parámetro dataTransferMode del comando 'Open'.

El mensaje se entrega a la capa IP cuando el tamaño de las ventanas (en la sección 2.2 se describe el mecanismo de la ventana deslizante de TCP) lo permite. Si el algoritmo de Nagle [7] está habilitado, TCP esperará a tener 1 MSS de datos para enviarlo.

- **Recepción de datos**

La conexión TCP almacena los segmentos entrantes en la cola de entrada ('receive queue'). Esta cola, igual que la cola de salida, puede ser de tres tipos: TCPMsgBasedRcvQueue, TCPDataStreamRCVQueue y TCPVirtualDataRcvQueue.

El buffer receptor tiene una capacidad finita, de acuerdo al parámetro advertisedWindow. Si este buffer se llena y recibe un nuevo segmento, éste será descartado; este tipo de pérdidas se registran en el vector tcpRcvQueueDrops.

Si el número de secuencia del segmento recibido es el esperado, los datos se pasan a la aplicación inmediatamente. La llamada recv() de Unix no está modelada.

Los datos de un mensaje, que pueden ser un objeto cMessage, un objeto ByteArray, o una simple cuenta de bytes, se pasan a la aplicación en un mensaje con un comando del tipo TCP_I_DATA.

- **Gestión de RESET**

Cuando ocurre un error a nivel TCP, se envía un segmento RST al otro extremo y se aborta la conexión. Este error puede ser:

- Llegada de un segmento en estado CLOSE.
- Un segmento entrante reconoce (ACK) datos que no se han enviado todavía.

El extremo de la conexión que recibe el segmento RST abortará la conexión. Si ésta todavía no se había establecido, el extremo pasivo volverá al estado LISTEN y esperará una nueva conexión entrante en vez de abortar.

- **Cierre de la conexión**

Cuando la aplicación no tiene más datos que enviar, cierra la conexión enviando un comando TCP_C_CLOSE al módulo TCP. Éste transmitirá todos los datos del buffer y en el último segmento activará la opción FIN. Si este último segmento no es reconocido (ACK) en cierto tiempo, volverá a retransmitirlo con backoff exponencial.

El extremo TCP que recibe el segmento FIN notificará a la aplicación que el otro extremo no tiene más datos que enviar, enviándole un mensaje TCP_I_PEER_CLOSED (que contiene el identificador de la conexión en la información de control).

Cuando ambas partes han cerrado la conexión, las aplicaciones reciben un mensaje TCP_I_CLOSED y el objeto que representa la conexión se borra (aunque en la práctica uno de los extremos TCP espera 2MSL ('Maximum Segment Lifetime', máximo tiempo de vida de segmento) antes de eliminar la conexión, de manera que no es posible reconectar con la misma dirección y número de puerto inmediatamente).

- **Abortar una conexión**

La aplicación puede abortar una conexión. Esto significa que no esperará más datos entrantes, y descartará los datos asociados a la conexión inmediatamente. Para ello, la

aplicación envía un mensaje TCP_C_ABORT, especificando el identificador adjunto en la información de control. TCP enviará un segmento RST al otro extremo de la comunicación, y eliminará el objeto que representa la conexión. La aplicación no debería reconectar con las mismas direcciones local y y remota y puertos en un MSL ('Maximum Segment Lifetime', máximo tiempo de vida de segmento), porque segmentos de la antigua conexión podrían ser aceptados, erróneamente, en la nueva.

3.3.2.3. Algoritmos TCP

La clase abstracta TCPAlgorithm encapsula todo el comportamiento TCP relacionado con las retransmisiones, el control de la congestión y el envío de ACKs: ACKs retrasados, 'slow start', 'fast retransmit', etc. Las diferentes versiones del protocolo TCP extienden esta clase. Esto simplifica mucho el diseño de TCPConnection y facilita la implementación de nuevas variantes TCP.

Para proporcionar una visión más detallada de lo que ofrece la clase TCPAlgorithm, se describen a continuación las principales funciones que implementa (que pueden modificarse en las subclases que la extienden para añadir funcionalidades y variaciones al control de la congestión):

- ackSent(): se llama después de enviar un ACK.
- connectionClosed(): se llama cuando la conexión se cierra, debiéndose cancelar todos los temporizadores activos.
- createStateVariables(): crea el bloque de estado (TCB) utilizado por la variante TCP de la conexión en marcha. Se espera que cada subclase de TCPAlgorithm tenga su propio bloque de estado.
- dataSent(uint32 fromseq): función que se llama después de enviar datos. Puede utilizarse para inicializar el temporizador de retransmisiones, las medida del RTT, etc. El argumento es el numero de secuencia del primer byte enviado.
- established(bool active): se llama cuando la conexión pasa de SYN_SENT o SYN_RCVD a ESTABLISHED. Es la función en la que se inicializan algunas variables (como por ejemplo establecer cwnd al valor de MSS establecido durante el inicio de la conexión). Si no se está en el lado activo, también se debe terminar el inicio de la conexión en 3 pasos ('3-way handhsake') enviando un ACK.
- getStateVariables(): crea y devuelve el estado de las variables TCP.
- initialize(): debe redefinirse para inicializar el objeto: crear temporizadores, etc. Es necesario porque el puntero TCPConnection todavía no está disponible en el constructor.
- processTimer(cMessage* timer, TCPEventCode& code): procesa los temporizadores específicos de esta clase TCPAlgorithm. TCPConnection invocará este método cada vez que se produzca un 'timeout' en un temporizador que no reconozca (TCPConnection sólo reconoce los temporizadores de 2MSL ('Maximum Segment Lifetime', tiempo de vida máximo de segmento), CONN-ESTAB (establecimiento de conexión), y FIN-WAIT-2).

- receivedAckForDataNotYetSent(uint32 seq): se llama después de recibir un ACK que reconoce datos que no se han enviado todavía. De acuerdo a [1] esta función debe enviar un ACK.

- receivedDataAck(uint32 firstSeqAcked): se llama después de recibir un ACK que reconoce datos nuevos (esto significa que la variable snd_una, que indica el siguiente número de secuencia que espera reconocerse en el siguiente ACK, puede avanzar). Cuando se llama a esta función las variables de estado (snd_una, snd_wnd) ya han sido actualizadas, siendo el argumento firstSeqAcked el valor previo de snd_una. Esto significa que el número de bytes que el ACK recibido reconoce es (snd_una – firstSeqAcked). Sin embargo, el contador de ACKs duplicados refleja el valor antiguo, ya que se necesita en las subclases de TCPAlgorithm.

- receivedDuplicateAck(): se llama después de recibir un ACK duplicado, estando el contador de ACKs duplicados actualizado ya.

- receivedOutOfOrderSegment(): se llama después de recibir un segmento fuera de orden, es decir, un segmento que está dentro de la ventana, pero no en el extremo izquierdo, i.e. se esperaba recibir otros segmentos antes que él.

- receiveSeqChanged(): se llama después de que la variable rcv_nxt, que registra el siguiente segmento que se espera recibir, avance, bien porque se ha recibido un segmento con datos nuevos, o bien porque se ha recibido un segmento FIN.

- restartRexmitTimer(): reinicia el temporizador de retransmisiones.

Las subclases de TCPAlgorithm existentes hasta el momento, que implementan versiones de TCP, son TCPReno, TCPNewReno, TCPNoCongestionControl y DumbTCP. En la Figura 3.3, en la siguiente página, se pueden observar las relaciones de herencia que hay entre ellas.

Ya se ha mencionado que el algoritmo concreto de TCP a usar en una simulación se puede elegir, para cada conexión (en OPEN), o como parámetro de módulo.

A continuación se describe brevemente qué características implementa cada una de las versiones.

- DumbTCP: es una implementación muy básica, con un 'timeout' de retransmisión codificado (2 segundos) como única “sofisticación”.

- TCPBaseAlg: es la clase base de las implementaciones INET de Tahoe, Reno y NewReno. Implementa los algoritmos básicos TCP para retransmisiones, 'persistence timers', ACKs retrasados, algoritmo de Nagle e incremento de ventana inicial ('Increased Initial Window). Se excluye el control de la congestión, que se deja a las subclases.

- TCPNoCongestionControl: implementación sin control de la congestión.

- TCPTahoe: extiende TCPBaseAlg con los algoritmos para el control de la congestión 'slow start', 'congestion avoidance' y 'fast retransmit'. Inicia a 'slow start' cuando se detecta que un paquete se ha perdido.

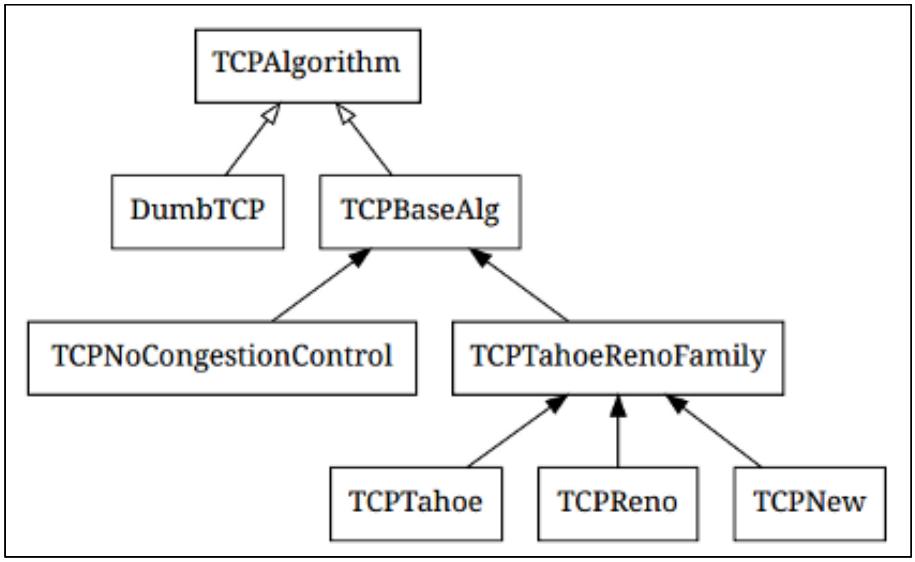

Figura 3.3: Versiones TCP en INET Framework
Figura de [50], Sección 2.4

- TCPReno: extiende el comportamiento de TCPTahoe con 'fast recovery'. Puede utilizar la información de la opción SACK, que permite un control de la congestión más refinado.

- TCPNewReno: implementa la variante TCP New Reno, que se recupera de manera más eficiente que Reno de múltiples pérdidas de paquetes dentro de un RTT. No sale de la fase de 'fast recovery' hasta que todos los datos que hay en tránsito en el instante en el que entró en dicha fase son reconocidos (ACK); así evita reducir *cwnd* múltiples veces.

4. Protocolos implementados

El proyecto desarrollado persigue como finalidad el estudio de versiones del protocolo TCP en entornos de red inalámbricos, en los que, entre otras diferencias, destaca el hecho de que las pérdidas de paquetes no se deben tanto a congestión en la red, pero sí a pérdidas esporádicas debidas a fallos en el medio.

Concretamente se han implementado las versiones Vegas y Westwood del protocolo TCP para el paquete INET Framwork [48], suite IP del simulador OMNeT++ [45].

4.1. Descripción de la propuesta realizada

La idea básica del proyecto es la siguiente. Tomando como base la versión existente TCP Reno del protocolo, presente en INET Framework, se implementarán los protocolos TCP Vegas y TCP Westwood.

El funcionamiento de estos protocolos se ha descrito en las secciones 2.7.2.1 y 2.7.2.2 respectivamente. Cabe destacar que, para su implementación, se han seguido fielmente las propuestas existentes para cada uno de ellos [22] [23].

Además, se han tomado como referencia las implementaciones existentes de TCP Vegas para x-kernel [53] y ns-2 [52], y de TCP Westwood para ns-2 [54].

4.2. Diseño

En la sección 3.3.2.3. se han mencionado las versiones existentes del protocolo TCP en el INET Framework. Se ha observado que todas ellas extienden la clase TCPAlgorithm, que es la que controla las retransmisiones, el control de la congestión y el envío de ACKs. Concretamente, la clase TCPBaseAlg es la clase base de las implementaciones INET de Tahoe, Reno y New Reno, e implementa los algoritmos básicos de TCP para retransmisiones, 'persistence timers', ACKs retrasados, algoritmo de Nagle e incremento de la ventana inicial ('Increased Initial Window'). Se excluye el control de la congestión, que se deja a las subclases.

Por ello, puesto que las propuestas Vegas y Westwood plantean cambios en el lado emisor de la comunicación, en el control de la congestión, con respecto a Reno, se ha decidido que dichas clases, TCPVegas y TCPWestwood, extiendan también la clase TCPBaseAlg (ver Figura 4.1 en la siguiente página).

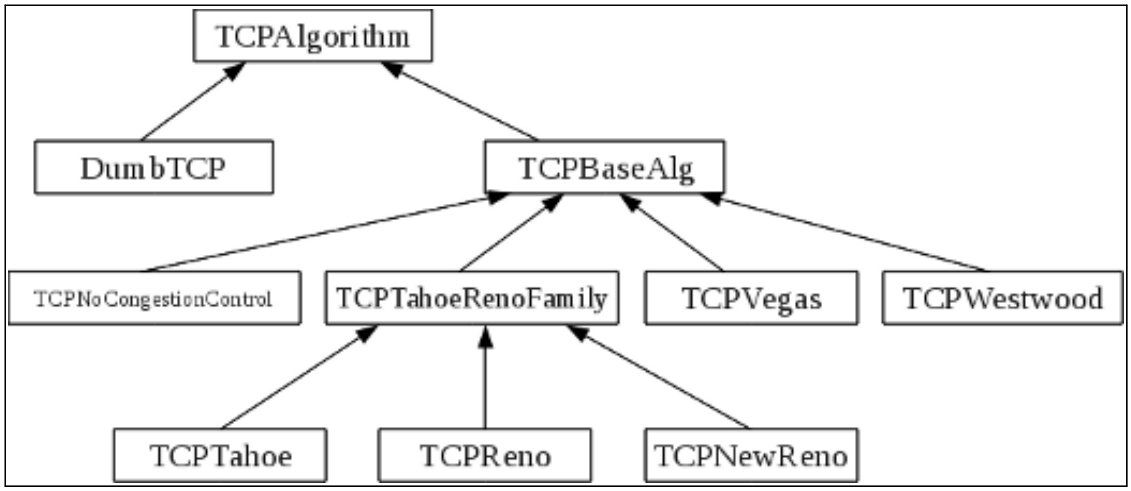

Figura 4.1: Integración de TCPVegas y TCPWestwood en la estructura de clases que extienden TCPAlgorithm

De esta manera se aprovecha y se explota la estructura de clases de INET Framework a la hora de añadir nuevas versiones del protocolo TCP que sólo plantean cambios en el control de la congestión, no siendo necesario modificar la clase `TCPConnection`, que es la que controla la comunicación TCP en aspectos como el manejo de PDUs (SYN, SYNACK, RST, FIN, etc.), temporizadores y eventos asociados con los cambios de estado TCP.

4.3. Detalles de implementación

La implementación de estos protocolos se ha realizado en C++, puesto que el objetivo es integrarlos en la suite IP para OMNeT++, INET Framework, desarrollada en C++.

El desarrollo se ha realizado sobre un sistema operativo Debian GNU/Linux 6.0.5 (squeeze) de 32 bits, con el apoyo del entorno gráfico de desarrollo Eclipse IDE for C/C++, que viene incorporado en OMNeT++ IDE en su versión 4.2.2.

En las dos próximas secciones se describen detalladamente los métodos que constituyen las clases TCPVegas y TCPWestwood.

4.3.1. TCP Vegas

Como se ha mencionado en la sección 4.2., la clase TCPVegas hereda de la clase TCPBaseAlg, y se ha implementado tomando como base la clase TCPReno.

```
class INET_API TCPVegas : public TCPBaseAlg {
protected:
      TCPVegasStateVariables *&state; // alias to TCLAlgorithm's 'state'
```

```
        virtual TCPStateVariables *createStateVariables() {
              return new TCPVegasStateVariables();
        }
        virtual void recalculateSlowStartThreshold();
        virtual void processRexmitTimer(TCPEventCode& event);
        virtual bool checkRTTTimer();
public:
        TCPVegas();
        virtual void receivedDataAck(uint32 firstSeqAcked);
        virtual void receivedDuplicateAck();
        virtual void dataSent(uint32 fromseq);
};
```

De esta manera se explota el mecanismo de herencia y se aprovecha la funcionalidad TCP que implementa la clase TCPBaseAlg. Así, sólo ha sido necesario crear un método nuevo, checkRTTTimer, mientras que el resto se han redefinido, para añadir funcionalidades propias de Vegas.

A continuación se describen los detalles particulares de cada método. Cabe destacar, como se podrá ver, que la funcionalidad básica que implementa TCPBaseAlg no se pierde en las funciones redefinidas, puesto que se llama a los métodos de TCPBaseAlg primero.

Una de las características más importantes de Vegas es el mecanismo, más refinado, que plantea para calcular los tiempos de ida y vuelta de los paquetes (RTTs), que luego se utilizan en el nuevo mecanismo de retransmisiones. Cuando se envía un paquete, se debe registrar el tiempo en el que se envía. Para ello, se redefine la función dataSent, que se llama después de enviar datos:

```
void TCPVegas::dataSent(uint32 fromseq) {

        TCPBaseAlg::dataSent(fromseq);

        // Si es el primer paquete que se envia, inicializ.
        if (fromseq == state->iss+1) {
              // rcv_wnd: maxima capacidad del buffer receptor
              state->v_maxwnd = state->rcv_wnd;

              state->v_sendtime = new simtime_t[state->v_maxwnd];
              state->v_transmits = new int[state->v_maxwnd];
              for(int i = 0; i < state->v_maxwnd; i++) {
                    state->v_sendtime[i] = -1;
                    state->v_transmits[i] = 0;
              }
        }
        // guardar el tiempo en el que se envia el paquete:
        simtime_t sendtime = simTime();
        for(uint32 i = fromseq; i < state->snd_max; i++) {
              int index = (i - (state->iss+1)) % state->v_maxwnd;
              state->v_sendtime[index] = sendtime;
              ++state->v_transmits[index];
        }
}
```

Se definen los vectores v_sendtime y v_transmits para registrar el tiempo en que se envía cada byte y el número de veces que se transmite (1 vez si no se retransmite nunca). El tamaño de dichos vectores se establece a rcv_wnd (suele ser 14 * MSS, 'Maximum Segment Size, tamaño

máximo de segmento), que es la máxima capacidad del buffer receptor, con lo que la ventana de congestión nunca superará dicho valor.

Además, cuando se itera sobre dichos vectores, los índices tienen que adaptarse, porque los valores que se tienen para calcular los bytes que se envían (snd_max – fromseq es el número de bytes que se envían) son números de secuencia de bytes, y no necesariamente al primer byte enviado se le asigna el número de secuencia 0, es decir, iss ('initial sequence number', número inicial de secuencia) no es un valor constante inicializado a 0. Por eso se calculan los índices de los vectores tal que:

```
int index = (i - (state->iss+1)) % state->v_maxwnd;
```

El motivo para registrar el tiempo y el número de transmisiones de cada byte en lugar de cada paquete, es el hecho de que no se asume que los paquetes sean todos del mismo tamaño, además de que, en las funciones que manejan la recepción de ACKs, se tienen variables que identifican números de secuencia de bytes, y no de paquetes. De esta manera no se pierde generalidad (pues el método es válido tanto si los paquetes son del mismo tamaño como si no), y no se hace necesario guardar más variables, que serían necesarias para identificar números de paquete.

De esta manera, cuando se recibe un ACK, se puede calcular el RTT de un segmento de manera sencilla, como se observa en receivedDataAck, con el 'timestamp' registrado (tSent) y el instante de tiempo actual (currentTime):

```
void TCPVegas::receivedDataAck(uint32 firstSeqAcked) {

      TCPBaseAlg::receivedDataAck(firstSeqAcked);

      simtime_t tSent = state->v_sendtime[(firstSeqAcked - (state->iss+1))
% state->v_maxwnd];
      simtime_t currentTime = simTime();
```

Con estos valores ('fine-grained timer') se puede realizar el cálculo del 'timeout' que propone Vegas para el nuevo mecanismo de retransmisiones (v_rtt_timeout).

```
if (tSent != 0 && num_transmits == 1) {
      simtime_t newRTT = currentTime - tSent;
      state->v_sumRTT += newRTT;
      ++state->v_cntRTT;

      if (newRTT > 0) {
            if(newRTT < state->v_baseRTT)
                  state->v_baseRTT = newRTT;

            simtime_t n = newRTT - state->v_sa/8;
            state->v_sa += n;
            n = n < 0 ? -n : n;
            n -= state->v_sd / 4;
            state->v_sd += n;
            state->v_rtt_timeout = ((state->v_sa / 4) + state->v_sd) / 2;
            state->v_rtt_timeout += (state->v_rtt_timeout / 16);
      }
}
```

Para ello, se hace uso de la media (v_sa) y la desviación (v_sd), siguiendo los cálculos utilizados en la implementación de Vegas del x-kernel [53], basados en el 'smoothing algorithm' definido en el RFC 793 [1].

Obsérvese que, siguiendo el Algoritmo de Karn [55], los paquetes retransmitidos, no cuentan para el cálculo de v_rtt_timeout. Por eso se comprueba que num_transmits valga 1, ya que guarda el número de transmisiones del paquete considerado:

```
int num_transmits =
      state->v_transmits[(firstSeqAcked - (state->iss+1)) % state->v_maxwnd];
```

En el método receivedDataAck también se llevan a cabo todas las operaciones, una vez por RTT, de ajuste de la ventana de congestión y el umbral de 'slow start', dependiendo de si el algoritmo está en fase de 'slow start' o 'congestion avoidance', y en base a la diferencia entre el 'throughput' esperado y el real. A continuación se describen los detalles de esta parte del código.

Primero se deben definir los diferentes umbrales que propone Vegas: α, β y γ. Estos valores están recomendados en la propuesta [22], en la que también se hacen simulaciones con valores de (α = 1) y (β = 3), aunque obteniendo menos mejoras sobre el 'throughput'.

```
      // Se necesita el valor en bytes, no en paquetes
      uint32 v_beta = 4 * state->snd_mss;
      uint32 v_alpha = 2 * state->snd_mss;
      uint32 v_gamma = 1 * state->snd_mss;
```

A continuación, se llevan a cabo los cálculos, una vez por RTT.

El valor del RTT para cálculo del 'throughput' real (newRTT) es la media de todos los RTT medidos de paquetes recibidos durante este RTT:

```
if (state->snd_una > state->v_begseq) {
      simtime_t newRTT;
      if(state->v_cntRTT > 0) {
            newRTT = state->v_sumRTT / state->v_cntRTT;
      }
      else {
            newRTT = currentTime - state->v_begtime;
      }
      tcpEV << "Vegas: newRTT calculated: " << newRTT << "\n";
      state->v_sumRTT = 0.0;
      state->v_cntRTT = 0;
```

También para el cálculo del 'throughput' real se necesitan los bytes transmitidos desde que se envía un segmento hasta que su ack es recibido (rttLen).

Para el cálculo del 'throughput' esperado se utiliza el mínimo de todos los RTT medidos (baseRTT, que, como se ha visto en la página anterior, se actualiza cuando se hacen los cálculos de v_rtt_timeout, con paquetes que no han sido retransmitidos), y el tamaño de la ventana actual, que se calcula como proponen los autores proponen en su implementación para el x-kernel [53]:

```
// Se decide si incr/decr cwnd
if (newRTT > 0) {

      // rttLen: bytes transmitidos desde que se envia un segmento
      // y su ack es recibido
      uint32 rttLen = state->snd_nxt - state->v_begseq;

      // si hay un solo pkt en transito, se actualiza baseRTT
      if (rttLen <= state->snd_mss)
            state->v_baseRTT = newRTT;

      // actual = rttLen/(current rtt)
      uint32 actual = rttLen/newRTT;

      // expected = (current window size)/baseRTT
      uint32 expected;
      uint32 acked = state->snd_una – firstSeqAcked;
      expected = (uint32)((state->snd_nxt - firstSeqAcked) +
      std::min(state->snd_mss - acked, (uint32)0)) / state->v_baseRTT;

      // diff = expected - actual
      uint32 diff;
      // convertir v_baseRTT a double
      double baseRTT_double = conn->convertSimtimeToTS(state->v_baseRTT);
      diff = (uint32)(expected - actual) * (baseRTT_double/1000) + 0.5;
```

Con el valor calculado de la diferencia entre el 'throughput' esperado y el real, se hacen los cálculos propios de 'slow start' o 'congestion avoidance' para calcular el incr/decr de cwnd:

```
// Slow start
state->v_incr_ss = false; // reset
if (state->snd_cwnd < state->ssthresh) {

      // modificacion de cwnd durante slow-start cada 2 rtt
      state->v_inc_flag = !state->v_inc_flag;
      if (!state->v_inc_flag) {
            state->v_incr = 0;
      }
      else {
            if (diff > v_gamma) {
            /* Cuando el ratio real cae por debajo del esperado un cierto
            valor (umbral gamma), Vegas cambia de slow start a incr/decr
            lineal */

                  recalculateSlowStartThreshold();
                  state->snd_cwnd -= (state->snd_cwnd / 8);

                  if (state->snd_cwnd < 2*state->snd_mss)
                        state->snd_cwnd = 2*state->snd_mss;

                  state->v_incr = 0;

            } else {
                  state->v_incr = state->snd_mss; //incr 1 segmento
                  state->v_incr_ss = true;
            }
      }
```


```
	} // fin slow start

	// Cong. avoidance
	else {
		// Incr/decr 1 mss dependiendo de alfa y beta:
		if (diff > v_beta) {
			state->v_incr = -state->snd_mss;
		}
		else if (diff < v_alpha) {
			state->v_incr = state->snd_mss;
		}
		else
			state->v_incr = 0;
		} // fin cong. avoidance
	}

	// registrar valores beqseq y begtime para el rtt siguiente
	state->v_begtime = currentTime;
	state->v_begseq = state->snd_nxt;

} // fin seccion 'una vez por rtt'
```

Y se incrementa o reduce la ventana de congestión en base a lo que indique la variable que guarda el incremento (v_incr) para este rtt. Hay que destacar que, como se establece cuando incrementar una vez por rtt, hay que comprobar si se ha sobrepasado el umbral de 'slow start' (ssthresh) antes de que dicho rtt termine:

```
	// si cwnd > ssthresh, no incrementar
	if (state->v_incr_ss && state->snd_cwnd >= state->ssthresh) {
		state->v_incr = 0;
	}

	// incr/decr cwnd
	if (state->v_incr > 0) {
		state->snd_cwnd += state->v_incr;

		if (cwndVector)
			cwndVector->record(state->snd_cwnd);

		tcpEV << "Vegas: incr cwnd linearly" << "\n";
	}
	else if (state->v_incr < 0) {
		state->snd_cwnd += state->v_incr;

		if (state->snd_cwnd < 2*state->snd_mss)
			state->snd_cwnd = 2*state->snd_mss;

		if (cwndVector)
			cwndVector->record(state->snd_cwnd);

		tcpEV << "Vegas: decr cwnd linearly" << "\n";
	}
```

Finalmente, en receivedDataAck, si el ACK recibido es el primero o el segundo después de una retransmisión (v_worried guarda este valor), Vegas comprueba si debería haber ocurrido un 'timeout'. Si es así, se retransmite el segmento implicado. Esto forma parte del nuevo mecanismo de

retransmisiones propuesto [22].

```
// comprobar 1o y 2o ack despues de una rtx
if (state->v_worried > 0) {
      state->v_worried -= state->snd_mss;
      bool expired = checkRTTTimer();

      if (expired && (state->snd_max - state->snd_una > 0)) {
            state->dupacks = DUPTHRESH;

            ++state->v_transmits[(state->snd_una - (state->iss+1)) %
            state->v_maxwnd];
            tcpEV << "Vegas: retransmission (v_rtt_timeout) " << "\n";

            // retransmitir un segmento desde snd_una
            conn->retransmitOneSegment(false);
      }
      else
            state->v_worried = 0;
}
```

Obsérvese que, en el código anterior, se llama a la función checkRTTTimer, que comprueba si el tiempo desde que se envió el paquete más antiguo que no se ha reconocido todavía (snd_una) es superior al valor del 'timeout' que calcula Vegas con el 'fine-grained timer':

```
bool TCPVegas::checkRTTTimer() {
      simtime_t tSent =
                  state->v_sendtime[(state->snd_una - (state->iss+1)) %
                  state->v_maxwnd];
      simtime_t currentTime = simTime();

      simtime_t elapse = currentTime – tSent;

      if(elapse >= state->v_rtt_timeout)
            return true;
      else
            return false;
}
```

Una vez explicado el código correspondiente al método cuando se recibe un ACK que reconoce nuevos datos, se pasa a explicar el código del método cuando se recibe un ACK duplicado, receivedDuplicateAck.

En el nuevo mecanismo de retransmisiones propuesto por Vegas, cuando se recibe un ACK duplicado, de nuevo se comprueba si debería haberse producido un 'timeout' (obsérvese que se llama a la función checkRTTTimer). Si es así, se retransmite el segmento implicado, de la misma manera que cuando se reciben n (3) ACKs duplicados (como en Reno):

```
void TCPVegas::receivedDuplicateAck() {
      TCPBaseAlg::receivedDuplicateAck();
      simtime_t tSent =
                  state->v_sendtime[(state->snd_una - (state->iss+1)) %
                  state->v_maxwnd];
      simtime_t currentTime = simTime();
      int num_transmits =
```


```
                state->v_transmits[(state->snd_una - (state->iss+1)) %
                state->v_maxwnd];

        // comprobar si Vegas timeout
        bool expired = checkRTTTimer();

        // rtx si Vegas timeout || 3 dupacks
        if (expired || state->dupacks == DUPTHRESH) { // DUPTHRESH = 3
              uint32 win = std::min(state->snd_cwnd, state->snd_wnd);

              state->v_worried = std::min((uint32)2*state->snd_mss,
                                          state->snd_nxt- state->snd_una);

              if (num_transmits > 1)
                    state->v_rtt_timeout *= 2; // exp. Backoff.
              else
                    state->v_rtt_timeout += (state->v_rtt_timeout / 8.);


              // Vegas reduce cwnd si el segment rtx. se envio despues de la
              // ultima reduccion
              if (state->v_cwnd_changed < tSent) {
                    win = win/state->snd_mss;

                    if (win <= 3)
                          win = 2;
                    else if (num_transmits > 1)
                          win = win/2; //win <<= 1
                    else win -= win/4; // win -= (win >>2)

                    state->snd_cwnd =
                                win * state->snd_mss + 3 * state->snd_mss;

                    state->v_cwnd_changed = currentTime;

                    if (cwndVector)
                          cwndVector->record(state->snd_cwnd);

                    // reiniciar rtx. timer
                    restartRexmitTimer();
              }

              // retransmitir un segmento desde snd_una
              ++state->v_transmits[(state->snd_una - (state->iss+1)) %
              state->v_maxwnd];
              conn->retransmitOneSegment(false);

              if(num_transmits == 1) {
                    state->dupacks = DUPTHRESH;
              }
        }
```

Del código anterior, cabe destacar los siguientes aspectos:

- Se actualiza la variable v_worried para que se compruebe si debería ocurrir un 'timeout' cuando se reciban los dos próximos segmentos ACKs no duplicados (o menos, dependiendo de los segmentos que hay en tránsito).

- Si el segmento ha sido enviado más de una vez (num_transmits > 1), se realiza 'backoff' exponencial, siguiendo el Algoritmo de Karn [55].

- La ventana de congestión sólo se reduce si el segmento considerado se envió después de la última reducción (la variable v_cwnd_changed guarda el instante en que la ventana de congestión se reduce), porque, como ya se ha explicado, Vegas asume que cualquier pérdida de paquete que sucediera antes de la última reducción de la ventana de congestión no implica que la red esté congestionada para el tamaño actual de la ventana [22].

El comportamiento cuando se reciben más de n (3) ACKs duplicados es el mismo que en Reno, se incrementa en 1 MSS la ventana de congestión (ya que un ACK duplicado significa que un paquete ha dejado la red):

```
else if (state->dupacks > DUPTHRESH) { // DUPTHRESH = 3
      state->snd_cwnd += state->snd_mss;
      if (cwndVector)
            cwndVector->record(state->snd_cwnd);
}
```

Para terminar con la explicación de los detalles de implementación de TCP Vegas, sólo queda mencionar los detalles de los métodos recalculateSlowStartThreshold y processRexmitTimer, que no plantean muchos cambios.

El método processRexmitTimer de Vegas, llamado cuando se produce un 'timeout' de retransmisión, se diferencia del método implementado en Reno en que, en lugar de reiniciar la ventana de congestión a 1 MSS ('Maximum Segment Size', tamaño máximo de segmento), ésta se reinicia a 2 MSS.

El método recalculateSlowStartThreshold es el mismo que el implementado en Reno, que establece el umbral de 'slow start' a la mitad de la ventana de congestión actual (2 MSS como mínimo).

4.3.2. TCP Westwood

La clase TCPWestwood también hereda de TCPBaseAlg. En este caso, se han redefinido los métodos recalculateSlowStartThreshold, processRexmitTimer, receivedDataAck, receivedDuplicateAck, y se ha creado el método recalculateBWE:

```
class INET_API TCPWestwood : public TCPBaseAlg {
protected:
      TCPWestwoodStateVariables *&state;// alias to TCLAlgorithm's 'state'

      virtual TCPStateVariables *createStateVariables() {
            return new TCPWestwoodStateVariables();
      }
      virtual void recalculateSlowStartThreshold();
      virtual void processRexmitTimer(TCPEventCode& event);
      virtual void recalculateBWE(uint32 cumul_ack);
```

```
public:
      TCPWestwood();

      virtual void receivedDataAck(uint32 firstSeqAcked);
      virtual void receivedDuplicateAck();
      virtual void dataSent(uint32 fromseq);
};
```

El método recalculateBWE es la pieza clave en la implementación de TCPWestwood. En él se hace la estimación del ancho de banda. Para ello se utiliza el ratio de ACKs recibidos (cumul_ack) en un RTT, y el valor de dicho RTT (timeAck).

Como se puede ver, se aplica, además, el filtro pasa-bajo, para tener en cuenta sólo las componentes de baja frecuencia del ancho disponible, puesto que, como ya se ha explicado en la descripción de TCP Westwood (sección 2.7.2.2), la congestión ocurre cuando la tasa de tráfico de baja frecuencia excede la capacidad del enlace[42].

La variable w_lastAckTime guarda el valor en el que se recibió el anterior ACK (así se calcula el RTT para la estimación del ancho de banda, de manera similar a como se hace en TCPVegas, con la variable v_begtime).

```
void TCPWestwood::recalculateBWE(uint32 cumul_ack) {
      simtime_t currentTime = simTime();

      simtime_t timeAck = currentTime - state->w_lastAckTime;

      // Actualizar BWE
      if(timeAck > 0) {
            double old_sample_bwe = state->w_sample_bwe;
            double old_bwe = state->w_bwe;
            state->w_sample_bwe = (cumul_ack) / timeAck;
            state->w_bwe = 0.9047*old_bwe + 0.0476*(state->w_sample_bwe +
                        old_sample_bwe);
            tcpEV << "recalculateBWE(), new BWE=" << state->w_bwe << "\n";
      }
      state->w_lastAckTime = currentTime;

}
```

Este método es llamado cada vez que se recibe un ACK, tanto si éste reconoce nuevos datos (ACK no duplicado):

```
void TCPWestwood::receivedDataAck(uint32 firstSeqAcked) {

      TCPBaseAlg::receivedDataAck(firstSeqAcked);

      simtime_t tSent =
                  state->w_sendtime[(firstSeqAcked - (state->iss+1)) %
                  state->w_maxwnd];
      simtime_t currentTime = simTime();
      simtime_t newRTT = currentTime - tSent;

      int num_transmits =
                  state->w_transmits[(firstSeqAcked - (state->iss+1)) %
```

```
                state->w_maxwnd];

        // Actualizar RTTmin
        if (newRTT < state->w_RTTmin && newRTT > 0 && num_transmits == 1) {
              state->w_RTTmin = newRTT;
        }

        // cumul_ack: ack's acumulativos que ackean 2 o + paquetes cuentan
        // como 1, pues DUPACKs ya los ha tenido en cuenta
        uint32 cumul_ack = state->snd_una – firstSeqAcked;

        if ((state->dupacks * state->snd_mss) >= cumul_ack) {
              cumul_ack = state->snd_mss; // cumul_ack = 1:
        }
        else {
              cumul_ack -= (state->dupacks * state->snd_mss);
        }

        // comprobacion de seguridad: si los pasos previos se han seguido
        // correctamente cumul_ack debe ser > 2:
        if (cumul_ack > (2 * state->snd_mss))
              cumul_ack = 2 * state->snd_mss;

        recalculateBWE(cumul_ack);
```

Como si se trata de un ACK duplicado:

```
void TCPWestwood::receivedDuplicateAck() {

        TCPBaseAlg::receivedDuplicateAck();

        // cumul_ack: si es un dupack, cuenta como 1
        uint32 cumul_ack = state->snd_mss;

        recalculateBWE(cumul_ack);
```

Hay que destacar que TCP Westwood también registra el RTT mínimo (w_RTTmin) de todos los RTT de cada paquete (tiempo desde que se envía hasta que se recibe su ACK). Se puede observar que el cálculo se lleva a cabo en receivedDataAck, de manera similar a como se calcula en TCPVegas, o sea, sin tener en cuenta los ACKs de paquetes que han sido retransmitidos (Algoritmo de Karn [55]). Este valor, w_RTTmin, es utilizado, como se explicará más adelante, para establecer el valor del umbral de 'slow start', en el método recalculateSlowStartThreshold.

Para calcular el RTT mínimo, se utiliza el mismo mecanismo que el descrito para TCPVegas, registrando el tiempo en que se envía cada byte, en el método dataSent.

TCP Westwood no propone ninguna acción adicional más, o cambio, cuando se recibe un ACK que reconoce nuevos datos. El código que sigue en el método receivedDataAck es el mismo que para Reno, manteniendo los algoritmos de 'slow start' y 'congestion avoidance' sin modificar.

Sin embargo, cuando se recibe un ACK duplicado, TCP Westwood sí introduce más cambios. La estimación del ancho de banda (w_bwe) que, como se ha observado, se calcula en el método recalculateBWE, se utiliza para establecer los valores de la ventana de congestión y el umbral de 'slow start' después de un episodio de congestión.

Los episodios de congestión son: la recepción de n (3) ACKs duplicados (en el método receivedDuplicateAck):

```
if (state->dupacks == DUPTHRESH) // DUPTHRESH = 3 {

      if (state->snd_cwnd < state->ssthresh) { // Slow start
            state->w_a = state->w_a + 0.25;
            if (state->w_a > 4)
                  state->w_a = 4;
      }

      else { // Cong. avoidance
            state->w_a = 1;
      }

      recalculateSlowStartThreshold();

      // reiniciar cwnd a ssthresh si lo supera
      if (state->snd_cwnd > state->ssthresh)
            state->snd_cwnd = state->ssthresh;
```

Y el 'timeout' de retransmisión (método processRexmitTimer):

```
void TCPWestwood::processRexmitTimer(TCPEventCode& event) {

      TCPBaseAlg::processRexmitTimer(event);
      if (event == TCP_E_ABORT)
            return;

      if (state->snd_cwnd < state->ssthresh) { // Slow start
            state->w_a = state->w_a + 1;
            if (state->w_a > 4)
                  state->w_a = 4;
      }
      else { // Cong. avoidance
            state->w_a = 1;
      }
      recalculateSlowStartThreshold();
      if (state->ssthresh < 2 * state->snd_mss) {
            state->ssthresh = 2 * state->snd_mss;
            state->snd_cwnd = state->snd_mss;
      }
```

En el código anterior se puede observar que, tal y como se ha descrito en el apartado 2.7.2.2 en la sección “Procedimiento tras un episodio de congestión”, los valores a los que se establece la ventana de congestión y el umbral de 'slow start' difieren, dependiendo de si se está en fase de 'slow start' o en fase de 'congestion avoidance'.

Hay que destacar que, en la propuesta [23], los autores permiten ciertos grados de libertad a la hora de establecer los valores de cwnd y ssthresh, que permiten “tunear” el algoritmo. En esta implementación se ha utilizado el factor de reducción (w_a) que ellos mismos proponen. El objetivo de dicho factor es evitar una posible sobreestimación del ancho de banda disponible, un hecho bastante común en periodos prolongados de congestión.

Así pues, se puede observar en el código que, a mayor frecuencia de episodios de congestión durante el 'slow start' (una indicación de que ssthresh ha sido establecido a un valor demasiado elevado), más crece el factor de reducción: en el caso de los 3 ACKs duplicados crece 0.25 cada vez, y en el caso del 'timeout' crece 1 cada vez; el valor máximo es 4. Y siguiendo la misma línea de razonamiento, si se está en fase de 'congestion avoidance', el factor se restaura a 1, ya que significa que ssthresh ha sido establecido de manera adecuada y no hay necesidad de reducir el impacto del ancho de banda estimado, BWE.

Puede observarse, también, que para establecer el valor de ssthresh, se llama al método recalculateSlowStartThreshold. En él se calcula dicho valor, haciendo uso del ancho de banda estimado en recalculateBWE (w_bwe), del RTT mínimo calculado en receivedDataAck (RTTmin), y del factor de reducción que se acaba de describir (w_a):

```
void TCPWestwood::recalculateSlowStartThreshold() {
      // convertir w_RTTmin a double
      double RTTmin_double = conn->convertSimtimeToTS(state->w_RTTmin);

      state->ssthresh =
            (uint32)(state->w_bwe * (RTTmin_double/1000)) / (state->w_a);

    if (ssthreshVector)
        ssthreshVector->record(state->ssthresh);
}
```

De esta manera, terminada la explicación de los detalles más importantes del código de TCPWestwood, se procederá a continuación a la validación de las implementaciones de ambas variantes de TCP implementadas: TCP Vegas y TCP Westwood.

5. Validación y pruebas

Con el objetivo de comprobar que las implementaciones realizadas funcionan correctamente, se han llevado a cabo una serie de pruebas que se describen a continuación.

Se han realizado una serie de simulaciones, comparando el comportamiento, en términos de 'throughput' obtenido (el rendimiento), de los protocolos implementados, TCP Vegas y TCP Westwood, con el comportamiento de la implementación ya existente de TCP Reno.

El simulador que se ha utilizado en todas las simulaciones, como ya se ha explicado, es OMNeT++, en su versión 4.2.2 [45].

El escenario utilizado (observar Figura 5.1) enlaza dos terminales, cliente y servidor, mediante un canal con una tasa de datos de 100 Mbps.

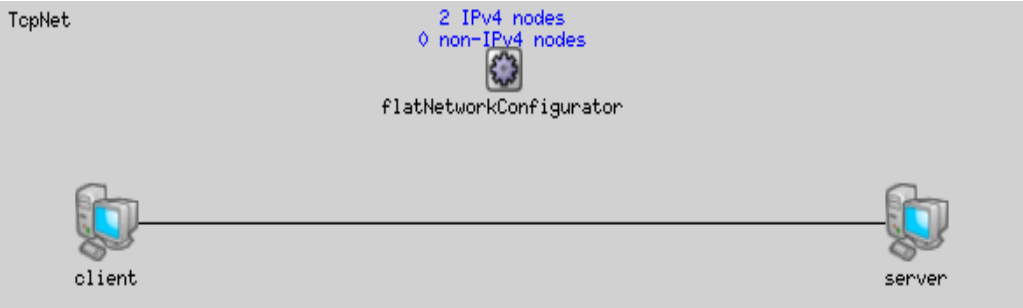

Figura 5.1: Escenario de simulaciones

En dicho escenario, el cliente se conecta al servidor enviándole una petición de 8B y, una vez establecida la conexión, el servidor comienza la transferencia del fichero, cuyo tamaño se ha establecido en 10MiB. El modo de transferencia de datos que se ha utilizado (recuérdese que INET Framework permite tres modos [50]) es “stream de bytes”, y se ha elegido un MSS ('Maximum Segment Size', tamaño máximo de segmento) de 1460.

Se han realizado simulaciones variando el retardo y la tasa de pérdidas de paquetes (PER, 'Packet Error Rate') del canal, tomando 5 muestras por configuración. Se han medido los tiempos y se ha calculado el ancho de banda que se ha alcanzado en el canal.

Para la ejecución de todas estas pruebas se ha utilizado un archivo de inicialización, omnetpp.ini, en el que se han establecido los parámetros de simulación mencionados.

```
[General]
network = TcpNet
repeat = 5
#record-eventlog = true

**.tcpApp[*].**.scalar-recording = true
**.tcpApp[*].**.vector-recording = true
**.mac.**.scalar-recording = true
**.mac.**.vector-recording = true
**.scalar-recording = true
```

```
**.vector-recording = true
**.per.param-record-as-scalar = true
TcpNet.retardo.param-record-as-scalar = true

**.numTcpApps = 1
**.tcpType = "TCP"
**.udpType = "UDP_None"
**.sctpType = "SCTP_None"

#Packet error rate
**.per = ${0, 0.001, 0.002, 0.005, 0.01, 0.02, 0.05}
#**.per = 0.005

#Retardo
TcpNet.retardo = 1ms
#TcpNet.retardo = ${1, 2, 5, 10, 20, 50, 100}ms

#TasaDatos
TcpNet.tasaDatos = 100Mbps

**.server.tcpApp[*].typename = "TCPSessionApp"
**.server.tcpApp[*].active = false
**.server.tcpApp[*].localPort = 1001
**.server.tcpApp[*].dataTransferMode = "bytestream"
**.server.tcpApp[*].tOpen = 0s
**.server.tcpApp[*].tSend = 0s
**.server.tcpApp[*].sendBytes = 10MiB    #File size
**.server.tcpApp[*].tClose = -1s
**.server.tcpApp[*].connectAddress = ""

**.client.tcpApp[*].typename = "TCPSessionApp"
**.client.tcpApp[*].active = true
**.client.tcpApp[*].connectPort = 1001
**.client.tcpApp[*].connectAddress = "server"
**.client.tcpApp[*].dataTransferMode = "bytestream"
**.client.tcpApp[*].localPort = -1
**.client.tcpApp[*].tOpen = 0s
**.client.tcpApp[*].tSend = 0s
**.client.tcpApp[*].sendBytes = 8B       #Request size
**.client.tcpApp[*].tClose = -1s

[Config TCPReno]
**.tcp.tcpAlgorithmClass = "TCPReno”
**.tcp.mss = 1460

[Config TCPVegas]
**.tcp.tcpAlgorithmClass = "TCPVegas"
**.tcp.mss = 1460

[Config TCPWestwood]
**.tcp.tcpAlgorithmClass = "TCPWestwood"
**.tcp.mss = 1460

sim-time-limit = 12000s
```

Como se puede observar, la metodología seguida ha sido dejar un parámetro fijo, retardo o porcentaje de pérdida de paquetes (PER), y variar el otro. Así, en las gráficas siguientes, cada punto representa la media de 5 muestras, para cada configuración, del ancho de banda medido en el canal.

En la Figura 5.2, se muestra cómo evoluciona el 'throughput' de TCP Reno, TCP Vegas y TCP Westwood, conforme se varía el valor del retardo, con un valor fijo del PER de 0.5%.

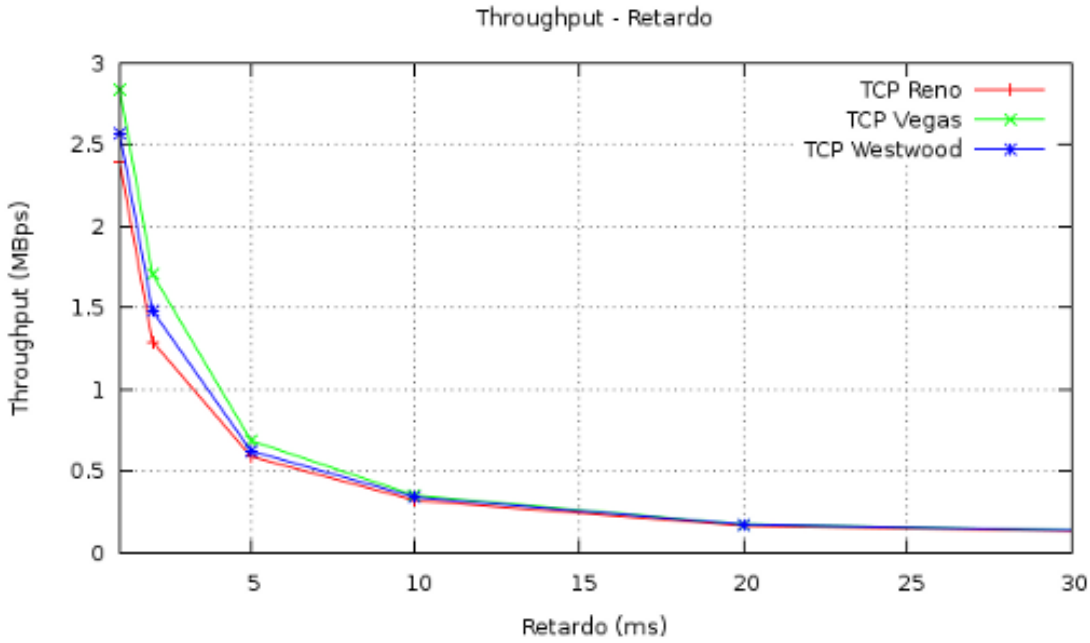

Figura 5.2: Comparativa Reno – Vegas – Westwood (PER de 0.5%)

Se observa que los tres protocolos tienen un comportamiento muy similar ante el aumento del retardo. Para valores del retardo más pequeños, se observa una pequeña mejora en el rendimiento de TCP Vegas y TCP Westwood, aunque conforme crece el valor del retardo, las diferencias se van reduciendo, y el rendimiento es prácticamente el mismo en los tres protocolos.

De igual manera, en la Figura 5.3, en la pñagina siguiente, se puede observar cómo evoluciona el 'throughput' de los tres protocolos a medida que va variando el PER ('Packet Error Rate'), con un valor fijo del retardo de 1ms.

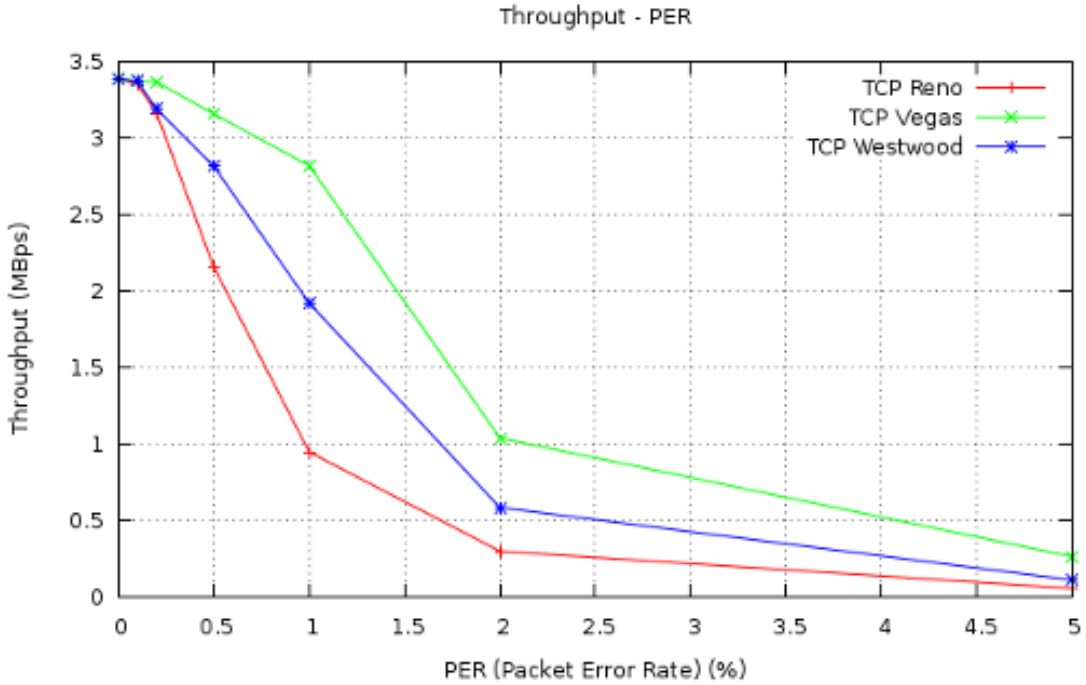

Figura 5.3: Comparativa Reno – Vegas – Westwood (Retardo de 1ms)

En esta gráfica también se observa que los tres protocolos evolucionan de una manera similar conforme se incrementa el porcentaje de paquetes perdidos, si bien las mejoras que presentan TCP Vegas y TCP Westwood con respecto a TCP Reno son algo más significativas, siendo Vegas el protocolo que mejor rendimiento experimenta.

En general, se han observado mejoras máximas en el rendimiento del 46% sobre Reno en TCP Vegas, y del 30% sobre Reno en TCP Westwood. A medida que los valores del PER ('packet Error rate') y del retardo aumentan, el comportamiento de los tres protocolos sigue una trayectoria similar, reduciéndose la mejora que se obtiene cuando los escenarios son más extremos (en términos de retardo y pérdida de paquetes).

A continuación, se van a comparar los resultados obtenidos en el presente trabajo con los que obtuvieron los autores en las respectivas propuestas. Los escenarios de simulación y las características no son idénticos, por lo que resultados obtenidos difieren, no significando esto, por lo tanto, que las implementaciones en sí sean incorrectas. Pasamos a analizar estas cuestiones.

En el caso de TCP Vegas, el simulador que se usa en la propuesta es un simulador basado en el *x-kernel*, desarrollado por los propios autores [22]. Se llevan a cabo diversas simulaciones, todas ellas dentro del escenario de simulación que se observa en la Figura 5.4, en la siguiente página.

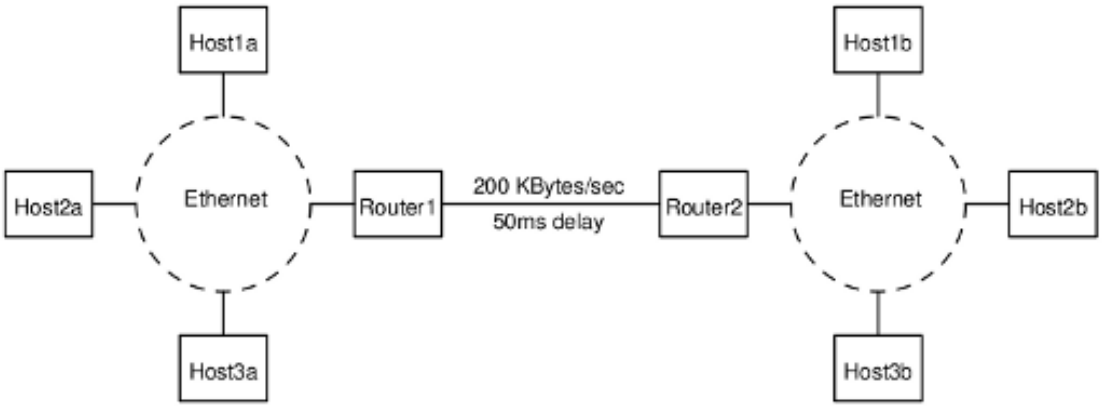

Figura 5.4: Escenario de simulaciones original de la propuesta de TCP Vegas [22]
Figura de [22], Figura 5

En la propuesta se simula una conexión TCP entre los hosts 'Host2a' y 'Host2b', en la que se transfieren 1MB de datos, y en la que el protocolo TRAFFIC (protocolo que soporta el simulador diseñado por los autores y que implementa tráfico TCP basado en *tcplib* [57]) genera un determinado tráfico de datos entre 'Host1a' y 'Host1b'.

Se simulan versiones de Reno, Vegas con valores de ($\beta = 1$) y ($\alpha = 3$), y Vegas con valores de ($\beta = 2$) y ($\alpha = 4$) (recuérdese que en la versión implementada para este trabajo, los valores de los 'thresholds' son estos últimos).

En la Tabla 5.1, en la siguiente página, se pueden observar los ratios del 'throughput' (comparados con Reno) obtenidos en las simulaciones de la propuesta original, y los que se han obtenido en este trabajo.

Los resultados mostrados de la propuesta son valores medios de 57 simulaciones, obtenidas usando diferentes semillas para *tcplib* y usando 10, 15 y 20 buffers en los routers. Mientras que los valores mostrados, obtenidos de las simulaciones realizadas para este proyecto, son los valores medios de los resultados mostrados en las gráficas anteriores (Vegas variando el PER, con retardo fijo de 1ms, y Vegas variando el retardo, con PER fijo de 0.5%).

	Propuesta original [22]		Resultados obtenidos	
Variante de TCP Vegas utilizada	Vegas-1,3	Vegas-2,4	Vegas-2,4 (retardo fijo: 1ms)	Vegas-2,4 (PER fijo: 0.5%)
Throughput Ratio	1.53	1.58	1.21	1.30

Tabla 5.1: Comaparación ratio 'throughput' entre propuesta original [22] y trabajo, para TCP Vegas

Los resultados recogidos en la tabla muestran una ganancia menor en las simulaciones realizadas con los protocolos implementados para este trabajo. Se considera que dicha diferencia es

debida a que las simulaciones para este trabajo se han realizado variando el retardo y el PER ('Packet Error Rate'), hasta valores elevados, de hasta 5% para el PER, o 100ms para el retardo. En las gráficas se ha observado que para estos valores, la mejora en el rendimiento con respecto a Reno se reduce, siendo el comportamiento de ambos protocolos muy similar. Por tanto, los valores recogidos para estos protocolos afectan a la media calculada, acercando el 'throughput' obtenido con Vegas al de Reno, reduciendo así la ganancia de Vegas sobre Reno. (Recuérdese que, como se ha mencionado, para valores no tan extremos del PER o del retardo, se han obtenido ratios del 'throughput' de Vegas sobre Reno alrededor de 1.46).

Además, hay que tener en cuenta las diferencias entre los escenarios de simulación. En general, en la propuesta original [22], de todas las simulaciones llevadas a cabo, incluidas pruebas reales sobre Internet, las mejoras de Vegas sobre Reno presentan entre un 37% y un 71% de ganancia en el 'throughput'. En nuestro caso, simulamos un canal de capacidad muy superior al utilizado por los autores en sus experimentos (10 Mbps vs. 1.6 Mbit/s), y se fuerzan pérdidas en el canal, ya que en la propuesta original las pérdidas solo son debidas a congestión en los routers.

Centrándonos ahora en el caso de la variante Westwood, en [58], los autores del protocolo llevan a cabo un análisis exhaustivo de su propuesta. Para eso realizan numerosas simulaciones en el simulador *ns-2*, de las cuales destacaremos las realizadas sobre el escenario que se muestra en la Figura 5.5.

La red simulada es una red mixta, con parte cableada de 10Mbps y un retardo inicial de 45ms, y parte inalámbrica de 2Mbps y retardo de 0.01ms. Se asume que los errores ocurren en ambas direcciones del enlace inalámbrico. Se realizan diferentes simulaciones variando tanto el PER (entre 0 – 5%), como el retardo de la parte cableada (entre 0 – 250ms), para representar la variedad de redes cableadas.

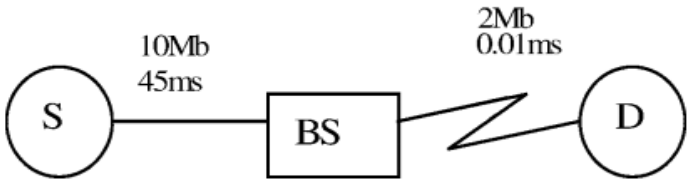

Figura 5.5: Escenario de simulaciones original de la propuesta de TCP Westwood [58]
Figura de [58]

En los análisis que se ofrecen para Westwood de estas simulaciones [58], no se muestran ratios medios del 'throughput', sino que se muestran las ganancias máximas obtenidas. Concretamente los autores presentan una ganancia máxima de Westwood sobre Reno del 394% con un valor del PER de 1%, y una ganancia máxima del 567% con un valor del retardo de 100ms. Estos valores distan bastante de los obtenidos en las simulaciones realizadas en este trabajo. No obstante, hay que tener en cuenta la diferente naturaleza de la red que se presenta en la propuesta. En ésta, el enlace inalámbrico, que es en el que ocurren las pérdidas, presenta un retardo fijo de 0.01ms, mientras que es el de la parte cableada el que se varía. En nuestro escenario de simulación, se ha variado tanto el PER como el retardo de todo el enlace, llevando las características de la red a situaciones extremas. En estos casos, tal y como se indica en en el paper original, es normal que la

ganancia de Westwood sobre Reno empiece a disminuir, puesto que el feedback que se utiliza para estimar el ancho de banda en Westwood llega demasiado tarde para que pueda considerarse significativo.

En general se considera que las simulaciones llevadas a cabo en la propuesta original están diseñadas para mostrar los mejores resultados que se pueden obtener con Westwood, y como en el presente trabajo no se han emulado dichas condiciones, los resultados difieren bastante.
erados enlaces 'lossy' (con alta probabilidad de pérdidas debidas al medio, y no a la congestión), y elevados anchos de banda. Además, en enlaces con diferentes flujos de datos de diferentes conexiones, TCP Westwood aumenta su ganancia sobre Reno cuando se trata de flujos mixtos (algunos usando Reno y otros Westwood), mientras que si se trata de escenarios en los que todas las conexiones utilizan la variante Westwood, el ancho de banda obtenido es muy similar al obtenido cuando todas las conexiones utilizan Reno.

Este razonamiento se ve apoyado, en parte, por los resultados que se observan en los análisis llevados a cabo en la propuesta [58], cuando las pruebas se realizan en escenarios reales de Internet. En este caso, se realizan una serie transmisiones entre la UCLA (University of California) y diferentes destinos en diferentes continentes, con diferentes retardos. (ver Figura 5.6).

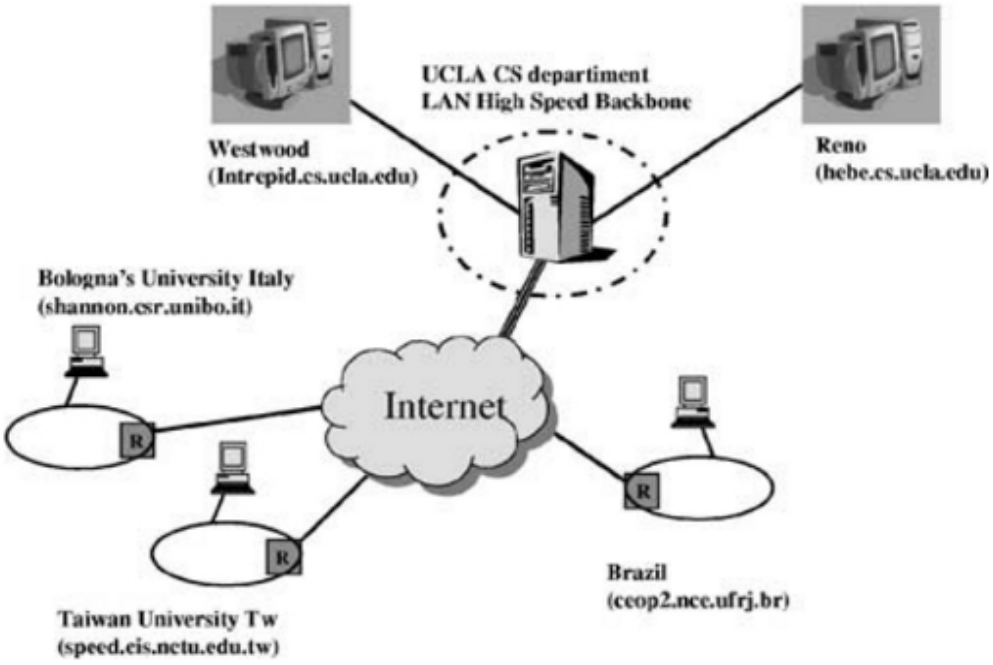

Figura 5.6: Escenario de simulaciones original, en Internet, de la propuesta de TCP Westwood [58]
Figura de [58]

Para estas pruebas, los autores sí muestran valores medios del 'throughput' obtenido con Westwood, comparándolo con Reno, por lo que sí se pueden mostrar los ratios obtenidos, y compararlos con los ratios de las simulaciones realizadas en el presente trabajo (observar. Tabla 5.2, en la siguiente página).

Los valores del 'throughput' de los cuales se han obtenido los ratios que se muestran, se obtuvieron, tal y como describen los autores, realizando la media de los resultados obtenidos de varias repeticiones de una transferencia, de un fichero de 10MB de tamaño. Se muestran los valores medios del ratio para los tres destinos usados en las pruebas.

En el caso de los ratios de las simulaciones realizadas en este trabajo, al igual que con Vegas, están obtenidos realizando la media de los valores que se muestran en las gráficas de Westwood, en el inicio de la sección.

	Propuesta original [58]	Resultados obtenidos	
	Westwood	Westwood (retardo fijo: 1ms)	Westwood (PER fijo: 0.5%)
Throughput Ratio	1.06 – 1.10 – 1.44	1.09	1.15

Tabla 5.2: Comaparación ratio 'throughput' entre propuesta original [58] y trabajo, para TCP Westwood

Se puede observar que respecto a Vegas, hay un mayor grado de similitud entre los valores obtenidos en el presente trabajo y los valores obtenidos de las pruebas reales en Internet de la propuesta original.

En general, en la propuesta original se considera que Westwood trabaja en condiciones óptimas cuando trabaja en enlaces inalámbricos (que pueden formar parte de redes mixtas), considerados enlaces 'lossy' (con alta probabilidad de pérdidas debidas al medio, y no a la congestión), y elevados anchos de banda. Además, en enlaces con diferentes flujos de datos de diferentes conexiones, TCP Westwood aumenta su ganancia sobre Reno cuando se trata de flujos mixtos (algunos usando Reno y otros Westwood), mientras que si se trata de escenarios en los que todas las conexiones utilizan la variante Westwood, el ancho de banda obtenido es muy similar al obtenido cuando todas las conexiones utilizan Reno.

6. Conclusiones

El protocolo TCP presenta una reducción del rendimiento en entornos inalámbricos, debido a la diferente naturaleza de las redes inalámbricas con respecto a las redes cableadas, para las que se creó y desarrolló el protocolo en un principio.

El trabajo presentado en esta memoria se centra en la implementación y posterior análisis de dos de las propuestas existentes como alternativa al protocolo TCP base, las variantes TCP Vegas y TCP Westwood, para la suite IP INET Framework del simulador OMNeT++.

Los resultados obtenidos validan las implementaciones realizadas, puesto que las mejoras que se observan en el rendimiento, alrededor de un 46% de mejora sobre Reno con TCP Vegas, y alrededor de un 30% de mejora sobre Reno con TCP Westwood, se consideran aceptables.

Estos resultados muestran que, si bien estas propuestas, Vegas y Westwood, presentan un cierto grado de mejora sobre Reno, todavía queda mucho camino por recorrer, puesto que las mejoras observadas se dan para determinadas condiciones de la red, reduciéndose dicho grado de mejora cuando las condiciones se tornan más agresivas, tanto en aumento del retardo, como en aumento del porcentaje de pérdida de paquetes. Este comportamiento se entiende, en tanto que, tanto TCP Vegas como TCP Westwood, proponen modificaciones únicamente en el lado emisor, sobre el algoritmo del control de la congestión. Además, parten de la implementación de Reno, sin modificar parte del comportamiento de ésta.

Se han propuesto numerosos trabajos que tienen en cuenta la dinámica de los paquetes cuando son redireccionados en los dispositivos intermedios de una conexión entre extremo y extremo. Es decir, el estudio para obtener mejoras en la gestión de los paquetes que hacen los routers y otros dispositivos intermedios, puede introducir mejoras significativas en el comportamiento de TCP, puesto que los dispositivos intermedios pueden “afinar” el feedback que utilizan los extremos TCP para asumir las características de la red en un determinado momento. Estas mejoras podrían ser muy útiles en TCP Vegas y TCP Westwod, si bien introducirían cierto grado de dificultad en el diseño de las implementaciones.

Para finalizar, cabe destacar que también existen en la literatura numerosas propuestas, que se han venido estudiando y desarrollando en los últimos años, que plantean alternativas al protocolo TCP más radicales, no suponiendo únicamente leves modificaciones de las implementaciones base. Muchas de estas propuestas se presentan prometedoras, si bien no hay que olvidar los problemas de integración y compatibilidad que suponen nuevos protocolos, si presentan mecanismos muy diferentes a los establecidos.

Como trabajo futuro consideramos que sería interesante proceder al desarrollo de otras variantes de TCP, así como de alternativas a este protocolo, y realizar un estudio comparativo más amplio.

7. Bibliografía

[1] J. Postel, USC/Information Sciences Institute, "Transmission Control Protocol", RFC 793, September 1981. Disponible en: http://www.ietf.org/rfc/rfc793.txt

[2] Dr. Wei Liu, Carolyn Matthews, Lydia Parziale, Nicolas Rosselot, Chuck Davis, Jason Forrester, David T. Britt, “TCP/IP Tutorial and Technical Overview”, p. 149-170, IBM Redbooks, December 2006.

[3] “Transmission Control Protocol”. Disponible en http://en.wikipedia.org/wiki/Transmission_Control_Protocol

[4] Andrew S. Tanenbaum, “Computer Networks” (Fourth de.), Prentice Hall, 2003-03-17.

[5] Douglas E. Comer, “Internetworking with TCP/IP: Principles, Protocols and Architecture. 1 (5th ed.), Prentice Hall, 2006.

[6] David D. Clark, MIT Laboratory for Computer Science, Computer Systems and Comunications group, “Window and Acknowledgement Strategy in TCP”, RFC 813, July 1982. Disponible en: http://www.ietf.org/rfc/rfc813

[7] John Nagle, Ford Aerospace and Communications Corporation, “Congestion Control in IN/TCP Internetworks”, RFC 896, January 1984. Disponible en: http://www.ietf.org/rfc/rfc896

[8] M. Allman, NASA Glenn/Sterling Software, V. Paxson, ACIRI / ICSI, W. Stevems, Consultant, “TCP Congestion Control”, RFC 2581, April 1999. Disponible en: http://www.ietf.org/rfc/rfc2581

[9] V. Jacobson, “Congestion Avoidance and Control”, Computer Communication Review, vol. 18, no. 4, p. 314-329, August 1988. Disponible en: ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z

[10] V. Jacobson, “Modified TCP Congestion Avoidance Algorithm”, end2end-interest mailing list, April 30, 1990. Disponible en: ftp://ftp.isi.edu/end2end/end2end-interest-1990.mail

[11] R. Braden, “Requirements for Internet Hosts – Communication Layers”, STD 3, RFC 1122, October 1989. Disponible en: http://www.ietf.org/rfc/rfc1122

[12] W. Stevens, NOAO, “TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast recovery Algorithms”, RFC 2001, January 1997. Disponible en: http://www.ietf.org/rfc/rfc2001

[13] M. Allman, BBN/NASA GRC, S. Floyd, ICIR, C. Partridge, BBN Technologies, “Increasing TCP's Initial Window”, RFC 3390, October 2002. Disponible en: http://www.ietf.org/rfc/rfc3390

[14] Vinton Cerf, Yogen Dalal, Carl Sunshine, “Specification of Internet Transmission Control Program”, RFC 675, December 1974. Disponible en: http://www.ietf.org/rfc/rfc675

[15] K. Ramakrishnan, TeraOptic Networks, S. Floyd, ACIRI, D. Black, EMC, “The Addition of Explicit Congestion Notification (ECN) to IP”, RFC 3168, September 2001. Disponible en: http://www.ietf.org/rfc/rfc3168

[16] W. Richard Stevens, Gary. R. Wright, “TCP/IP Illustrated: The Implementation, Vol. 2.”, Pearson Education, 1995.

[17] V. Jacobson, LBL, R. Braden, ISI, D. Borman, Cray Research, “TCP Extensions for High Performance”, RFC 1323, May 1992. Disponible en: http://www.ietf.org/rfc/rfc1323

[18] T. Henderson, Boeing, S. Floyd, ICSI, A. Gurtov, University of Oulu, Y. Nishida, WIDE Project, “The NewReno Modification to TCP's Fast Recovery Algorithm”, RFC 6582, April 2012. Disponible en: http://tools.ietf.org/rfc/rfc6582

[19] “Retransmission (data networks)”. Disponible en: http://en.wikipedia.org/wiki/Retransmission_(data_networks)

[20] M. Mathis, J. Mahdavi, PSC, S. Floyd, LBNL, A. Romanov, Sun Microsystems, “TCP Selective Acknowledgment Options”, RFC 2018, October 1996. Disponible en: http://www.ietf.org/rfc/rfc2018

[21] Matthew Mathis, Jamshid Mahdavi, Pittsburgh Supercomputing Center, “Forward Acknowledgment: Refining TCP Congestion Control”, 1996.

[22] Lawrence S. Brakmo, Sean W. O'Malley, Larry L. Peterson, Department of Computer Science, Universtity of Arizona, “TCP Vegas: New Techniques for Congestion detection and Avoidance”.

[23] S. Mascolo, C. Casetti, M. Gerla, S. S. Lee, M. Sanadidi, Computer Science Department – UCLA, “TCP Westwood: congestion control with faster recovery”, Los Angeles.

[24] James Kurose, Keith Ross, “Computer Networking – A Top-Down Approach”, 4th ed., p. 284, Addison Wesley, 2008.

[25] Ashish Natani, Jagannadha Jakilinki, Mansoor Mohsin, Vijay Sharma, “TCP for Wireless Networks”, University of Texas at Dallas, 2001.

[26] Kostas Pentikousis, “TCP in wired-cum-wireless environments”, IEEE Communications Surveys, 2000.

[27] Hari Balakrishnan, Venkata N. Padmanabhan, Srinivasan Seshan, Randy H. Katz, “A Comparison of Mechanisms for Improving TCP Performance over Wireless Links”, Computer Science Division, Department of EECS, University of California at Berkeley.

[28] P. Karn. “The Qualcomm CDMA Digial Cellular System”, USENIX Symp. on Mobile and Location-Independent Computing, p. 35-40, August 1993.

[29] S. Nanda, R. Ejzak, B. T. Doshi, “A Retransmision Scheme for Circuit-Mode Data on Wireless Links”, IEEE Journal on Selected Areas in Communications, 12(8), October 1994.

[30] E. Ayanoglu, S. paul, T. F. LaPorta, K. K. Sabnani, R. D. Gitlin. “AIRMAIL: A Link-Layer protocol for Wireless Networks”, ACM ACM/Baltzer Wireless Networks Journal, vol. 1, p. 47-60, February 1995.

[31] H. Balakrishnan, S. Seshan, R. H. Katz, “Improving Reliable Transport and Handoff

Performance in Cellular Wireless Networks", ACM Wireless Networks, 1(4), December 1995.

[32] Christina Parsa, J. J. Garcia-Luna-Aceves, "TULIP: A Link-Level Protocol for Improving TCP over Wireless Links", University of California, 1999.

[33] A. Bakre, B. R. Badrinath, "I-TCP: Indirect TCP for Mobile Hosts", 15th International Conf. On Distributed Computing Systems (ICDCS), May 1995.

[34] S. Keshav, S. Morgan, "SMART retransmission: Performance with Overload and Random Losses, IEEE INFOCOM, 1997.

[35] C. P. Fu, S. C. Liew, "TCP Veno: TCP Enhancement for Transmission over Wireless Access Networks", IEEE JSAC, vol. 21, no. 2, pp. 216-228, February 2004.

[36] K. Xu, Y. Tian, N. Ansari, "TCP-Jersey for Wireless IP Communications", IEEE JSAC, vol. 22, no. 4, pp. 747-756, May 2004.

[37] Christina Parsa, J. J. Garcia-Luna-Aceves, "Improving TCP Congestion Control over Internets with Heterogeneous Transmission Media", University of California.

[38] Kartik Chandran, Sudarshan Raghunathan, S. Venkatesan, Ravi Prakash, "A Feedback Based Scheme For Improving TCP Performance In Ad-Hoc Wireless Networks", University of Texas at Dallas, 1998.

[39] Z. Wang, J. Crowcroft, "Eliminating Periodic Packet Losses in 4.3-Tahoe BSD TCP Congestion Control Algorithm", vol. 22, no. 2, p. 9-16, ACM Computer Communication Review, April 1992.

[40] R. Jain, "A Delay-Based Approach for Congestion Avoidance in Interconnected Heterogeneous Computer Networks", ACM Computer Communication Review, vol. 19, no. 5, p. 56-71, October 1989.

[41] Z. Wang, J. Crowcroft, "A New Congestion Control Scheme: Slow Start and Search (Tri-S)", ACM Computer Communication Review, vol. 21, no. 1, p. 32-43, January 1991.

[42] J. C. Hoe, "Improving the Start-up Behavior of a Congestion Control Scheme for TCP", ACM SIGCOMM'96, Stanford, CA, USA, August 1996.

[43] "Bilinear Transform". Disponible en: http://en.wikipedia.org/wiki/Bilinear_transform

[44] M. Allman, NASA GRC/BBN, H. Balakrishnan, MIT, S. Floyd, ACIRI, "Enhaancing TCP's Loss Recovery Using Limited Transmit", RFC 3042, January 2001. Disponible en: http://tools.ietf.org/rfc/rfc3042

[45] "OMNeT++ Network Simulation Framework". Disponible en: http://www.omnetpp.org/

[46] "OMNeT++ version 4.2. User Manual". Disponible en: http://www.omnetpp.org/doc/omnetpp/manual/usman.html

[47] "OMNEST". Disponible en: http://www.omnest.com/company.php

[48] “INET Framework”. Disponible en: http://inet.omnetpp.org/

[49] “Mobility Framework for OMNeT++”. Disponible en: http://mobility-fw.sourceforge.net/

[50] “INET Framework for OMNeT++. Manual”, Generated on June 21, 2012. Disponible en: http://inet.omnetpp.org/doc/INET/inet-manual-draft.pdf

[51] “TCPAlgorithm Class Reference”, INET Framework for OMNeT++/OMNEST. Disponible en: http://inet.omnetpp.org/doc/INET/doxy/class_t_c_p_algorithm.html#details

[52] Implementation of U. of Arizona's TCP Vegas, based on USC's NetBSD-Vegas. Disponible en: http://www-rp.lip6.fr/ns-doc/ns226-doc/html/tcp-vegas_8cc-source.htm

[53] Implementación de TCP Vegas para x-kernel. Disponible en: ftp://ftp.cs.arizona.edu/xkernel/new-protocols/Vegas.tar.Z

[54] Implementación de TCP Westwood para ns-2. Disponible en: http://www.cs.ucla.edu/~nrl/hpi/tcpw/tcpw_ns2/old/tcp-westwood.cc

[55] Phil Karn, Craig Partridge, “Improving Round-Trip Time Estimates in Reliable Transport Protocols”, ACM SIGCOMM'87, p. 2-7, 1987.

[56] Jian Liu, Suresh Singh, “ATCP: TCP for Mobile Ad Hoc Networks”, IEEE Journal on Selected Areas In Communications, vol. 19, no. 7, July, 2001.

[57] P. Danzig and S. Jamin, “tcplib: A library of TCP internetwork traffic characteristics”, Computer Science Department, USC, Technical Report CS-SYS-91-495, 1991.

[58] Claudio Casetti, Mario Gerla, Saverio Mascolo, M. Y. Sanadidi, Ren Wang, “TCP Westwood: End-to-End Congestion Control for Wired/Wireless Networks”, Wireless Nerworks, 8, p. 467-479, Kluwer Academic Publishers, The Netherlands, 2002.