

10

CAPÍTULO

PARTE II. INTERNET, ARQUITECTURA TCP/IP,
PROTOCOLOS Y SERVICIOS



PROTOCOLOS EXTREMO A EXTREMO

- 10.1. Introducción
- 10.2. Protocolo de datagrama de usuario (UDP)
- 10.3. Protocolo de control de transmisión (TCP)
- 10.4. Modelado analítico de TCP

10.1. Introducción

Continuando con el estudio de la arquitectura TCP/IP, tras la presentación de la capa de red y de sus protocolos, en este capítulo abordaremos las funciones propias de la capa de transporte, considerando especialmente las soluciones adoptadas en los protocolos asociados. De igual manera a como sucede en el modelo OSI, la capa de transporte en TCP/IP es la primera de las capas denominadas extremo a extremo (Figura 10.1). Esto es, involucra directamente a las estaciones finales o *hosts* y no a los dispositivos intermedios de la subred (ver Capítulo 1). El diseño de esta capa, como comprobaremos en el capítulo, adopta el principio de diseño de situar la complejidad en los extremos (ver Apartado 8.1).

A diferencia de lo que ocurre en la capa de red, donde solo existe el protocolo IP para llevar a cabo las funcionalidades de direccionamiento y encaminamiento, para la capa de transporte se adopta una aproximación polarizada en la que se especifican dos alternativas: UDP y TCP. El primer protocolo se caracteriza por ofrecer, al igual que IP, un servicio no fiable y no orientado a conexión. Por el contrario, TCP ofrece un servicio orientado a conexión que incluye el control de flujo, control de errores y control de congestión.

En el siguiente apartado se explica el protocolo UDP. Frente a este, con una complejidad mucho mayor, en el apartado tercero se estudian los servicios ofrecidos por TCP. Además, en ese mismo apartado se consideran las variantes o «sabores» de TCP más relevantes. Nuestro estudio también incluye un modelado analítico de TCP, tanto en latencia como en tasa o velocidad de transmisión, que permitirá

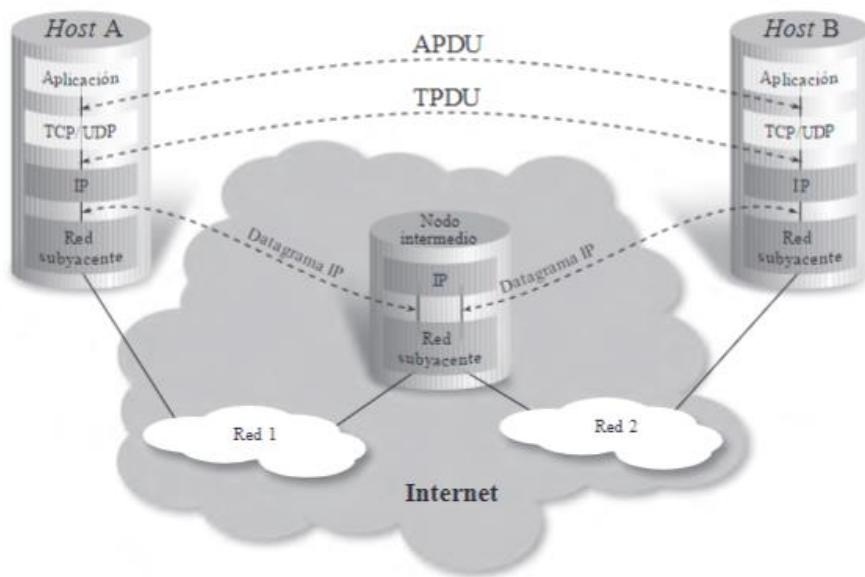


Figura 10.1. Transmisión extremo a extremo en Internet.

por un lado profundizar en su funcionamiento y por otro disponer, aunque solo sea de forma aproximada, de una metodología para la evaluación cuantitativa de sus prestaciones.

10.2. Protocolo de datagrama de usuario (UDP)

UDP se encapsula sobre IP, tomando el campo *protocolo* del datagrama IP el valor 17 (ver RFC 1700). UDP se especificó en el año 1980 en el RFC 768, siendo su principal característica ofrecer un servicio no fiable y no orientado a conexión. Por ello, y por su semejanza con el servicio ofrecido por IP, la TDPU («Transport Protocol Data Unit») de UDP se denomina indistintamente *datagrama de usuario*, *datagrama UDP* o *paquete UDP*. El formato del paquete UDP es el que se muestra en la Figura 10.2(a), siendo los campos que lo componen los siguientes:

- *Porigen*: Campo de 16 bits que especifica el puerto origen. La funcionalidad de este campo, junto con la del siguiente, se explica más adelante.
- *Pdestino*: Campo de 16 bits que identifica al puerto destino.
- *LongitudUDP*: 16 bits que indican el número de octetos de que consta el datagrama UDP completo.
- *Comprobación*: Campo redundante que contiene la suma de comprobación usual, es decir, el complemento a uno de la suma en complemento a uno de todo el datagrama y de una pseudo-cabecera, explicada más adelante.
- *Datos*: Este campo contiene la PDU de capa superior.

Los cuatro primeros campos mencionados constituyen la cabecera del datagrama UDP (sombreada en la Figura 10.2), la cual tiene una longitud fija de 64 bits. El servicio que ofrece UDP es muy sencillo. Concretamente, es un servicio no fiable sin control de flujo ni control de errores.

En el cálculo de la suma de comprobación de UDP, además de considerar todo el datagrama, incluyendo la cabecera, se incluyen cuatro campos de la capa inferior (IP): *dirección IP origen*, *dirección IP destino*, *protocolo* del paquete IP y campo *longitudUDP* del datagrama. Esta estructura se denomina *pseudo-cabecera UDP* (Figura 10.2(b)).

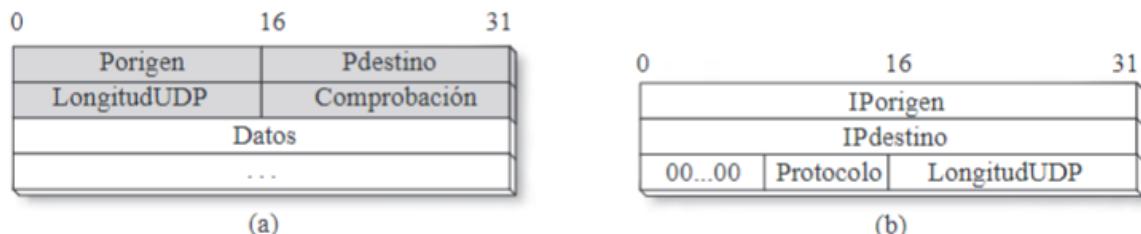


Figura 10.2. Datagrama de usuario UDP (a) y pseudo-cabecera UDP para el cálculo del campo comprobación (b).

En general, un protocolo se denomina *cross-layer* cuando considera información de otras capas, como por ejemplo la pseudo-cabecera UDP. En este caso, a pesar de vulnerar el principio de diseño de Internet de modularidad y acoplamiento débil (ver Apartado 8.1), el considerar en el cálculo del campo *comprobación* las direcciones IP tiene como objetivo proporcionar una garantía extra de fiabilidad para evitar que información no deseada llegue a direcciones IP equivocadas. La motivación es proteger a UDP de posibles errores en la capa IP, como por ejemplo traducciones NAT erróneas.

Dicho lo anterior, sobre el servicio ofrecido por IP, UDP solo añade la *multiplexación*. Es decir, introduce un esquema de direccionamiento extra —basado en los puertos— que permite identificar las aplicaciones origen y destino del datagrama (Figura 10.3). Pensemos, por ejemplo, en una aplicación de correo electrónico, otra de transferencia de ficheros y una tercera de acceso web. ¿Cómo podemos discriminar entre ellas dentro de una misma estación final si la dirección IP de esta es única? Aunque pudieramos recurrir a los identificadores de proceso asignados por el sistema operativo (SO) de la máquina en cuestión, la razón principal para no hacerlo es que desde una entidad remota no es posible conocerlos, ya que son asignados de forma dinámica.

Desde este punto de vista, interesa direccionar a las diferentes aplicaciones destino sin, para ello, precisar conocer el proceso específico asignado por el SO en cada momento. Por ejemplo, es deseable poder contactar con un servidor de correo electrónico sin necesidad de conocer el identificador del proceso en el *host* destino que corresponde a dicha aplicación. Por tanto, en lugar de considerar los identificadores de proceso para indicar el destino de una comunicación TCP/IP, se recurre al concepto de *puerto* (introducido brevemente en el Apartado 8.4). Así, toda comunicación entre una aplicación

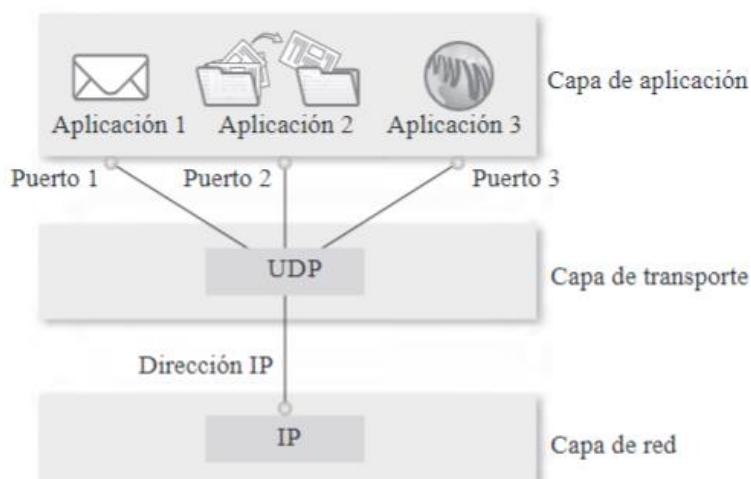


Figura 10.3. Puertos y multiplexación UDP.

Tabla 10.1. Ejemplos de aplicaciones que usan UDP y sus puertos asociados.

Puerto	Aplicación/Servicio	Descripción
7	echo	Eco
13	daytime	Fecha
37	time	Hora
53	domain	Servicio de nombres de domino
69	tftp	Transferencia simple de ficheros
123	ntp	Protocolo de tiempo de red

origen y otra destino precisa especificar no solo las respectivas direcciones IP, indicadas en el datagrama IP, sino también los puertos origen y destino correspondientes dentro de cada *host*¹. A modo de ejemplo, en la Tabla 10.1. se indican algunos servicios o aplicaciones estándares que usan UDP y los puertos asociados. La IANA ha definido tres rangos para los puertos (para más detalles consultar el RFC 6335):

- puertos del sistema: del 0 al 1023,
- puertos de usuario: del 1024 al 49151 y
- puertos dinámicos y/o privados: del 49152 al 65535.

Por último, en relación al protocolo UDP es de mencionar que su carácter no orientado a conexión (es decir, no exige ninguna negociación o interacción explícita con el destino) y su sencillez lo convierten en la opción adecuada a nivel de transporte para aquellas aplicaciones que impliquen interacciones solicitud/respuesta sencillas (como las multimedia o juegos *on-line*), en las que la fiabilidad no sea un requisito estricto, y en las que se prime la interactividad (sin retardos introducidos por los mecanismos de realimentación), aún a costa de asumir cierta probabilidad de error.

10.3. Protocolo de control de transmisión (TCP)

El protocolo TCP («Transmission Control Protocol») forma, junto a IP, el corazón de la arquitectura de Internet, dándole su nombre: TCP/IP. Las especificaciones de TCP se encuentran detalladas en el RFC 793, debiendo destacarse como características principales de este protocolo las siguientes:

1. TCP ofrece un servicio orientado a conexión, permitiendo el envío secuencial de los datos, es decir, estos se reciben en el mismo orden en que fueron transmitidos. Es lo que se conoce como *envío orientado a flujo* («stream oriented»).
2. TCP ofrece un servicio punto a punto, es decir, no se puede utilizar para comunicaciones *multicast*.
3. TCP es full-duplex.
4. TCP ofrece un servicio extremo a extremo fiable que incluye mecanismos de detección y recuperación de errores, control de flujo y control de congestión.

Según lo anterior, a diferencia de UDP, TCP implementa todas las funciones necesarias para llevar a cabo un control de la transmisión extremo a extremo entre dos estaciones finales. Esto hace que

¹ Retomando la terminología OSI utilizada en la primera parte del libro, un puerto se corresponde con un TSAP; es decir, un punto de acceso al servicio en la capa de transporte.

TCP sea un protocolo más complejo, ya que como veremos más adelante, a diferencia de UDP, implica mecanismos de realimentación que pueden llegar a incrementar el retardo o latencia que experimenta la aplicación usuaria del mismo. No obstante, si bien UDP se puede adoptar como protocolo de transporte para aplicaciones en un entorno local, donde la probabilidad de error sea baja, el uso de TCP es aconsejable para la transmisión remota de datos a larga distancia para aplicaciones que exijan un servicio fiable.

Los servicios que ofrece TCP, estudiados en los siguientes apartados, son:

1. *Control de errores y de flujo.* Este servicio garantiza la recepción correcta y ordenada del flujo o mensaje, por parte de la aplicación destino tal y como se generó en la aplicación origen. Es más, se encarga de soslayar las diferencias que haya en cuanto a la tasa de generación y consumo de información entre las dos aplicaciones involucradas, liberando a estas de la necesidad de ocuparse de este problema.
2. *Establecimiento y cierre de la conexión.* Dada la naturaleza orientada a conexión de TCP, este incluye mecanismos para el establecimiento y el cierre de la conexión con carácter previo y posterior, respectivamente, a la transmisión de los datos.
3. *Control de congestión.* Adicionalmente a los procesos anteriores, TCP ofrece un servicio de control de congestión (ver el Capítulo 7), que reduce el posible agotamiento de los recursos disponibles en la subred (es decir, del ancho de banda de las líneas y de la capacidad de almacenamiento temporal en los routers).
4. Por último, TCP, al igual que UDP, permite la *multiplexación* de diferentes aplicaciones origen y destino.

En la Figura 10.4(a) se muestra la TPDU de TCP, denominada *segmento TCP*. La explicación de los distintos campos del segmento TCP se puede realizar considerando los servicios ofrecidos por este protocolo. Antes de pasar al estudio de los campos más importantes, en las cabeceras TCP se definen los siguientes campos:

		0	4	10	16	31							
		Porigen		Pdestino									
Secuencia													
Acuse													
Hlen	000	Control: N C E U A P R S F		Ventana									
Comprobación			Puntero										
Opciones													
Datos													
.....													

(a)

IPorigen		
IPdestino		
00...00	Protocolo	LongitudTCP

(b)

Figura 10.4. Segmento TCP (a) y pseudo-cabecera para el cálculo del campo comprobación (b).

- *Hlen*: Campo de 4 bits, de valor mínimo igual a 5, utilizado para indicar la longitud de la cabecera TCP (campos sombreados en la Figura 10.4(a)) en palabras de 32 bits.
- *Reservado*: Campo de 3 bits sin uso específico (normalmente con valor 000).
- *Datos*: Campo en el que se transporta el mensaje generado por la aplicación. Aunque la longitud de este campo no se especifica explícitamente, se puede calcular a partir de la siguiente sencilla operación: $tlenIP - hlenIP*4 - hlenTCP*4$, donde $tlenIP$ es el número total de octetos del datagrama IP (cabecera IP + segmento TCP) y $hlenIP$ y $hlenTCP$ las longitudes, en palabras de 32 octetos, de la cabecera IP y de la cabecera TCP, respectivamente.
- *Opciones*: Este campo, de longitud variable, puede presentar dos formatos distintos: (a) un único octeto que define el *tipo* de opción o (b) un octeto de *tipo* de opción más un octeto que indica la *longitud* de los datos correspondientes a la opción, seguida de los *datos* como tales. Inicialmente solo existen especificados tres tipos de opciones:
 - 0 → fin de la lista de opciones,
 - 1 → sin operación, utilizado como relleno para hacer que la longitud de la cabecera del segmento TCP sea múltiplo de 32 bits, y
 - 2 → tamaño máximo del segmento (MSS, «Maximum Segment Size»).

10.3.1. Multiplexación

El servicio de multiplexación, al igual que ocurre en UDP, permite que varias aplicaciones ejecutadas en un mismo *host* (origen o destino) puedan comunicarse entre sí usando TCP. Para direccionar las aplicaciones concretas, los campos de 16 bits *porigen* y *pdestino* del segmento TCP especifican, respectivamente, los puertos origen y destino de la comunicación y, en definitiva, identifican el proceso generador y consumidor de la información transportada.

En la Tabla 10.2 se muestran, a modo de ejemplo, algunas aplicaciones usuarias de TCP y los puertos estándares asociados. De entre todos estos servicios podemos destacar el de transferencia de ficheros (*ftp*), el de correo electrónico (*smtp*) y el de acceso a hipertextos (*http*). Como se puede comprobar

Tabla 10.2. Ejemplos de aplicaciones TCP y puertos asociados.

Puerto	Aplicación/Servicio	Descripción
7	echo	Eco
13	daytime	Fecha
20	ftp-data	Transferencia de ficheros: datos
21	ftp	Transferencia de ficheros: control
23	telnet	Acceso remoto
25	smtp	Correo electrónico
37	time	Hora
42	nameserver	Servicio de nombres
53	domain	Servicio de nombres de dominio
79	finger	Servicio finger
80	http	Acceso hipertexto (web)

(Tablas 10.1 y 10.2), hay servicios que pueden usar tanto TCP como UDP (por ejemplo, *eco*, *daytime* o el *servicio de nombres*); es decir, se trata de servicios que con independencia del protocolo de transporte utilizado, dispondrán de un proceso accesible en el mismo puerto. Es lo que se conoce como *servicios multiprotocolo* (explicados con más detalle en el Apartado 11.2.3).

Antes de estudiar los servicios característicos de TCP, hemos de comentar que, a diferencia de UDP, dado su carácter orientado a conexión, TCP multiplexa y demultiplexa aplicaciones (transportando los datos con fiabilidad en modo *full-duplex*) estableciendo conexiones entre las partes. Una conexión estará únicamente identificada por cuatro valores: la dirección IP y puerto orígenes, además de la dirección IP y puerto remotos (o destinos). En el siguiente apartado se estudia cómo se establecen las conexiones TCP, paso previo al intercambio de datos, y cómo se cierran estas una vez concluida la transmisión de información.

10.3.2. Control de la conexión

Como se ha establecido al comienzo del Apartado 10.3, TCP ofrece un servicio fiable orientado a conexión. Esto significa que, previo al intercambio de los datos, entre las dos aplicaciones finales se establece una conexión. De forma análoga, finalizada la transmisión se procederá al cierre de la conexión.

Una conexión TCP es como una especie de circuito virtual, que implica acoplar las entidades origen y destino mediante la definición de un estado común a ambas. La conexión puede explicarse como una «tubería» (*full-duplex*) establecida entre las dos aplicaciones a través de la cual todos los segmentos que entran por uno de sus extremos (la aplicación origen) salen por el otro (la aplicación destino). Además, la transmisión de los segmentos que hace TCP a través de esta «tubería» se ofrece libre de errores, respetando el mismo orden con el que se generaron, gracias a la implementación de procedimientos de control de flujo, errores y congestión. Estas funcionalidades exigen que el proceso emisor y el receptor estén sincronizados.

Para ello son necesarias las conexiones TCP, las cuales implican una fase previa al intercambio de datos, denominada establecimiento de la conexión, en la que se define ese estado común antes aludido, y mediante la que se reservan recursos del sistema (*buffer* o memoria temporal) y, tras el intercambio, una fase final o cierre de la conexión, en la que se liberan los recursos reservados del sistema, tanto en el emisor como en el receptor.

Ambos procedimientos, establecimiento y cierre, se llevan a cabo, como no puede ser de otro modo, usando campos definidos en el segmento TCP mostrado en la Figura 10.54(a); en concreto, a través de los bits *R*, *S* y *F* del campo de *control*. Seguidamente se discuten ambos procesos.

— *Establecimiento de la conexión*. Desde el punto de vista de las entidades TCP, en la capa de red contamos con el servicio no fiable que ofrece IP; por tanto, el establecimiento de la conexión debe tener en cuenta (con probabilidad mayor que cero) que los segmentos intercambiados se pueden perder. Estrictamente hablando es imposible garantizar el éxito en esta tarea, ya que el canal utilizado (ofrecido por IP) no está libre de errores. Para paliar este hecho, en el establecimiento de la conexión se adopta el esquema conocido como *procedimiento a tres pasos* («*three-way handshake*») que consiste, como indica su nombre, en el intercambio de tres segmentos entre la entidad origen y la destino (Figura 10.5(a)):

- Solicitud de conexión*: Uno de los dos extremos envía un segmento (sin datos, solo cabecera) con el bit *S* (SYN, de «SYNchronize») activado, indicando el deseo de iniciar una conexión. En ese mismo segmento, en el campo *secuencia* se especifica un número

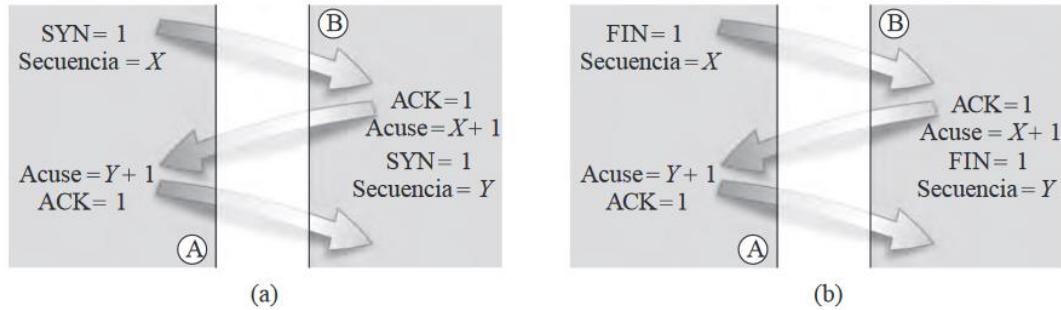


Figura 10.5. Procedimiento de tres pasos para el establecimiento (a) y cierre (b) de una conexión TCP entre dos entidades A y B.

arbitrario (de 32 bits) a partir del cual se numeran los segmentos. De esta forma sincronizamos, es decir, establecemos, un estado común entre las entidades, utilizando un número diferente en cada sentido de la conexión.

Para justificar la adopción de este procedimiento pensemos en una transmisión de datos entre dos aplicaciones e imaginemos que, una vez finalizada la conexión correspondiente, se estableciese con posterioridad otra entre las mismas aplicaciones. Si en el transcurso de esta nueva conexión se recibiese (retrasado por la subred) un segmento de datos correspondiente a la primera, ¿es posible que este segmento se aceptase como válido en la nueva conexión? Aunque improbable, la respuesta es sí.

Es evidente que la probabilidad de que ocurra una situación como la comentada se reduce haciendo que las sucesivas conexiones se inicien con una numeración distinta de los segmentos. Es lo que se pretende con el campo *secuencia* en el segmento de solicitud de conexión o SYN (bit *S*=1).

- b) *Aceptación de conexión y solicitud en el otro sentido:* En caso de aceptar la conexión, el otro extremo reserva recursos en el SO remoto (típicamente mediante la llamada a procedimientos de reserva de memoria dinámica), tras lo cual responde con un segmento en el que se activa el bit ACK (de «ACKnowledge») de la cabecera TCP y se hace *acuse* = *X*+1, donde *X* es el número de secuencia especificado por el emisor en el campo *secuencia* en el paso a).

Dado el carácter *full-duplex* de las conexiones TCP, estas se deben establecer en ambos sentidos de la transmisión. Así, en el segmento de respuesta a la solicitud de conexión se activa adicionalmente el bit SYN y se elige un número de secuencia *Y*, arbitrario e independiente del especificado por el emisor, para la transmisión en sentido contrario.

- c) *Conexión establecida:* La conexión quedará establecida una vez que el extremo que originó el mensaje inicial de solicitud de conexión (A en la Figura 10.5(a)), tras reservar recursos del SO, genere un segmento TCP con *acuse*=*Y*+1 y ACK=1 en respuesta a la solicitud generada por el otro extremo (B).

— *Cierre de la conexión.* El procedimiento de cierre de la conexión es análogo al seguido en su establecimiento, con la salvedad de que ahora el bit del campo *control* utilizado no es SYN sino F (de FIN en la Figura 10.4(a)). Como se muestra en la Figura 10.5(b), el cierre de la conexión se desarrolla también en tres pasos:

- a) *Solicitud de cierre:* Cualquiera de las dos entidades TCP, cuando no tenga más datos que transmitir, enviará un segmento TCP con el bit FIN=1 y el campo *secuencia* = *X* en uno de los sentidos, indicando su deseo de cerrar la conexión.

- b) *Aceptación de cierre y solicitud en el otro sentido:* El otro extremo debe confirmar la solicitud recibida, enviando para ello un segmento con el bit ACK activado y con el campo *acuse* = X+1. En este punto, si esta entidad ha concluido igualmente con la transmisión de los datos en el otro sentido, en este mismo segmento activará el bit FIN y especificará el correspondiente campo *secuencia* = Y. Esto último no es necesariamente obligatorio, en el sentido de que si la aplicación tiene datos pendientes que transmitir, la solicitud de cierre se hará posteriormente al envío de los datos pendientes.
- c) *Conexión finalizada:* La conexión TCP queda definitivamente cerrada cuando el extremo que originó el primer FIN genere un segmento TCP con *acuse*=Y+1 y ACK=1 como respuesta al segmento del Apartado b).

Tanto el procedimiento de establecimiento como el de cierre quedan definidos a través del autómata de estados finitos TCP mostrado en la Figura 10.6, en el cual aparecen los siguientes estados y transiciones entre ellos:

- El estado *CLOSED* corresponde a la no existencia de conexión alguna e identifica al estado inicial y final de toda transmisión. Mientras no tenga lugar evento alguno permaneceremos en este estado. De él se puede salir de dos formas: como solicitante de establecimiento de conexión (extremo A en Figura 10.5(a)), lo que se conoce como *apertura activa*², o como destinatario de la misma (extremo B en Figura 10.5(a)), lo que se corresponde con una *apertura pasiva*³. El primer caso consiste en el envío de un segmento SYN (primer paso en la Figura 10.5(a)), transición que nos lleva al estado *SYN_SENT* indicado en la Figura 10.6. Por su parte, una apertura pasiva implica la transición a un estado de escucha (*LISTEN* en el autómata) en el que se está a la espera de recibir un segmento SYN por parte de una entidad remota.
- Como hemos comentado, en el estado *LISTEN* estaremos en disposición de recibir un segmento SYN. Ante este evento responderemos con el envío de un segmento ACK de confirmación, además de activar el bit SYN para solicitar el establecimiento en el otro sentido. Así pues, la transición *syn/syn+ack* correspondiente a la recepción y envío comentados desde *LISTEN* nos lleva al estado *SYN_RCVD*.

Desde *LISTEN* también podemos ir al estado *SYN_SENT* sin más que emitir un segmento TCP con el bit SYN activo.

- Estando en *SYN_RCVD*, bastará con recibir la confirmación a nuestra solicitud para dar por establecida la conexión (tercer y último paso en Figura 10.5(a)), pasando así al estado *ESTABLISHED*.
- Por su parte, en el estado *SYN_SENT* bastará la recepción de una confirmación además de una solicitud de establecimiento por parte del otro extremo, solicitud que será confirmada, para dar por establecida la conexión. Esta transición corresponde a la especificada como *syn+ack/ack* desde el estado *SYN_SENT* hasta el estado *ESTABLISHED* en el autómata.
- El autómata recoge una transición del estado *SYN_SENT* al estado *SYN_RCVD*. Dicha transición, consistente en la recepción de un segmento SYN y en el envío de uno ACK, representa una situación en la que tiene lugar una solicitud de conexión simultánea por parte de los dos extremos.
- El estado *ESTABLISHED* se caracteriza por la existencia de una conexión efectiva entre ambas entidades finales. En este estado se produce el intercambio de datos entre los dos extremos, llevándose a cabo el control de flujo y de errores (explicados en los Apartados 10.3.3 y 10.3.4). Finalizada la transmisión, se procederá al cierre de la conexión y abandono del estado *ESTABLISHED*.

² La apertura activa la realiza la entidad *cliente* (ver Capítulo 11).

³ La apertura pasiva es propia del *servidor* (ver Capítulo 11).

Como sucedía con el estado *CLOSED*, del estado *ESTABLISHED* se puede salir de dos formas: como solicitante inicial del cierre de la conexión (extremo A en Figura 10.5(b)), *cierre activo*, o como receptor de dicha solicitud (extremo B en Figura 10.5(b)), *cierre pasivo*. En el primer caso transitaremos hacia el estado de conexión cerrada (*CLOSED*) a través de los estados *FIN_WAIT_1*, *FIN_WAIT_2* o *CLOSING*, y *TIME_WAIT*, mientras que en un cierre pasivo alcanzaremos el estado *CLOSED* a través de los estados *CLOSE_WAIT* y *LAST_ACK*. Estas transiciones y estados se comentan a continuación.

- Si en el estado *ESTABLISHED* emitimos un segmento FIN hacia el otro extremo, pasaremos al estado *FIN_WAIT_1*. En este estado pueden darse tres situaciones posibles, todas ellas desembocando en el estado *TIME_WAIT*:

1. El receptor confirma nuestra solicitud de cierre mediante un segmento ACK y, simultáneamente, solicita el cierre de la conexión en el sentido opuesto, solicitud que confirmaremos. En tal caso se produce la transición directa desde *FIN_WAIT_1* a *TIME_WAIT*.
2. El receptor solicita el cierre en el otro sentido a través de un segmento FIN, el cual confirmaremos pasando al estado *CLOSING*. Desde este se transitará a *TIME_WAIT* ante la recepción de la confirmación correspondiente a nuestra solicitud de cierre inicial (emitida de *ESTABLISHED* a *FIN_WAIT_1*).
3. Otra posibilidad es que, dada la existencia de datos pendientes de enviar en el sentido contrario, el receptor responda solamente confirmando nuestra solicitud de cierre mediante un segmento ACK. En este caso pasaremos al estado *FIN_WAIT_2*. En este estado solo faltarán que el otro extremo solicite con posterioridad el cierre de conexión en su sentido de envío mediante un segmento FIN y que confirmemos este mediante un segmento ACK: transición *fin/ack* de *FIN_WAIT_2* a *TIME_WAIT*.

Si pensamos detenidamente en los procesos de cierre indicados en 2 y 3, el número de pasos llevado a cabo ha sido cuatro, en lugar de los tres establecidos en el «three-way handshake», debido a la descomposición del paso intermedio en dos: uno para la confirmación del cierre en un sentido y otro para la solicitud de cierre en el sentido opuesto.

- Antes de dar por finalizada completamente la conexión desde el estado *TIME_WAIT* (transición a *CLOSED*), hemos de esperar un intervalo de tiempo dado ante la potencial recepción de segmentos retrasados por la subred y correspondientes a la conexión actual. Este intervalo de tiempo, denominado *tiempo de vida de segmento máximo* (MSL, «Maximum Segment Lifetime»), se especifica igual a 2 minutos en el RFC 793. Por ello, el intervalo de tiempo en cuestión se conoce como 2MSL, nombre con el que en ocasiones se designa también al estado *TIME_WAIT*.
- El cierre de la conexión también puede verse desde el punto de vista del extremo receptor de la solicitud inicial de cierre (B en Figura 10.5(b)), lo que corresponde a un cierre pasivo. En tal caso, la recepción de un segmento FIN y la correspondiente confirmación del mismo nos llevaría desde el estado *ESTABLISHED* al estado *CLOSE_WAIT*. Para dar por concluida la conexión solo faltaría la solicitud por nuestra parte del cierre en el otro sentido (transición al estado *LAST_ACK*) y la recepción del segmento de confirmación correspondiente (transición *ack/-* en Figura 10.6 desde el estado *LAST_ACK* al estado *CLOSED*).

Además del temporizador asociado al estado *TIME_WAIT*, en los mecanismos de establecimiento y cierre de conexión, para evitar comportamientos no deseados, es decir, para ser robustos frente a posibles pérdidas o retrasos no controlados de los segmentos (recuérdese que TCP utiliza el servicio no fiable de IP), se utilizan temporizadores de forma tal que si no se recibe adecuadamente o a tiempo la respuesta esperada, se genera una interrupción que desbloqueará el proceso. Posteriormente, en el Apartado 10.3.6 se estudian los temporizadores asociados a toda conexión TCP.

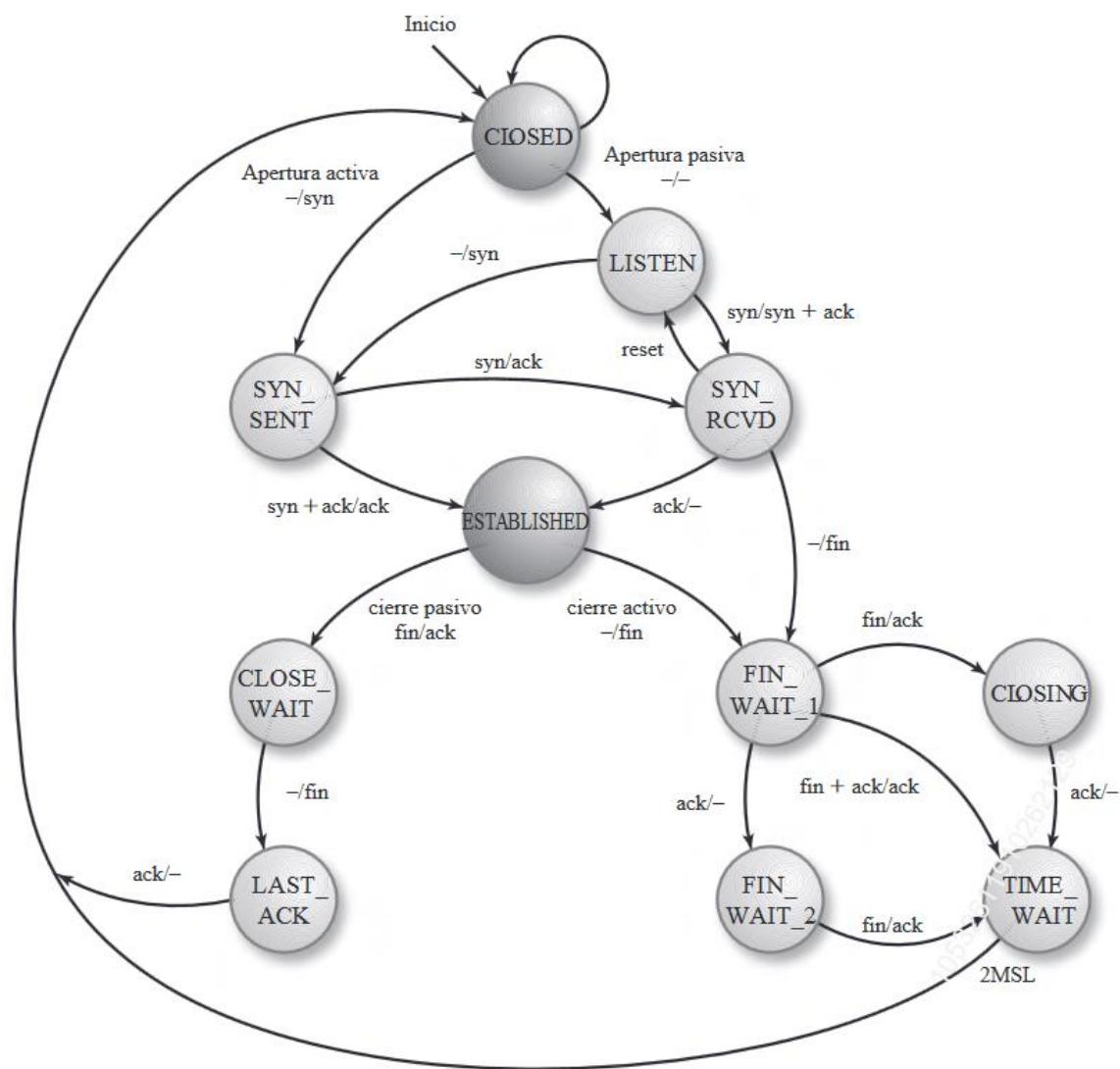


Figura 10.6. Autómata de estados finitos de TCP. La etiqueta a / b es «segmento a recibido, segmento b transmitido».

Se define el ya referido tamaño máximo de segmento o MSS como la máxima cantidad de datos (expresada en bytes) que se pueden transportar en un segmento. Cuanto mayor sea el MSS la utilización de la capacidad del canal será mayor, si bien lo deseable es elegir el MSS tal que se evite la fragmentación del correspondiente paquete IP resultante. Para ello sería necesario conocer la menor MTU de todas las subredes implicadas en la ruta hasta el destino. Nótese que, dado el carácter no orientado a conexión de IP, los extremos finales solo pueden conocer la MTU de las subredes directamente conectadas. Como se comentará más adelante en el Apartado 10.3.7, TCP define un campo opcional o extensión para negociar este parámetro.

Un último comentario acerca del control de una conexión en TCP es la posibilidad de reiniciar esta mediante la activación del cuarto bit (*R*, de «ReSeT») del campo *control* del segmento TCP. Este segmento de reinicio («reset») permite la desconexión ante situaciones anormales.

10.3.3. Control de errores

El control de errores ofrecido por TCP se basa en un esquema de realimentación con confirmaciones positivas y acumulativas. Es decir, las unidades de datos o segmentos que se envían han de ser confirmadas positivamente por parte del receptor. Se denominan acumulativas porque, como ya se estudió

en el Capítulo 4, no es necesario confirmar uno a uno todos los segmentos recibidos, sino que es posible confirmar más de uno de forma simultánea.

Dado su carácter *full-duplex*, TCP permite utilizar los segmentos de datos para incorporar en el campo *acuse* (Figura 10.4(a)) las confirmaciones relativas a segmentos de datos recibidos en sentido contrario. Esta técnica, denominada *piggybacking* (véase Apartado 4.4.3), permite ahorrar ancho de banda pues no es necesario generar segmentos específicos de confirmación, sino que se aprovechan los segmentos de datos generados en sentido contrario.

El control de errores en TCP, por cuestiones de eficacia, adopta un esquema de ventana deslizante (ver Capítulo 4). La principal diferencia entre el esquema utilizado en TCP y las técnicas de ventana deslizante de n bits estudiadas previamente, radica en la separación que se hace en TCP de las funciones de control de errores y de flujo (estudiado en el siguiente apartado). Esto quiere decir que mientras que en un esquema de ventana deslizante la recepción de la confirmación correspondiente a una ventana ya transmitida autorizaba implícitamente, y de forma automática, a desplazar la ventana de emisión y, en consecuencia, a poder continuar la transmisión de datos hacia el receptor, en TCP esto no es así. Una cosa es la confirmación de los segmentos enviados y otra bien distinta la autorización para transmitir nuevos datos (es decir, el control de flujo). A continuación se describen ambos procesos.

El control de errores en TCP se basa en un esquema ARQ con temporizadores y ventanas deslizantes de emisión y recepción:

1. Cada segmento se numera por parte del emisor y se envía. En ese mismo momento se almacena una copia del segmento en la ventana de emisión y se inicia un contador o temporizador asociado hasta que se recibe la confirmación positiva de este segmento, generada por el receptor (puede que con *piggybacking*), lo cual se traduce en dar por realizada la transmisión y consecuentemente se desplaza si procede la ventana de emisión. Nótese que al adoptar un esquema de ventana, se permite enviar de forma consecutiva más de un segmento sin necesidad de que los anteriores sean confirmados.
2. En el receptor, tras el análisis del campo de *comprobación* y de ver que el número de secuencia recibido está dentro de los definidos por la ventana de recepción, se devuelve la confirmación positiva, puede que acumulativa.
3. Si el temporizador asociado al segmento en el emisor alcanza un valor determinado denominado «*timeout*» se concluye que ha habido un error y se procede al reenvío de los datos correspondientes.

Los campos del segmento TCP involucrados en el control de errores son los siguientes (Figura 10.4(a)):

- *Secuencia*: Campo de 32 bits que indica la posición (o número de byte) que ocupa el primer octeto del campo *datos* del segmento dentro del mensaje generado por la aplicación. Este esquema permite de forma fácil la entrega ordenada de los segmentos a la aplicación en el receptor, incluso tras recibirlos desordenados, cosa posible por el carácter no fiable del servicio ofrecido por IP.
- *Comprobación*: Campo que se incluye en cada segmento transmitido y permite al receptor determinar si el segmento se ha recibido correctamente o no. El campo *comprobación* se obtiene como el complemento a 1 de la suma complemento a 1 de las palabras de 16 bits del segmento TCP completo, incluidos los datos, y, además, de la *pseudo-cabecera TCP* que, como en el caso de UDP, se construye al efecto (Figura 10.4(b)).
- *Acuse*: Campo de confirmación de 32 bits que indica el número de byte que se espera recibir en el destino. Téngase en cuenta que en TCP no hay confirmaciones negativas. Dado su carácter

acumulativo, recibir un determinado valor en el campo *acuse* confirma todos los bytes pendientes de confirmación hasta justo el anterior.

- *ACK*: Bit (*A*, de *ACK*) del campo de *control* en la cabecera TCP. Este bit indica la validez o no de la confirmación especificada en el campo *acuse*. Es decir, si *A*=1 el campo *acuse* debe ser tenido en cuenta por el receptor, mientras que si *A*=0 no.

De acuerdo con los RFC 1122, 2525 y 2581, en la Tabla 10.3 se identifican diferentes eventos que pueden ocurrir cuando un receptor recibe paquetes (ordenados o no según su número de secuencia), detallándose cómo el receptor debe devolver las confirmaciones.

Para completar el estudio del control de errores en TCP, nótese que la eficacia o prestaciones del procedimiento dependerán de la correcta estimación del *timeout*. Evidentemente, su valor siempre debería estar próximo al tiempo de ida y vuelta entre el origen y el destino, también denominado RTT («Round Trip Time»). Ahora bien, si el valor del *timeout* (que en definitiva dispara la retransmisión de segmentos) fuera demasiado pequeño, se producirían retransmisiones prematuras e innecesarias, desperdiando por tanto recursos en la red. Por el contrario, si el *timeout* fuese demasiado grande el emisor tardaría mucho tiempo en darse cuenta de posibles problemas, retrasando innecesariamente las retransmisiones de segmentos. La cuestión crítica es por tanto cuál es el valor y cómo ajustar el *timeout*.

Hay que tener presente que TCP debe operar en un conjunto muy heterogéneo de escenarios que van desde una red local (en la que los RTT pueden ser muy pequeños, del orden de pocos milisegundos) hasta interacciones WAN extremo a extremo que involucren muchos *routers* con enlaces de muy diversa índole (satélite, cable submarino, etc.). Además, la propia red, dependiendo de la carga de tráfico ofrecida, puede tener una dinámica muy cambiante en el tiempo, lo que se traduce en que en una conexión dada, dependiendo de la ocupación de las colas en los *routers* intermedios, la latencia entre segmentos consecutivos pueda variar en un factor 4, 5 o más.

Ante la dificultad de poder modelar analíticamente el fenómeno, la estimación del *timeout* pasa por adoptar un esquema adaptable, y así es como TCP resuelve este problema. De este modo, el método para la estimación del *timeout* seguido en TCP es el siguiente:

Tabla 10.3. Gestión de ACK. Posibles eventos y acciones en el receptor TCP.

Evento	Acción en el receptor TCP
Llega un segmento en orden, sin errores y sin discontinuidad, es decir, su número de secuencia coincide con el límite inferior de la ventana de recepción. Además, todos los datos anteriores están ya confirmados.	Pasar el segmento a la aplicación. Retrasar el envío del ACK. Esperar hasta 500 ms para recibir el siguiente segmento. Si no llega, enviar el ACK.
Llega un segmento en orden, sin errores y sin discontinuidad, pero hay pendiente un ACK retrasado.	Pasar el segmento a la aplicación. Enviar inmediatamente un único ACK acumulativo que confirme los dos segmentos.
Llega un segmento desordenado sin errores con una discontinuidad, es decir, su número de secuencia, aunque permitido por la ventana de recepción, es mayor que el límite inferior de la ventana.	Almacenar el segmento y no pasarlo a la aplicación. Enviar ACK duplicado (repetición de la anterior) que confirme hasta el byte anterior correspondiente al límite inferior de la ventana.
Llega un segmento desordenado sin errores que completa parcial o totalmente una discontinuidad.	Pasar el segmento recibido (y los contiguos que estén almacenados) a la aplicación. Enviar el ACK que corresponda (confirmando hasta el último byte que se haya pasado a la aplicación).

1. Cada vez que se transmite un segmento, el emisor registra el instante de tiempo en que se produce el envío. También se considera el tiempo en que se recibe la confirmación correspondiente a dicho segmento. La diferencia entre ambos tiempos es el RTT instantáneo para el segmento actual, RTT_{actual} .
2. Para cada nuevo segmento confirmado se obtiene un nuevo RTT, RTT_{nuevo} , definido como un suavizado temporal o filtro pasa-bajo mediante la expresión:

$$RTT_{nuevo} = \alpha \cdot RTT_{viejo} + (1-\alpha) \cdot RTT_{actual}, \alpha \in [0,1] \quad (10.1)$$

donde RTT_{viejo} es el valor del RTT anterior y α es un parámetro adimensional, o peso, que permite regular qué importancia tiene en la estimación el valor instantáneo frente al valor histórico.

3. Por último, el valor del temporizador t_{out} se fija a partir de un suavizado de la desviación mediante las siguientes expresiones:

$$\begin{aligned} dev_{nuevo} &= (1 - \beta) \cdot dev_{vieja} + \beta \cdot (|RTT_{nuevo} - RTT_{actual}|), \quad \beta \in (0,1) \\ t_{out} &= RTT_{nuevo} + \eta \cdot dev, \quad \eta = 4 \end{aligned} \quad (10.2)$$

La estimación adaptable del *timeout* presenta una dificultad. Cuando se produce un *timeout* (y por lo tanto el segmento se retransmite), si se recibe una confirmación posterior el emisor no puede determinar si esta corresponde al segmento originalmente transmitido, que simplemente se ha retrasado, o si, por el contrario, corresponde a la confirmación de la retransmisión. Este problema se denomina *ambigüedad en las confirmaciones* (véase Capítulo 4), ambigüedad que dispara el denominado *algoritmo de Karn*. Este procedimiento establece que la estimación del *timeout* no se actualiza para segmentos que sean retransmisiones. El algoritmo de Karn presenta un inconveniente: el RTT no se adapta precisamente cuando más falta hace, esto es, cuando se realizan retransmisiones. Surge así la técnica conocida como *retroceso del temporizador* («timer backoff»), en la cual, cada vez que expira el temporizador asociado a un segmento TCP, se hace que

$$t_{outnuevo} = \gamma \cdot t_{outviejo} \quad (10.3)$$

siendo 2 un valor típico del parámetro γ .

10.3.4. Control de flujo

En TCP, el control de flujo se lleva a cabo usando el campo *ventana* de los segmentos (Figura 10.4(a)). Este campo, de 16 bits de longitud, indica el número de octetos que el receptor autoriza al emisor para que este los envíe de forma consecutiva. Esta técnica de control de flujo se conoce como *esquema crediticio*. Así, si un emisor recibe un segmento con un valor de *ventana* igual a 0 significa que, independientemente de que todos los segmentos precedentes hayan sido o no confirmados positivamente, el receptor no autoriza a enviar más datos hasta nueva orden.

De acuerdo con lo establecido, el procedimiento seguido en TCP es el siguiente:

- El receptor especifica al emisor un tamaño de ventana máximo (en bytes) autorizado para transmitir. Es lo que se conoce como *ventana ofertada*.
- El emisor transmite datos de acuerdo con lo que se denomina *ventana útil*, la cual se calcula como $ventana\ útil = ventana\ ofertada - bytes\ en\ tránsito$, donde *bytes en tránsito* se refiere a los octetos ya transmitidos hacia el receptor y de los cuales aún no se ha recibido confirmación.

Hay que tener en cuenta que tras un anuncio de *ventana ofertada* = 0 por parte del receptor, el emisor se bloqueará. Si la aplicación receptora consume más segmentos (es decir, libera recursos) se generará un nuevo segmento ACK anunciando la nueva *ventana ofertada* con tamaño > 0; nótese que

en TCP los segmentos de confirmación ACK no se confirman, luego si este segmento se perdiera se produciría un bloqueo en la transmisión, ya que el emisor no recibiría el anuncio y el receptor supone que emisor sí lo ha recibido. Para desbloquear este posible evento se define un temporizador en el receptor que tiene en cuenta esta situación, tal que si desde que se envía el ACK transcurre un cierto tiempo en el que no se reciben segmentos, el ACK se volverá a repetir.

En el RFC 813 se describe la dificultad que tiene el mecanismo de control de flujo en TCP denominada *síndrome de la ventana tonta* («silly window syndrome»). Este problema se da cuando el emisor, el receptor o ambos son muy lentos. Por ejemplo, imaginemos una aplicación en el receptor consumiendo datos muy lentamente. En esta situación puede ocurrir que cada vez se anuncien ventanas más pequeñas, con lo que se generarán segmentos cada vez más reducidos y, a su vez, se liberará cada vez menos espacio por cada segmento procesado por el receptor. Ello generaría anuncios más pequeños y se entraría en un ciclo en el que se reduciría la eficiencia de la transmisión debido a la disminución en la relación entre los bits de datos y los suplementarios de cabecera. Es más, puede resultar que el sistema tienda a un esquema de control de flujo parada-espera (ver Apartado 4.4.2)

Para evitar esta situación, en el RFC 813 se proponen dos posibles soluciones:

- a) Desde el punto de vista del emisor bastaría con no proceder al envío de datos mientras el tamaño de la *ventana útil* no supere un cierto valor. Para ello bastará ir acumulando tamaños de *ventana ofertada* recibidos hasta alcanzar el umbral fijado. La solución propuesta es el algoritmo de Nagle (RFC 896) que consiste básicamente en:
 - 1. Si se generan datos en el emisor y no hay segmentos pendientes de ser confirmados, enviar los datos inmediatamente en un segmento.
 - 2. Si se generan datos pero hay algo pendiente de confirmar, esperar a que todo lo pendiente se confirme o esperar a que el segmento a transmitir tenga un tamaño mínimo prefijado.
- b) Desde el punto de vista del receptor tampoco es difícil adoptar una solución. En este caso bastaría con hacer *ventana ofertada* = 0 si el valor real acumulado de esta fuese inferior a una cierta cantidad de octetos; típicamente, el mínimo entre el tamaño de un segmento y la mitad de la cantidad de memoria disponible.

También en el RFC 813 se propone un método para mejorar la eficiencia de transmisión en TCP. Conocida como *ventana optimista*, la técnica consiste en el envío por parte del emisor de una cantidad de datos mayor a la realmente ofertada por el receptor; es decir, $\text{ventana útil} > (\text{ventana ofertada} - \text{bytes en tránsito})$. Esta idea se fundamenta en la suposición de que la transmisión se va a desarrollar con éxito, de forma que los bytes en tránsito se van a recibir correctamente y, en consecuencia, se producirá una liberación de espacio adicional al especificado mediante la *ventana ofertada*. En definitiva, la idea subyacente en la técnica de ventana optimista es la igualdad entre la *ventana útil* y la *ventana ofertada*. Indudablemente, el riesgo que se corre con este esquema de transmisión es el reenvío de datos no autorizados ante potenciales errores en la comunicación.

Para concluir el estudio del control de errores y de flujo en TCP, hemos de mencionar aunque sea de forma breve algunas cuestiones adicionales:

1. TCP permite la transferencia de datos urgentes, es decir, permite que ciertos bytes se entreguen a la aplicación sin respetar el orden con el que fueron generados. Para ello, basta activar el bit *U* (de «URGent») del campo *control* (ver RFC 6093). Dicho bit significa que en la posición especificada por el campo *puntero* del segmento TCP finalizan los datos urgentes contenidos en el mismo, los cuales siempre comienzan en el byte 0. Los datos urgentes se pasarán a la capa de aplicación sin respetar el orden preestablecido por los números de secuencia. Cuando *U* = 0 el campo *puntero* no se considera.

2. Como se ha comentado anteriormente respecto del problema de la ventana tonta, TCP trata de acumular datos para una segmentación y envío eficientes de los mismos. Este hecho puede interferir con ciertas aplicaciones interactivas en las que la cantidad de datos a enviar por unidad de tiempo es reducida (pensemos por ejemplo una aplicación del tipo *chat* en la que los usuarios mantienen una conversación mediante teclado). En tales casos interesa una transmisión inmediata de los datos para evitar altos retardos en la comunicación. Esto se consigue forzando el envío a través de la activación del tercer bit del campo *control: P* (de «PuSH»).

10.3.5. Control de congestión en TCP

Como ya se planteó en el Apartado 7.1, es claro que, como consecuencia directa de la congestión de los recursos de la subred, los datos a transmitir entre dos estaciones finales pueden experimentar retardos crecientes. Esto puede provocar su retransmisión debido a la expiración del temporizador asociado en el emisor. A su vez, estas nuevas retransmisiones provocarán una mayor congestión. Es fácil, pues, constatar que entraremos así en un ciclo en el que nuevas retransmisiones provocan mayor congestión y esta, a su vez, nuevas retransmisiones. Como se comentó, se puede llegar incluso al *colapso de la red* (Figura 7.3).

En el Apartado 7.1.3 se estudiaron diferentes variantes que se pueden adoptar para controlar la congestión: en bucle abierto o cerrado, con notificación implícita o explícita, basado en regular la tasa o la ventana. También se evidenció la conveniencia de adoptar una respuesta AIMD (ver Apartado 7.1.4).

A partir de estos precedentes, en TCP se ha especificado un mecanismo en bucle cerrado, con notificación implícita basado en ventanas que adopta una variante del esquema AIMD. Para explicar la variante adoptada para el control de congestión en TCP definimos los siguientes parámetros:

- Ventana_permitida, cantidad máxima de datos que el emisor puede transmitir de forma consecutiva sin esperar confirmación.
- Ventana_del_receptor, directamente relacionada con el control de flujo y definida como la cantidad de datos que el receptor, de acuerdo con su disponibilidad, autoriza a enviar al emisor.
- Ventana_de_congestión, cantidad de datos que el procedimiento de control de congestión permite transmitir al emisor.

La relación entre estas tres variables es

$$\text{ventana_permitida} = \min(\text{ventana_del_receptor}, \text{ventana_de_congestión})$$

Partiendo de estas definiciones, en TCP se adopta una aproximación de prueba y error consistente en aumentar la ventana_de_congestión (lo que implica aumentar la tasa de transmisión mayor) mientras no se produzca congestión (evento que se identifica al producirse un *timeout*). TCP adopta por tanto una variante de AIMD en la que para adaptar la ventana_de_congestión se identifican tres comportamientos o zonas bien diferenciadas:

- *Inicio lento*. Una vez establecida la conexión, la zona de *inicio lento* («slow start» en inglés; procedimiento inicialmente especificado en el RFC 2001, siendo la última versión el RFC 5681) comienza con una ventana_de_congestión igual a 1 MSS. En esta zona, la ventana aumenta en 1 MSS cada vez que se recibe la confirmación de un segmento. En realidad, se puede observar que el procedimiento de inicio lento hace crecer la ventana exponencialmente.
- *Prevención de congestión*. Para que la ventana no crezca demasiado rápido, lo que podría producir congestión, se adopta el esquema de prevención de congestión. Este procedimiento se aplica cuando el valor de la ventana_de_congestión es mayor que un umbral dado y

consiste en aumentar la ventana de congestión a lo sumo 1 MSS por cada RTT. Para ello se sigue la siguiente estrategia: si MSS es el tamaño del segmento expresado en bytes, por cada ACK recibido

```
ventana_de_congestión = ventana_de_congestión +
                         MSS * MSS / ventana_de_congestión;
```

lo que equivale aproximadamente a incrementar la ventana de congestión 1 MSS cuando se reciben todos los ACK pendientes.

- *Decremento multiplicativo*. Cada vez que expire el temporizador⁴ en el emisor se reduce drásticamente la ventana_de_congestión a 1 MSS y el umbral se actualiza a la mitad del valor de ventana_de_congestión antes de expirar el temporizador. Igualmente, como se explicó en el Apartado 10.3.3, además de reducir la ventana, cuando expira el temporizador el valor del *timeout* se incrementa linealmente.

El procedimiento anterior corresponde a una de las primeras especificaciones (denominadas «flavors» o sabores) para el control de congestión. Concretamente, se identifica como TCP-Tahoe (en referencia a la ciudad donde el IETF se reunió para adoptarlo). Desde entonces se han ido proponiendo un buen número de diferentes «sabores» de TCP. Téngase en cuenta que la adopción de un sabor u otro no impide que las implementaciones interoperen. Además de Tahoe, por su interés académico también es relevante mencionar TCP-Reno, que incluye el procedimiento «fast retransmit». En este caso, si el emisor recibe 3 ACK repetidos (Tabla 10.3) se concluye que se está experimentando cierta congestión, siempre menos severa que en el caso de producirse un *timeout*. Cuando se produce este evento, en TCP-Reno el umbral y la ventana_de_congestión se reducen a la mitad del valor de esta última.

Para una mayor claridad, el control de congestión en TCP lo podemos explicar de forma algorítmica con el siguiente pseudo-código ejecutado en el emisor, en el que además de la definición de las tres ventanas implicadas se define la variable umbral:

```
ventana_permitida = min(ventana_del_receptor, ventana_de_congestión);
Inicialmente ventana_de_congestión = 1 (MSS);
Si ventana_de_congestión < umbral, por cada ACK recibido
    ventana_de_congestión++;
    #(crecimiento exponencial);
Si ventana_de_congestión > umbral, por cada ACK recibido
    ventana_de_congestión += MSS*MSS / ventana_de_congestión;
    #(crecimiento lineal);
Si 3 ACK repetidos consecutivos:      #(TCP-Reno)
    umbral = ventana_de_congestión /2;
    ventana_de_congestión = umbral;
Si expira timeout:
    umbral = ventana_de_congestión /2;
    ventana_de_congestión = 1;
```

Como se ha comentado previamente, el número de sabores diferentes para TCP no ha dejado de crecer. Por su relevancia es también destacable TCP-CuBIC, adoptado por defecto a partir del kernel de Linux 2.6.19. En este caso la ventana_de_congestión se define como una función cúbica

⁴ TCP asume que todas las pérdidas de segmentos se deben indiscriminadamente a la aparición de congestión en la subred (aunque hayan sido resultado de un error en la comunicación).

del tiempo transcurrido desde el último episodio de congestión experimentado; es decir, a diferencia de TCP-Reno, la evolución de la ventana no depende de RTT, lo que lo convierte en un esquema más equitativo.

En el RFC 2309 se discute la conveniencia del empleo de mecanismos de gestión activa de colas (AQM, «Active Queue Management»), estudiados en el Apartado 7.2.4, para detectar la ocurrencia de congestión antes de que se produzca el desbordamiento de las memorias de los nodos intermedios y la consecuente pérdida de segmentos debido a la congestión. Desde este punto de vista, en el RFC 2481 (sustituido por el 3168) se propone el uso del esquema denominado ECN («Explicit Congestion Notification») para indicar la ocurrencia de congestión incipiente entre los extremos de la comunicación empleando los bits 6 y 7 del campo *tipo de servicio* (TS) del paquete IP (Apartado 9.1.1). La activación del bit 6, denominado ECT («ECN-Capable Transport»), indica la capacidad ECN que presentan las entidades de transporte, mientras que el bit 7 se reserva como bit de notificación explícita de congestión (bit CE, «Congestion Experienced»). Ante la recepción de un paquete EC, el flujo de emisión TCP se modificará de modo análogo a como se procede ante la expiración de un temporizador de retransmisión. Más recientemente, en el RFC 3540 se propone una modificación, todavía en estado experimental, para protegerse contra cancelaciones accidentales o maliciosas de paquetes marcados con ECN.

En este mismo contexto de uso de notificaciones explícitas, cabe reseñar que en el RFC 896 se refiere el uso de los paquetes ICMP de *ralentización del origen* (ver Apartado 9.2) para el control de congestión a nivel TCP como sigue: cuando un *host* recibe un paquete ICMP «source quench», supone la ventana permitida a valor cero hasta que no se reciba un número dado de confirmaciones, momento en el cual el envío TCP vuelve a su operación normal. Hay que mencionar que esta funcionalidad está desaconsejada por las razones argumentadas en el RFC 6633.

10.3.6. Temporizadores de TCP

En las secciones anteriores se han explicado los diferentes procedimientos que se ejecutan en las entidades TCP. Como conclusión se observa que la provisión del servicio ofrecido por TCP es bastante compleja (especialmente comparada con el servicio ofrecido por UDP). La complejidad de TCP permite que las aplicaciones usuarias del servicio puedan olvidarse de los problemas y detalles de la transmisión, relegando por tanto todas las tareas a TCP (proceso que habitualmente está controlado por el SO).

Para dar una idea de su complejidad, en este apartado se identifican y explican los siete temporizadores que son necesarios en toda conexión TCP. En general, los temporizadores se usan para evitar los problemas que pueden desencadenarse por pérdidas o retrasos no deseados en la entrega de los paquetes IP, que en definitiva son los responsables de transportar los segmentos TCP. Concretamente, en una conexión TCP se necesitan los siguientes temporizadores:

1. *Temporizador de establecimiento de la conexión*: Se inicia en la entidad servidora que recibe una solicitud de conexión (segmento SYN) y se cancela cuando el servidor recibe el ACK correspondiente. Su objetivo es limitar el intervalo máximo de tiempo durante el cual el servidor reserva recursos al cliente en espera de establecer la conexión. Su valor por defecto es 75 s.
2. *Temporizador de retransmisión*: Se inicia en el emisor cada vez que se envía un segmento. Su límite máximo, como se ha comentado previamente, es adaptable y depende de un suavizado temporal de los RTT medidos.
3. *Temporizador de ACK retrasados*: Temporizador que se inicia en el receptor coincidiendo con la llegada de un segmento ordenado sin discontinuidad en el número de secuencia (ver Tabla 10.3). Su objetivo es esperar un cierto tiempo hasta que se reciba otro segmento y confirmar ambos acumulativamente. Su valor inicial es 200 ms, aumentado exponencialmente hasta 500 ms.

4. *Temporizador de persistencia*: Se usa para evitar posibles bloqueos que pudieran ocurrir cuando se anuncia un tamaño de ventana de control de flujo igual a 0. Al recibir este anuncio, el emisor inicia el temporizador de persistencia, el cual se reinicia cuando se recibe un anuncio de ventana mayor que cero. Si alcanza su valor máximo, se envía un segmento de prueba (o mantenimiento) al otro extremo para comprobar de esta manera su accesibilidad.
5. *Temporizador de mantenimiento (keepalive)*: Sirve para descartar que, aunque no se reciban datos del otro extremo, la entidad remota sigue operativa. Su valor es configurable. Si no se recibe confirmación del otro extremo, se concluye que la conexión ha terminado y se cierra en consecuencia.
6. *Temporizador FIN-WAIT_2*: Se habilita cuando desde una entidad TCP se envía un segmento FIN. Su valor máximo determina el intervalo de tiempo durante el cual esta entidad se mantiene activa esperando a recibir segmentos de datos válidos desde el otro extremo. Cuando una conexión pasa del estado FIN_WAIT_1 al estado FIN_WAIT_2, (Figura 10.6) la conexión no puede enviar más datos. Cuando expira el temporizador, si no ha habido cambio en el estado correspondiente, la conexión se pierde, por lo que el protocolo no se queda permanente en el estado FIN_WAIT_2. El valor de este temporizador puede llegar hasta 10 minutos.
7. *Temporizador TIME-WAIT*: Sirve para asegurar que todos los paquetes pendientes se confirman tras alcanzarse el estado TIME_WAIT (Figura 10.6), es decir, tras recibirse un segmento FIN. Su valor por defecto es igual a 2 veces el tiempo de vida máximo de un segmento (MSL), el cual puede valer 30 s, 1 minuto o 2 minutos. Durante este período se impide establecer nuevas conexiones entre las mismas direcciones IP y puertos, para evitar que paquetes de la conexión previa se acepten en conexiones posteriores.

10.3.7. Extensiones TCP

TCP fue desarrollado a finales de la década de 1970 y, como tal, estaba pensado para dar respuesta a las necesidades propias de los entornos de red de la época. Aquellas redes de entonces distan mucho de las redes con las que nos encontramos en la actualidad, en especial por lo que respecta a las velocidades alcanzadas. En este sentido, en el RFC 1323 se recogen algunas mejoras a TCP que persiguen aumentar la eficiencia de este protocolo cuando se implementa sobre redes de mayor velocidad a las existentes cuando fue diseñado.

Las mejoras a las que nos referimos son las siguientes:

- *Ventana escalada*. Como el campo *ventana* del segmento TCP tiene una longitud de 16 bits, la máxima cantidad de datos que un receptor TCP autoriza a transmitir a un emisor sin esperar confirmación es $2^{16} - 1 = 65.535$ bytes. Esto es escaso si pensamos en redes con velocidades de transmisión próximas a los gigabits por segundo, lo que se traduce en un decaimiento brutal en la eficiencia de la transmisión.
- Para solucionar este problema se propone una nueva opción TCP consistente en un factor de escala aplicable a cada ventana ofrecida. Esta opción se especifica en segmentos SYN de inicio de conexión y consta de tres octetos. El primero indica el *tipo* de opción en cuestión mediante el valor 3. El segundo byte toma el valor 3 para indicar que la *longitud* total de la opción es 3 bytes. El tercer y último octeto es un factor de escala (logarítmica) que, con un valor máximo de 2^{14} , posibilita tamaños de ventana de hasta $2^{14} \times 2^{16} = 2^{30} = 1$ Gigabyte.
- *Estimación del tiempo de ida y vuelta y opción de sello de tiempo*. Las especificaciones originales de TCP consideraban la estimación del RTT para un único segmento por ventana. Para mejorar esta estimación, se propone el cálculo del RTT para cada segmento enviado mediante un *sello de tiempo*. De esta forma, cuando se genera un segmento TCP se incluye dicha opción,

la cual será devuelta en un paquete de confirmación en el que el bit ACK del campo *control* debe estar activo.

Esta opción TCP consta de los siguientes campos:

- *Tipo*: Campo de 1 octeto a valor 8 que indica el tipo de opción.
 - *Longitud*: Campo de 1 byte con valor 10 para indicar que la longitud total de la opción es 10 octetos.
 - *TS envío*: Campo de 4 octetos donde se especifica el instante de tiempo en que se envió el segmento.
 - *TS respuesta*: Campo de 4 bytes para indicar el instante de tiempo en que se genera el segmento ACK de respuesta. El valor de este campo debe ser 0 cuando no sea válido.
- *Protección contra números de secuencia ya recibidos*. La técnica PAWS («Protect Against Wrapped Sequence numbers») hace uso de la opción *sello de tiempo* de TCP para rechazar segmentos duplicados que podrían corromper una conexión TCP abierta. Un segmento se considera como duplicado si se recibe con un sello de tiempo menor que alguno de los recientemente recibidos para la conexión actual. Esta consideración se fundamenta en el valor monótono no decreciente de la variable *sello de tiempo*.
- *Confirmaciones selectivas*. Para mejorar su eficacia, TCP adopta un mecanismo de ACK acumulativos. Esto hace que ante la pérdida aislada de un segmento seguido de un buen número de segmentos con éxito, no haya forma de evitar la retransmisión de todos, incluso de los que se han recibido correctamente. Para evitar esta eventualidad, en el RFC 2018 se especifican las confirmaciones selectivas (SACK, «Selective ACKnowledgement»). Con esta opción, además de usar el esquema convencional, un segmento puede confirmar selectivamente bloques de bytes aislados, especificando en este caso el número de secuencia y final del bloque.

10.4. Modelado analítico de TCP

Llegados a este punto, conocemos el funcionamiento de los procedimientos más relevantes para la provisión del servicio fiable y orientado a conexión que TCP ofrece. La cuestión a estudiar ahora es en qué medida (cuantitativa) los servicios ofrecidos por TCP afectan a las prestaciones extremo a extremo que la aplicación usuaria de TCP recibe.

Dar respuesta a esta pregunta es una tarea compleja, si bien con la ayuda de ciertas simplificaciones podemos llegar a un modelado analítico del comportamiento de TCP que puede ser útil para la mayor parte de los casos prácticos de interés.

Los parámetros a estimar son la velocidad de transmisión efectiva que la aplicación usuaria de TCP experimenta y la latencia extremo a extremo, todo ello bajo la adopción de ciertas simplificaciones. En este sentido, es conveniente recalcar que las conclusiones que se extraigan a partir de la evaluación realizada siempre han de situarse en el contexto de las hipótesis y simplificaciones adoptadas.

10.4.1. Cálculo de la velocidad de transmisión

En este análisis supondremos que no estamos en la zona de inicio lento, es decir, asumimos una entidad TCP que siempre está en la zona de prevención de congestión, en la que la ventana aumenta linealmente. Particularmente, supondremos que el tamaño de esta crece en 1 MSS cada RTT. Además, suponemos que los segmentos ACK nunca se pierden y que todos llegan en orden.

En el modelado identificamos dos procesos fundamentales: por un lado la dinámica de la ventana de congestión, que impone un límite y en definitiva acota la velocidad de transmisión, y por otro la

pérdida de paquetes, que se detectan en el emisor por las interrupciones de *timeout*, lo que, como ya se ha explicado previamente, se asocia incondicionalmente a un episodio de congestión.

Supondremos un sistema estacionario, es decir caracterizado por una probabilidad de pérdidas de segmentos constante, p . Suponemos también un modelo periódico, es decir, el número de segmentos enviados en cada periodo (T) viene dado por $1/p$, o, expresado de otra forma: entre dos pérdidas consecutivas se transmiten en media $(1/p) - 1$ segmentos. En la Figura 10.7 se muestra la evolución temporal de la tasa de transmisión para la zona de prevención de congestión, que se calcula como el tamaño de la ventana (expresado en número de segmentos) por RTT. Nótese que si suponemos que la ventana tiene un tamaño igual a W segmentos cuando se produce un *timeout*, la ventana reducirá su tamaño a la mitad, dando lugar a la evolución típica de dientes de sierra mostrada en la figura.

El número de paquetes N trasmítidos en cada periodo se puede calcular simplemente a partir del área de la superficie de cada periodo (sombreada en la Figura 10.7). Es decir, viene dado por

$$N = \frac{1}{2} \frac{T}{RTT} \left(\frac{W}{2} + W \right) = \frac{1}{p} \quad (10.4)$$

En la zona de prevención de congestión la ventana crece, como hemos supuesto, 1 MSS cada RTT, luego para pasar de tener un tamaño igual a $W/2$ segmentos hasta W , debe transcurrir un tiempo dado por

$$T = RTT \cdot \frac{W}{2} \quad (10.5)$$

Es decir, sustituyendo en (10.5), se tiene que

$$W = \sqrt{\frac{8}{3p}} \quad (10.5)$$

Luego, si tenemos en cuenta que la tasa o velocidad de transmisión media, $\bar{X}(p)$, expresada en segmentos por segundo para la fuente TCP la podemos calcular como el cociente entre el número de paquetes por cada periodo, se obtiene que

$$\bar{X}(p) = \frac{\frac{1}{p}}{RTT \cdot \frac{W}{2}} = \frac{1}{RTT} \sqrt{\frac{3}{2p}} \text{ (segmentos/seg)} \quad (10.7)$$

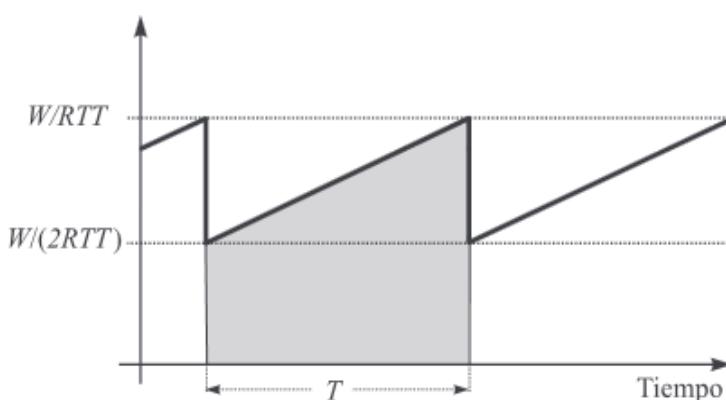


Figura 10.7. Evolución de la tasa en TCP con las suposiciones adoptadas.

Es decir, como se observa a partir de la expresión obtenida, y aceptando todas las hipótesis consideradas, la fuente TCP proporciona una tasa media que tiene una eficiencia inversamente proporcional al RTT así como una dependencia inversamente proporcional con la raíz cuadrada de la probabilidad de error.

Si se considera un modelo más preciso que incluya el hecho de que cada vez que se produce una pérdida se dispara un *timeout*, que resulta en un valor nuevo dado por $t_{out} = 2 \cdot t_{out_anterior}$, y si se considera la limitación impuesta por la ventana del receptor (W_m) con la que se realiza el control de flujo, una estimación más precisa obtenida por Padhye y otros, viene dada por la expresión:

$$\bar{X}(p) \approx \min \left(\frac{W_m}{RTT}, \frac{1}{RTT \sqrt{\frac{2p}{3}} + t_{out} \min \left(1,3 \sqrt{\frac{3p}{8}} \right) p (1 + 32p^2)} \right) \text{ (segmentos/seg)} \quad (10.8)$$

10.4.2. Cálculo de la latencia

Como ya sabemos por el Capítulo 6, se puede definir la latencia como el tiempo transcurrido desde que una aplicación solicita un objeto (de un determinado tamaño) hasta que es recibido completamente. En el análisis que sigue (desarrollado por J. Kurose y K. Ross), al igual que en el estudio de la tasa efectiva realizado anteriormente, adoptamos una serie de simplificaciones que facilitan el análisis. En este caso supondremos que no hay episodios de congestión, es decir, no hay pérdidas de segmentos durante la transmisión del objeto. Además, también supondremos que no hay errores, lo que se traduce en que no hay retransmisiones de segmentos. Supondremos además que el tamaño de las cabeceras de los protocolos es despreciable en comparación con los datos, y, sin pérdida de generalidad, vamos a aceptar que el objeto y los segmentos tienen un tamaño tal que permiten que el objeto se transmita con un número entero de segmentos. Todos los segmentos que no sean de datos asumiremos que tienen un tiempo de transmisión despreciable frente al tiempo de propagación y, por último, restringiremos este análisis a la zona de inicio lento (recuérdese que corresponde con un crecimiento exponencial del tamaño de la ventana de congestión).

Vamos a adoptar la siguiente notación:

- Sea R la velocidad de transmisión (tasa en bits/s) de la línea.
- Sea S el MSS en bits.
- Sea O el tamaño del objeto expresado en bits.
- Sea RTT el RTT en segundos.
- Sea W el tamaño de ventana_de_congestión en número de segmentos.

La transmisión del objeto con TCP siempre implica las tres fases características de un servicio orientado a conexión: establecimiento, transmisión de datos y cierre de la conexión. En nuestro modelo analítico, con las suposiciones adoptadas suponemos que el objeto se solicita coincidiendo con la devolución del ACK en el tercer mensaje del establecimiento de conexión (SYN, SYN+ACK, ACK). Suponemos que los tiempos de procesamiento son despreciables y que no se adopta la técnica de los ACK retrasados (recuérdense los 500 ms de espera típicos para la generación de ACK; ver Tabla 10.3).

Un parámetro característico y que va a determinar el comportamiento del sistema es $RTT + S/R$, ya que, como se puede comprobar, coincide con el tiempo transcurrido desde que se transmite el primer bit de un segmento hasta que vuelve el ACK correspondiente (Figura 10.8).

La relación de este parámetro con WS/R (el tiempo de transmisión de una ventana) determinará el comportamiento del sistema, ya que si $WS/R > (RTT + S/R)$, se verifica que el ACK del segmento

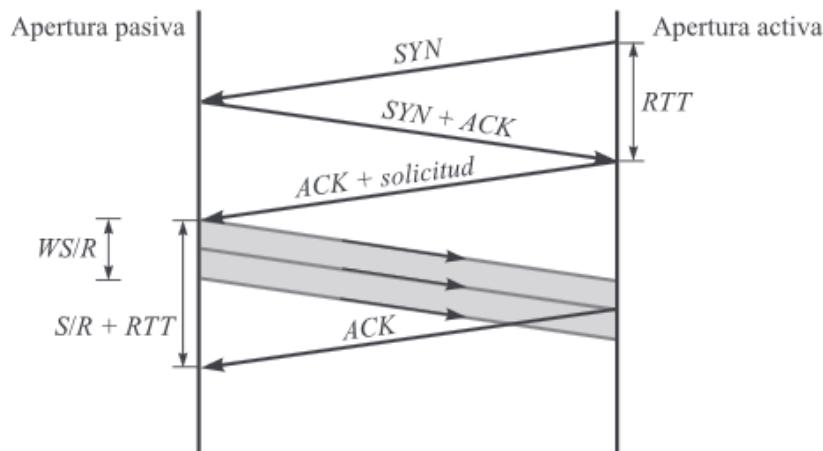


Figura 10.8. Establecimiento conexión y transmisión de una ventana que se cierra antes de que vuelva el primer ACK.

vuelve al emisor antes de que la ventana se transmita completa, es decir, el emisor no pararía de transmitir en ningún momento. Por el contrario, si en las mismas condiciones se verifica que $WS/R < (RTT + S/R)$ como ocurre en la Figura 10.8, la ventana se agota antes de que se reciba el primer ACK, es decir, la transmisión se detiene. Cuando se recibe el ACK, a este le seguirán $(W-1)$ segmentos tipo ACK separados S/R segundos entre sí. En el caso mostrado en la figura se distinguen por tanto períodos de transmisión (con W segmentos) y períodos de parada, de duración $S/R = RTT + WS/R$. Si se cumple que $k = O/(WS)$ (o su redondeo a un número entero) es el número de ventanas necesarias para transmitir el objeto, tendremos que

$$\text{latencia} = 2RTT + \frac{O}{R} + (K - 1) \left[\frac{S}{R} + RTT - \frac{WS}{R} \right] \quad (10.9)$$

Expresión esta en la que se ha despreciado el cierre de la conexión y que se ha obtenido sin más que sumar los términos correspondientes al tiempo involucrado en el establecimiento de la conexión ($2RTT$), el tiempo implicado en transmitir el objeto (O/R) y el número de paradas ($K - 1$) por el tiempo que dura cada parada [$S/R + RTT - WS/R$]. Evidentemente, para la situación en la que no cesamos de enviar se cumpliría que

$$\text{latencia} = 2RTT + \frac{O}{R} \quad (10.10)$$

En el estudio anterior se ha supuesto implícitamente que la ventana no cambia de tamaño, lo cual no es cierto, ya que para adaptarse a las condiciones de la línea, y sin con ello perder demasiadas prestaciones, TCP adopta una aproximación de prueba y error que consiste en ir aumentando el tamaño de la ventana de congestión mientras no haya errores. Como hemos indicado en la introducción, centramos el estudio en la zona de inicio lento; es decir, suponemos un tamaño inicial de 1 MSS para la ventana, que irá creciendo de forma exponencial conforme se van recibiendo las confirmaciones ACK.

En este caso, suponiendo de nuevo que el valor de RTT es constante, podemos identificar tres términos diferentes que afectarán a la latencia implicada en servir el objeto. Al igual que en el caso anterior, habrá un término relativo al establecimiento de la conexión y solicitud del objeto ($2RTT$), y un tiempo para servir el objeto a la velocidad de la línea (O/R). Adicionalmente, en el caso que nos ocupa aparecerá el tiempo empleado en las paradas que introduce la ventana de congestión debido al inicio lento.

De nuevo, la relación entre WS/R y $RTT + S/R$ jugará un papel importante. Sea P el número de paradas que impone la ventana de congestión para servir el objeto. Si el objeto fuera infinitamente grande, llegaría un momento (nótese que W va creciendo exponencialmente a partir de un cierto valor de ventana) que $WS/R > (RTT + S/R)$; es decir, llegaría un momento en el que ya no haya paradas. Sea Q el número de paradas necesarias en ese caso.

Por otro lado, como el objeto no es infinito, sea K el número de ventanas que se necesitan para transmitirlo. En cualquier caso, el número de paradas estará acotado por $K - 1$. Es decir, podemos obtener P , el número de paradas, de forma genérica, sin tener en cuenta la relación entre S/R y RTT como $P = \min\{k-1, Q\}$. Más adelante volveremos a calcular explícitamente P .

Teniendo en cuenta la Figura 10.9, el tiempo desde que se envía un segmento hasta que se recibe su ACK asociado es $S/R + RTT$. El tiempo en transmitir la k -ésima ventana (con 2^{k-1} segmentos) es $2^{k-1} S/R$, luego el tiempo de parada tras transmitir la k -ésima ventana viene dado por

$$\left[\frac{S}{R} + RTT - 2^{k-1} \frac{S}{R} \right]^+$$

donde la notación $[]^+$ indica que siempre es un valor no negativo.

Por tanto, teniendo en cuenta los tres términos contribuyentes en la latencia, se tiene que

$$\begin{aligned} \text{latencia} &= \frac{O}{R} + 2RTT + \sum_{k=1}^P \text{tiempo_parada}_k \\ &= \frac{O}{R} + 2RTT + \sum_{k=1}^P \left[\frac{S}{R} + RTT - 2^{k-1} \frac{S}{R} \right] \\ &= \frac{O}{R} + 2RTT + P \left[RTT + \frac{S}{R} \right] - (2^P - 1) \frac{S}{R} \end{aligned} \quad (10.11)$$

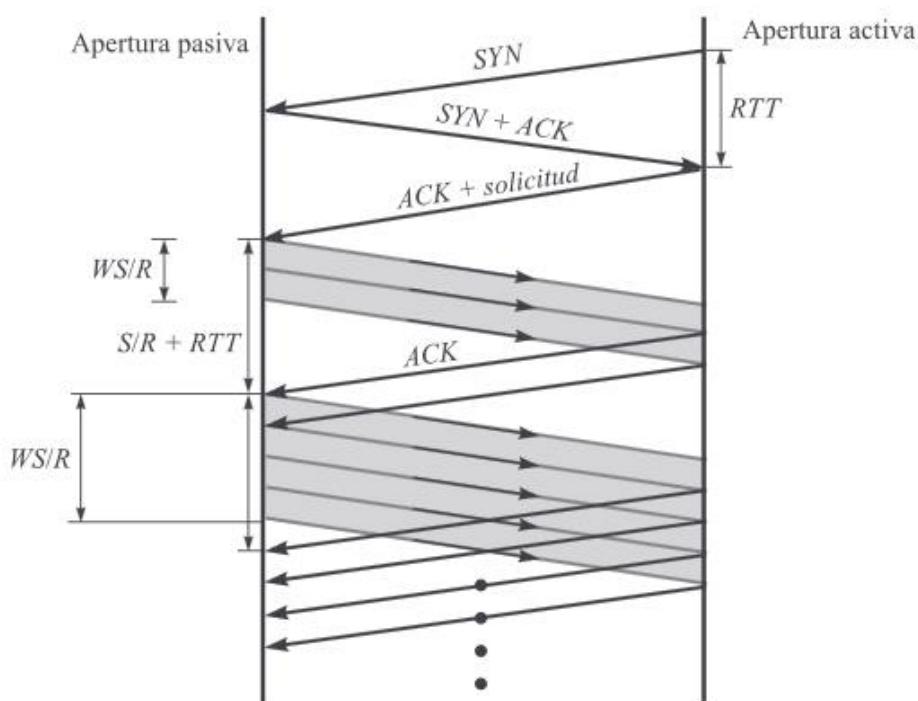


Figura 10.9. Evolución de la ventana de congestión en inicio lento.

Donde $P = \min\{K - 1, Q\}$ es el número de paradas y K , el número de ventanas para transmitir el objeto, resulta

$$\begin{aligned}
 K &= \min \{k : 2^0 S + 2^1 S + \dots + 2^{k-1} S \geq O\} \\
 &= \min \left\{ k : 2^0 + 2^1 + \dots + 2^{k-1} \geq \frac{O}{S} \right\} \\
 &= \min \left\{ k : 2^k - 1 \geq \frac{O}{S} \right\} \\
 &= \min \left\{ k : k \geq \log_2 \left(\frac{O}{S} + 1 \right) \right\} \\
 &= \left[\log_2 \left(\frac{O}{S} + 1 \right) \right]
 \end{aligned} \tag{10.12}$$

mientras que Q , el número de paradas que la ventana impone si el objeto fuera de tamaño infinito, viene dado por

$$Q = \left[\log_2 \left(1 + \frac{\frac{RTT}{S}}{\frac{R}{S}} \right) \right] + 1$$

RESUMEN

Dedicado al análisis de las funciones y protocolos implementados en la capa de transporte en Internet, este capítulo ha abordado el estudio de UDP y de TCP. Respecto del primero de ellos se ha comentado su carácter no fiable y no orientado a conexión, siendo en este caso la multiplexación el único servicio que ofrece.

Frente al anterior, el protocolo TCP se ha presentado como de una gran complejidad y potencia por cuanto que a través de él se controla de forma exhaustiva la transmisión de datos extremo a extremo. Se han estudiado las funciones implementadas para el control de flujo y de errores, para el control de conexión y el de congestión, además de la propia de multiplexación análoga a la llevada a cabo en UDP.

Por lo que se refiere a la primera de ellas, se ha evidenciado la separación efectiva entre el control de errores y el de flujo, comentándose respecto de este último el esquema crediticio considerado y el problema de la ventana tonta. Respecto del control de la conexión se ha explicado el procedimiento de tres pasos seguido tanto en el establecimiento como en el cierre de la misma y se ha presentado el autómata de estados finitos que define el funcionamiento de TCP. En el estudio del control de congestión llevado a cabo por TCP se han comentado las técnicas de decremento multiplicativo, de inicio lento y de prevención de congestión, así como la importancia de una adaptación dinámica y adecuada de los temporizadores de retransmisión asociados a los segmentos. En este sentido, y para tener un mejor conocimiento del servicio de TCP, se han identificado los siete temporizadores necesarios.

También se han introducido de forma breve algunas extensiones TCP, cuyo objetivo es la mejora de la eficiencia de este protocolo en entornos de red actuales, y, por último, se ha realizado un modelo analítico en términos de la tasa o velocidad de transmisión media en la zona de prevención de congestión y de la latencia en la zona del inicio lento.

EJERCICIOS

1. Identifique las diferencias entre TCP y UDP.
2. Dado que UDP no ofrece ninguna fiabilidad, ¿por qué usarlo?
3. ¿Sería posible diseñar un protocolo extremo a extremo fiable solo con confirmaciones negativas? ¿Por qué?
4. Explique cualitativamente qué ocurriría en TCP si el decremento de la ventana de congestión fuera aditivo.
5. Demuestre gráficamente que la pérdida de un segmento ACK no genera entregas duplicadas en la aplicación.
6. A un proceso en un *host* 1 se le asigna el puerto *p*, y a un proceso en un *host* 2 se le asigna el puerto *q*. ¿Sería posible establecer 2 o más conexiones simultáneas entre estos puertos?
7. ¿Por qué el tamaño máximo para el campo de datos de los segmentos TCP es 65.495 bytes?
8. Teniendo en cuenta el efecto del inicio lento, en una línea sin congestión con 10 ms de tiempo de propagación, una ventana de recepción de 24 Kbytes y un tamaño máximo de segmento de 2 Kbytes, ¿cuánto tiempo se emplea en enviar la primera ventana?
9. Suponiendo que la ventana de congestión es 36 Kbytes y que se dispara un *timeout*, ¿cuál será el valor de la ventana de congestión si las 4 siguientes ráfagas de transmisiones son exitosas? Suponga que el tamaño máximo de segmento (MSS) es 1 Kbyte.
10. En un escenario con RTT=100 ms, una velocidad de transmisión de 1 Gbps y segmentos de longitud 10.000 bytes (datos y cabeceras), ¿qué tamaño debería tener la ventana de emisión en un esquema de ventana deslizante para que la utilización del canal fuera mayor o igual del 90 %?
11. Identifique bajo qué condiciones el inicio lento afecta más a la latencia en TCP. Justifique la respuesta.
12. Suponga el envío de un fichero de tamaño elevado sobre una conexión TCP y suponga que el RTT es constante.
 - a) Si la ventana de congestión es 1 MSS, ¿cuánto tiempo como mínimo se necesitará para que la ventana de congestión sea 7 MSS? (suponga que no hay pérdidas y que no se entra en la zona de prevención de congestión)
 - b) ¿Cuál será el rendimiento o *throughput* medio tras 6 RTT?
 - c) Si la ventana de congestión es 101 MSS y se está en la zona de prevención de congestión, ¿cuánto tiempo se necesitará para que la ventana de congestión sea 107 MSS?
 - d) ¿Cuál será la tasa media tras 6 RTT?
13. Dibuje la figura en la que se representa el tamaño de la ventana de congestión (en segmentos) de un emisor TCP en función del tiempo (turnos), suponiendo TCP-Reno y si
 - el umbral inicial es 32,
 - en la ventana 16 se reciben 3 ACK consecutivos y
 - expira el temporizador para la ventana 22.
 - a) ¿Cuál es el valor del umbral en los turnos 18 y 24?
 - b) ¿En qué turno se envía el segmento 70?
 - c) Suponiendo que se detecta una pérdida tras el turno 26 por la recepción de un ACK tripulado, ¿cuál será el tamaño de la ventana de congestión y el valor del umbral?

BIBLIOGRAFÍA

- Allman, M.; Paxson, V.; Stevens, W.: *TCP Congestion Control*. RFC 2581. Abril, 1999.
- Braden, B.; Clark, C.; Crowcroft, J.; Davie, B.; Deering, S.; Estrin, D.; Floyd, S.; Jacobson, V.; Minshall, G.; Partridge, C.; Peterson, L.; Ramakrishnan, K.; Shenker, S.; Wroclawski, J.; Zhang, L.: *Recommendations on Queue Management and Congestion Avoidance in the Internet*. RFC 2309. Abril, 1998.
- Clark, D.D.: *Window and Acknowledgement Strategy in TCP*. RFC 813. Julio, 1982.
- Comer, D.E.: *Internetworking with TCP/IP. Volume I: Principles, Protocols and Architecture*. 3^a edición. Prentice Hall, 1995.
- Jacobson, V.; Braden, R.; Borman, D.: *TCP Extensions for High Performance*. RFC 1323. Mayo, 1992.
- Kurose, J.F.; Ross, K.W.: *Computer Networking. A Top-Down Approach Featuring the Internet*. Addison Wesley, 6a. edición, 2012.
- Nagle, J.: *Congestion control in IP/TCP internetworks*. RFC 896. Enero, 1984.
- Postel, J.: *User Datagram Protocol*. RFC 768. Agosto, 1980.
- Postel, J.: *Transmission Control Protocol*. RFC 793. Septiembre, 1981.
- Ramakrishnan, K.; Floyd, S.: *A Proposal to add Explicit Congestion Notification (ECN) to IP*. RFC 2481. Enero, 1999.
- Ramakrishnan, K.; Floyd, S.; Black, D.: *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC 3168. Septiembre, 2001.
- Stevens, W.: *TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms*. RFC 2001. Enero, 1997.
- Stevens, W.R.: *TCP/IP Illustrated, Vol. 1. The Protocols*. Ed. Addison Wesley, 2000.