CAPÍTULO 4

EL PROTOCOLO TCP

4.1 Introducción al protocolo TCP

En la red Internet, el protocolo principal del nivel de transporte es el *Transmission Control Protocol (TCP)* [31]. La pila protocolaria TCP/IP viene representada en la figura 4.1 [9]:

HTTP-FTP-Telnet-DNS-POP3/IMAP SMTP-SNMP-RTP	***Aplicación***
TCP-UDP	***Transporte***
IP-ICMP	***Red***
Ethernet-IEEE 802.3-IEEE 802.5-FDDI-ISDN- Frame Relay-IEEE 802.11-SONET/SDH-PPP-HDLC	***Network Access***

Figura 4.1: Niveles del TCP/IP

El TCP, definido inicialmente en el RFC 793 [32], ha sido modificado y corregido en el curso de los años para adaptarlo a cada tipología de red (RFC 1122, RFC 1323, RFC 2018, RFC 2581, RFC 3782). Actualmente, la mayor parte de las investigaciones y de los estudios envueltos en el TCP investigan el comportamiento de

este protocolo sobre las redes wireless y es de esto el argumento que se verá expuesto en los próximos capítulos.

El protocolo TCP ha sido proyectado para proporcionar un flujo fiable de byte, desde la fuente hasta el destino, sobre una red no fiable. Mientras que el protocolo IP, inferior al protocolo TCP, proporciona un servicio de tipo *best-effort*, es decir, "hace lo mejor" para entregar los datagramas entre dos servidores en comunicación, no logra garantizar que los paquetes alcancen efectivamente un destino y que se respete el orden de entrega y la integridad de los datos.

Por tanto, el protocolo TCP beneficiándose del servicio del IP, proporciona un servicio fiable de transferencia de datos entre los procesos. Sus principales funciones son:

- Reconstruir el flujo originariamente transmitido en el caso de fenómenos de pérdida, duplicación o entrega fuera de secuencia de las unidades informativas y de congestión;
- Adoptar técnicas para el control de flujo y congestión [4].

Con el control de flujo, el TCP previene que el emisor de una comunicación envíe mensajes superiores a la capacidad actual del buffer del destino, mientras que con el control de congestión, limita la cantidad de datos introducidos en la red de manera que se evita sobrecargar a los encaminadores y la consiguiente pérdida de paquetes.

Además, el TCP efectúa la *multiplexación* de más flujos informativos de usuario sobre la misma conexión de transporte y soporta transmisiones *full-duplex*. La conexión TCP es de tipo *point-to-point*, es decir, entre un único receptor y un único emisor. Por lo tanto, el llamado *multicasting* no es realizable con el TCP. Se dice que: *"Para el TCP, dos servidores son una compañía, tres son multitud"* [4].

El servicio ofrecido por el TCP se llama *byte-oriented* en cuanto los niveles superiores escriben o leen de los byte de la conexión TCP, pero el mismo TCP no transmite los byte uno cada vez, sino que los organiza en un paquete llamado *segmento* (figura 4.2). La cantidad de datos que pueden ser extraídos e insertados en segmentos vienen limitados por la dimensión máxima del segmento (MSS, Maximum Segment Size) que depende de la implementación del TCP relativa al determinado sistema operativo (valores usuales son 1500, 536, 512 byte) [10].

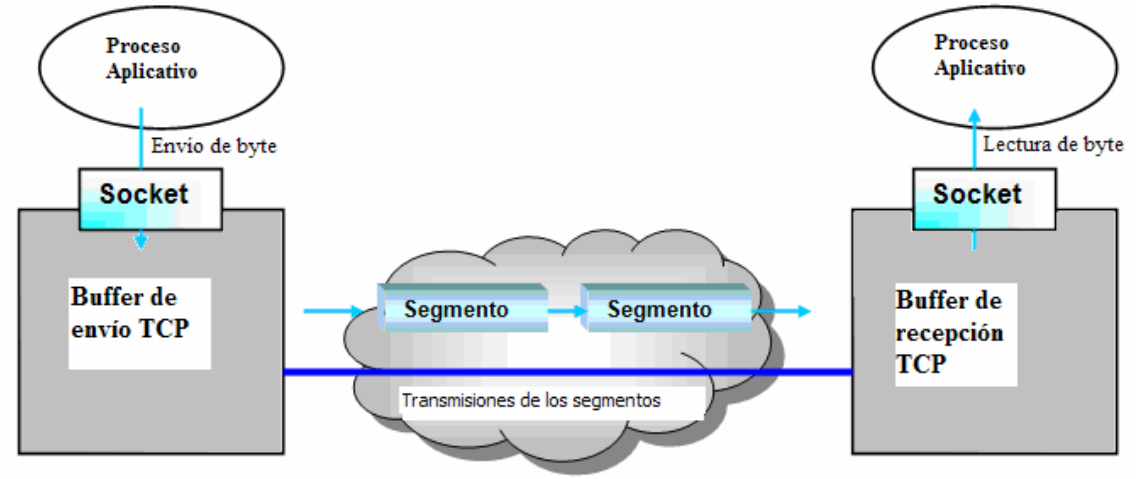

Figura 4.2: Servicio *byte-oriented* del TCP

Establecida la conexión TCP, el proceso *cliente* pasa una corriente de byte a través del *socket* (la puerta del proceso). Una vez atravesada la puerta, el TCP direcciona estos bytes en un buffer de envío (*send buffer*) y de vez en cuando se extraen algunos y los organiza en un segmento adhiriéndolos a una cabecera TCP. El segmento viene bajado al nivel de red y encapsulado en un datagrama IP, este último viene después enviado a la red. En el otro extremo, el TCP recibe los segmentos y los pone en el buffer de recepción, del cual la aplicación lee la corriente de bytes [10].

4.2 El formato del segmento TCP

La estructura del segmento TCP [9] viene mostrada en la figura 4.3:

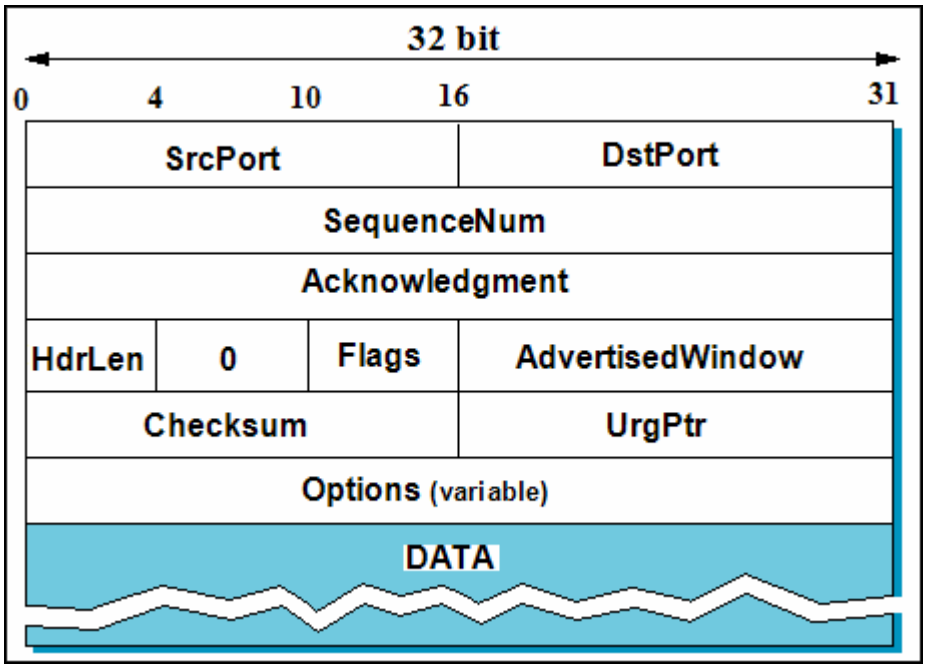

Figura 4.3: Formato del segmento TCP.

Este consta de un campo de cabecera (header) y de un campo de datos (payload).

- La cabecera comprende números de puerta fuente (*Source Port*) y de destino (*Destination Port*) que, juntos a los valores de la dirección IP fuente y destino, identifican unívocamente una conexión TCP y hacen posible la comunicación entre dos procesos en servidores distintos (esto permite incluso tener más conexiones entre dos servidores iguales, dado que cada uno está identificado unívocamente).
- En la cabecera hay otros dos campos: los campos *Sequence Number y Acknowledgment Number* que tienen, cada uno, longitud de 32 bit y se utilizan para hacer fiable el servicio de transferencia de datos.

El TCP, de hecho, ve los datos como un flujo de byte no estructurado pero ordenado: el número de secuencia para un segmento es el número del primer byte del segmento al interno del flujo.

El Sequence Number contiene el número del primer byte de datos contenido en el segmento; el campo Acknowledgment Number, sin embargo, contiene el número de secuencia del sucesivo byte esperado por el destinatario.

- Viene después el campo *Advertised Window* que se utiliza para el control del flujo. Indica el número de byte que el receptor está dispuesto a aceptar. Este ocupa 16 bit en la cabecera TCP.

Estos tres campos (SequenceNum, Acknowledgment, Advertised Window) son una parte crítica del servicio de transferencia fiable para los datos del TCP.

- El campo *Header Length (HdrLEN)* expresa la longitud de la cabecera TCP en palabras de 32 bit. Es necesario tener en cuenta en recepción esta información, ya que el campo *Option* es de dimensiones variables. Son 4 bit y debe ser al menos igual a 5 (20 byte, de hecho, es la longitud mínima de la cabecera de un segmento TCP).
- En la cabecera encontramos, además, el campo *Flags*, que contiene 6 bit, los cuales describimos a continuación:

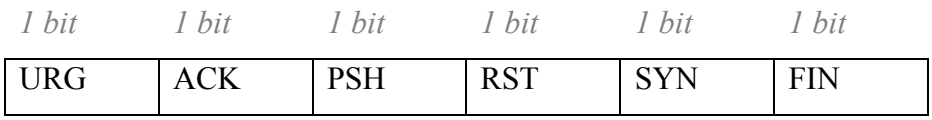

Figura 4.4: Campo Flags

- El flag URG (*Urgent Pointer*): es usado para indicar que en el relativo segmento hay datos que la unidad superior del remitente ha indicado como urgentes. La dislocación (offset) del último byte de estos datos urgentes se indica en el campo *urgent data pointer* a 16 bit.
- El flag ACK (*Acknowledgment*): sirve para indicar que el valor portado en el relativo campo es válido. Este bit vale 1 si el campo *ack number* contiene un dato válido, sino, quiere decir que el segmento TCP no contiene una verificación y el campo ack number se ignora.
- El flag PSH (*Push*): cuando está puesto a 1 le indica al protocolo TCP que transmita todos los datos presentes en el buffer de recepción y los datos apenas recibidos, sin esperar el rellenado del buffer.

- El flag RST (*Reset*): este bit puesto a 1 indica que es necesario restablecer la conexión ya que se hace inestable debido a una avería o cualquier otro motivo. A menudo viene usado para rechazar un intento de conexión.
- El flag SYN (*Synchronization*): sirve para identificar los mensajes que instauran la comunicación, cuando su valor es 1.
- El flag FIN (*Final*): puesto a 1 indica que el emisor no tiene más datos para enviar, cerrando la conexión TCP.

Entre el campo Header Length y el campo Flags hay 6 bit que no se utilizan.

- La cabecera comprende también, un campo *Checksum*, de 16 bit, que verifica la corrección de la cabecera TCP, de los datos TCP y de un pseudo preámbulo que contiene la dirección IP de fuente y destino. El número de byte del segmento TCP es el identificativo para el protocolo TCP que vale 6.
- El campo *Urgent Pointer (UrgPtr)* indica el desplazamiento en byte respecto al primer byte del segmento donde finalizan los datos urgentes. Tan solo sirve si el flag URG vale 1.
- El campo *Option*, finalmente, es un campo de longitud variable que puede contener subcampos diseñados con la finalidad de proporcionar un método para alcanzar nuevos servicios extras. Como, por ejemplo, para negociar entre emisor y receptor la dimensión máxima del segmento. En el RFC 1323 [33] se propone la opción *Window scale* que indica un factor de escala de la ventana para el uso en las redes a alta velocidad [26] [33].

4.3 La gestión de la conexión TCP

Sabemos ya que el TCP es un protocolo orientado a conexión; esto implica que una conexión entre emisor y destinatario deba ser instaurada antes de iniciar un

intercambio de datos. Las modalidades de las creaciones de las conexiones son significativas y no despreciables dado que esta fase podría aportar un posterior retardo a la transmisión. Con el fin de establecer una conexión, el TCP usa un procedimiento denominado "*three way handshake*", según el cual los dos servidores se intercambian paquetes (figura 4.5).

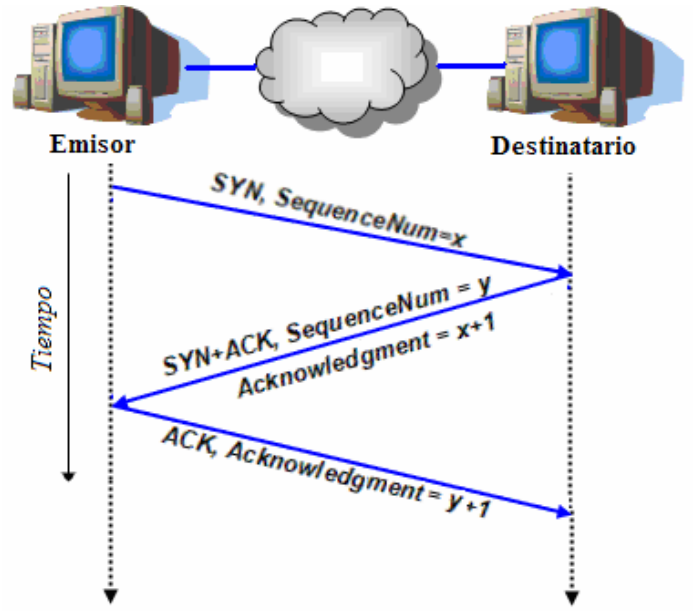

Figura 4.5: Three way handshake del TCP

El primer paso de este procedimiento prevé que el emisor envíe un segmento con el bit SYN y ponga en el campo SequenceNum el número de secuencia *(x)* que utilizará para transmitir los datos. El valor del número de secuencia es un número pseudocasual (generalmente distinto de 0) de modo que evita que los segmentos de la nueva conexión se confundan con los de una conexión precedente entre los mismos servidores.

Si el destinatario acepta comunicarse con el emisor en modalidad bidireccional, le responde con un segmento de autorización y destina, tanto un buffer como las variables, a la conexión. En este segmento se ponen simultáneamente a 1 los campos SYN y ACK; se impone *(y)* en el campo SequenceNum, con la finalidad de enviar un número de secuencia válido para la comunicación en sentido contrario; y los Acknowledgment con *x+1,* para efectuar la comprobación del número de secuencia del

emisor. De este modo, emisor y destinatario concuerdan los parámetros de la conexión, como el SecuenceNum y Acknowledgment.

Para la recepción del segmento que autoriza la conexión, también el emisor destina un buffer y unas variables. Desde este momento puede sucederse el intercambio de segmentos entre las dos entidades TCP; los segmentos tendrán el flag SYN a 0 dado que la conexión está establecida.

Cualquiera de los dos procesos participantes puede cerrar la conexión en cualquier momento enviando un segmento con el flag FIN puesto a 1. La liberación de la conexión bidireccional requiere que se libere ambas conexiones unidireccionales: cada una de las partes debe enviar al otro el segmento con el bit FIN puesto a 1, y esto debe ser verificado con otro segmento con el bit ACK a 1 (figura 4.6). Cuando termina la conexión, los recursos utilizados (buffer y variables) se liberan [4] [26].

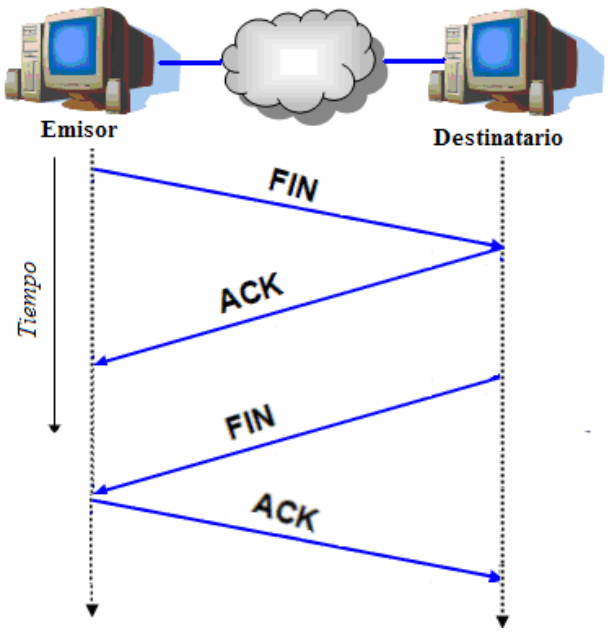

Figura 4.6: Cierre de una conexión TCP.

4.4 El control de flujo en el TCP

En cada extremidad de la conexión los servidores reservan un buffer de transmisión y recepción. Cuando los datos llegan al destinatario correctamente y en secuencia, vienen colocados en el buffer. El proceso asociado los leerá, pero no necesariamente en el instante de su llegada porque podría estar ocupado en otras operaciones. Si la velocidad de lectura de los datos de la aplicación receptora es inferior a la de transmisión del emisor, puede suceder que el buffer de recepción se sature y los segmentos se pierdan [10].

Para prevenir este fenómeno, el TCP proporciona un servicio de control de flujo, es decir, un servicio de adaptación de la velocidad de transmisión a la de lectura, siguiendo un algoritmo de tipo *sliding window (ventana deslizante).* Este algoritmo se basa en el mantenimiento en el emisor de una variable llamada *receive window*, cuyo valor viene actualizado por el campo Advertised Window presente en los segmentos TCP, y que viene utilizado para limitar el número máximo de byte en transmisión sin confirmación, es decir, "en vuelo" (o alternativamente, los *outstanding packet*) [4]. Recordamos que el campo Avertised Window indica la cantidad de espacio disponible en el buffer de recepción, es decir, el número de byte que el receptor está dispuesto a aceptar sin que el buffer se sature.

El algoritmo ventana deslizante obtiene una transferencia fiable a través de la utilización de las respuestas (acknowledgment) y de los *timeout* (pausa). Después de cada segmento transmitido, la fuente activa un reloj llamado *timeout* y en caso de que en este intervalo de tiempo no se reciba un ACK como confirmación de la correcta recepción del segmento transmitido, se encarga de retransmitirlo. Además, si la fuente ha enviado el máximo número de segmentos permitidos por la variable receive window, está debe esperar a recibir, al menos, un ACK para poder transmitir otros datos. Este comportamiento es común tanto a todos los algoritmos de tipo ARQ (*Automatic Repeat reQuest*) como a los de tipo ventana deslizante [4]. Un ejemplo de funcionamiento del algoritmo sliding window, en el caso de una ventana igual a 3, se muestra en la figura 4.7.

Para optimizar la eficiencia del protocolo se utilizan ACK de tipo acumulativo: en el caso de que un ACK se perdiese en la red, la recepción de un ACK con el valor del Acknowledgment mayor de aquel esperado, confirma que todos los datos precedentes se han recibido correctamente.

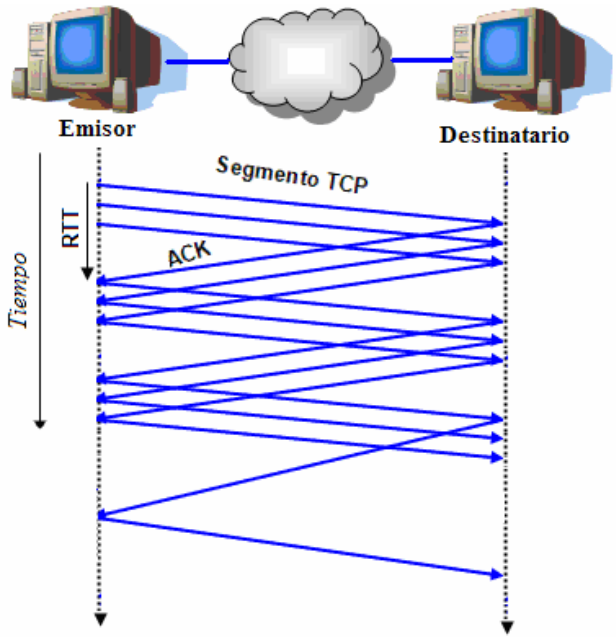

Figura 4.7: Algoritmo ventana deslizante con ventana igual a 3.

4.4.1 Estimación del tiempo de *round trip* y de *timeout*

Se ha visto como el TCP, para recuperar pérdidas de paquetes, utiliza un mecanismo *timeout*/retransmisión. Para que tal mecanismo resulte eficiente es necesario calcular un adecuado tiempo de *timeout* (tiempo de espera) [34].

Obviamente, el *timeout* debe ser mayor que el tiempo de *round trip* (RTT, *Round Trip Time*), es decir, del tiempo transcurrido desde que un segmento viene enviado hasta que es verificado por el destinatario. La mayor parte de las versiones del TCP efectúa una única medida cada vez de RTT (*SampleRTT*), es decir, en cada instante de tiempo el SampleRTT viene estimado por los segmentos no retransmitidos y por uno solo de los segmentos transmitidos. El valor de SampleRTT fluctuará de segmento en segmento a causa de la congestión en los routers y a la carga variable sobre los terminales. Es necesario, por tanto, obtener una media pesada de estos diversos valores,

llamada *EstimatedRTT*. Ya que las muestras recientes reflejan mejor la actual congestión de la red, se efectúa una media de tipo EWMA (*Exponential Weighted Moving Average*), cuya expresión es:

$$EstimatedRTT = (1 - \alpha) * EstimatedRTT + \alpha * SampleRTT \quad (4.1)$$

donde un valor típico de α es 0.125.

Para tener una estimación de RTO, es necesario medir su variabilidad *DevRTT*, es decir, cuanto SampleRTT se aleja de EstimatedRTT:

$$DevRTT = (1 - \beta) * DevRTT + \beta * | SampleRTT - EstimatedRTT | \quad (4.2)$$

donde el valor recomendado de β es 0.25.

El método propuesto por la RFC 2988 [36] para determinar el intervalo de timeout para las retransmisiones es el siguiente:

$$TimeoutInterval = EstimatedRTT + 4 * DevRTT \quad (4.3)$$

El timeout debe ser mayor (o igual) a EstimatedRTT, pero no mucho, ya que si no, en el caso de que un paquete se pierda, el TCP no lo retransmite velozmente introduciendo un posterior retardo de transferencia. Se impone, por tanto, el timeout igual al EstimatedRTT más un gran margen, cuando hay fluctuaciones amplias en los valores de SampleRTT y pequeños en caso contrario (en concordancia con la expresión 4.3).

4.5 El control de congestión del TCP

El control de la congestión representa otra componente extremadamente importante en el protocolo TCP. El problema de la congestión se presenta cada vez que la carga ofrecida a la red es superior a la que ésta puede administrar [2]. Una red se dice congestionada cuando hay demasiados paquetes que se disputan el mismo enlace

causando la saturación de los buffer de los routers y las consiguientes pérdidas de paquetes. El paquete perdido da lugar a un evento de pérdida al emisor (un timeout o la recepción de ACK duplicados, como veremos en seguida) que es considerado por el emisor como una indicación de congestión.

El protocolo TCP utiliza un control de la congestión *end-to-end*, ya que la capa IP no proporciona ningún *feedback* explícito alrededor de la congestión de la red [26]. El procedimiento seguido por el TCP consiste en enviar en red los segmentos hasta provocar la congestión para después tratar de administrarla: cada emisor limita el ritmo con el que introduce tráfico en su conexión, en función de la congestión percibida en la red. Si un emisor TCP nota que existe poca congestión en el trayecto entre él y el destino, entonces aumenta el ritmo de la transmisión; al contrario, si percibe que la red está congestionada, lo reduce. Esta estrategia es adoptada porque se hace necesario utilizar al máximo los recursos compartidos (anchura de banda de los enlaces, buffer de recepción y de los conmutadores) para obtener otras prestaciones. El TCP estima la banda disponible para cada conexión y transmiten los segmentos en base a ella [4].

Así como el valor del campo AdvertisedWindow regula la ventana de los paquetes en vuelo en el algoritmo ventana deslizante para el control de flujo, el TCP define un parámetro interno llamado *CongestionWindow* para implementar el control de congestión. Esta variable impone una limitación adicional al número de paquetes en vuelo. Específicamente, la cantidad de los datos no confirmados ("en vuelo") que un servidor puede tener al interno de una conexión TCP no debe superar el mínimo entre CongestionWindow y AdvertisedWindow, de modo que se satisfaga las exigencias, tanto del control de flujo como del de congestión.

Para concentrar la atención sobre el control de congestión, asumimos que el buffer de recepción del TCP sea bastante grande como para poder ignorar la limitación impuesta por la ventana de recepción; además, por simplicidad, medimos el AdvertisedWindow y la CongestionWindow en segmentos, en lugar de en byte.

El algoritmo de control de la congestión del TCP, introducido por primera vez por Van Jacobson [35], se basa en tres componentes principales: incremento aditivo, decremento multiplicativo (AIMD); *slow start;* y los mecanismos de *fast retransmit y fast recovery*.

4.5.1 *Additive Increase, Multiplicative Decrease (AIMD)*

La finalidad del control de congestión del TCP es aquella de hacer reducir al emisor su ritmo de envío (disminuyendo la dimensión de la CongestionWindow) cuando se verifica una pérdida [10].

El hecho de que un paquete no sea entregado a destino puede depender, sobretodo, del hecho que haya sido descartado por cualquier router por motivos de congestión y no por errores de transmisión. El evento timeout está directamente conectado al nivel de congestión de la red, de hecho, cuando un paquete viene descartado por un router, al lado emisor de la conexión TCP le caduca un timeout y la confirmación del paquete perdido no llegará nunca.

El TCP utiliza el mecanismo AIMD (Additive Increase Multiplicative Decrease) para adaptar la CongestionWindow al nivel de congestión percibido. El TCP debe aumentar su ritmo de transmisión si no siente congestión: la fase llamada "incremento aditivo" *(additive increase)* aumento el valor de la CongestionWindow de 1 MMS cada vez que viene confirmado un número de segmentos iguales al actual valor de CongestionWindow. Esta fase de incremento lineal de la ventana de congestión se conoce también como "*congestion avoidance"* [37].

Para reducir la ventana de congestión, el TCP usa un procedimiento del tipo "decremento multiplicativo" *(multiplicative decrease)*, que en caso de un timeout debido a eventos de pérdidas o de fuertes congestiones, pone el valor de la variable *SlowStartThreshold* a la mitad del actual CongestionWindow, poniendo después la CongestionWindow a 1.

Desde el momento en que más conexiones transeúntes por los mismos routers congestionados experimentan eventos de pérdida y reducen su ritmo de transmisión, el efecto global debe ser aquel de evitar la congestión [4].

Recapitulando, un emisor TCP disminuye su ritmo de envío si nota congestión, viceversa, debe aumentar su ritmo de envío si no se percibe congestión alguna, es decir, cuando no se verifican eventos de pérdidas.

4.5.2 *Slow Start*

Cuando se inicia una nueva conexión TCP se hace necesario estimar la banda disponible en ese momento e incrementar, en consecuencia, la CongestionWindow.

Inicialmente, cuando la fuente no tiene ninguna idea de la banda disponible, la CongestionWindow se inicializa a 1 segmento. La fase de *Slow Start* incrementa, pues, tal valor exponencialmente, aumentándolo en una unidad por cada ACK recibido, es decir, duplicándolo a cada RTT, hasta alcanzar el valor asumido por el SlowStartThreshold. Hace falta considerar, en efecto, que poniendo la ventana de congestión igual a 1 segmento y sólo incrementándola linealmente, se corre el riesgo de bajo-utilizar la banda disponible y someterse a un largo retraso antes de que el ritmo de envío suba a un nivel aceptable. El mecanismo AIMD sólo es eficiente cuando el emisor TCP usa valores de CongestionWindow cercanos a la banda disponible que ofrecen en ese momento la red.

La fase de Slow Start acaba cuando el valor de CongestionWindow supera el del umbral SlowStartThreshold. Al principio de la conexión, este último parámetro es programado a valores muy altos, de modo que no influye en la consideración inicial de la banda disponible. Cuando ocurre un timeout, para poder tener conocimientos del nivel de congestión de la red, se actualiza la variable umbral, poniéndola a la mitad del valor actual del CongestionWindow (de acuerdo con el valor de la misma que deriva del decremento multiplicativo). El nuevo SlowStartThreshold determina la dimensión de la ventana a la que tiene que acabar la siguiente fase de Slow Start, y a la que tiene que iniciar la de Congestion Avoidance, donde es ejecutado el algoritmo AIMD [4] [37]. En el diagrama temporal de la figura 4.8 vienen representadas la evolución de la ventana de congestión del TCP y la variable umbral. Se hacen notar las fases de Slow Start que se suceden después de cada acontecimiento de timeout.

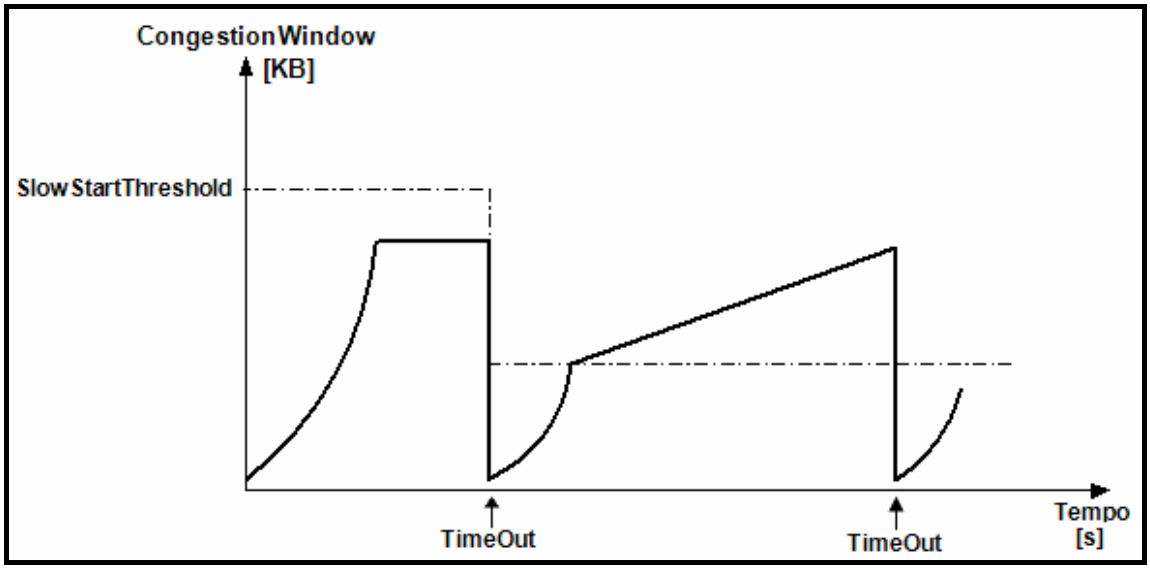

Figura 4.8: Evolución de la ventana de congestión del TCP.

4.5.3 *Fast Retransmit* y *Fast Recovery*

Para remediar el problema de los largos períodos de inactividad atados al timeout, se ha introducido el mecanismo de *Fast Retransmit*. Con este mecanismo el TCP no tiene innecesariamente que esperar a que venza el timeout para poder retransmitir un segmento. Está previsto que el receptor mande el ACK relativo a un segmento, aunque no se hayan recibido aún los segmentos anteriores a éste. El transmisor deduce del número de ACK duplicados qué recibe, si el segmento se ha perdido o bien está sufriendo solamente retrasos. En efecto, un ACK duplicado indica el hecho tal que el receptor no puede mandar un ACK relativo a un segmento llegado fuera de orden, ya que está esperando recibir de uno precedente.

El mecanismo de Fast Retransmit prevé, que en caso de se reciban que 3 ACK duplicados (3DUPACK) relativos al mismo segmento, se procede a la retransmisión del segmento, sin esperar a que venza el timeout. Este mecanismo permite beneficios en el throughput y reduce el número de timeout.

La versión del TCP denominada TCP Reno, después de haber recibido 3DUPACK, pone el SlowStartThreshold a la mitad del valor actual de la ventana de congestión, pero en vez de reducir bruscamente este última hasta llevarla al valor de 1 segmento, la reduce a la mitad solamente, eliminando así la fase de Slow Start siguiente ("retomada veloz", *fast recovery)*.

El motivo que lleva al control de congestión del TCP Reno a comportarse de forma distinta después de un acontecimiento de timeout y después de la recepción de tres ACK duplicados es que, en el segundo caso, aunque se haya perdido un paquete, la llegada de los 3DUPACK al remitente testimonia que la red es capaz de entregar, al menos, algún segmento, aunque otros se hayan perdido a causa de la congestión [4]. La recepción de los tres ACK duplicados por lo tanto, representa en todo caso un síntoma del alto nivel de congestión de la red, pero menos grave que los timeout.

Es interesante el que una vieja versión del TCP denominada TCP Tahoe, reduzca incondicionalmente la ventana de congestión a 1 segmento y entre en la fase de Slow Start después de cualquier acontecimiento de pérdida [4].

El diagrama temporal de la figura 4.8, exhibe el curso del CongestionWindow y el SlowStartThreshold en el caso en que se adopten los mecanismos de Fast Retransmit y Fast Recovery.

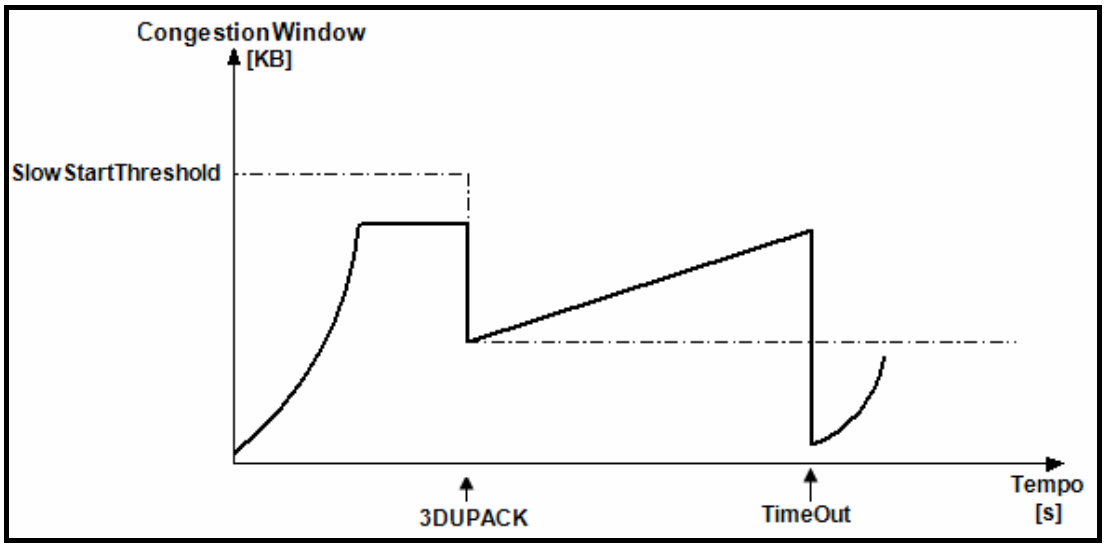

Figura 4.8: Evolución de la ventana de congestión del TCP con Fast Retransmit y Fast Recovery.

Además del Fast Recovery y al Fast Retransmit, el TCP Reno también le añade al TCP Tahoe el soporte opcional por los *retardados ACK*. Este mecanismo incrementa la eficiencia permitiendo al receptor de no generar el segmento de ACK para todos los segmentos que le llegan, sino sólo después de una cierta cantidad de ellos (ACK acumulativo) [38].

4.6 El control de la congestión en las otras versiones del TCP

Las variantes del TCP Reno se distinguen entre ellos por el modo de administrar la congestión.

El TCP NewReno [3] se comporta como el TCP Reno aunque se diferencia por el momento de salir de la fase Fast Recovery. Sin detenernos mucho en detalles, decimos que mientras el TCP Reno sale de la fase Fast Recovery en cuanto ha recibido el ACK del segmento perdido; el TCP NewReno sale de la misma cuando ha recibido las verificaciones de todos los segmentos pertenecientes a la misma ventana del segmento que ha causado el 3DUPACK. Este comportamiento permite no tener posteriores reducciones del CongestionWindow por los 3DUPACK muy cercanos, es decir, relativos a segmentos de la misma ventana.

El algoritmo de control de congestión propuesto por el TCP Vegas [39] intenta evitar la congestión manteniendo un buen rendimiento, puesto que advierte la congestión en los router y disminuye el ritmo de envío antes de que se produzca la pérdida de los paquetes. La inminente pérdida de paquetes viene prevista observando los RTT: cuanto mayor sean estos últimos, mayor es la congestión en los router.

Otras versiones como el TCP Westwood [40] y el Westwood+ [41], aportan, sobre todo, notables mejorías en el control de congestión en redes inalámbricas, estimando la banda disponible para reducir oportunamente el CongestionWindow cuando se verifica una congestión.