# 6

# LA CAPA DE TRANSPORTE

Junto con la capa de red, la capa de transporte es el corazón de la jerarquía de protocolos. La capa de red provee una entrega de paquetes punto a punto mediante el uso de datagramas o circuitos virtuales. La capa de transporte se basa en la capa de red para proveer transporte de datos de un proceso en una máquina de origen a un proceso en una máquina de destino, con un nivel deseado de confiabilidad que es independiente de las redes físicas que se utilizan en la actualidad. Ofrece las abstracciones que necesitan las aplicaciones para usar la red. Sin esta capa, todo el concepto de protocolos por capas tendría muy poco sentido. En este capítulo estudiaremos la capa de transporte con detalle, incluyendo sus servicios y la elección de diseño de la API para lidiar con las cuestiones de confiabilidad, conexiones y control de la congestión, los protocolos como TCP y UDP, y el desempeño.

## 6.1 EL SERVICIO DE TRANSPORTE

En las siguientes secciones veremos una introducción al servicio de transporte. Analizaremos el tipo de servicio que se proporciona a la capa de aplicación. Para que el tema del servicio de transporte sea más concreto, examinaremos dos conjuntos de primitivas de la capa de transporte. Primero analizaremos uno muy sencillo (pero hipotético) para mostrar las ideas básicas. Después veremos la interfaz que se utiliza comúnmente en Internet.

### 6.1.1 Servicios que se proporcionan a las capas superiores

La meta fundamental de la capa de transporte es proporcionar un servicio de transmisión de datos eficiente, confiable y económico a sus usuarios, procesos que normalmente son de la capa de aplicación. Para lograr este objetivo, la capa de transporte utiliza los servicios proporcionados por la capa de red. El hardware o software de la capa de transporte que se encarga del trabajo

se llama **entidad de transporte**, la cual puede localizarse en el kernel (núcleo) del sistema operativo, en un paquete de biblioteca que forma parte de las aplicaciones de red, en un proceso de usuario separado o incluso en la tarjeta de interfaz de red. Las primeras dos opciones son las más comunes en Internet. En la figura 6-1 se ilustra la relación (lógica) entre las capas de red, transporte y aplicación.

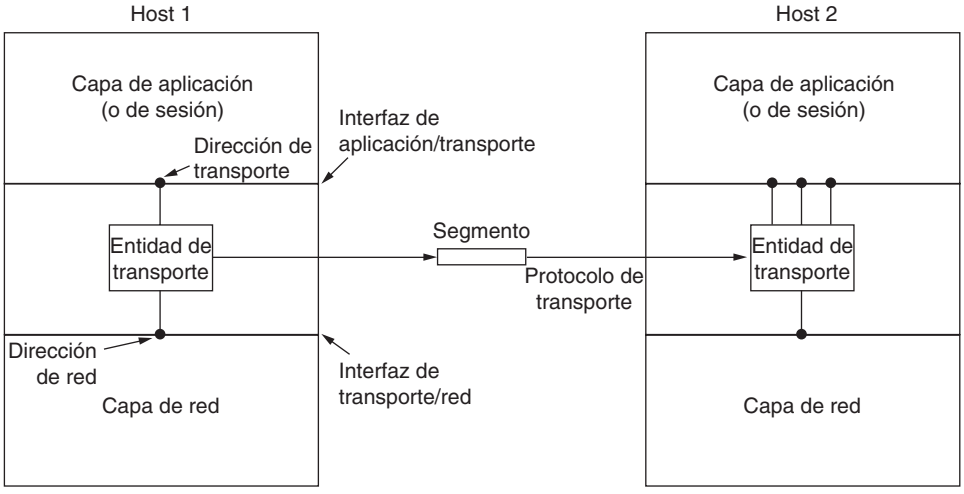


**Figura 6-1.** Las capas de red, transporte y aplicación.

Así como hay dos tipos de servicio de red, orientado a conexión y sin conexión, también hay dos tipos de servicio de transporte. El que está orientado a conexión es parecido en muchos sentidos al servicio de red orientado a conexión. En ambos casos, las conexiones tienen tres fases: establecimiento, transferencia de datos y liberación. El direccionamiento y el control de flujo también son similares en ambas capas. Además, el servicio de transporte sin conexión es muy parecido al servicio de red sin conexión. Sin embargo, puede ser difícil proveer un servicio de transporte sin conexión encima de un servicio de red orientado a conexión, ya que es ineficiente establecer una conexión para enviar un solo paquete y deshacerla justo después.

La pregunta obvia es: si el servicio de la capa de transporte es tan parecido al de la capa de red, ¿por qué hay dos capas diferentes? ¿Por qué no es suficiente una sola capa? La respuesta es sutil, pero crucial. El código de transporte se ejecuta por completo en las máquinas de los usuarios, pero la capa de red se ejecuta en su mayor parte en los enrutadores, los cuales son operados por la empresa portadora (por lo menos en el caso de una red de área amplia). ¿Qué sucede si la capa de red ofrece un servicio inadecuado? ¿Qué tal si esa capa pierde paquetes con frecuencia? ¿Qué ocurre si los enrutadores fallan de vez en cuando?

Problemas, eso es lo que ocurre. Los usuarios no tienen un control real sobre la capa de red, por lo que no pueden resolver los problemas de un mal servicio usando mejores enrutadores o incrementando el manejo de errores en la capa de enlace de datos, puesto que no son dueños de los enrutadores. La única posibilidad es poner encima de la capa de red otra capa que mejore la calidad del servicio. Si en una red sin conexión se pierden paquetes o se rompen, la entidad de transporte puede detectar el problema y compensarlo mediante el uso de retransmisiones. Si, en una red orientada a conexión, se informa a la entidad de transporte a la mitad de una transmisión extensa que su conexión de red ha sido terminada de manera abrupta, sin indicación de lo que ha sucedido a los datos actualmente en tránsito, la entidad puede establecer una nueva conexión de red con la entidad de transporte remota. A través de esta nueva conexión de red, la entidad puede enviar una solicitud a su igual para preguntarle cuáles datos llegaron y cuáles no, y como sabe en dónde se encontraba, puede reiniciar a partir de donde se originó la interrupción.

En esencia, la existencia de la capa de transporte hace posible que el servicio de transporte sea más confiable que la red subyacente. Además, las primitivas de transporte se pueden implementar como llamadas a procedimientos de biblioteca para que sean independientes de las primitivas de red. Las llamadas al servicio de red pueden variar de manera considerable de una red a otra (por ejemplo, las llamadas basadas en una Ethernet sin conexión pueden ser muy distintas de las llamadas en una red WiMAX orientada a conexión). Al ocultar el servicio de red detrás de un conjunto de primitivas de servicio de transporte, aseguramos que para cambiar la red simplemente hay que reemplazar un conjunto de procedimientos de biblioteca por otro que haga lo mismo con un servicio subyacente distinto.

Gracias a la capa de transporte, los programadores de aplicaciones pueden escribir código de acuerdo con un conjunto estándar de primitivas; estos programas pueden funcionar en una amplia variedad de redes sin necesidad de preocuparse por lidiar con diferentes interfaces de red y distintos niveles de confiabilidad. Si todas las redes reales fueran perfectas, tuvieran las mismas primitivas de servicio y se pudiera garantizar que nunca jamás cambiaran, tal vez la capa de transporte sería innecesaria. Sin embargo, en el mundo real esta capa cumple la función clave de aislar a las capas superiores de la tecnología, el diseño y las imperfecciones de la red.

Por esta razón, muchas personas han establecido una distinción cualitativa entre las capas 1 a 4, por una parte, y la(s) capa(s) por encima de la 4, por la otra. Las cuatro capas inferiores se pueden ver como el **proveedor del servicio de transporte**, mientras que la(s) capa(s) superior(es) son el **usuario del servicio de transporte**. Esta distinción entre proveedor y usuario tiene un impacto considerable en el diseño de las capas, además de que posiciona a la capa de transporte en un puesto clave, ya que constituye el límite principal entre el proveedor y el usuario del servicio confiable de transmisión de datos. Es el nivel que ven las aplicaciones.

### 6.1.2 Primitivas del servicio de transporte

Para permitir que los usuarios accedan al servicio de transporte, la capa de transporte debe proporcionar algunas operaciones a los programas de aplicación; es decir, una interfaz del servicio de transporte. Cada servicio de transporte tiene su propia interfaz. En esta sección examinaremos primero un servicio de transporte sencillo (hipotético) y su interfaz para ver los aspectos esenciales. En la siguiente sección veremos un ejemplo real.

El servicio de transporte es similar al servicio de red, pero también hay algunas diferencias importantes. La principal es que el propósito del servicio de red es modelar el servicio ofrecido por las redes reales, con todos sus problemas. Las redes reales pueden perder paquetes, por lo que el servicio de red por lo general no es confiable.

En cambio, el servicio de transporte orientado a conexión sí es confiable. Claro que las redes reales no están libres de errores, pero ése es precisamente el propósito de la capa de transporte: ofrecer un servicio confiable en una red no confiable.

Como ejemplo, considere dos procesos en una sola máquina, conectados mediante una canalización en UNIX (o cualquier otra herramienta de comunicación entre procesos). Ambos consideran que la conexión entre ellos es 100% perfecta. No quieren saber de confirmaciones de recepción, paquetes perdidos, congestión ni nada por el estilo. Lo que quieren es una conexión 100% confiable. El proceso *A* pone datos en un extremo de la canalización y el proceso *B* los saca por el otro extremo. Ésta es la esencia del servicio de transporte orientado a conexión: ocultar las imperfecciones del servicio de red para que los procesos de usuarios puedan dar por hecho simplemente la existencia de un flujo de bits libre de errores, incluso cuando estén en distintas máquinas.

Como nota al margen, la capa de transporte también puede proporcionar un servicio no confiable (de datagramas). Sin embargo, hay muy poco que decir al respecto, además del hecho de que "son datagra-

mas", por lo que en este capítulo nos concentraremos principalmente en el servicio de transporte orientado a conexión. No obstante, hay algunas aplicaciones que se benefician del transporte sin conexión, como la computación cliente-servidor y la multimedia de flujo continuo, por lo que veremos algo sobre ellas más adelante.

Una segunda diferencia entre los servicios de red y de transporte es a quién están dirigidos. El servicio de red lo usan únicamente las entidades de transporte. Pocos usuarios escriben sus propias entidades de transporte y, por lo tanto, pocos usuarios o programas llegan a ver los aspectos internos del servicio de red. En contraste, muchos programas (y, por lo tanto, programadores) ven las primitivas de transporte. En consecuencia, el servicio de transporte debe ser conveniente y fácil de usar.

Para tener una idea de lo que podría ser un servicio de transporte, considere las cinco primitivas listadas en la figura 6-2. Esta interfaz de transporte ciertamente es sencilla, pero muestra la esencia de lo que debe hacer una interfaz de transporte orientada a conexión: permitir que los programas de aplicación establezcan, usen y después liberen las conexiones, lo cual es suficiente para muchas aplicaciones.

| Primitiva | Paquete enviado | Significado |
|---|---|---|
| LISTEN | (ninguno) | Se bloquea hasta que algún proceso intenta conectarse. |
| CONNECT | CONNECTION REQ. | Intenta activamente establecer una conexión. |
| SEND | DATA | Envía información. |
| RECEIVE | (ninguno) | Se bloquea hasta que llegue un paquete DATA. |
| DISCONNECT | DISCONNECTION REQ. | Solicita que se libere la conexión |

**Figura 6-2.** Las primitivas para un servicio simple de transporte.

Para ver cómo podrían usarse estas primitivas, considere una aplicación con un servidor y cierta cantidad de clientes remotos. Para empezar, el servidor ejecuta una primitiva LISTEN, por lo general mediante la llamada a un procedimiento de biblioteca que hace una llamada al sistema para bloquear al servidor hasta que aparezca un cliente. Cuando un cliente desea comunicarse con el servidor, ejecuta una primitiva CONNECT. La entidad de transporte ejecuta esta primitiva, para lo cual bloquea al invocador y envía un paquete al servidor. En la carga útil de este paquete se encuentra un mensaje de capa de transporte encapsulado, dirigido a la entidad de transporte del servidor.

Aquí es pertinente una nota rápida sobre la terminología. A falta de un mejor término, usaremos **segmento** para indicar los mensajes que se envían de una entidad de transporte a otra. TCP, UDP y otros protocolos de Internet usan este término. Algunos protocolos antiguos usaban el nombre de **TPDU** (**Unidad de Datos del Protocolo de Transporte**, del inglés *Transport Protocol Data Unit*). Este término ya no se utiliza mucho, pero todavía se puede ver en publicaciones y libros antiguos.

Así, los segmentos (intercambiados por la capa de transporte) están contenidos en paquetes (intercambiados por la capa de red). A su vez, estos paquetes están contenidos en tramas (intercambiadas por la capa de enlace de datos). Cuando llega una trama, la capa de enlace de datos procesa el encabezado de la trama y, si la dirección de destino coincide para la entrega local, pasa el contenido del campo de carga útil de la trama a la entidad de red. Esta última procesa de manera similar el encabezado del paquete y después pasa el contenido de la carga útil del paquete a la entidad de transporte. Este anidamiento se ilustra en la figura 6-3.

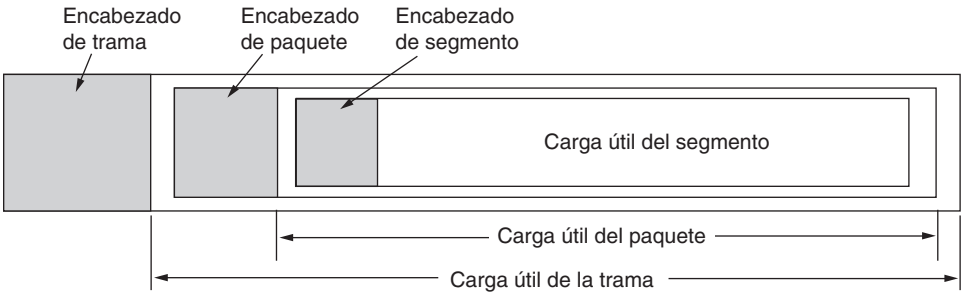


**Figura 6-3.** Anidamiento de segmentos, paquetes y tramas.

Ahora regresemos a nuestro ejemplo de cliente-servidor. La llamada CONNECT del cliente ocasiona el envío de un segmento CONNECTION REQUEST (solicitud de conexión) al servidor. Al llegar este segmento, la entidad de transporte verifica que el servidor esté bloqueado en LISTEN (es decir, que esté interesado en manejar solicitudes). Si es así, entonces desbloquea el servidor y envía un segmento CONNECTION ACCEPTED (conexión aceptada) de regreso al cliente. Al llegar este segmento, el cliente se desbloquea y se establece la conexión.

Ahora pueden intercambiarse datos mediante las primitivas SEND y RECEIVE. En la forma más simple, cualquiera de las dos partes puede emitir un RECEIVE (bloqueo) para esperar que la otra parte emita un SEND. Al llegar el segmento, el receptor se desbloquea. Entonces puede procesar el segmento y enviar una respuesta. Mientras ambos lados puedan llevar el control de quién tiene el turno para transmitir, este esquema funciona bien.

Observe que en la capa de transporte, incluso un intercambio de datos unidireccional es más complicado que en la capa de red. También se confirmará (tarde o temprano) la recepción de cada paquete de datos enviado. Asimismo, la recepción de los paquetes que llevan segmentos de control se confirmará de manera implícita o explícita. Estas confirmaciones se manejan mediante las entidades de transporte usando el protocolo de capa de red, por lo que no son visibles para los usuarios de transporte. De la misma forma, las entidades de transporte tienen que preocuparse por los temporizadores y las retransmisiones. Los usuarios de transporte no se enteran de ningún aspecto de esta mecánica. Para ellos una conexión es un conducto de bits confiable: un usuario mete bits en él y por arte de magia aparecen en el otro lado, con el mismo orden. Esta habilidad de ocultar la complejidad es la razón por la cual los protocolos en capas son herramientas tan poderosas.

Cuando ya no es necesaria una conexión, hay que liberarla para desocupar espacio en las tablas de las dos entidades de transporte. La desconexión tiene dos variantes: asimétrica y simétrica. En la variante asimétrica, cualquiera de los dos usuarios de transporte puede emitir una primitiva DISCONNECT, con lo cual se envía un segmento DISCONNECT a la entidad de transporte remota. A su llegada se libera la conexión.

En la variante simétrica, cada dirección se cierra por separado, independientemente de la otra. Cuando una de las partes emite una primitiva DISCONNECT, quiere decir que ya no tiene más datos por enviar pero aún está dispuesta a recibir datos de la otra parte. En este modelo, una conexión se libera cuando ambas partes han emitido una primitiva DISCONNECT.

En la figura 6-4 se presenta un diagrama de estado para establecer y liberar una conexión para estas sencillas primitivas. Cada transición se activa mediante algún evento, ya sea una primitiva ejecutada por el usuario de transporte local o la llegada de un paquete. Por simplicidad, supongamos que la confirmación de recepción de cada segmento se realiza por separado. También supondremos que se usa un modelo de desconexión simétrica, en donde el cliente se desconecta primero. Cabe señalar que este modelo es muy poco sofisticado. Más adelante analizaremos modelos más realistas cuando describamos la forma en que funciona TCP.

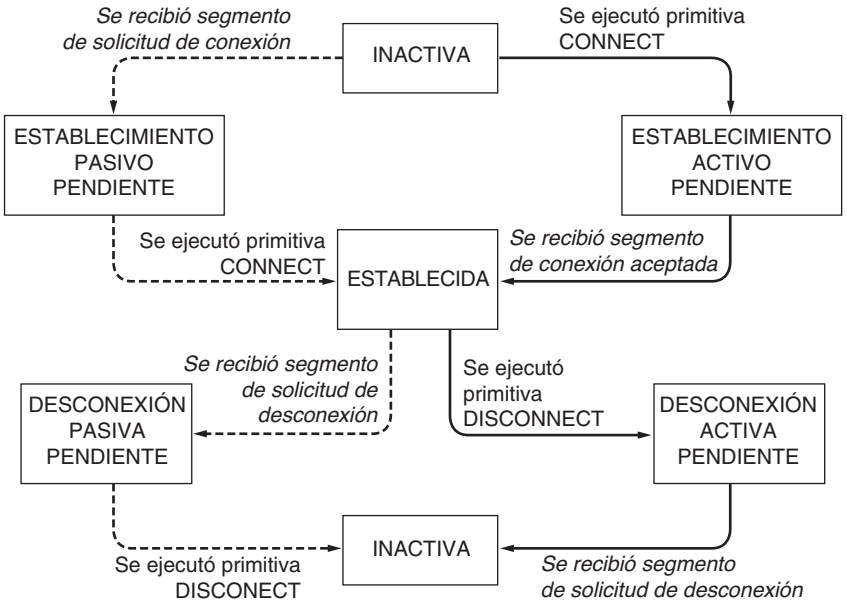


**Figura 6-4.** Un diagrama de estado para un esquema simple de manejo de conexiones. Las transiciones etiquetadas en cursiva se producen debido a la llegada de paquetes. Las líneas continuas muestran la secuencia de estados del cliente. Las líneas punteadas muestran la secuencia de estados del servidor.

### 6.1.3 Sockets de Berkeley

Inspeccionemos brevemente otro conjunto de primitivas de transporte: las primitivas de socket que se utilizan para TCP. Los sockets se liberaron primero como parte de la distribución de software de Berkeley UNIX 4.2BSD en 1983. Su popularidad aumentó muy rápido. Las primitivas se utilizan mucho en la actualidad para la programación de Internet en muchos sistemas operativos, en especial los sistemas basados en UNIX; también hay una API estilo sockets para Windows, llamada *winsock*.

Las primitivas se listan en la figura 6-5. En esencia, siguen el modelo de nuestro primer ejemplo pero ofrecen más características y flexibilidad. No analizaremos aquí los segmentos correspondientes. Eso lo haremos después.

| Primitiva | Significado |
|---|---|
| SOCKET | Crea un nuevo punto terminal de comunicación. |
| BIND | Asocia una dirección local con un socket. |
| LISTEN | Anuncia la disposición de aceptar conexiones; indica el tamaño de la cola. |
| ACCEPT | Establece en forma pasiva una conexión entrante. |
| CONNECT | Intenta establecer activamente una conexión. |
| SEND | Envía datos a través de la conexión. |
| RECEIVE | Recibe datos de la conexión. |
| CLOSE | Libera la conexión. |

**Figura 6-5.** Las primitivas de socket para TCP.

Los servidores ejecutan las primeras cuatro primitivas de la lista en ese orden. La primitiva SOCKET crea un nuevo punto terminal y le asigna espacio en las tablas de la entidad de transporte. Los parámetros de la llamada especifican el formato de direccionamiento que se utilizará, el tipo de servicio deseado (por ejemplo, flujo confiable de bytes) y el protocolo. Una llamada SOCKET con éxito devuelve un descriptor de archivo ordinario que se utiliza con las siguientes llamadas, de la misma manera que lo hace una llamada OPEN.

Los sockets recién creados no tienen direcciones de red. Éstas se asignan mediante la primitiva BIND. Una vez que un servidor ha destinado una dirección a un socket, los clientes remotos se pueden conectar a él. La razón para que la llamada SOCKET no cree directamente una dirección es que algunos procesos se encargan de sus direcciones (por ejemplo, han estado usando su misma dirección durante años y todos la conocen), mientras que otros no lo hacen.

A continuación viene la llamada LISTEN, que asigna espacio para poner en cola las llamadas entrantes por si varios clientes intentan conectarse al mismo tiempo. A diferencia de la llamada LISTEN de nuestro primer ejemplo, en el modelo de sockets LISTEN no es una llamada bloqueadora.

Para bloquearse en espera de una conexión entrante, el servidor ejecuta una primitiva ACCEPT. Cuando llega un segmento que solicita una conexión, la entidad de transporte crea un socket nuevo con las mismas propiedades que el original y devuelve un descriptor de archivo para él. A continuación, el servidor puede ramificar un proceso o hilo para manejar la conexión en el socket nuevo y regresar a esperar la siguiente conexión en el socket original. ACCEPT devuelve un descriptor de archivo que se puede utilizar para leer y escribir de la forma estándar, al igual que con los archivos.

Ahora veamos el lado cliente. Aquí también se debe crear primero un socket mediante la primitiva SOCKET, pero no se requiere BIND puesto que la dirección usada no le importa al servidor. La primitiva CONNECT bloquea al invocador y comienza el proceso de conexión. Al completar este proceso (es decir, cuando se recibe un segmento apropiado del servidor) el proceso cliente se desbloquea y se establece la conexión. Ambos lados pueden usar ahora SEND y RECEIVE para transmitir y recibir datos a través de la conexión full-dúplex. Las llamadas de sistema READ y WRITE de UNIX también se pueden utilizar si no son necesarias las opciones especiales de SEND y RECEIVE.

La liberación de las conexiones a los sockets es simétrica. La conexión se libera cuando ambos lados ejecutan una primitiva CLOSE.

Los sockets se han vuelto muy populares, además de ser el estándar de facto para abstraer servicios de transporte para las aplicaciones. La API de sockets se utiliza comúnmente con el protocolo TCP para ofrecer un servicio orientado a conexión, conocido como **flujo confiable de bytes**, que consiste en el conducto de bits confiable que describimos antes. Sin embargo, se podrían usar otros protocolos para implementar este servicio mediante el uso de la misma API. Debería ser igual para todos los usuarios del servicio de transporte.

Un punto fuerte de la API de sockets es que cualquier aplicación la puede utilizar para otros servicios de transporte. Por ejemplo, se pueden usar sockets con un servicio de transporte sin conexión. En este caso, CONNECT establece la dirección del transporte del igual remoto, mientras que SEND y RECEIVE envían y reciben datagramas hacia/desde el igual remoto (también es común usar un conjunto expandido de llamadas —por ejemplo, SENDTO y RECEIVEFROM— que enfaticen los mensajes y no limiten una aplicación a un solo igual de transporte). Los sockets también se pueden usar con protocolos de transporte que proporcionen un flujo continuo de mensajes en vez de un flujo continuo de bytes, y que dispongan o no de un control de congestión. Por ejemplo, **DCCP** (**Protocolo de Control de Congestión de Datagramas**, del inglés *Datagram Congestion Controlled Protocol*) es una versión de UDP con control de congestión (Kohler y colaboradores, 2006). Depende de los usuarios de transporte comprender qué servicio están recibiendo.

Sin embargo, no es probable que los sockets vayan a ser la última palabra en las interfaces de transporte. Por ejemplo, las aplicaciones trabajan a menudo con un grupo de flujos relacionados, como un

navegador web que solicita varios objetos al mismo servidor. Con los sockets, lo más natural es que los programas de aplicación usen un flujo por cada objeto. Esta estructura significa que el control de congestión se aplica por separado para cada flujo, no entre todo el grupo, lo cual es subóptimo. Esto libra a la aplicación de la carga de tener que administrar el conjunto. Se han desarrollado protocolos e interfaces más recientes que soportan grupos de flujos relacionados de una forma más efectiva y simple para la aplicación. Dos ejemplos son **SCTP** (**Protocolo de Control de Transmisión de Flujo**, del inglés *Stream Control Transmission Protocol* ), que se define en el RFC 4960, y **SST** (**Transporte Estructurado de Flujo**, del inglés *Structured Stream Transport*) (Ford, 2007). Estos protocolos deben modificar un poco la API para obtener los beneficios de los grupos de flujos relacionados; además soportan características tales como una mezcla de tráfico orientado a conexión y sin conexión, e incluso múltiples rutas de red. El tiempo dirá si tienen éxito o fracasan.

### 6.1.4 Un ejemplo de programación de sockets: un servidor de archivos de Internet

Veamos ahora el código de cliente y servidor de la figura 6-6 como ejemplo del uso real de las llamadas de sockets. Ahí se muestra un servidor de archivos de Internet muy primitivo, junto con un cliente de ejemplo que lo utiliza. El código tiene muchas limitaciones (que analizaremos más adelante), pero en principio el código del servidor se puede compilar y ejecutar en cualquier máquina UNIX conectada a Internet y el código del cliente también se puede compilar y ejecutar en cualquier otra máquina UNIX en Internet, en cualquier parte del mundo. El código del cliente se puede ejecutar con los parámetros apropiados para obtener cualquier archivo al que el servidor tenga acceso en su máquina. El archivo se escribe a la salida estándar que, por supuesto, se puede redirigir a un archivo o conducto.

Veamos primero el código del servidor. Comienza con algunos encabezados estándar, los últimos tres contienen las principales definiciones y estructuras de datos relacionadas con Internet. A continuación se encuentra una definición de *SERVER_PORT* como 12345. Este número se eligió de manera arbitraria. Cualquier número que se encuentre entre 1024 y 65535 funcionará siempre y cuando otro proceso no lo esté utilizando; los puertos debajo de 1023 están reservados para los usuarios privilegiados.

Las siguientes líneas del servidor definen dos constantes necesarias. La primera determina el tamaño de bloque en bytes que se utiliza para la transferencia de archivos. La segunda determina cuántas conexiones pendientes se pueden almacenar antes de empezar a descartar las conexiones adicionales que vayan llegando.

```
/* Esta página contiene un programa cliente que puede solicitar un archivo desde el programa servidor de la siguiente
    página. El servidor responde
* enviando el archivo completo.
*/
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 12345                    /* arbitrario, pero el cliente y el servidor deben coincidir */
#define BUF_SIZE 4096                        /* tamaño de bloque para transferencia */

int main(int argc, char **argv)
{
 int c, s, bytes;
 char buf[BUF_SIZE];                         /* búfer para el archivo entrante */
 struct hostent *h;                          /* información sobre el servidor */
 struct sockaddr_in channel;                 /* contiene la dirección IP */
```

(*continúa*)

```
 if (argc != 3) fatal("Usar: cliente nombre-servidor nombre-archivo");                    (continuación)
 h = gethostbyname(argv[1]);                          /* busca la dirección IP del host */
 if (!h) fatal("falla en gethostbyname ");

 s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
 if (s < 0) fatal("socket");
 memset(&channel, 0, sizeof(channel));
 channel.sin_family= AF_INET;
 memcpy(&channel.sin_addr.s_addr, h->h_addr, h->h_length);
 channel.sin_port= htons(SERVER_PORT);

 c = connect(s, (struct sockaddr *) &channel, sizeof(channel));
 if (c < 0) fatal("falla en conexion");

 /* Se ha establecido la conexión. Se envía el nombre del archivo incluyendo el byte 0 al final. */
 write(s, argv[2], strlen(argv[2])+1);

 /* Obtiene el archivo y lo escribe en la salida estándar. */
 while (1) {
     bytes = read(s, buf, BUF_SIZE);                  /* lee del socket */
     if (bytes <= 0) exit(0);                                  /* verifica el final del archivo */
     write(1, buf, bytes);                                     /* escribe en la salida estándar */
 }
}

fatal(char *string)
{
 printf("%s\n", string);
 exit(1);
}
```

**Figura 6-6.** Código del cliente que utiliza sockets. El código del servidor está en la siguiente página.

```
#include <sys/types.h>                                /* Éste es el código del servidor */
#include <sys/fcntl.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 12345                             /* arbitrario, pero el cliente y el servidor deben coincidir */
#define BUF_SIZE 4096                                 /* tamaño de bloque para la transferencia */
#define QUEUE_SIZE 10

int main(int argc, char *argv[])
{
 int s, b, l, fd, sa, bytes, on = 1;
 char buf[BUF_SIZE];                                  /* búfer para el archivo saliente */
 struct sockaddr_in channel;                          /* contiene la dirección IP */

 /* Construye la estructura de la dirección para enlazar el socket. */
 memset(&channel, 0, sizeof(channel));                /* canal cero */
 channel.sin_family = AF_INET;
 channel.sin_addr.s_addr = htonl(INADDR_ANY);
 channel.sin_port = htons(SERVER_PORT);

 /* Apertura pasiva. Espera una conexión. */
 s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);       /* crea el socket */
 if (s < 0) fatal("falla en socket ");
 setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *) &on, sizeof(on));                    (continúa)
```

(*continuación*)

```
 b = bind(s, (struct sockaddr *) &channel, sizeof(channel));
 if (b < 0) fatal("falla en bind ");
 l = listen(s, QUEUE_SIZE);                    /* especifica el tamaño de la cola */
 if (l < 0) fatal("falla en listen");

 /* El socket ahora está configurado y enlazado. Espera una conexión y la procesa. */
 while (1) {
     sa = accept(s, 0, 0);                     /* se bloquea para la solicitud de conexión */
     if (sa < 0) fatal("falla en accept");

     read(sa, buf, BUF_SIZE);                  /* lee el nombre del archivo desde el socket */

     /* Obtiene y regresa el archivo.*/
     fd = open(buf, O_RDONLY);                 /* abre el archivo para regresarlo */
     if (fd < 0) fatal("falla en open");

     while (1) {
        bytes = read(fd, buf, BUF_SIZE); /* lee del archivo */
        if (bytes <= 0) break;                 /* verifica el final del archivo */
        write(sa, buf, bytes);                 /* escribe bytes en el socket */
     }
     close(fd);                                /* cierra el archivo */
     close(sa);                                /* cierra la conexión */
 }
}
```

El código del servidor empieza después de las declaraciones de las variables locales. Primero inicializa una estructura de datos que contendrá la dirección IP del servidor. Esta estructura de datos pronto se anexará al socket del servidor. La llamada a *memset* establece en 0s toda la estructura de datos. Las siguientes tres asignaciones llenarán tres de sus campos. El último de éstos contiene el puerto del servidor. Las funciones *htonl* y *htons* están relacionadas con la conversión de valores a un formato estándar a fin de que el código se ejecute correctamente, tanto en las máquinas *little-endian* (por ejemplo, Intel x86) como en las máquinas big-endian (por ejemplo, la SPARC). Su semántica exacta no es importante aquí.

A continuación el servidor crea un socket y verifica si hay errores (lo cual se indica mediante $s < 0$). En una versión de producción del código, el mensaje de error puede ser un poco más explicativo. La llamada a *setsockopt* es necesaria para permitir la reutilización del puerto, a fin de que el servidor se pueda ejecutar de manera indefinida, sorteando una solicitud tras otra. A continuación, la dirección IP se enlaza con el socket y se realiza una verificación para ver si la llamada a *bind* tuvo éxito. El último paso en la inicialización es la llamada a *listen* para anunciar que el servidor está dispuesto a aceptar llamadas entrantes e indicar al sistema que almacene la cantidad de estas llamadas, que se especifica en QUEUE_SIZE, en caso de que lleguen más solicitudes mientras el servidor aún esté procesando la actual. Si la cola está llena y llegan solicitudes adicionales, se descartan en silencio.

En este punto el servidor entra a su ciclo principal, del cual nunca sale. La única forma de detenerlo es desde afuera. La llamada a *accept* bloquea el servidor hasta que algún cliente trata de establecer una conexión con él. Si la llamada a *accept* tiene éxito, se devuelve un descriptor de archivo que se puede utilizar para leer y escribir, de la misma forma en la que se pueden usar los descriptores de archivo para leer y escribir hacia/desde las canalizaciones. Sin embargo, a diferencia de las canalizaciones, que son unidireccionales, los sockets son bidireccionales, por lo que *sa* (el socket aceptado) se puede utilizar para leer de la conexión y también para escribir en ella. Un descriptor de archivo de canalización es para leer o escribir, pero no para ambas cosas.

Una vez que se establece la conexión, el servidor lee en ella el nombre del archivo. Si el nombre aún no está disponible, el servidor se bloquea y lo espera. Una vez que obtiene el nombre, el servidor abre el archivo y luego entra en un ciclo que alterna entre leer bloques del archivo y escribirlos en el socket hasta que el archivo se haya copiado por completo. A continuación, el servidor cierra el archivo y la conexión, y espera hasta que aparezca la siguiente conexión. Este ciclo se repite de manera indefinida.

Ahora analicemos el código del cliente. Para entender su funcionamiento es necesario comprender cómo se invoca. Suponiendo que se llama *cliente*, una llamada típica es:

```
cliente flits.cs.vu.nl /usr/tom/nombredearchivo >f
```

Esta llamada sólo funciona si el servidor ya se está ejecutando en *flits.cs.vu.nl,* si existe el archivo */usr/tom/nombredearchivo* y si el servidor tiene acceso de lectura a él. Si la llamada es exitosa, el archivo se transfiere a través de Internet y se escribe en *f*, después de lo cual finaliza el programa cliente. Puesto que el servidor continúa después de una transferencia, el cliente puede iniciarse una y otra vez para obtener otros archivos.

El código del cliente inicia con algunas inclusiones y declaraciones. La ejecución verifica primero si se invocó el código con el número correcto de argumentos (*argc* = 3 significa el nombre del programa más dos argumentos). Observe que *argv* [1] contiene el nombre del servidor (por ejemplo, *flits.cs.vu.nl* ) y que *gethostbyname* lo convierte en una dirección IP. Esta función utiliza DNS para buscar el nombre. En el capítulo 7 estudiaremos el sistema DNS.

A continuación se crea e inicializa un socket. Después de esto, el cliente intenta establecer una conexión TCP con el servidor mediante *connect*. Si el servidor está activo y ejecutándose en la máquina especificada y enlazado a *SERVER_PORT*, y si está inactivo o tiene espacio en su cola *listen*, la conexión se establecerá (en algún momento dado). El cliente envía el nombre del archivo mediante esta conexión, para lo cual escribe en el socket. El número de bytes enviados es un byte mayor que el propio nombre, puesto que también hay que enviar el byte 0 de terminación del nombre para indicar al servidor en dónde termina dicho nombre.

Ahora el cliente entra en un ciclo, lee el archivo bloque por bloque desde el socket y lo copia a la salida estándar. Cuando termina, simplemente abandona la conexión.

El procedimiento *fatal* imprime un mensaje de error y termina. El servidor necesita el mismo procedimiento, sólo que lo omitimos debido a la falta de espacio en la página. Puesto que el cliente y el servidor se compilan por separado y, por lo general, se ejecutan en computadoras diferentes, no pueden compartir el código de *fatal*.

Estos dos programas (así como el material adicional relacionado con este libro) se pueden obtener del sitio web del libro

```
http://www.pearsonhighered.com/tanenbaum
```

Sólo como aclaración, este servidor no es lo último en tecnología de servidores. Su proceso de verificación de errores no es muy bueno y su proceso de reporte de errores es regular. Puesto que maneja todas las solicitudes estrictamente en forma secuencial (ya que cuenta con un solo hilo), su desempeño es pobre. Es evidente que nunca ha escuchado sobre la seguridad; además, utilizar sólo llamadas de sistema UNIX no es la manera de obtener independencia de la plataforma. También da por sentados algunos detalles que son técnicamente ilegales, como suponer que el nombre del archivo cabe en el búfer y se transmite instantáneamente. Pese a estas deficiencias, es un servidor de archivos de Internet funcional. En los ejercicios, invitamos al lector a mejorarlo. Para mayor información sobre la programación con sockets, consulte a Donahoo y Calvert (2008, 2009).

## 6.2 ELEMENTOS DE LOS PROTOCOLOS DE TRANSPORTE

El servicio de transporte se implementa mediante un **protocolo de transporte** entre las dos entidades de transporte. En ciertos aspectos, los protocolos de transporte se parecen a los protocolos de enlace de datos que estudiamos con detalle en el capítulo 3. Ambos se encargan del control de errores, la secuenciación y el control de flujo, entre otros aspectos.

Sin embargo, existen diferencias considerables entre los dos, las cuales se deben a disimilitudes importantes entre los entornos en que operan ambos protocolos, como se muestra en la figura 6-7. En la capa de enlace de datos, dos enrutadores se comunican de forma directa mediante un canal físico, ya sea cableado o inalámbrico, mientras que, en la capa de transporte, ese canal físico se sustituye por la red completa. Esta diferencia tiene muchas implicaciones importantes para los protocolos.

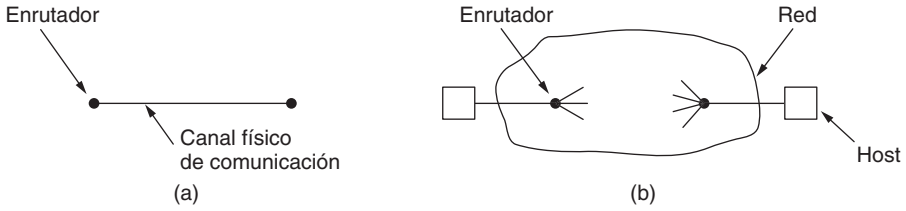


**Figura 6-7.** (a) Entorno de la capa de enlace de datos. (b) Entorno de la capa de transporte.

Por una parte, a través de los enlaces de punto a punto tales como los cables o la fibra óptica, por lo general no es necesario que un enrutador especifique con quién quiere comunicarse; cada línea de salida conduce de manera directa a un enrutador específico. En la capa de transporte se requiere el direccionamiento explícito de los destinos.

Por otro lado, el proceso de establecer una conexión a través del cable de la figura 6-7(a) es sencillo: el otro extremo siempre está ahí (a menos que se caiga, en cuyo caso no estará ahí). Sea como fuere, no hay mucho que hacer. Aún en enlaces inalámbricos el proceso no es muy diferente. Sólo basta enviar un mensaje para que llegue a todos los demás destinos. Si no se confirma la recepción del mensaje debido a un error, se puede reenviar. En la capa de transporte, el establecimiento inicial de la conexión es complicado, como veremos.

Otra diferencia (muy irritante) entre la capa de enlace de datos y la capa de transporte es la existencia potencial de capacidad de almacenamiento en la red. Cuando un enrutador envía un paquete a través de un enlace, éste puede llegar o perderse, pero no puede andar de un lado a otro durante un rato, esconderse en un rincón alejado del mundo y aparecer de repente después de otros paquetes que se hayan enviado mucho después. Si la red usa datagramas, los cuales se enrutan por separado en el interior, hay una probabilidad nada despreciable de que un paquete pueda tomar la ruta escénica, llegar tarde y fuera del orden esperado, o incluso que lleguen duplicados de ese paquete. Las consecuencias de la capacidad de la red de retrasar y duplicar paquetes algunas veces pueden ser desastrosas y puede requerir el uso de protocolos especiales para transportar la información de la manera correcta.

Una última diferencia entre las capas de enlace de datos y de transporte es de cantidad, más que de tipo. Se requieren búferes y control de flujo en ambas capas, pero la presencia de una cantidad de conexiones grande y variable en la capa de transporte, con un ancho de banda que fluctúa a medida que las conexiones compiten entre sí, puede requerir un enfoque distinto del que se usa en la capa de enlace de datos. Algunos de los protocolos que vimos en el capítulo 3 asignan una cantidad fija de búferes a cada línea de modo que, al llegar una trama, siempre hay un búfer disponible. En la capa de transporte, la gran cantidad de conexiones que se deben manejar, además de las variaciones en el ancho de banda que

puede recibir cada conexión, hacen menos atractiva la idea de dedicar muchos búferes a cada una. En las siguientes secciones examinaremos todos estos importantes temas, además de otros.

### 6.2.1 Direccionamiento

Cuando un proceso de aplicación (por ejemplo, un usuario) desea establecer una conexión con un proceso de aplicación remoto, debe especificar a cuál se conectará (el transporte sin conexión tiene el mismo problema: ¿a quién debe enviarse cada mensaje?). El método que se emplea por lo general es definir direcciones de transporte en las que los procesos puedan escuchar las solicitudes de conexión. En Internet, estos puntos terminales se denominan **puertos**. Usaremos el término genérico **TSAP** (**Punto de Acceso al Servicio de Transporte**, del inglés *Transport Service Access Point*) para indicar un punto terminal específico en la capa de transporte. Así, no es sorpresa que los puntos terminales análogos en la capa de red (es decir, direcciones de capa de red) se llamen **NSAP** (**Punto de Acceso al Servicio de Red**, del inglés *Network Service Access Points*). Las direcciones IP son ejemplos de NSAP.

En la figura 6-8 se ilustra la relación entre el NSAP, el TSAP y la conexión de transporte. Los procesos de aplicación, tanto clientes como servidores, se pueden enlazar por sí mismos a un TSAP para establecer una conexión a un TSAP remoto. Estas conexiones se realizan a través de puntos NSAP en cada host, como se muestra. El propósito de tener puntos TSAP es que, en algunas redes, cada computadora tiene un solo NSAP, por lo que se necesita alguna forma de diferenciar los múltiples puntos terminales de transporte que comparten ese punto NSAP.

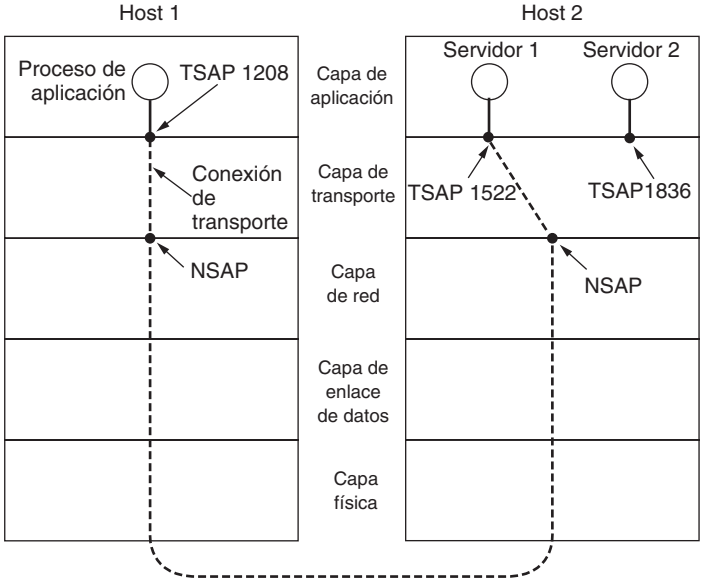


**Figura 6-8.** Los puntos TSAP y NSAP, junto con las conexiones de transporte.

El siguiente es un posible escenario para una conexión de transporte:

1. Un proceso servidor de correo se enlaza con el TSAP 1522 en el host 2 para esperar una llamada entrante. La manera en que un proceso se enlaza con un TSAP está fuera del modelo de red y depende por completo del sistema operativo local. Por ejemplo, se podría usar una llamada como nuestra LISTEN.

2. Un proceso de aplicación en el host 1 quiere enviar un mensaje de correo, por lo que se enlaza con el TSAP 1208 y emite una solicitud CONNECT. Esta solicitud especifica el TSAP 1208 en el host 1 como el origen y el TSAP 1522 en el host 2 como destino. En última instancia, esta acción provoca que se establezca una conexión de transporte entre el proceso de aplicación y el servidor.
3. El proceso de aplicación envía el mensaje de correo.
4. El servidor de correo responde para decir que entregará el mensaje.
5. Se libera la conexión de transporte.

Observe que en el host 2 podría haber otros servidores enlazados a otros puntos TSAP en espera de conexiones entrantes que lleguen a través del mismo NSAP.

El panorama anterior está muy bien, excepto que hemos ocultado un pequeño problema bajo la alfombra: ¿cómo sabe el proceso de usuario del host 1 que el servidor de correo está conectado al TSAP 1522? Una posibilidad es que el servidor de correo se haya estado enlazando con el TSAP 1522 durante años y que gradualmente todos los usuarios de la red hayan aprendido esto. En este modelo, los servicios tienen direcciones TSAP estables que se listan en archivos en lugares bien conocidos. Por ejemplo, el archivo */etc/services* de los sistemas UNIX, que lista cuáles servidores están enlazados de manera permanente a cuáles puertos, incluyendo el hecho de que el servidor de correo se encuentra en el puerto TCP 25.

Aunque las direcciones TSAP estables funcionan bien con una cantidad pequeña de servicios clave que nunca cambian (por ejemplo, el servidor web), en general, es común que los procesos de usuario deseen comunicarse con otros procesos de usuario que no tienen una dirección TSAP conocida por adelantado, o que sólo existan durante un tiempo corto.

Para manejar esta situación, podemos utilizar un esquema alternativo en el que existe un proceso especial llamado **asignador de puertos** (*portmapper*). Para encontrar la dirección TSAP correspondiente a un nombre de servicio específico, como "BitTorrent", un usuario establece una conexión al asignador de puertos (que escucha en un TSAP bien conocido). Entonces, el usuario envía un mensaje en el que especifica el nombre del servicio y el asignador de puertos le regresa la dirección TSAP. A continuación, el usuario libera la conexión con el asignador de puertos y establece una nueva conexión con el servicio deseado.

En este modelo, cuando se crea un nuevo servicio, éste se debe registrar con el asignador de puertos para proporcionarle su nombre de servicio (por lo general, una cadena ASCII) y su TSAP. El asignador de puertos registra esta información en su base de datos interna, de manera que cuando empiecen a llegar las consultas más tarde, conozca las respuestas.

La función del asignador de puertos es análoga a la de un operador de asistencia de directorios en el sistema telefónico: provee una asociación de nombres con números. Al igual que en el sistema telefónico, es imprescindible que la dirección del TSAP bien conocido que utilice el asignador de puertos sea en realidad bien conocida. Si usted no conoce el número del operador de información, no puede llamarle para averiguarlo. Si cree que el número que marca para solicitar información es obvio, intente hacerlo en algún otro país cuando tenga oportunidad.

Muchos de los procesos servidor que pueden existir en una máquina sólo se utilizarán pocas veces. Es un desperdicio tenerlos a todos activos y escuchando en una dirección TSAP todo el día. En la figura 6-9 se muestra un esquema alternativo en forma simplificada. Este esquema se conoce como **protocolo de conexión inicial**. En vez de que cada uno de los servidores existentes escuche en un TSAP bien conocido, cada máquina que desea ofrecer servicios a usuarios remotos tiene un **servidor de procesos** especial, el cual actúa como proxy de los servidores que se usan menos. Este servidor se llama *inetd* en los sistemas UNIX, y escucha a un conjunto de puertos al mismo tiempo, en espera de una solicitud de conexión. Los usuarios potenciales de un servicio comienzan por emitir una solicitud CONNECT, en la que especifican la dirección TSAP del servicio que desean. Si no hay ningún servidor esperándolos, consiguen una conexión al servidor de procesos, como se muestra en la figura 6-9(a).

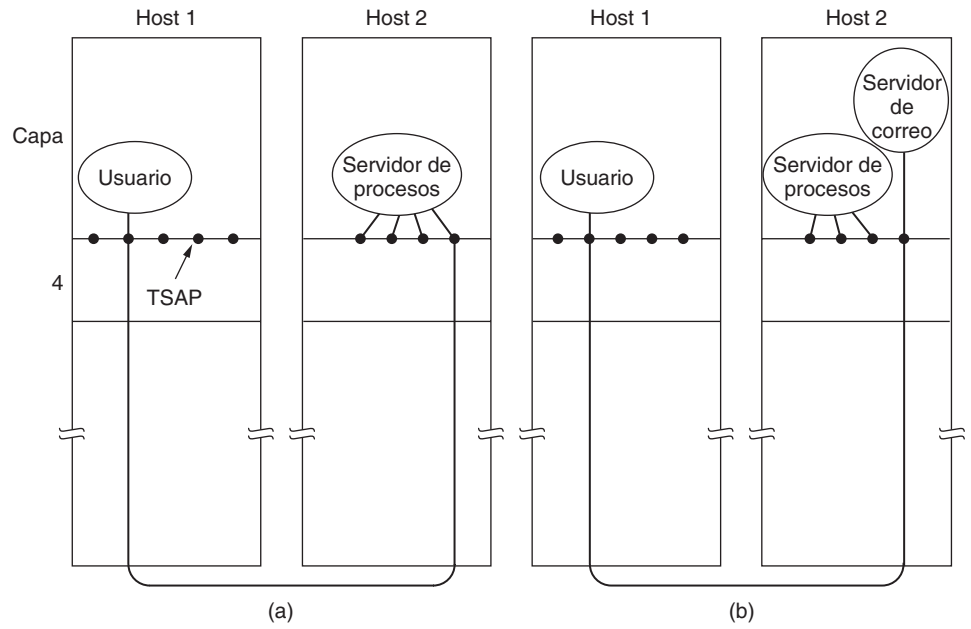


**Figura 6-9.** Cómo establece un proceso de usuario en el host 1 una conexión con un servidor de correo en el host 2, mediante un servidor de procesos.

Tras obtener la solicitud entrante, el servidor de procesos genera el servidor solicitado y le permite heredar la conexión existente con el usuario. El nuevo servidor hace el trabajo solicitado, mientras que el servidor de procesos regresa a escuchar nuevas solicitudes, como se muestra en la figura 6-9(b). Este método sólo se puede aplicar cuando los servidores se crean bajo demanda.

### 6.2.2 Establecimiento de una conexión

El proceso de establecer una conexión suena fácil, pero en realidad es sorprendentemente complicado. A primera vista parecería suficiente con que una entidad de transporte enviara tan sólo un segmento CONNECTION REQUEST al destino y esperara una respuesta CONNECTION ACCEPTED. El problema ocurre cuando la red puede perder, retrasar, corromper y duplicar paquetes. Este comportamiento causa complicaciones serias.

Imagine una red que está tan congestionada que las confirmaciones de recepción casi nunca regresan a tiempo, y cada paquete expira y se retransmite dos o tres veces. Suponga que la red usa datagramas en su interior y que cada paquete sigue una ruta diferente. Algunos de los paquetes podrían atorarse en un congestionamiento de tráfico dentro de la red y tardar mucho tiempo en llegar. Es decir, se pueden retardar en la red y reaparecer mucho después, cuando el emisor piense que se han perdido.

La peor pesadilla posible es la siguiente. Un usuario establece una conexión con un banco y envía mensajes para indicar al banco que transfiera una gran cantidad de dinero a la cuenta de una persona no del todo confiable. Por desgracia, los paquetes toman la ruta escénica hacia el destino y parten para explorar un rincón lejano de la red. Entonces el temporizador del emisor expira y envía todos los paquetes de nuevo. Esta vez los paquetes toman la ruta más corta y se entregan con rapidez, por lo que el emisor libera la conexión.

Por desgracia, en un momento dado el lote inicial de paquetes sale finalmente de su escondite y llega al destino en orden; le pide al banco que establezca una nueva conexión y que transfiera dinero (otra vez).

El banco no tiene manera de saber que estos paquetes son duplicados. Debe suponer que ésta es una segunda transacción independiente, y transfiere de nuevo el dinero.

Este escenario puede parecer poco probable o incluso inverosímil, pero el punto es el siguiente: los protocolos se deben diseñar de manera que sean correctos en todos los casos. Sólo es necesario implementar eficientemente los casos comunes para obtener un buen desempeño de la red, pero el protocolo debe ser capaz de lidiar con los casos no comunes sin quebrantarse. En caso de que no pueda, hemos construido una red poco confiable que puede fallar sin avisarnos cuando las condiciones se pongan difíciles.

Durante el resto de esta sección estudiaremos el problema de los duplicados con retardo, con un énfasis en los algoritmos para establecer conexiones de una manera confiable, de modo que no ocurran pesadillas como la anterior. El meollo del problema es que los duplicados con retardo se consideran paquetes nuevos. No podemos evitar que los paquetes se dupliquen y retrasen. Pero si esto ocurre, los paquetes deben ser rechazados como duplicados y no procesados como paquetes nuevos.

El problema se puede atacar de varias maneras, ninguna de las cuales es muy satisfactoria. Una es usar direcciones de transporte desechables. En este enfoque, cada vez que se requiere una dirección de transporte, se genera una nueva. Cuando una conexión es liberada, se descarta la dirección y no se vuelve a utilizar. Así, los paquetes duplicados con retardo nunca encuentran su camino hacia un proceso de transporte y no pueden hacer ningún daño. Sin embargo, esta estrategia dificulta la conexión con un proceso.

Otra posibilidad es dar a cada conexión un identificador único (es decir, un número de secuencia que se incremente con cada conexión establecida) elegido por la parte iniciadora y puesto en cada segmento, incluyendo el que solicita la conexión. Después de liberar cada conexión, cada entidad de transporte puede actualizar una tabla que liste conexiones obsoletas como pares (entidad de transporte de igual, identificador de conexión). Cada vez que entre una solicitud de conexión, se puede verificar con la tabla para saber si pertenece a una conexión previamente liberada.

Por desgracia, este esquema tiene una falla básica: requiere que cada entidad de transporte mantenga una cierta cantidad de información histórica durante un tiempo indefinido. Esta historia debe persistir tanto en la máquina de origen como en la de destino. De lo contrario, si una máquina falla y pierde su memoria, ya no sabrá qué identificadores de conexión ya han utilizado sus iguales.

Más bien, necesitamos un enfoque diferente para simplificar el problema. En lugar de permitir que los paquetes vivan eternamente dentro de la red, debemos idear un mecanismo para eliminar a los paquetes viejos que aún andan vagando por ahí. Con esta restricción, el problema se vuelve algo más manejable.

El tiempo de vida de un paquete puede restringirse a un máximo conocido mediante el uso de una (o más) de las siguientes técnicas:

1. Un diseño de red restringido.
2. Colocar un contador de saltos en cada paquete.
3. Marcar el tiempo en cada paquete.

La primera técnica incluye cualquier método que evite que los paquetes hagan ciclos, combinado con una manera de limitar el retardo, incluyendo la congestión a través de la trayectoria más larga posible (ahora conocida). Es difícil, dado que el alcance de las interredes puede variar, desde una sola ciudad hasta un alcance internacional. El segundo método consiste en inicializar el contador de saltos con un valor apropiado y decrementarlo cada vez que se reenvíe el paquete. El protocolo de red simplemente descarta cualquier paquete cuyo contador de saltos llega a cero. El tercer método requiere que cada paquete lleve la hora en la que fue creado, y que los enrutadores se pongan de acuerdo en descartar cualquier paquete que haya rebasado cierto tiempo predeterminado. Este último método requiere que los relojes de los enrutadores estén sincronizados, lo que no es una tarea fácil, además, en la práctica un contador de saltos es una aproximación suficientemente cercana a la edad.

En la práctica, necesitaremos garantizar no sólo que un paquete está eliminado, sino que todas sus confirmaciones de recepción también están eliminadas, por lo que ahora introduciremos un periodo *T*, que es un múltiplo pequeño del tiempo de vida máximo verdadero del paquete. El tiempo de vida máximo del paquete es una constante conservadora para una red; en Internet se toma de manera arbitraria como 120 segundos. El múltiplo depende del protocolo y simplemente tiene el efecto de hacer a *T* más largo. Si esperamos un tiempo de *T* segundos después de enviar un paquete, podemos estar seguros de que todos sus rastros ya han desaparecido, y que ni él ni sus confirmaciones de recepción aparecerán repentinamente de la nada para complicar el asunto.

Al limitar los tiempos de vida de los paquetes, es posible proponer una manera práctica y a prueba de errores para rechazar segmentos duplicados con retardo. El método descrito a continuación se debe a Tomlinson (1975); después Sunshine y Dalal (1978) lo refinaron. Las variantes de este método se utilizan mucho en la práctica, incluyendo en TCP.

La base del método es que el origen etiquete los segmentos con números de secuencia que no se vayan a reutilizar durante *T* segundos. El periodo *T* y la tasa de paquetes por segundo determinan el tamaño de los números de secuencia. De esta manera, sólo un paquete con un número de secuencia específico puede estar pendiente en cualquier momento dado. Aún puede haber duplicados de este paquete, en cuyo caso el destino debe descartarlos. Sin embargo, ya no se da el caso en que un duplicado con retardo de un paquete pueda vencer a un nuevo paquete con el mismo número de secuencia y que el destino lo acepte.

Para resolver el problema de una máquina que pierde toda la memoria acerca de su estado tras una falla, una posibilidad es exigir a las entidades de transporte que estén inactivas durante *T* segundos después de una recuperación. El periodo inactivo permitirá que todos los segmentos antiguos expiren, por lo que el emisor podrá empezar de nuevo con cualquier número de secuencia. Sin embargo, en una interred (interconexión de redes) compleja el periodo *T* puede ser grande, por lo que esta estrategia no es muy atractiva.

En cambio, Tomlinson propuso equipar cada host con un reloj. Los relojes de los distintos hosts no necesitan estar sincronizados. Se supone que cada reloj tiene la forma de un contador binario que se incrementa a sí mismo a intervalos uniformes. Además, la cantidad de bits del contador debe ser igual o mayor que la cantidad de bits en los números de secuencia. Por último, y lo más importante, se supone que el reloj continúa operando aunque el host falle.

Cuando se establece una conexión, los *k* bits de menor orden del reloj se usan como número de secuencia inicial de *k* bits. Por tanto, y a diferencia de los protocolos del capítulo 3, cada conexión comienza a numerar sus segmentos con un número de secuencia inicial diferente. El espacio de secuencia también debe ser lo bastante grande para que, para cuando los números de secuencia se reinicien, los segmentos antiguos con el mismo número de secuencia hayan desaparecido hace mucho tiempo. En la figura 6-10(a) se muestra esta relación lineal entre tiempo y números secuenciales iniciales. La región prohibida muestra los tiempos en donde los números de secuencia de los segmentos son ilegales previo a su uso. Si se envía cualquier segmento con un número de secuencia en esta región, se podría retrasar y hacerse pasar por un paquete distinto con el mismo número de secuencia que se emitirá un poco más tarde. Por ejemplo, si el host falla y se reinicia en la marca de tiempo de 70 segundos, utilizará los números de secuencia iniciales con base en el reloj para continuar a partir de donde se quedó; el host no empieza con un número de secuencia inferior en la región prohibida.

Una vez que ambas entidades de transporte han acordado el número de secuencia inicial, se puede usar cualquier protocolo de ventana deslizante para el control de flujo de datos. Este protocolo de ventana encontrará y descartará exactamente los paquetes duplicados después de que éstos hayan sido aceptados. En realidad, la curva de números de secuencia iniciales (que se indica mediante la línea gruesa) no es lineal, sino una escalera, ya que el reloj avanza en pasos discretos. Por sencillez, ignoraremos este detalle.

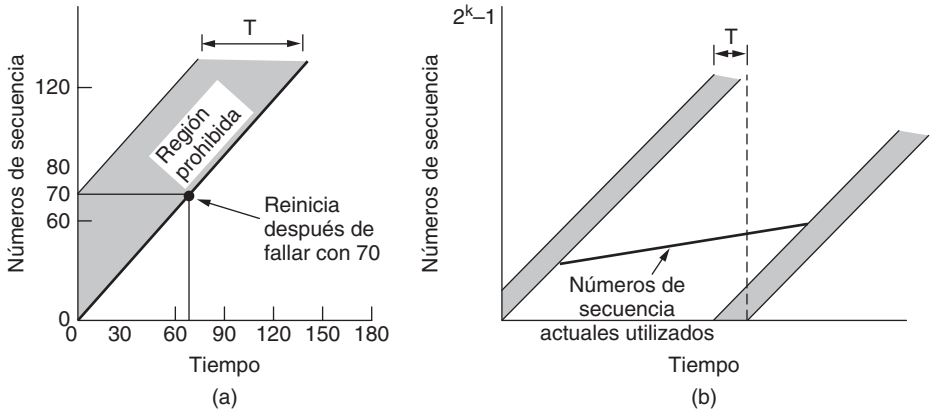


**Figura 6-10.** (a) Los segmentos no pueden entrar en la región prohibida. (b) El problema de resincronización.

Para mantener los números de secuencia fuera de la región prohibida, necesitamos tener cuidado con dos aspectos. Podemos meternos en problemas de dos maneras distintas. Si un host envía muchos datos con demasiada rapidez en una conexión recién abierta, el número de secuencia actual contra la curva de tiempo puede subir en forma más pronunciada que el número de secuencia inicial contra la curva de tiempo, lo cual provoca que el número de secuencia entre a la región prohibida. Para evitar que esto ocurra, la tasa máxima de datos en cualquier conexión es de un segmento por cada pulso de reloj. Esto significa que la entidad de transporte debe esperar hasta que el reloj emita un pulso antes de abrir una nueva conexión después de un reinicio por falla, no sea que el mismo número se utilice dos veces. Ambos puntos están a favor de un pulso corto de reloj (1 μseg o menor). Pero el reloj no puede pulsar demasiado rápido en relación con el número de secuencia. Para una tasa de reloj de $C$ y un espacio de números de secuencia de tamaño $S$, la condición es que $S/C > T$ para que los números de secuencia no se reinicien con tanta rapidez.

Entrar a la región prohibida por la parte inferior al enviar con demasiada rapidez no es la única forma de meterse en problemas. De la figura 6-10(b) podemos ver que, a cualquier tasa de datos menor que la tasa de reloj, la curva de números de secuencia actuales utilizados vs el tiempo entrará en un momento dado a la región prohibida desde la izquierda, mientras los números de secuencia se reinician. Entre mayor sea la pendiente de los números de secuencia actuales, más se retardará este evento. Al evitar esta situación se limita el grado de lentitud con que los números de secuencia pueden avanzar en una conexión (o qué tanto pueden durar las conexiones).

El método basado en reloj resuelve el problema de no poder diferenciar los segmentos duplicados con retardo de los segmentos nuevos. Sin embargo, hay un inconveniente práctico en cuanto a su uso para establecer conexiones. Como por lo general no recordamos los números de secuencia de una conexión a otra en el destino, aún no tenemos forma de saber si un segmento CONNECTION REQUEST que contiene un número de secuencia inicial es un duplicado de una conexión reciente. Este inconveniente no existe durante una conexión, ya que el protocolo de la ventana deslizante sí recuerda el número de secuencia actual.

Para resolver este problema específico, Tomlinson (1975) desarrolló el **acuerdo de tres vías** (*three-way handshake*). Este protocolo de establecimiento implica que un igual verifique con el otro que la solicitud de conexión sea realmente actual. El procedimiento normal de establecimiento al iniciar el host 1 se muestra en la figura 6-11(a). El host 1 escoge un número de secuencia, $x$, y envía al host 2 un segmento CONNECTION REQUEST que contiene ese número. El host 2 responde con un segmento ACK para confirmar la recepción de $x$ y anunciar su propio número de secuencia inicial, $y$. Por último, el host 1 confirma la recepción del número de secuencia inicial seleccionado por el host 2 en el primer segmento de datos que envía.

Ahora veamos la manera en que funciona el acuerdo de tres vías en presencia de segmentos de control duplicados con retardo. En la figura 6-11(b), el primer segmento es un CONNECTION REQUEST duplicado con retardo de una conexión antigua. Este segmento llega al host 2 sin el conocimiento del host 1. El host 2 reacciona a este segmento y envía al host 1 un segmento ACK, para solicitar en efecto la comprobación de que el host 1 haya tratado realmente de establecer una nueva conexión. Cuando el host 1 rechaza el intento del host 2 por establecer una conexión, el host 2 se da cuenta de que fue engañado por un duplicado con retardo y abandona la conexión. De esta manera, un duplicado con retardo no causa daño.

El peor caso ocurre cuando en la subred deambulan tanto un segmento CONNECTION REQUEST con retardo como un ACK. Este caso se muestra en la figura 6-11(c). Como en el ejemplo anterior, el host 2 recibe un CONNECTION REQUEST con retardo y lo contesta. En este momento es imprescindible tener en cuenta que el host 2 ha propuesto usar *y* como número de secuencia inicial para el tráfico del host 2 al host 1, sabiendo bien que no existen todavía segmentos que contengan el número de secuencia *y* ni confirmaciones de recepción de *y*. Cuando llega el segundo segmento con retardo al host 2, el hecho de

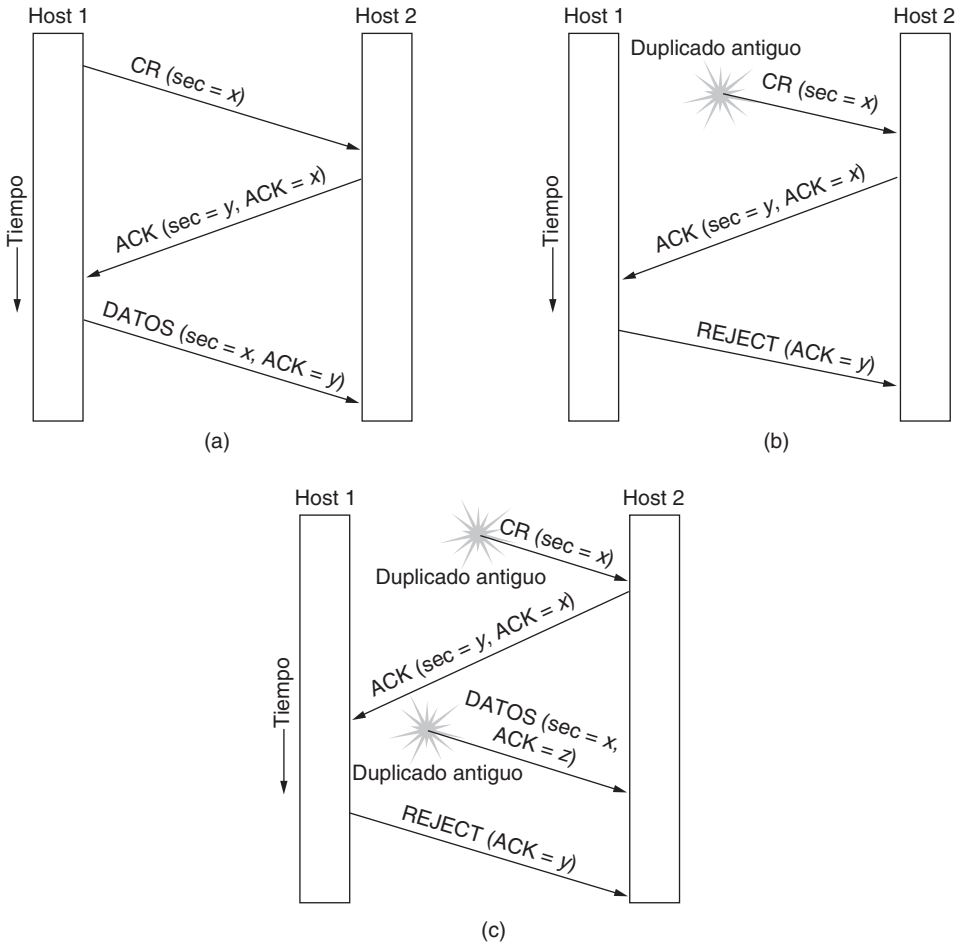


**Figura 6-11.** Tres escenarios del protocolo para establecer una conexión mediante un acuerdo de tres vías. CR significa CONNECTION REQUEST. (a) La operación normal. (b) Un CONNECTION REQUEST duplicado antiguo que aparece de la nada. (c) CONNECTION REQUEST duplicado y ACK duplicado.

que se confirmó la recepción de *z* en lugar de *y* indica al host 2 que éste también es un duplicado antiguo. Lo importante que debemos tener en cuenta aquí es que no hay una combinación de segmentos antiguos que puedan provocar la falla del protocolo y permitan establecer una conexión accidental cuando nadie la quiere.

TCP usa este acuerdo de tres vías para establecer las conexiones. Dentro de una conexión se utiliza una estampa de tiempo para extender el número de secuencia de 32 bits, de manera que no se reinicie en un intervalo menor al tiempo de vida máxima del paquete, incluso para las conexiones de gigabits por segundo. Este mecanismo es una corrección al TCP, la cual fue necesaria debido a que este protocolo se utilizaba en enlaces cada vez más rápidos. Se describe en el RFC 1323 y se llama **PAWS** (**Protección Contra el Reinicio de Números de Secuencia**, del inglés *Protection Against Wrapped Sequence Numbers*). Entre conexiones, para los números de secuencia y antes de que PAWS pudiera entrar en acción, TCP utilizaba originalmente el esquema basado en reloj que describimos antes. Sin embargo, se detectó una vulnerabilidad de seguridad: mediante el reloj, un atacante podía predecir con facilidad el siguiente número de secuencia inicial y enviar paquetes que engañaran al acuerdo de tres vías para establecer una conexión falsificada. Para cerrar este agujero, se utilizan números de secuencia inicial seudoaleatorios para las conexiones en la práctica. Sin embargo, aún es importante que no se repitan los números de secuencia iniciales durante un intervalo, incluso cuando parezcan aleatorios a quien los esté observando. En caso contrario, los duplicados con retardo pueden provocar un caos.

### 6.2.3 Liberación de una conexión

Es más fácil liberar una conexión que establecerla. No obstante, hay más obstáculos de los que uno podría imaginar. Como mencionamos antes, hay dos estilos para terminar una conexión: liberación asimétrica y liberación simétrica. La liberación asimétrica es la manera en que funciona el sistema telefónico: cuando una de las partes cuelga, se interrumpe la conexión. La liberación simétrica trata la conexión como dos conexiones unidireccionales distintas y requiere que cada una se libere por separado.

La liberación asimétrica es abrupta y puede provocar la pérdida de datos. Considere el escenario de la figura 6-12. Una vez que se establece la conexión, el host 1 envía un segmento que llega en forma apropiada al host 2. A continuación, el host 1 envía otro segmento. Por desgracia, el host 2 emite un DISCONNECT antes de que llegue el segundo segmento. El resultado es que se libera la conexión y se pierden los datos.

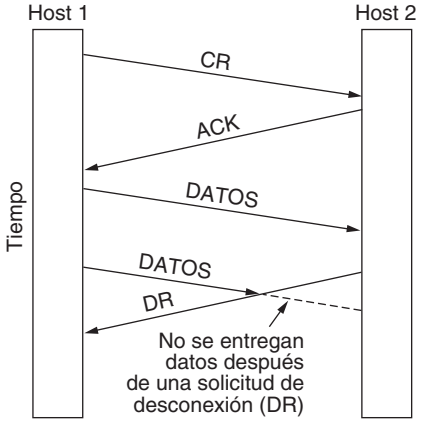


**Figura 6-12.** Desconexión abrupta con pérdida de datos.

Es obvio que se requiere un protocolo de liberación más sofisticado para evitar la pérdida de los datos. Una posibilidad es usar la liberación simétrica, en la que cada dirección se libera en forma independiente de la otra. Aquí, un host puede continuar recibiendo datos, aun después de haber enviado un segmento DISCONNECT.

La liberación simétrica es ideal cuando cada proceso tiene una cantidad fija de datos por enviar y sabe con certeza cuándo los ha enviado. En otras situaciones, el proceso de determinar si se ha efectuado o no todo el trabajo y si debe terminar o no la conexión no es tan obvio. Podríamos pensar en un protocolo en el que el host 1 diga: "Ya terminé. ¿Terminaste también?" Si el host 2 responde: "Ya terminé también. Adiós", la conexión se puede liberar sin problemas.

Por desgracia, este protocolo no siempre funciona. Hay un problema famoso que tiene que ver con ese asunto. Se conoce como el **problema de los dos ejércitos**. Imagine que un ejército blanco está acampado en un valle, como se muestra en la figura 6-13. En los dos cerros que rodean al valle hay ejércitos azules. El ejército blanco es más grande que cualquiera de los dos ejércitos azules por separado, pero juntos éstos son más grandes que el ejército blanco. Si cualquiera de los dos ejércitos azules ataca por su cuenta, será derrotado, pero si los dos atacan a la vez obtendrán la victoria.

Los ejércitos azules quieren sincronizar sus ataques. Sin embargo, su único medio de comunicación es el envío de mensajeros a pie a través del valle, donde podrían ser capturados y se perdería el mensaje (es decir, tienen que usar un canal de comunicación no confiable). La pregunta es: ¿existe un protocolo que permita que los ejércitos azules ganen?

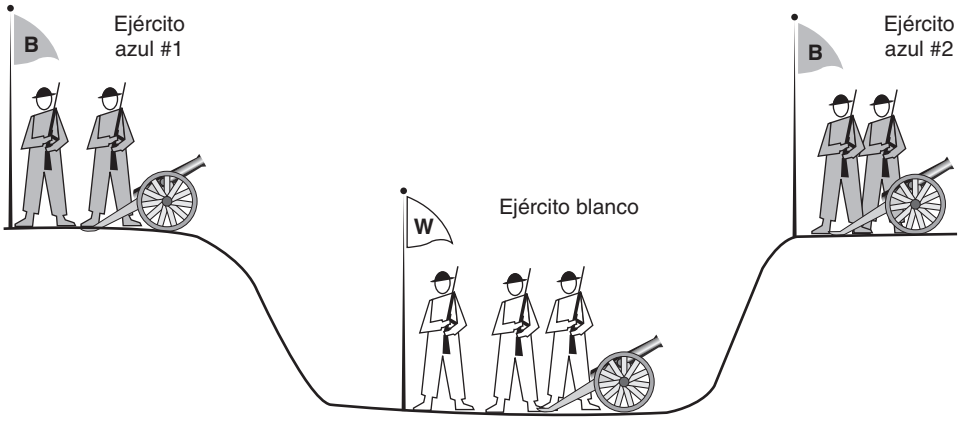


**Figura 6-13.** El problema de los dos ejércitos.

Supongamos que el comandante del ejército azul #1 envía un mensaje que dice: "Propongo que ataquemos al amanecer del 29 de marzo. ¿Qué les parece?" Ahora supongamos que llega el mensaje y que el comandante del ejército azul #2 está de acuerdo, y que su respuesta llega con seguridad al ejército azul #1. ¿Ocurrirá el ataque? Es probable que no, porque el comandante #2 no sabe si su respuesta llegó. Si no llegó, el ejército azul #1 no atacará, y sería tonto de su parte emprender el combate.

Mejoremos ahora el protocolo para convertirlo en un acuerdo de tres vías. El iniciador de la propuesta original debe confirmar la recepción de la respuesta. Suponiendo que no se pierden mensajes, el ejército azul #2 recibirá la confirmación de recepción, pero ahora el que dudará será el comandante del ejército azul #1. A fin de cuentas, no sabe si ha llegado su confirmación de recepción y, si no llegó, sabe que el ejército #2 no atacará. Podríamos probar ahora un protocolo de acuerdo de cuatro vías, pero tampoco ayudaría.

De hecho, podemos demostrar que no existe un protocolo que funcione. Supongamos que existiera algún protocolo. O el último mensaje del protocolo es esencial, o no lo es. Si no lo es, podemos eliminarlo (así como los demás mensajes no esenciales) hasta que quede un protocolo en el que todos los mensajes sean esenciales. ¿Qué ocurre si el mensaje final no pasa? Acabamos de decir que es esencial, por lo que, si se pierde, el ataque no ocurrirá. Dado que el emisor del mensaje final nunca puede estar seguro de su llegada, no se arriesgará a atacar. Peor aún, el otro ejército azul sabe esto, por lo que tampoco atacará.

Para ver la relevancia del problema de los dos ejércitos en relación con la liberación de conexiones, simplemente sustituya "atacar" por "desconectar". Si ninguna de las partes está preparada para desconectarse hasta estar convencida de que la otra está preparada para desconectarse también, nunca ocurrirá la desconexión.

En la práctica podemos evitar este dilema al eludir la necesidad de un acuerdo y pasar el problema al usuario de transporte, de modo que cada lado pueda decidir por su cuenta si se completó o no la comunicación. Éste es un problema más fácil de resolver. En la figura 6-14 se ilustran cuatro escenarios de liberación mediante el uso de un acuerdo de tres vías. Aunque este protocolo no es infalible, es adecuado en la mayoría de los casos.

En la figura 6-14(a) vemos el caso normal en el que uno de los usuarios envía un segmento DR de solicitud de desconexión (DISCONNECTION REQUEST) con el fin de iniciar la liberación de una conexión. Al llegar, el receptor devuelve también un segmento DR e inicia un temporizador, por si acaso se pierde su DR. Cuando este DR llega, el emisor original envía de regreso un segmento ACK y libera la conexión. Finalmente, cuando llega el segmento ACK, el receptor también libera la conexión. Liberar una conexión significa que la entidad de transporte remueve la información sobre la conexión de su tabla de conexiones abiertas y avisa de alguna manera al dueño de la conexión (el usuario de transporte). Esta acción es diferente a aquélla en la que el usuario de transporte emite una primitiva DISCONNECT.

Si se pierde el último segmento ACK, como se muestra en la figura 6-14(b), el temporizador salva la situación. Al expirar el temporizador, la conexión se libera de todos modos.

Ahora consideremos el caso en el que se pierde el segundo DR. El usuario que inicia la desconexión no recibirá la respuesta esperada, su temporizador expirará y todo comenzará de nuevo. En la figura 6-14(c) vemos la manera en que esto funciona, suponiendo que la segunda vez no se pierden segmentos, y que todos se entregan correctamente y a tiempo.

Nuestro último escenario, la figura 6-14(d), es el mismo que en la figura 6-14(c), excepto que ahora suponemos que todos los intentos repetidos de retransmitir el segmento DR también fallan debido a los segmentos perdidos. Después de *N* reintentos, el emisor simplemente se da por vencido y libera la conexión. Mientras tanto, expira el temporizador del receptor y también se sale.

Aunque por lo general basta con este protocolo, en teoría puede fallar si se pierden el DR inicial y *N* retransmisiones. El emisor se dará por vencido y liberará la conexión, pero el otro lado no sabrá nada sobre los intentos de desconexión y seguirá plenamente activo. Esta situación provoca una conexión semiabierta.

Pudimos haber evitado este problema al impedir que el emisor se diera por vencido después de *N* reintentos y obligarlo a seguir insistiendo hasta recibir una respuesta. No obstante, si permitimos que expire el temporizador en el otro lado, entonces el emisor continuará por siempre, pues nunca llegará una respuesta. Si no permitimos que expire el temporizador en el lado receptor, el protocolo queda suspendido en la figura 6-14(d).

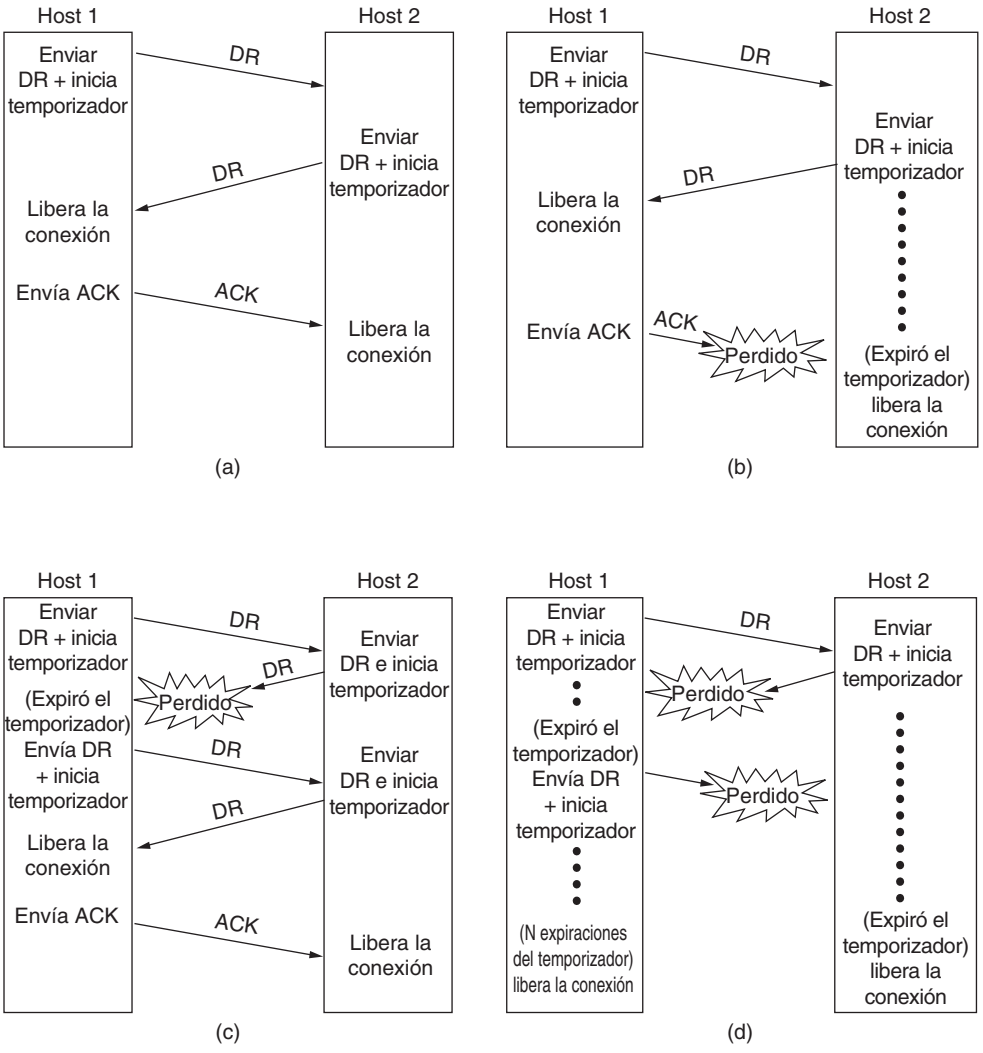


**Figura 6-14.** Cuatro escenarios de un protocolo para liberar una conexión. (a) Caso normal del acuerdo de tres vías. (b) Pérdida del último ACK. (c) Respuesta perdida. (d) Respuesta perdida y pérdida de los segmentos DR subsecuentes.

Una manera de eliminar las conexiones semiabiertas es tener una regla que diga que, si no han llegado segmentos durante ciertos segundos, se libera automáticamente la conexión. De esta manera, si un lado llega a desconectarse, el otro lado detectará la falta de actividad y también se desconectará. Esta regla también se encarga del caso donde se rompe la conexión (porque la red ya no puede entregar paquetes entre los hosts) sin que ninguno de los extremos se desconecte. Por supuesto que si se pone en práctica esta regla, es necesario que cada entidad de transporte tenga un temporizador que se detenga y se reinicie cada vez que se envíe un segmento. Si expira este temporizador se transmite un segmento ficticio, simplemente para evitar que el otro lado se desconecte. Por otra parte, si se usa la regla de desconexión automática y se pierden demasiados segmentos ficticios seguidos en una conexión que de otro modo estaría en reposo, primero se desconectará de forma automática un lado y después el otro.

No insistiremos más en este punto, pero ya debe quedar claro que liberar una conexión sin pérdida de datos no es ni remotamente tan simple como parece a primera instancia. Aquí la lección es que el usuario de transporte debe estar involucrado a la hora de decidir cuándo desconectarse; el problema no se puede resolver tan sólo mediante las entidades de transporte. Para ver la importancia de la aplicación, considere que mientras TCP realiza por lo general un cierre simétrico (en donde cada lado cierra de manera independiente su mitad de la conexión con un paquete FIN cuando termina de enviar sus datos), muchos servidores web envían al cliente un paquete RST que provoca un cierre repentino de la conexión, lo cual es más parecido a un cierre asimétrico. Esto funciona sólo porque el servidor web conoce el patrón del intercambio de datos. Primero recibe una solicitud del cliente, que incluye todos los datos que enviará ese cliente, y después envía una respuesta al cliente. Cuando el servidor web termina con su respuesta, se han enviado todos los datos en ambas direcciones. El servidor puede enviar una advertencia al cliente y cerrar en forma repentina la conexión. Si el cliente recibe esta advertencia, liberará su estado de conexión en ese momento. Si no recibe la advertencia, en algún momento detectará que el servidor ya no se está comunicando con él y liberará el estado de la conexión. En cualquiera de los casos, los datos se transfieren con éxito.

### 6.2.4 Control de errores y almacenamiento en búfer

Ya que examinamos cómo establecer y liberar conexiones con cierto detalle, veamos ahora la manera en que se manejan las conexiones mientras están en uso. Las cuestiones clave son el control de errores y el control de flujo. El control de errores consiste en asegurar que los datos se entreguen con el nivel deseado de confiabilidad, por lo general que todos los datos se entreguen sin errores. El control de flujo consiste en evitar que un transmisor rápido sature a un receptor lento.

Ya vimos ambas cuestiones antes, cuando estudiamos la capa de enlace de datos. Las soluciones que se utilizan en la capa de transporte son los mismos mecanismos que estudiamos en el capítulo 3. A continuación veremos una muy breve recapitulación:

1. Una trama transporta un código de detección de errores (por ejemplo, CRC o suma de verificación), el cual se utiliza para comprobar que la información se haya recibido de manera correcta.
2. Una trama transporta un número de secuencia para identificarse a sí misma; el emisor la retransmite hasta que reciba una confirmación de recepción exitosa por parte del receptor. A esto se le conoce como **ARQ** (**Solicitud Automática de Repetición**, del inglés *Automatic Repeat reQuest*).
3. Hay un número máximo de tramas pendientes que el emisor permitirá en un momento dado, y se detendrá si el receptor no envía confirmaciones de recepción de las tramas con la rapidez suficiente. Si este máximo es de un paquete, el protocolo se denomina **parada y espera**. Las ventanas más grandes permiten canalizaciones y mejoran el desempeño en enlaces largos y rápidos.
4. El protocolo de la **ventana deslizante** combina estas características y también se utiliza para soportar la transferencia de datos bidireccional.

Dado que estos mecanismos se utilizan en tramas de la capa de enlace, es natural preguntarse por qué se utilizarían también en segmentos de la capa de transporte. Sin embargo, en la práctica hay una pequeña duplicación entre las capas de enlace y de transporte. A pesar de que se utilizan los mismos mecanismos, existen diferencias en cuanto a función y grado.

Para una diferencia en función, considere la detección de errores. La suma de verificación de la capa de enlace protege una trama mientras atraviesa un solo enlace. La suma de verificación de la capa de

transporte protege un segmento mientras atraviesa una trayectoria de red completa. Es una verificación de punto a punto; que no es lo mismo que realizar una verificación en cada enlace. Saltzer y colaboradores (1984) describen una situación en que los paquetes se corrompieron dentro de un enrutador. Las sumas de verificación de la capa de enlace protegían a los paquetes sólo mientras viajaban a través de un enlace, no mientras estaban dentro del enrutador. Por ende, los paquetes se entregaron en forma incorrecta, incluso cuando eran correctos de acuerdo con las verificaciones en cada enlace.

Éste y otros ejemplos condujeron a Saltzer y colaboradores a que articularan el **argumento punto a punto**. De acuerdo con este argumento, la verificación de la capa de transporte que se efectúa de un punto a otro es esencial para la precisión, y aunque las verificaciones de la capa de enlace no son esenciales, sí son valiosas para mejorar el desempeño (ya que sin ellas se puede enviar innecesariamente un paquete corrupto a través de toda la ruta).

Como una diferencia de grado, considere las retransmisiones y el protocolo de ventana deslizante. Con excepción de los enlaces de satélite, la mayoría de los enlaces inalámbricos sólo pueden tener una trama pendiente del emisor en cualquier momento dado. Esto es, el producto de ancho de banda-retardo para el enlace es tan pequeño que ni siquiera se puede almacenar una trama completa dentro del enlace. En este caso, un tamaño de ventana pequeño es suficiente para un buen desempeño. Por ejemplo, 802.11 usa un protocolo de parada y espera, en donde transmite o retransmite cada trama y espera a recibir una confirmación de recepción antes de pasar a la siguiente trama. Un tamaño de ventana más grande aumentaría la complejidad sin mejorar el desempeño. Para los enlaces cableados y de fibra óptica, como Ethernet (conmutada) o las redes troncales de ISP, la tasa de errores es tan baja que se pueden omitir las retransmisiones de la capa de enlace, porque las retransmisiones punto a punto repararán la pérdida de tramas residuales.

Por otro lado, muchas conexiones TCP tienen un producto de ancho de banda-retardo mucho mayor que un solo segmento. Considere una conexión que envía datos a través de Estados Unidos a 1 Mbps y con un tiempo de ida y vuelta de 100 mseg. Incluso para esta conexión lenta se almacenarán 200 Kbits de datos en el receptor, en el tiempo requerido para enviar un segmento y recibir una confirmación de recepción. Para estos casos se debe usar una ventana deslizante grande. El protocolo de parada y espera paralizaría el desempeño. En nuestro ejemplo, limitaría el desempeño a un segmento cada 200 mseg, o 5 segmentos/seg sin importar qué tan rápida sea en realidad la red.

Dado que, por lo general, los protocolos de transporte usan ventanas deslizantes más grandes, analizaremos con más cuidado la cuestión de colocar los datos en un búfer. Como un host puede tener muchas conexiones, cada una de las cuales se trata por separado, puede requerir una cantidad considerable de búferes para las ventanas deslizantes. Se necesitan búferes tanto en el emisor como en el receptor. Sin duda, se requieren en el emisor para contener todos los segmentos transmitidos, para los cuales todavía no se ha enviado una confirmación de recepción. Se precisan ahí debido a que estos segmentos se pueden perder y tal vez necesiten retransmitirse.

Sin embargo, como el emisor está usando búferes, tal vez el receptor pueda o no dedicar búferes específicos a conexiones específicas, según lo considere apropiado. Por ejemplo, el receptor puede mantener un solo grupo de búferes compartido por todas las conexiones. Cuando entre un segmento, se hará un intento por adquirir un nuevo búfer en forma dinámica. Si hay uno disponible, se acepta el segmento; en caso contrario, se descarta. Como el emisor está preparado para retransmitir los segmentos perdidos por la red, no hay daño permanente al permitir que el receptor descarte segmentos, aunque se desperdician algunos recursos. El emisor simplemente sigue intentando hasta que recibe una confirmación de recepción.

La mejor solución de compromiso entre usar búferes en el origen o usarlos en el destino depende del tipo de tráfico transmitido por la conexión. Para el tráfico en ráfagas con bajo ancho de banda, como el que produce una terminal interactiva, es razonable no dedicar ningún búfer, sino adquirirlos en forma dinámica en ambos extremos, confiando en el uso de búferes en el emisor si hay que descartar segmentos ocasionalmente. Por otra parte, para la transferencia de archivos y demás tráfico de alto ancho de banda,

es mejor si el receptor dedica una ventana completa de búferes para permitir que los datos fluyan a la máxima velocidad. Ésta es la estrategia que utiliza TCP.

Todavía queda pendiente la cuestión de cómo organizar el grupo de búferes. Si la mayoría de los segmentos tienen el mismo tamaño aproximado, es natural organizar los búferes como un grupo de búferes de tamaño idéntico, con un segmento por búfer, como en la figura 6-15(a). Pero si hay una variación amplia en cuanto al tamaño de los segmentos, desde solicitudes cortas de páginas web hasta paquetes extensos en transferencias de archivos de igual a igual, un grupo de búferes de tamaño fijo presenta problemas. Si el tamaño de búfer se selecciona de manera que sea igual al segmento más grande, se desperdiciará espacio cada vez que llegue un segmento corto. Si el tamaño se selecciona de manera que sea menor al tamaño máximo de segmento, se requerirán varios búferes para los segmentos largos, con la complejidad inherente.

Otra forma de enfrentar el problema del tamaño de los búferes es usar búferes de tamaño variable, como en la figura 6-15(b). La ventaja aquí es un mejor uso de la memoria, al costo de una administración de búferes más complicada. Una tercera posibilidad es dedicar un solo búfer circular grande por conexión, como en la figura 6-15(c). Este sistema es simple y elegante, además de que no depende de los tamaños de los segmentos, pero hace buen uso de la memoria sólo cuando todas las conexiones están muy cargadas.

A medida que se abren y cierran conexiones y cambia el patrón del tráfico, el emisor y el receptor necesitan ajustar en forma dinámica sus asignaciones de búferes. En consecuencia, el protocolo de transporte debe permitir que un host emisor solicite espacio de búfer en el otro extremo. Se podrían asignar búferes por conexión o en forma colectiva, para todas las conexiones entre los dos hosts. De manera alternativa, al conocer su situación de uso de búferes (pero sin conocer el tráfico ofrecido), el receptor podría decir al emisor "Reservé *X* búferes para ti". Si el número de conexiones abiertas aumentara, tal vez sería necesario reducir una asignación, de modo que el protocolo debe tener en cuenta esta posibilidad.

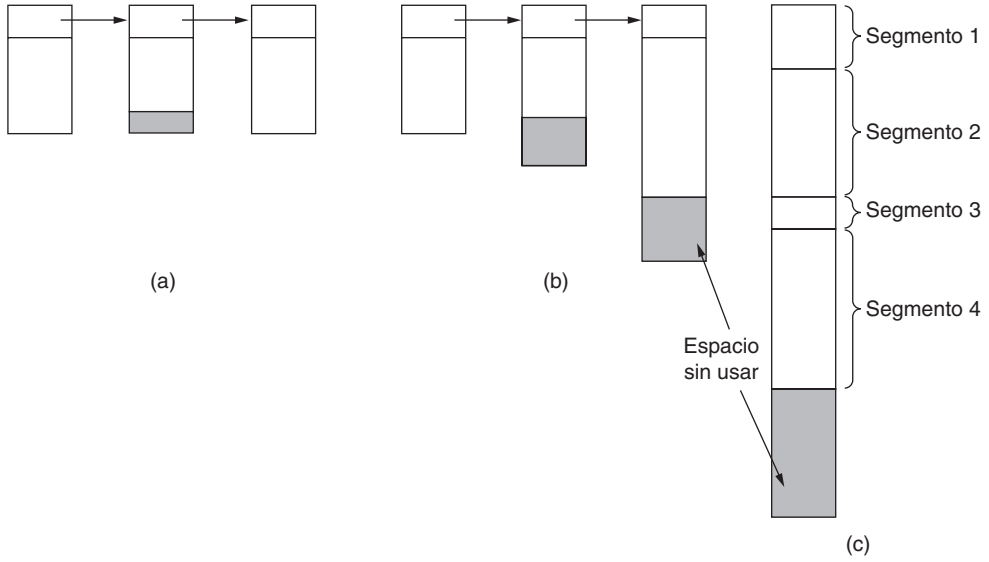


**Figura 6-15.** (a) Búferes encadenados de tamaño fijo. (b) Búferes encadenados de tamaño variable. (c) Un gran búfer circular por conexión.

Una manera razonable y general de administrar la asignación dinámica de búferes es desacoplar los búferes de las confirmaciones de recepción, en contraste a los protocolos de ventana deslizante del capítulo 3. En efecto, administración dinámica de búfer significa una ventana de tamaño variable. Al principio

el emisor solicita cierto número de búferes, con base en sus necesidades esperadas. Después el receptor otorga tantos de éstos como pueda. Cada vez que el emisor transmite un segmento debe disminuir su asignación, y cuando ésta llegue a cero debe detenerse por completo. El receptor superpone por separado las confirmaciones de recepción y las asignaciones de búfer en el tráfico inverso. TCP usa este esquema y transmite las asignaciones de búfer en un campo de encabezado llamado *Tamaño de ventana*.

La figura 6-16 muestra un ejemplo de la manera en que podría trabajar la administración dinámica de ventanas en una red de datagramas con números de secuencia de 4 bits. En este ejemplo, los datos fluyen en segmentos del host *A* al host *B*; las confirmaciones de recepción y las asignaciones de búferes fluyen en segmentos en dirección opuesta. En un principio *A* quiere ocho búferes, pero se le otorgan solamente cuatro. Después envía tres segmentos, de los cuales se pierde el tercero. El segmento 6 confirma la recepción de todos los segmentos hasta el número de secuencia 1, incluyéndolo, con lo cual permite que *A* libere esos búferes; además informa a *A* que tiene permiso de enviar tres segmentos más que empiecen después de 1 (es decir, los segmentos 2, 3 y 4). *A* sabe que ya ha enviado el número 2, por lo que piensa que debe enviar los segmentos 3 y 4, y procede a hacerlo. En este punto se bloquea y debe esperar una nueva asignación de búfer. Sin embargo, las retransmisiones inducidas por expiraciones del temporizador (línea 9) pueden ocurrir durante el bloqueo, pues usan búferes que ya se han asignado. En la línea 10, *B* confirma la recepción de todos los segmentos hasta el 4, inclusive, pero se niega a permitir que *A* continúe. Tal situación es imposible con los protocolos de ventana fija del capítulo 3. El siguiente segmento de *B* a *A* asigna otro búfer y permite a *A* continuar. Esto ocurrirá cuando *B* tenga espacio de búfer, quizá debido a que el usuario de transporte aceptó más segmentos de datos.

| | A | Mensaje | B | Comentarios |
|---|---|---|---|---|
| 1 | → | < solicita 8 búferes > | → | A quiere 8 búferes. |
| 2 | ← | <ack = 15, buf = 4> | ← | B sólo otorga los mensajes 0 a 3. |
| 3 | → | <seq = 0, data = m0> | → | A tiene 3 búferes libres ahora. |
| 4 | → | <seq = 1, data = m1> | → | A tiene 2 búferes libres ahora. |
| 5 | → | <seq = 2, data = m2> | • • • | Se perdió el mensaje, pero A piensa que le queda 1 libre. |
| 6 | ← | <ack = 1, buf = 3> | ← | B confirma la recepción de 0 y 1, permite 2-4. |
| 7 | → | <seq = 3, data = m3> | → | A tiene un búfer libre. |
| 8 | → | <seq = 4, data = m4> | → | A tiene 0 búferes libres y debe detenerse. |
| 9 | → | <seq = 2, data = m2> | → | El temporizador de A expira y retransmite. |
| 10 | ← | <ack = 4, buf = 0> | ← | Todo confirmado, pero A sigue bloqueado. |
| 11 | ← | <ack = 4, buf = 1> | ← | A puede enviar el 5 ahora. |
| 12 | ← | <ack = 4, buf = 2> | ← | B encontró un nuevo búfer en alguna parte. |
| 13 | → | <seq = 5, data = m5> | → | A tiene 1 búfer libre. |
| 14 | → | <seq = 6, data = m6> | → | A está bloqueado nuevamente. |
| 15 | ← | <ack = 6, buf = 0> | ← | A sigue bloqueado. |
| 16 | • • • | <ack = 6, buf = 4> | ← | Interbloqueo potencial. |

**Figura 6-16.** Asignación dinámica de búferes. Las flechas muestran la dirección de la transmisión. Los puntos suspensivos (…) indican un segmento perdido.

Pueden surgir problemas con los esquemas de asignación de búferes de este tipo en las redes de datagramas si hay posibilidad de perder segmentos de control (que casi siempre es así). Observe la línea 16. *B* ha asignado ahora más búferes a *A*, pero el segmento de asignación se perdió. Dado que los segmentos de control no están en secuencia ni ha expirado su temporizador, *A* se encuentra en interbloqueo. Para evitar esta situación, cada host debe enviar periódicamente segmentos de control con la confirmación de

recepción y el estado de búferes de cada conexión. De esta manera se puede interrumpir el interbloqueo, tarde o temprano.

Hasta ahora hemos supuesto tácitamente que el único límite impuesto sobre la tasa de datos del emisor es la cantidad de espacio de búfer disponible en el receptor. A menudo éste no es el caso. Hace algún tiempo la memoria era costosa, pero en la actualidad los precios han disminuido mucho. Los hosts pueden estar equipados con tanta memoria que la falta de búferes deja de ser un problema, incluso para conexiones de área amplia. Desde luego que esto depende de que el tamaño del búfer se establezca con la capacidad suficiente, lo cual no siempre ha sido así para TCP (Zhang y colaboradores, 2002).

Si el espacio de búfer ya no limita el flujo máximo, aparecerá otro cuello de botella: la capacidad de transporte de la red. Si enrutadores adyacentes pueden intercambiar cuando mucho $x$ paquetes/seg y hay $k$ trayectorias separadas entre un par de hosts, no hay manera de que esos hosts puedan intercambiar más de $kx$ segmentos/seg, sin importar la cantidad de espacio de búfer disponible en cada terminal. Si el emisor presiona demasiado (es decir, envía más de $kx$ segmentos/seg), la red se congestionará pues será incapaz de entregar los segmentos a la velocidad con que llegan.

Lo que se necesita es un mecanismo que limite las transmisiones del emisor con base en la capacidad de transporte de la red, en lugar de basarse en la capacidad de almacenamiento en búfer del receptor. Belsnes (1975) propuso el uso de un esquema de control de flujo de ventana deslizante, en el que el emisor ajusta en forma dinámica el tamaño de la ventana para igualarla a la capacidad de transporte de la red. Esto significa que una ventana deslizante dinámica puede implementar tanto el control de flujo como el de congestión. Si la red puede manejar $c$ segmentos/seg y el tiempo de ida y vuelta (incluyendo transmisión, propagación, encolamiento, procesamiento en el receptor y devolución de la confirmación de recepción) es de $r$, entonces la ventana del emisor debe ser $cr$. Con una ventana de este tamaño, el emisor normalmente opera con el canal a su máxima capacidad. Cualquier pequeña disminución en el desempeño de la red causará que se bloquee. Como la capacidad de red disponible para cualquier flujo dado varía con el tiempo, hay que ajustar el tamaño de ventana con frecuencia para rastrear los cambios en la capacidad de transporte. Como veremos más adelante, TCP usa un esquema similar.

### 6.2.5 Multiplexión

La multiplexión o la compartición de varias conversaciones a través de conexiones, circuitos virtuales y enlaces físicos, desempeña un papel importante en varias capas de la arquitectura de red. En la capa de transporte puede surgir la necesidad de usar la multiplexión de varias formas. Por ejemplo, si sólo hay una dirección de red disponible en un host, todas las conexiones de transporte de esa máquina tendrán que utilizarla. Cuando llega un segmento, se necesita algún mecanismo para saber a cuál proceso asignarlo. Esta situación, conocida como **multiplexión**, se muestra en la figura 6-17(a). En esta figura, cuatro conexiones de transporte distintas utilizan la misma conexión de red (por ejemplo, dirección IP) al host remoto.

La multiplexión también puede ser útil en la capa de transporte por otra razón. Por ejemplo, supongamos que un host cuenta con varias trayectorias de red que puede utilizar. Si un usuario necesita más ancho de banda o una mayor confiabilidad de la que le puede proporcionar una de las trayectorias de red, una solución es tener una conexión que distribuya el tráfico entre varias trayectorias de red por turno rotatorio (*round-robin*), como se indica en la figura 6-17(b). Este *modus operandi* se denomina **multiplexión inversa**. Con $k$ conexiones de red abiertas, el ancho de banda efectivo se podría incrementar por un factor de $k$. El **SCTP** (**Protocolo de Control de Transmisión de Flujo**, del inglés *Stream Control Transmission Protocol*) es un ejemplo de multiplexión inversa. Este protocolo puede operar una conexión mediante el uso de múltiples interfaces de red. En contraste, TCP utiliza un solo punto terminal de red. La multiplexión inversa también se encuentra en la capa de enlace, cuando se usan varios enlaces de baja velocidad en paralelo como un solo enlace de alta velocidad.

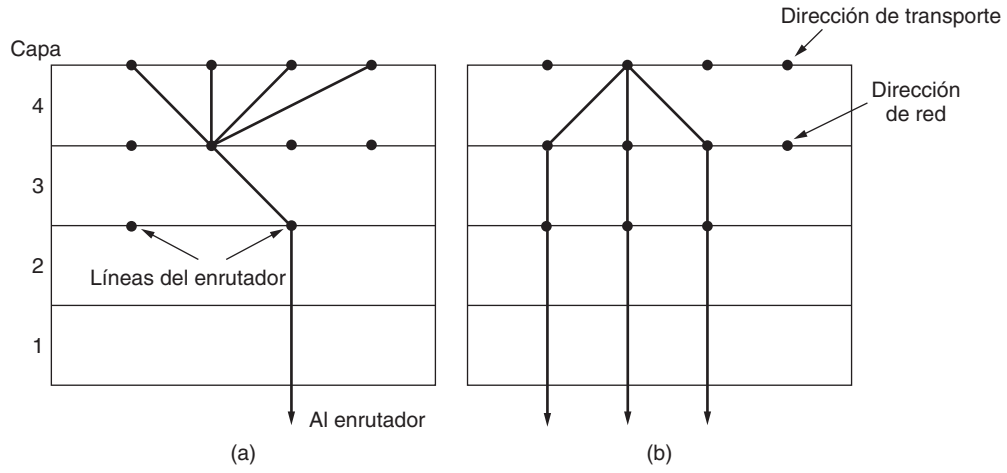


**Figura 6-17.** (a) Multiplexión. (b) Multiplexión inversa.

### 6.2.6 Recuperación de fallas

Si los hosts y los enrutadores están sujetos a fallas o las conexiones son de larga duración (por ejemplo, descargas extensas de software o medios), la recuperación de estas fallas se vuelve un tema importante. Si la entidad de transporte está totalmente dentro de los hosts, la recuperación de fallas de la red y de los enrutadores es sencilla. Las entidades de transporte esperan la pérdida de segmentos todo el tiempo y saben cómo lidiar con esto mediante el uso de retransmisiones.

Un problema más complicado es cómo recuperarse de las fallas del host. En particular, tal vez sea conveniente que los clientes sean capaces de continuar trabajando cuando los servidores fallan y se reinician muy rápido. Para ilustrar la dificultad, supongamos que un host, el cliente, envía un archivo grande a otro host, el servidor de archivos, mediante el uso de un protocolo simple de parada y espera. La capa de transporte en el servidor simplemente pasa los segmentos entrantes al usuario de transporte, uno por uno. De pronto, a mitad de la transmisión falla el servidor. Al reactivarse, sus tablas se reinicializan, por lo que ya no sabe precisamente en dónde se encontraba.

En un intento por recuperar su estado previo, el servidor podría enviar un segmento de difusión a todos los demás hosts, para anunciar que acaba de fallar y solicitar a sus clientes que le informen sobre el estado de todas las conexiones abiertas. Cada cliente puede estar en uno de dos estados: un segmento pendiente (*S1*) o ningún segmento pendiente (*S0*). Con base en esta información de estado, el cliente debe decidir si retransmitirá o no el segmento más reciente.

A primera vista parecería obvio: el cliente debe retransmitir sólo si tiene un segmento pendiente sin confirmación de recepción (es decir, que se encuentre en el estado *S1*) al momento de enterarse de la falla. Sin embargo, una inspección más cercana revela dificultades con esta metodología ingenua. Considere, por ejemplo, la situación en la que la entidad de transporte del servidor envía primero una confirmación de recepción y luego, una vez que se envía esta confirmación de recepción, escribe en el proceso de aplicación. Escribir un segmento en el flujo de salida y enviar una confirmación de recepción son dos eventos diferentes que no se pueden hacer al mismo tiempo. Si ocurre una falla después de enviar la confirmación de recepción pero antes de que la escritura se complete, el cliente recibirá la confirmación de recepción y estará por ende en el estado *S0* cuando llegue el anuncio de recuperación de la falla. Entonces el cliente no retransmitirá, pues pensará (en forma errónea) que llegó el segmento. Esta decisión del cliente provoca que falte un segmento.

En este punto el lector podría pensar: "Ese problema se resuelve fácil. Todo lo que hay que hacer es reprogramar la entidad de transporte para que primero haga la escritura y luego envíe la confirmación de recepción". Intente de nuevo. Imagine que se ha hecho la escritura pero que la falla ocurre antes de enviar la confirmación de recepción. El cliente se encontrará en el estado *S1*, y por tanto retransmitirá, lo que provocará un segmento duplicado sin detectar en el flujo de salida que va al proceso de aplicación del servidor.

Sin importar cómo se programen el emisor y el receptor, siempre hay situaciones en las que el protocolo no se puede recuperar de manera apropiada. Podemos programar el servidor en una de dos formas: enviar confirmación de recepción primero o escribir primero. El cliente se puede programar en una de cuatro formas: siempre retransmitir el último segmento, nunca retransmitir el último segmento, retransmitir sólo en el estado *S0* o retransmitir sólo en el estado *S1*. Esto nos da ocho combinaciones pero, como veremos, para cada combinación existe cierto conjunto de eventos que hacen fallar al protocolo.

Son posibles tres eventos en el servidor: enviar una confirmación de recepción (*A*), escribir al proceso de salida (*W*) y fallar (C). Los tres eventos pueden ocurrir en seis órdenes diferentes: *AC*(*W*), *AWC*, *C*(*AW*), *C*(*WA*), *WAC* y *WC*(*A*), donde los paréntesis se usan para indicar que ni *A* ni *W* pueden ir después de *C* (es decir, una vez que el servidor falla, así se queda). En la figura 6-18 se muestran las ocho combinaciones de las estrategias de cliente y servidor, junto con las secuencias de eventos válidas para cada una. Observe que para cada estrategia hay alguna secuencia de eventos que provoca que el protocolo falle. Por ejemplo, si el cliente siempre retransmite, el evento *AWC* generará un duplicado no detectado, incluso aunque los otros dos eventos funcionen de manera apropiada.

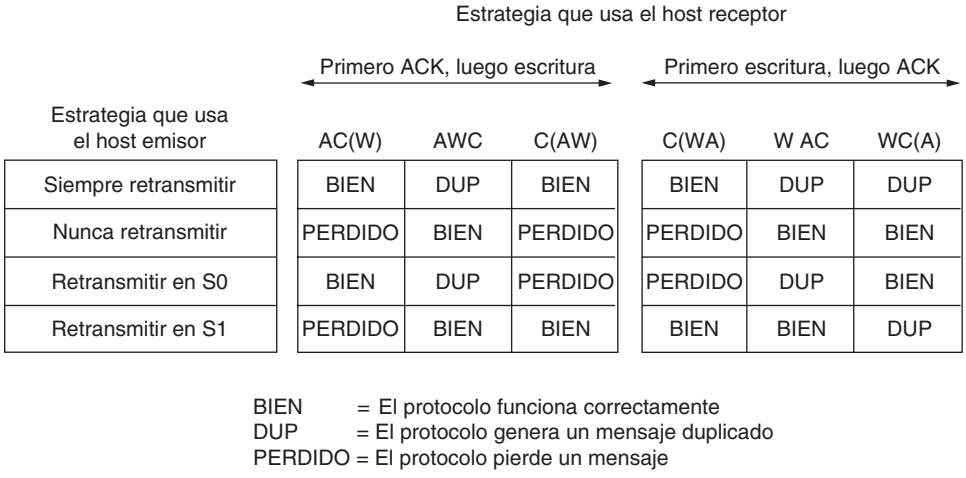


**Figura 6-18.** Distintas combinaciones de estrategias de cliente y servidor.

Hacer más elaborado el protocolo no sirve de nada. Aunque el cliente y el servidor intercambien varios segmentos antes de que el servidor intente escribir, para que el cliente sepa con exactitud lo que está a punto de ocurrir, el cliente no tiene manera de saber si ha ocurrido una falla justo antes o justo después de la escritura. La conclusión es inevitable: según nuestra regla básica de que no debe haber eventos simultáneos (es decir, que eventos separados ocurren uno después de otro y no al mismo tiempo), la falla de un host y su recuperación no pueden hacerse transparentes a las capas superiores.

Dicho en términos más generales, podemos replantear este resultado como "la recuperación de una falla en la capa *N* sólo se puede llevar a cabo en la capa *N* + 1", y esto es sólo si la capa superior retiene suficiente información del estado como para reconstruir la condición en la que se encontraba antes de que

ocurriera el problema. Esto es consistente con el caso que mencionamos antes, en el que la capa de transporte puede recuperarse de fallas en la capa de red, siempre y cuando cada extremo de una conexión lleve el registro del estado en el que se encuentra.

Este problema nos lleva a la cuestión de averiguar lo que en realidad significa una confirmación de recepción de extremo a extremo. En principio, el protocolo de transporte es de extremo a extremo y no está encadenado como en las capas inferiores. Ahora considere el caso de un usuario que introduce solicitudes de transacciones para una base de datos remota. Suponga que la entidad de transporte remota está programada para pasar primero los segmentos a la siguiente capa superior y luego emitir las confirmaciones de recepción. Incluso en este caso, el hecho de que la máquina de un usuario reciba una confirmación de recepción no significa necesariamente que el host remoto se quedó encendido el tiempo suficiente como para actualizar la base de datos. Quizá es imposible lograr una verdadera confirmación de recepción de extremo a extremo, en donde si se recibe significa que en realidad se hizo el trabajo y, si no se recibe, significa que no se realizó el trabajo. Este punto lo analizan con mayor detalle Saltzer y colaboradores (1984).

## 6.3 CONTROL DE CONGESTIÓN

Si las entidades de transporte en muchas máquinas envían demasiados paquetes a la red con exagerada rapidez, ésta se congestionará y se degradará el desempeño a medida que se retrasen y pierdan paquetes. El proceso de controlar la congestión para evitar este problema es la responsabilidad combinada de las capas de red y de transporte. La congestión ocurre en los enrutadores, por lo que se detecta en la capa de red. Sin embargo, se produce en última instancia debido al tráfico que la capa de transporte envía a la red. La única manera efectiva de controlar la congestión es que los protocolos de transporte envíen paquetes a la red con más lentitud.

En el capítulo 5 estudiamos los mecanismos de control de congestión en la capa de red. En esta sección estudiaremos la otra mitad del problema: los mecanismos de control de congestión en la capa de transporte. Después de describir los objetivos del control de congestión, describiremos la forma en que los hosts pueden regular la velocidad a la que envían los paquetes a la red. Internet depende mucho de la capa de transporte para el control de la congestión, por lo cual se han construido algoritmos específicos en TCP y otros protocolos.

### 6.3.1 Asignación de ancho de banda deseable

Antes de describir cómo regular el tráfico, debemos comprender lo que estamos tratando de lograr, al ejecutar un algoritmo de control de congestión. Esto es, debemos especificar el estado en el que un buen algoritmo de control de congestión operará la red. El objetivo es algo más que simplemente evitar la congestión. Se trata también de encontrar una buena asignación de ancho de banda a las entidades de transporte que utilizan la red. Una buena asignación producirá un buen desempeño, ya que utiliza todo el ancho de banda disponible pero evita la congestión, tratará con igualdad a las entidades de transporte que compitan entre sí, y rastreará con rapidez los cambios en las demandas de tráfico. A continuación explicaremos con más detalle cada uno de estos criterios por separado.

#### Eficiencia y potencia

Una asignación eficiente del ancho de banda entre las entidades de transporte utilizará toda la capacidad disponible de la red. Sin embargo, no es correcto pensar que si hay un enlace de 100 Mbps, cinco entidades

de transporte deben recibir 20 Mbps cada una. Por lo general deben recibir menos de 20 Mbps para un buen desempeño. La razón es que, con frecuencia, el tráfico es en ráfagas. Recuerde que en la sección 5.3 describimos el **caudal útil** (o tasa de paquetes útiles que llegan al receptor) como función de la carga ofrecida. En la figura 6-19 se muestran esta curva y otra similar para el retardo en función de la carga ofrecida.

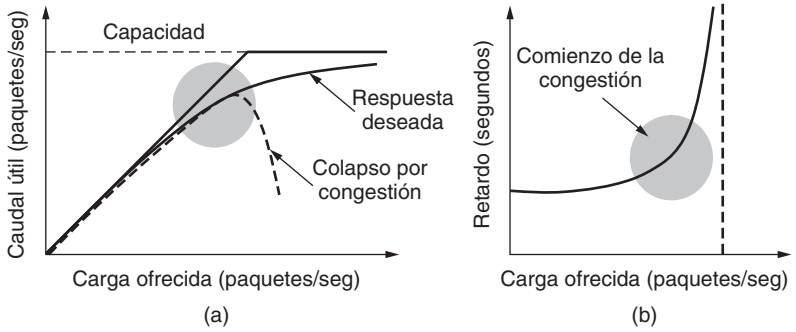


**Figura 6-19.** (a) Caudal útil y (b) retardo en función de la carga ofrecida.

A medida que aumenta la carga en la figura 6-19(a), el caudal útil aumenta en un principio con la misma proporción, pero a medida que la carga se acerca a la capacidad, el caudal útil aumenta en forma más gradual. Esta reducción se debe a que en ocasiones se pueden acumular las ráfagas de tráfico y provocar algunas pérdidas en los búferes dentro de la red. Si el protocolo de transporte está mal diseñado y retransmite paquetes que se hayan retardado pero que no se hayan perdido, la red puede entrar en un colapso por congestión. En este estado los emisores envían demasiados paquetes ya gran velocidad pero cada vez se realiza menos trabajo útil.

En la figura 6-19(b) se muestra el retardo correspondiente. En un principio, este retardo es fijo y representa el retardo de propagación a través de la red. A medida que la carga se acerca a la capacidad el retardo aumenta, lentamente al principio y después con mucha mayor rapidez. De nuevo, esto se debe a las ráfagas de tráfico que tienden a acumularse cuando la carga es alta. En realidad el retardo no puede llegar hasta infinito, excepto en un modelo en el que los enrutadores tengan búferes infinitos. En cambio, se perderán paquetes después de experimentar el retardo máximo en los búferes.

Tanto para el caudal útil como para el retardo, el desempeño se empieza a degradar cuando comienza la congestión. Por instinto, obtendremos el mejor desempeño de la red si asignamos ancho de banda hasta el punto en que el retardo empieza a aumentar con rapidez. Este punto está por debajo de la capacidad. Para identificarlo, Kleinrock (1979) propuso la métrica de **potencia**, en donde

$$Potencia = \frac{Carga}{Retardo}$$

En un principio la potencia aumentará con la carga ofrecida, mientras el retardo permanezca en un valor pequeño y aproximadamente constante, pero llegará a un máximo y caerá a medida que el retardo aumente con rapidez. La carga con la potencia más alta representa una carga eficiente para que la entidad de transporte la coloque en la red.

### Equidad máxima-mínima

En la discusión anterior no hablamos sobre cómo dividir el ancho de banda entre distintos emisores de transporte. Esto suena como una pregunta simple de responder (dar a todos los emisores una misma fracción del ancho de banda), pero implica varias consideraciones.

Tal vez la primera consideración sea preguntar qué tiene que ver este problema con el control de la congestión. Después de todo, si la red proporciona a un emisor cierta cantidad de ancho de banda para que la utilice, el emisor debe usar sólo esa cantidad de ancho de banda. Sin embargo, es común el caso en que las redes no tienen una reservación estricta de ancho de banda para cada flujo o conexión. Tal vez sí la tengan para algunos flujos en los que se soporta la calidad del servicio, pero muchas conexiones buscarán usar el ancho de banda que esté disponible o la red las agrupará bajo una asignación común. Por ejemplo, los servicios diferenciados de la IETF separan el tráfico en dos clases y las conexiones compiten por el ancho de banda dentro de cada clase. A menudo, los enrutadores IP tienen conexiones que compiten por el mismo ancho de banda. En esta situación, el mecanismo de control de congestión es quien asigna el ancho de banda a las conexiones que compiten entre sí.

Una segunda consideración es lo que significa una porción equitativa para los flujos en una red. Es lo bastante simple si $N$ flujos usan un solo enlace, en cuyo caso todos pueden tener $1/N$ parte del ancho de banda (aunque la eficiencia dictará que deben usar un poco menos, si el tráfico es en ráfagas). Pero ¿qué ocurre si los flujos tienen distintas trayectorias de red que se traslapan? Por ejemplo, un flujo puede atravesar tres enlaces y los otros flujos pueden atravesar un enlace. El flujo de tres enlaces consume más recursos de red. Puede ser más equitativo en cierto sentido para darle menos ancho de banda que a los flujos de un enlace. Sin duda debe ser posible soportar más flujos de un enlace al reducir el ancho de banda de tres enlaces. Este punto demuestra una tensión inherente entre la equidad y la eficiencia.

No obstante, adoptaremos una noción de equidad que no depende de la longitud de la trayectoria de red. Incluso con este simple modelo, es un poco complicado otorgar a las conexiones una fracción equitativa del ancho de banda, ya que las distintas conexiones tomarán distintas trayectorias a través de la red, y estas trayectorias tendrán distintas capacidades unas de otras. En este caso, es posible que un flujo sufra un congestionamiento en un enlace descendente y tome una porción más pequeña de un enlace ascendente que los otros flujos; reducir el ancho de banda de los otros flujos les permite disminuir su velocidad, pero no ayuda en nada al flujo con el congestionamiento.

La forma de equidad que se desea con frecuencia para el uso de red es la **equidad máxima-mínima**. Una asignación tiene equidad máxima-mínima si el ancho de banda que se otorga a un flujo no se puede incrementar sin tener que disminuir el ancho de banda otorgado a otro flujo con una asignación que no sea mayor. Esto es, si se incrementa el ancho de banda de un flujo, la situación sólo empeorará para los flujos con menos recursos.

Ahora veamos un ejemplo. En la figura 6-20 se muestra la asignación de equidad máxima-mínima para una red con cuatro flujos: *A*, *B*, *C* y *D*. Cada uno de los enlaces entre enrutadores tiene la misma capacidad, que se considera 1 unidad, aunque en el caso general los enlaces tendrán distintas capacidades. Tres flujos compiten por el enlace inferior izquierdo entre los enrutadores *R4* y *R5*. Por lo tanto, cada uno de estos flujos recibe 1/3 del enlace. El flujo restante *A* compite con *B* en el enlace de *R2* a *R3*. Como *B* tiene una asignación de 1/3, *A* recibe los 2/3 restantes del enlace. Observe que los demás enlaces tienen capacidad de sobra. Sin embargo, esta capacidad no se puede dar a ninguno de los flujos sin reducir la capacidad de otro flujo inferior. Por ejemplo, si se otorga más del ancho de banda en el enlace entre *R2* y *R3* al flujo *B*, habrá menos ancho de banda para el flujo *A*. Esto es razonable, puesto que el flujo *A* ya tiene de antemano más ancho de banda. Sin embargo, hay que reducir la capacidad del flujo *C* o *D* (o de ambos) para otorgar más ancho de banda a *B*, y estos flujos tendrán menor ancho de banda que *B*. Por ende, la asignación tiene equidad máxima-mínima.

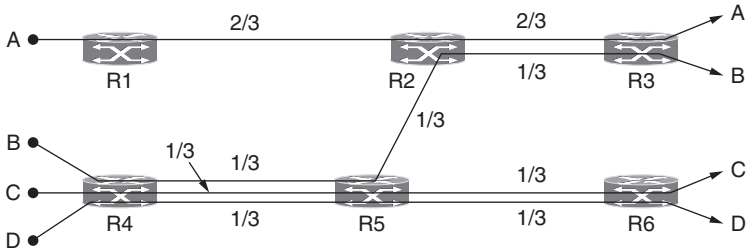


**Figura 6-20.** Asignación de ancho de banda con equidad máxima-mínima para cuatro flujos.

Las asignaciones máxima-mínima se pueden calcular con base en un conocimiento global de la red. Una forma intuitiva de pensar sobre ellas es imaginar que la tasa para todos los flujos empieza en cero y se incrementa lentamente. Cuando la tasa llega a un cuello de botella para cualquier flujo, entonces ese flujo deja de incrementarse. Los demás flujos seguirán incrementándose y compartirán la capacidad disponible por igual, hasta que también lleguen a sus respectivos cuellos de botella.

Una tercera consideración es el nivel sobre el cuál una red se puede considerar equitativa: al nivel de las conexiones, de las conexiones entre un par de hosts o de todas las conexiones por host. Ya examinamos esta cuestión cuando vimos el WFQ (Encolamiento justo ponderado) en la sección 5.4 y concluimos que cada una de estas definiciones tiene sus problemas. Por ejemplo, definir la equidad por host significa que un servidor ocupado se comportará igual que un teléfono móvil, mientras que al definir la equidad por conexión se anima a los hosts a que abran más conexiones. Dado que no hay una respuesta clara, lo más común es considerar la equidad por conexión, pero por lo general no es imprescindible una equidad precisa. Es más importante en la práctica que ninguna conexión se quede sin ancho de banda, a que todas las conexiones reciban la misma cantidad exacta de ancho de banda. De hecho, con TCP es posible abrir varias conexiones y competir por el ancho de banda en forma más agresiva. Esta táctica la utilizan aplicaciones que requieren grandes cantidades de ancho de banda, como BitTorrent para compartir archivos de igual a igual.

### Convergencia

Un último criterio es que el algoritmo de control de congestión debe converger rápidamente hacia una asignación equitativa y eficiente del ancho de banda. El análisis anterior del punto de operación deseable supone un entorno de red estático. Sin embargo, en una red siempre entran y salen conexiones, por lo que el ancho de banda necesario para una conexión dada también variará con el tiempo; por ejemplo, cuando un usuario navega por páginas web y en algunas ocasiones descarga videos extensos.

Debido a la variación en la demanda, el punto de operación ideal para la red varía con el tiempo. Un buen algoritmo de control de congestión debe converger con rapidez hacia el punto de operación ideal; y debe rastrear ese punto a medida que cambia con el tiempo. Si la convergencia es muy lenta, el algoritmo nunca estará cerca del punto de operación cambiante. Si el algoritmo no es estable, puede fracasar al tratar de converger hacia el punto correcto en algunos casos, o incluso puede oscilar alrededor del punto correcto.

En la figura 6-21 se muestra un ejemplo de una asignación de ancho de banda que cambia con el tiempo y converge con rapidez. En un principio, el flujo 1 tiene todo el ancho de banda. Un segundo después, el flujo 2 empieza y también necesita ancho de banda. La asignación cambia rápidamente para otorgar a cada uno de estos flujos la mitad del ancho de banda. A los 4 segundos aparece un tercer flujo. Sin embargo, este flujo utiliza sólo el 20% del ancho de banda, lo cual es menos que su fracción equitativa (que es un tercio).

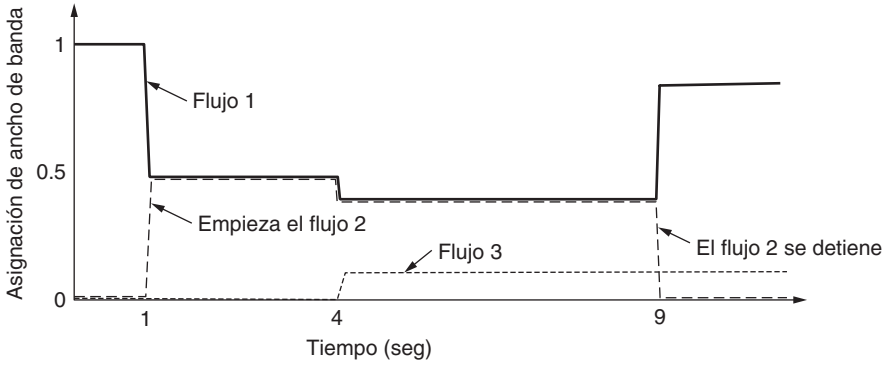


**Figura 6-21.** Ejemplo de una asignación de ancho de banda que cambia con el tiempo.

Los flujos 1 y 2 se ajustan rápidamente y dividen el ancho de banda disponible, para que cada uno tenga 40% del ancho de banda. A los 9 segundos termina el segundo flujo y el tercero permanece sin cambios. El primer flujo captura rápidamente 80% del ancho de banda. En todo momento, el ancho de banda total asignado es de aproximadamente 100%, de tal forma que la red se utilice por completo y los flujos que compiten obtengan un trato equitativo (pero no tienen que usar más ancho de banda del que necesitan).

### 6.3.2 Regulación de la tasa de envío

Ahora es tiempo para el platillo principal. ¿Cómo regulamos las tasas de envío para obtener una asignación de ancho de banda deseable? Podemos limitar la tasa de envío mediante dos factores. El primero es el control de flujo, en el caso en que haya un uso insuficiente de búfer en el receptor. El segundo es la congestión, en el caso en que haya una capacidad insuficiente en la red. En la figura 6-22 podemos ver este problema ilustrado mediante hidráulica. En la figura 6-22(a) vemos un tubo grueso que conduce a un receptor de baja capacidad. Ésta es una situación limitada por control de flujo. Mientras que el emisor no envíe más agua de la que pueda contener la cubeta, no se perderá agua. En la figura 6-22(b), el factor limitante no es la capacidad de la cubeta, sino la capacidad de transporte interno de la red. Si entra demasiada agua con mucha rapidez, se regresará y una parte se perderá (en este caso, por desbordamiento del embudo).

Estos casos pueden parecer similares para el emisor, puesto que al transmitir demasiado rápido se pierden paquetes. Sin embargo, tienen distintas causas y requieren diferentes soluciones. Ya hemos hablado sobre una solución de control de flujo con una ventana de tamaño variable. Ahora consideraremos una solución de control de congestión. Como puede ocurrir cualquiera de estos problemas, en general el protocolo de transporte tendrá que llevar a cabo ambas soluciones y reducir la velocidad si ocurre cualquiera de los dos problemas.

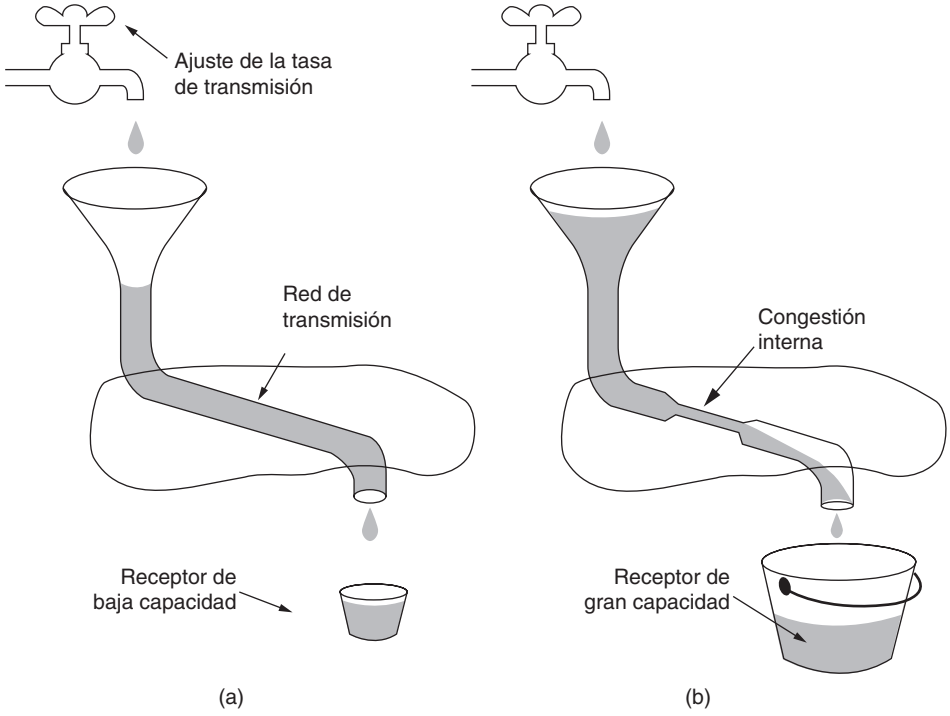


**Figura 6-22.** (a) Una red veloz que alimenta a un receptor de baja capacidad. (b) Una red lenta que alimenta a un receptor de alta capacidad.

La forma en que un protocolo de transporte debe regular la tasa de envío depende de la forma de retroalimentación que devuelva la red. Las distintas capas de la red pueden devolver distintos tipos de retroalimentación, la cual puede ser explícita o implícita, y también precisa o imprecisa.

Un ejemplo de un diseño explícito y preciso es cuando los enrutadores indican a las fuentes la velocidad a la que pueden enviar. Los diseños en la literatura, como XCP (Protocolo Explícito de Congestión, del inglés *eXplicit Congestion Protocol*), operan de esta forma (Katabi y colaboradores, 2002). Un diseño explícito e impreciso es el uso de ECN (Notificación Explícita de Congestión, del inglés *Explicit Congestion Notification*) con TCP. En este diseño, los enrutadores establecen bits en paquetes que experimentan congestión para advertir a los emisores que reduzcan la velocidad, pero no les dicen cuánto deben reducirla.

En otros diseños no hay una señal explícita. FAST TCP mide el retardo de ida y vuelta, y usa esa métrica como señal para evitar la congestión (Wei y colaboradores, 2006). Por último, en la forma de control de congestión más prevalente en Internet en la actualidad, TCP con enrutadores *drop-tail* (descartar final) o RED, se deduce la pérdida de paquetes y se utiliza para indicar que esa red se ha congestionado. Existen muchas variantes de esta forma de TCP, incluyendo CUBIC TCP, que se utiliza en Linux (Ha y colaboradores, 2008). También son posibles las combinaciones. Por ejemplo, Windows incluye el diseño Compound TCP, que utiliza la pérdida de paquetes y el retardo como señales de retroalimentación (Tan y colaboradores, 2006). En la figura 6-23 se resumen estos diseños.

Si se proporciona una señal explícita y precisa, la entidad de transporte puede usarla para ajustar su tasa con el nuevo punto de operación. Por ejemplo, si XCP indica a los emisores la tasa que deben usar, éstos simplemente usarán esa tasa. Sin embargo, en los otros casos se requieren algunas conjeturas. A falta de una señal de congestión, los emisores deben reducir sus tasas. Cuando se proporcione una señal de congestión, los emisores deben reducir sus tasas. La forma en que se deben aumentar o reducir las tasas se proporciona mediante una **ley de control**. Estas leyes tienen un efecto importante en el desempeño.

| Protocolo | Señal | ¿Explícito? | ¿Preciso? |
|---|---|---|---|
| XCP | Tasa a usar. | Sí | Sí |
| TCP con ECN | Advertencia de congestión. | Sí | No |
| FAST TCP | Retardo de extremo a extremo. | No | Sí |
| Compound TCP | Pérdida de paquete y retardo extremo a extremo. | No | Sí |
| CUBIC TCP | Pérdida de paquetes. | No | No |
| TCP | Pérdida de paquetes. | No | No |

**Figura 6-23.** Señales de algunos protocolos de control de congestión.

Chiu y Jain (1989) estudiaron el caso de la retroalimentación por congestión binaria y concluyeron que **AIMD** (**Incremento Aditivo/Decremento Multiplicativo**, del inglés *Additive Increase Multiplicative Decrease*) es la ley de control apropiada para llegar a un punto de operación eficiente y equitativo.

Para exponer este caso, construyeron un argumento gráfico para el caso simple de dos conexiones que compiten por el ancho de banda de un solo enlace. El gráfico de la figura 6-24 muestra el ancho de banda asignado al usuario 1 en el eje x y al usuario 2 en el eje y. Cuando la asignación es equitativa, ambos usuarios reciben la misma cantidad de ancho de banda. Esto se muestra mediante la línea de equidad punteada. Cuando las asignaciones suman un 100% de la capacidad del enlace, la asignación es eficiente. Esto se muestra mediante la línea de eficiencia punteada. La red proporciona una señal de congestión a ambos usuarios cuando la suma de sus asignaciones cruza esta línea. La intersección de esas líneas es el punto de operación deseado, cuando ambos usuarios tienen el mismo ancho de banda y se utiliza todo el ancho de banda de la red.

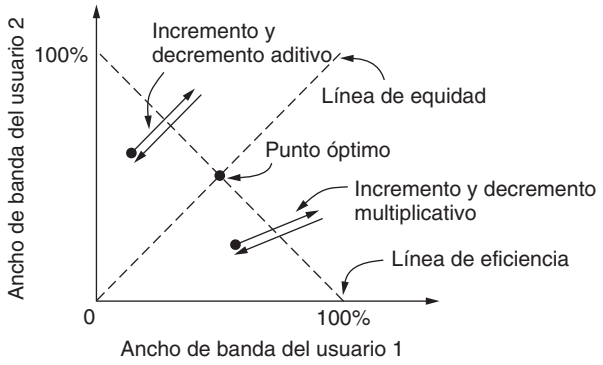


**Figura 6-24.** Ajustes de ancho de banda aditivo y multiplicativo.

Considere lo que ocurre desde alguna asignación inicial si tanto el usuario 1 como el 2 incrementan en forma aditiva su respectivo ancho de banda a través del tiempo. Por ejemplo, cada uno de los usuarios puede incrementar su tasa de envío por 1 Mbps cada segundo. En un momento dado, el punto de operación cruza la línea de eficiencia y ambos usuarios reciben de la red una señal de congestión. En esta etapa deben reducir sus asignaciones. Sin embargo, un decremento aditivo simplemente provocaría que oscilaran a lo largo de una línea aditiva. Esta situación se muestra en la figura 6-24. El comportamiento mantendrá el punto de operación cerca de lo eficiente, pero no necesariamente será equitativo.

De manera similar, considere el caso en que ambos usuarios incrementan en forma multiplicativa su ancho de banda a través del tiempo, hasta que reciben una señal de congestión. Por ejemplo, los usuarios pueden incrementar su tasa de envío en 10% cada segundo. Si después decrementan en forma multiplicativa sus tasas de envío, el punto de operación de los usuarios simplemente oscilará a lo largo de una línea multiplicativa. Este comportamiento también se muestra en la figura 6-24. La línea multiplicativa tiene una pendiente diferente a la línea aditiva (apunta hacia el origen, mientras que la línea aditiva tiene un ángulo de 45 grados). Pero de ninguna otra forma es mejor. En ningún caso los usuarios convergerán hacia las tasas de envío óptimas que sean tanto equitativas como eficientes.

Ahora considere el caso en que los usuarios incrementan en forma aditiva sus asignaciones de ancho de banda y después las decrementan en forma multiplicativa cuando se indica una congestión. Este comportamiento es la ley de control AIMD y se muestra en la figura 6-25. Podemos ver que la trayectoria trazada por este comportamiento converge hacia el punto óptimo, que es tanto equitativo como eficiente. Esta convergencia ocurre sin importar cuál sea el punto inicial, por lo cual el AIMD se considera en extremo útil. Con base en el mismo argumento, la única combinación restante, incremento multiplicativo y decremento aditivo, divergirá del punto óptimo.

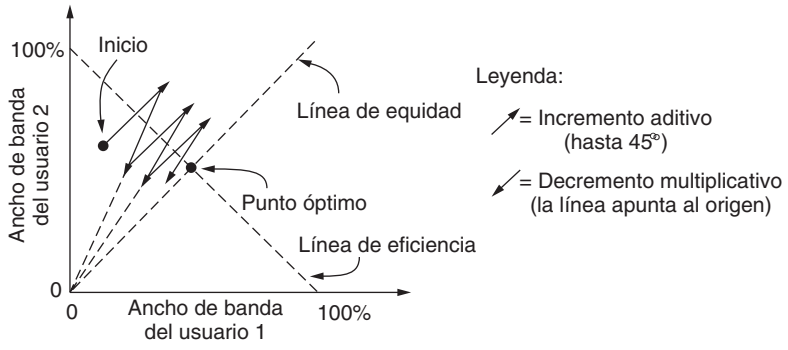


**Figura 6-25.** La ley de control de Incremento Aditivo/Decremento Multiplicativo (AIMD).

AIMD es la ley de control que utiliza TCP, con base en este argumento y en otro razonamiento de estabilidad (que es fácil llevar a la red a la congestión y difícil recuperarse, por lo que la política de incremento debe ser gentil y la política de decremento agresiva). No es muy equitativa, ya que las conexiones TCP ajustan el tamaño de su ventana por cierta cantidad en cada tiempo de ida y vuelta. Las distintas conexiones tendrán distintos tiempos de ida y vuelta. Esto conduce a una predisposición en donde las conexiones a los hosts cercanos reciben más ancho de banda que las conexiones a los hosts distantes, siendo todo lo demás igual.

En la sección 6.5 describiremos con detalle la forma en que TCP implementa una ley de control AIMD para ajustar la tasa de envío y proveer control de congestión. Esta tarea es más difícil de lo que parece, ya que las tasas se miden a través de cierto intervalo y el tráfico es en ráfagas. En vez de ajustar la tasa directamente, una estrategia de uso común en la práctica es ajustar el tamaño de una ventana deslizante. TCP usa esta estrategia. Si el tamaño de la ventana es $W$ y el tiempo de ida y vuelta es $RTT$, la tasa equivalente es $W/RTT$. Esta estrategia es fácil de combinar con el control de flujo, que de antemano usa una ventana, además de que tiene la ventaja de que el emisor controla los paquetes usando confirmaciones de recepción y, por ende, reduce la velocidad en un $RTT$ si deja de recibir los informes de que los paquetes están saliendo de la red.

Como cuestión final, puede haber muchos protocolos de transporte distintos que envíen tráfico a la red. ¿Qué ocurrirá si los distintos protocolos compiten con diferentes leyes de control para evitar la congestión? Se producirán asignaciones de ancho de banda desiguales. Como TCP es la forma dominante de control de congestión en Internet, hay una presión considerable de la comunidad en cuanto a diseñar nuevos protocolos de transporte para que compitan de manera equitativa con TCP. Los primeros protocolos de medios de flujo continuo provocaron problemas al reducir de manera excesiva la tasa de transmisión de TCP debido a que no competían en forma equitativa. Esto condujo a la noción del control de congestión **TCP-friendly** (amigable para TCP) , en donde se pueden mezclar los protocolos de transporte que sean o no de TCP sin efectos dañinos (Floyd y colaboradores, 2000).

### 6.3.3 Cuestiones inalámbricas

Los protocolos de transporte como TCP que implementan control de congestión deben ser independientes de la red subyacente y de las tecnologías de capa de enlace. Es una buena teoría, pero en la práctica hay problemas con las redes inalámbricas. La cuestión principal es que la pérdida de paquetes se usa con frecuencia como señal de congestión, incluyendo a TCP como vimos antes. Las redes inalámbricas pierden paquetes todo el tiempo debido a errores de transmisión.

Con la ley de control AIMD, una tasa de transmisión real alta requiere niveles muy pequeños de pérdida de paquetes. Los análisis realizados por Padhye y colaboradores (1998) mostraron que la tasa

de transmisión real aumenta con base en la raíz cuadrada inversa de la tasa de pérdida de paquetes. Lo que esto significa en la práctica es que la tasa de pérdidas para las conexiones TCP rápidas es muy pequeña: 1% es una tasa de pérdidas moderada, y para cuando la tasa de pérdidas llega a 10%, la conexión ha dejado de trabajar por completo. Sin embargo, para las redes inalámbricas como las LAN 802.11, son comunes las tasas de pérdidas de tramas de por lo menos un 10%. Esta diferencia significa que, sin medidas de protección, los esquemas de control de congestión que utilizan la pérdida de paquetes como señal regularán de manera innecesaria las conexiones que pasen a través de enlaces inalámbricos para generar tasas muy bajas.

Para funcionar bien, las únicas pérdidas de paquetes que debe observar el algoritmo de control de congestión son las pérdidas debido a un ancho de banda insuficiente, no las pérdidas debido a errores de transmisión. Una solución a este problema es enmascarar las pérdidas inalámbricas mediante el uso de retransmisiones a través del enlace inalámbrico. Por ejemplo, 802.11 usa un protocolo de parada y espera para entregar cada trama, y reintenta las transmisiones varias veces si es necesario antes de reportar la pérdida de un paquete a la capa superior. En el caso normal, cada paquete se entrega a pesar de los errores de transmisión transitorios que no son visibles para las capas superiores.

La figura 6-26 muestra una trayectoria con un enlace cableado y uno inalámbrico, para el que se utiliza la estrategia de enmascaramiento. Hay que tener en cuenta dos aspectos. En primer lugar, el emisor no necesariamente sabe que la trayectoria incluye un enlace inalámbrico, ya que todo lo que ve es el enlace cableado al que está conectado. Las trayectorias de Internet son heterogéneas, por lo que no hay un método general para que el emisor sepa qué tipo de enlaces conforman la trayectoria. Esto complica el problema de control de congestión, ya que no hay una manera fácil de usar un protocolo para enlaces inalámbricos y otro para enlaces cableados.

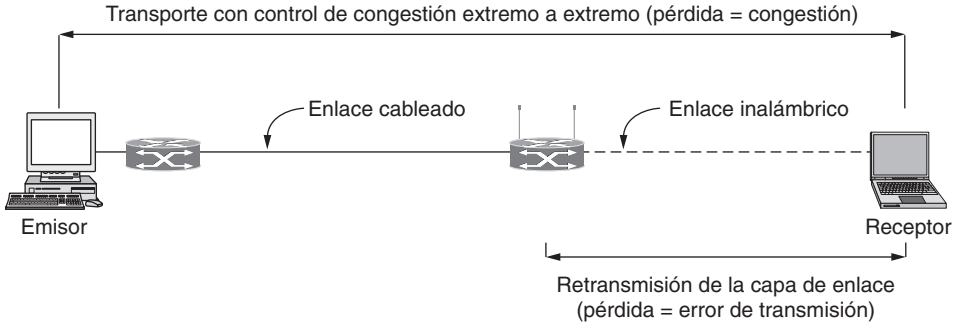


**Figura 6-26.** Control de congestión a través de una trayectoria con un enlace inalámbrico.

El segundo aspecto es un acertijo. La figura muestra dos mecanismos controlados por la pérdida: las retransmisiones de tramas de la capa de enlace y el control de congestión de la capa de transporte. El acertijo es: cómo pueden coexistir estos dos mecanismos sin confundirse. Después de todo, una pérdida sólo debe provocar que un mecanismo actúe, ya que puede ser un error de transmisión o una señal de congestión. No puede ser ambos. Si los dos mecanismos actúan (al retransmitir la trama y reducir la tasa de envío), entonces regresamos al problema original de los transportes que operan con demasiada lentitud a través de los enlaces inalámbricos. Considere este acertijo por un momento y vea si puede resolverlo.

La solución es que los dos mecanismos actúan en distintas escalas de tiempo. Las retransmisiones de la capa de enlace ocurren en el orden de microsegundos a milisegundos para los enlaces inalámbricos como 802.11. Los temporizadores de pérdidas en los protocolos de transporte se activan en el orden de milisegundos a segundos. La diferencia es de tres órdenes de magnitud. Esto permite a los enlaces ina-

lámbricos detectar las pérdidas de tramas y retransmitirlas para reparar los errores de transmisión mucho antes de que la entidad de transporte deduzca la pérdida de paquetes.

La estrategia de enmascaramiento es suficiente para permitir que la mayoría de los protocolos de transporte operen bien a través de la mayoría de los enlaces inalámbricos. Sin embargo, no siempre es una solución adecuada. Algunos enlaces inalámbricos tienen tiempos de ida y vuelta largos, como los satélites. Para estos enlaces se deben usar otras técnicas para enmascarar la pérdida, como FEC (Corrección de Errores hacia Adelante), o el protocolo de transporte debe usar una señal que no sea de pérdida para el control de congestión.

Un segundo aspecto relacionado con el control de congestión a través de enlaces inalámbricos es la capacidad variable. Esto es, la capacidad de un enlace inalámbrico cambia con el tiempo, algunas veces en forma brusca, a medida que los nodos se desplazan y la relación señal —ruido varía con las condiciones cambiantes del canal. Esto es distinto a los enlaces cableados, cuya capacidad es fija. El protocolo de transporte se debe adaptar a la capacidad cambiante de los enlaces inalámbricos; de lo contrario se congestionará la red o no se podrá usar la capacidad disponible.

Una posible solución a este problema es simplemente no preocuparse por ello. Esta estrategia es viable, puesto que los algoritmos de control de congestión ya deben manejar el caso en que nuevos usuarios entren a la red, o que los usuarios existentes cambien sus tasas de envío. Aun cuando la capacidad de los enlaces cableados es fija, el comportamiento cambiante de otros usuarios se presenta a sí mismo como una variabilidad en el ancho de banda que está disponible para un usuario dado. Por ende, es posible usar TCP a través de una ruta con un enlace inalámbrico 802.11 y obtener un desempeño razonable.

Sin embargo, cuando hay mucha variabilidad inalámbrica, los protocolos de transporte diseñados para los enlaces cableados pueden tener problemas para mantenerse a la par y provocar un desempeño deficiente. La solución en este caso es un protocolo de transporte que esté diseñado para enlaces inalámbricos. Una configuración muy desafiante es una red de malla inalámbrica en la que se deben atravesar varios enlaces inalámbricos, es necesario cambiar rutas debido a la movilidad y muchas pérdidas. La investigación en esta área es continua. Consulte a Li y colaboradores (2009) para obtener un ejemplo de diseño de un protocolo de transporte inalámbrico.

## 6.4 LOS PROTOCOLOS DE TRANSPORTE DE INTERNET: UDP

Internet tiene dos protocolos principales en la capa de transporte, uno sin conexión y otro orientado a conexión. Los protocolos se complementan entre sí. El protocolo sin conexión es UDP. Prácticamente no hace nada más que enviar paquetes entre aplicaciones, y deja que las aplicaciones construyan sus propios protocolos en la parte superior según sea necesario. El protocolo orientado a conexión es TCP. Hace casi todo. Realiza las conexiones y agrega confiabilidad mediante las retransmisiones, junto con el control de flujo y el control de congestión, todo en beneficio de las aplicaciones que lo utilizan.

En las siguientes secciones estudiaremos los protocolos UDP y TCP. Empezaremos con UDP porque es el más simple. También veremos dos aplicaciones de UDP. Como éste es un protocolo de capa de transporte que por lo general se ejecuta en el sistema operativo y los protocolos que utilizan UDP por lo general se ejecutan en el espacio de usuario, estos usos se podrían considerar como aplicaciones. Sin embargo, las técnicas que utilizan son convenientes para muchas aplicaciones y se considera que pertenecen a un servicio de transporte, por lo que las veremos aquí.

### 6.4.1 Introducción a UDP

La suite de protocolos de Internet soporta un protocolo de transporte sin conexión, conocido como **UDP** (**Protocolo de Datagrama de Usuario**, del inglés *Use Datagram Protocol*). UDP proporciona una forma

para que las aplicaciones envíen datagramas IP encapsulados sin tener que establecer una conexión. El protocolo UDP se describe en el RFC 768.

UDP transmite **segmentos** que consisten en un encabezado de 8 bytes seguido de la carga útil. En la figura 6-27 se muestra ese encabezado. Los dos **puertos** sirven para identificar los puntos terminales dentro de las máquinas de origen y destino. Cuando llega un paquete UDP, su carga útil se entrega al proceso que está conectado al puerto de destino. Este enlace ocurre cuando se utiliza la primitiva BIND o algo similar, como vimos en la figura 6-6 para TCP (el proceso de enlace es el mismo para UDP). Piense en los puertos como apartados postales que las aplicaciones pueden rentar para recibir paquetes. Diremos más sobre ellas cuando describamos a TCP, que también usa puertos. De hecho, el valor principal de contar con UDP en lugar de simplemente utilizar IP puro es la adición de los puertos de origen y destino. Sin los campos de puerto, la capa de transporte no sabría qué hacer con cada paquete entrante. Con ellos, entrega el segmento incrustado a la aplicación correcta.

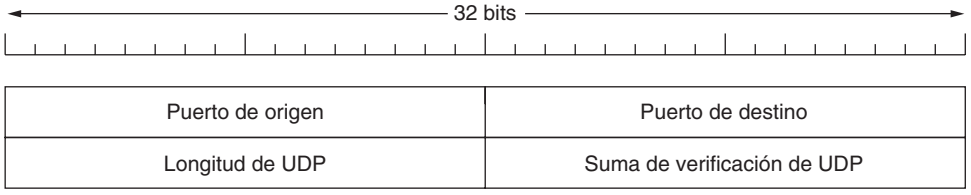


**Figura 6-27.** El encabezado UDP.

El puerto de origen se necesita principalmente cuando hay que enviar una respuesta al origen. Al copiar el campo *Puerto de origen* del segmento entrante en el campo *Puerto de destino* del segmento que sale, el proceso que envía la respuesta puede especificar cuál proceso de la máquina emisora va a recibirlo.

El campo *Longitud UDP* incluye el encabezado de 8 bytes y los datos. La longitud mínima es de 8 bytes, para cubrir el encabezado. La longitud máxima es de 65 515 bytes, lo cual es menor al número que cabe en 16 bits debido al límite de tamaño en los paquetes IP.

También se proporciona un campo *Suma de verificación* opcional para una confiabilidad adicional. Se realiza una suma de verificación para el encabezado, los datos y un pseudoencabezado IP conceptual. Al hacer este cálculo, el campo *Suma de verificación* se establece en cero y el campo de datos se rellena con un byte cero adicional si su longitud es un número impar. El algoritmo de suma de verificación consiste simplemente en sumar todas las palabras de 16 bits en complemento a uno y sacar el complemento a uno de la suma. Como consecuencia, cuando el receptor realiza el cálculo en todo el segmento (incluyendo el campo *Suma de verificación*), el resultado debe ser 0. Si no se calcula la suma de verificación, se almacena como 0, ya que por una feliz coincidencia de la aritmética de complemento a uno, un 0 calculado se almacena como 1. Sin embargo, no tiene sentido desactivarlo a menos que no importe la calidad de los datos (por ejemplo, en la voz digitalizada).

En la figura 6-28 se muestra el pseudoencabezado para el caso de IPv4. Éste contiene las direcciones IPv4 de 32 bits de las máquinas de origen y de destino, el número de protocolo para UDP (17) y la cuenta de bytes para el segmento UDP (incluyendo el encabezado). Es distinto pero análogo a IPv6. Es útil incluir el pseudoencabezado en el cálculo de la suma de verificación de UDP para detectar paquetes mal entregados, pero al incluirlo también se viola la jerarquía de protocolos debido a que las direcciones IP en él pertenecen a la capa IP, no a la capa UDP. TCP usa el mismo pseudoencabezado para su suma de verificación.

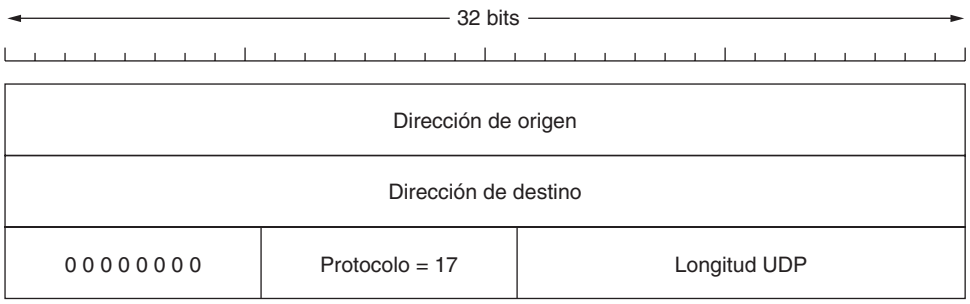


**Figura 6-28.** El pseudoencabezado IPv4 que se incluye en la suma de verificación de UDP.

Quizá valga la pena mencionar de manera explícita algunas de las cosas que UDP *no* realiza. No realiza control de flujo, control de congestión o retransmisión cuando se recibe un segmento erróneo. Todo lo anterior le corresponde a los procesos de usuario. Lo que sí realiza es proporcionar una interfaz para el protocolo IP con la característica agregada de demultiplexar varios procesos mediante el uso de los puertos y la detección de errores extremo a extremo opcional. Esto es todo lo que hace.

Para aplicaciones que necesitan tener un control preciso sobre el flujo de paquetes, control de errores o temporización, UDP es justo lo que se necesita. Un área en la que UDP es especialmente útil es en las situaciones cliente-servidor. Con frecuencia, el cliente envía una solicitud corta al servidor y espera una respuesta corta. Si se pierde la solicitud o la respuesta, el cliente simplemente puede esperar a que expire su temporizador e intentar de nuevo. El código no sólo es simple, sino que se requieren menos mensajes (uno en cada dirección) en comparación con un protocolo que requiere una configuración inicial, como TCP.

Una aplicación que utiliza de esta manera a UDP es DNS (el Sistema de Nombres de Dominio), el cual analizaremos en el capítulo 7. En resumen, un programa que necesita buscar la dirección IP de algún host, por ejemplo, www.cs.berkeley.edu, puede enviar al servidor DNS un paquete UDP que contenga el nombre de dicho host. El servidor responde con un paquete UDP que contiene la dirección IP del host. No se necesita configuración por adelantado ni tampoco una liberación posterior. Sólo dos mensajes que viajan a través de la red.

### 6.4.2 Llamada a procedimiento remoto

En cierto sentido, enviar un mensaje a un host remoto y obtener una respuesta es muy parecido a realizar la llamada a una función en un lenguaje de programación. En ambos casos, usted inicia con uno o más parámetros y obtiene un resultado. Esta observación ha llevado a la gente a tratar de que las interacciones de solicitud-respuesta en las redes se asignen en forma de llamadas a procedimientos. Dicho arreglo hace que las aplicaciones de red sean mucho más fáciles de programar y de manejar. Por ejemplo, imagine un procedimiento llamado *obtener_direccion_IP (nombre_de_host)* que envía un paquete UDP a un servidor DNS y espera una respuesta, y en caso de que no llegue ninguna con la suficiente rapidez, expira su temporizador y lo intenta de nuevo. De esta forma, todos los detalles de la conectividad pueden ocultarse al programador.

El trabajo clave en esta área fue realizado por Birrell y Nelson (1984). En sí, lo que ellos sugirieron fue permitir que los programas invocaran procedimientos localizados en hosts remotos. Cuando un proceso en la máquina 1 llama a otro procedimiento en la máquina 2, el proceso invocador en 1 se suspende y la ejecución del procedimiento invocado se lleva a cabo en la máquina 2. Se puede transportar información del proceso invocador al proceso invocado en los parámetros, y se puede regresar información en el

resultado del procedimiento. El paso de mensajes es transparente para el programador de la aplicación. Esta técnica se conoce como **RPC** (**Llamada a Procedimiento Remoto**, del inglés *Remote Procedure Call*) y se ha vuelto la base de muchas aplicaciones de red. Por tradición, el procedimiento invocador se conoce como cliente y el proceso invocado como servidor, por lo que aquí también utilizaremos esos nombres.

El objetivo de RPC es hacer que una llamada a procedimiento remoto sea lo más parecida posible a una local. En la forma más simple, para llamar a un procedimiento remoto, el programa cliente se debe enlazar con un pequeño procedimiento de biblioteca, llamado **stub del cliente**, que representa al procedimiento servidor en el espacio de direcciones del cliente. Asimismo, el servidor se enlaza con una llamada a procedimiento denominada **stub del servidor**. Estos procedimientos ocultan el hecho de que la llamada a procedimiento del cliente al servidor no es local.

En la figura 6-29 se muestran los pasos reales para realizar una RPC. El paso 1 consiste en que el cliente llame al stub del cliente. Ésta es una llamada a procedimiento local, y los parámetros se meten en la pila de la forma tradicional. El paso 2 consiste en que el stub del cliente empaque los parámetros en un mensaje y realice una llamada de sistema para enviar el mensaje. Al proceso de empacar los parámetros se le conoce como **marshaling (empaquetar)**. El paso 3 consiste en que el sistema operativo envíe el mensaje de la máquina cliente a la máquina servidor. El paso 4 consiste en que el sistema operativo pase el paquete entrante al stub del servidor. Por último, el paso 5 consiste en que el stub del servidor llame al procedimiento servidor con los parámetros sin empacar (*unmarshaling*). La respuesta sigue la misma ruta en la dirección opuesta.

El elemento clave a observar aquí es que el procedimiento cliente, escrito por el usuario, simplemente realiza una llamada a procedimiento normal (es decir, local) al stub del cliente, que tiene el mismo nombre que el procedimiento servidor. Puesto que el procedimiento cliente y el stub del cliente están en el mismo espacio de direcciones, los parámetros se pasan de la forma usual. De manera similar, el procedimiento servidor es llamado por un procedimiento en su espacio de direcciones con los parámetros esperados. Para el procedimiento servidor, nada es inusual. De esta forma, en lugar de que la E/S se realice en sockets, la comunicación de red se realiza mediante la simulación de una llamada a procedimiento normal.

A pesar de la elegancia conceptual de RPC, hay algunas desventajas ocultas. La más grande es el uso de parámetros de apuntador. Por lo general, no hay problema al pasar un apuntador a un procedimiento. El procedimiento invocado puede utilizar el apuntador de la misma manera que el invocador, porque ambos procedimientos se encuentran en el mismo espacio de direcciones virtual. Con RPC, el paso de apuntadores es imposible porque el cliente y el servidor están en diferentes espacios de direcciones.

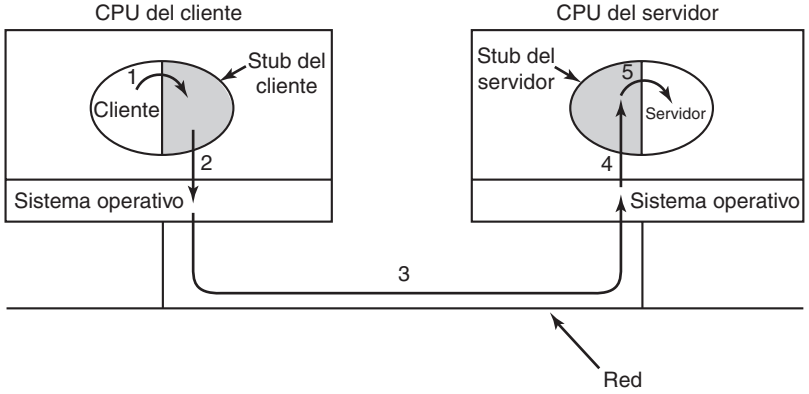


**Figura 6-29.** Pasos para realizar la llamada a un procedimiento remoto. Los stubs están sombreados.

En algunos casos, se pueden utilizar trucos para pasar apuntadores. Suponga que el primer parámetro es un apuntador a un entero, *k*. El stub del cliente puede empacar a *k* y enviarlo a lo largo del servidor. A continuación, el stub del servidor crea un apuntador a *k* y lo pasa al procedimiento servidor, justo como lo espera. Cuando el procedimiento servidor regresa el control al stub del servidor, este último envía a *k* de regreso al cliente, en donde el nuevo *k* se copia sobre el anterior, por si el servidor lo cambió. En efecto, la secuencia estándar de llamada por referencia se ha reemplazado por una llamada mediante copiar-restaurar. Por desgracia, este truco no siempre funciona; por ejemplo, si el apuntador apunta hacia un grafo u otra estructura de datos compleja. Por esta razón, se deben colocar algunas restricciones sobre los parámetros para los procedimientos que se llaman de manera remota, como veremos más adelante.

Un segundo problema es que en los lenguajes de tipos flexibles, como C, es perfectamente legal escribir un procedimiento que calcule el producto interno de dos vectores (arreglos), sin especificar la longitud de cada uno. Estos vectores se pueden terminar mediante un valor especial conocido sólo por los procedimientos invocador e invocado. Bajo estas circunstancias, es en esencia imposible que el stub del cliente empaquete los parámetros debido a que no tiene forma de determinar su longitud.

Un tercer problema es que no siempre es posible deducir los tipos de los parámetros, ni siquiera mediante una especificación formal o del mismo código. Un ejemplo de esto es el procedimiento *printf*, el cual puede tener cualquier número de parámetros (por lo menos uno), y éstos pueden ser una mezcla arbitraria de tipos enteros, cortos, largos, caracteres, cadenas, así como de números de punto flotante de varias longitudes, entre otros. Tratar de llamar a *printf* como un procedimiento remoto sería prácticamente imposible debido a que C es demasiado permisivo. Sin embargo, una regla que especifique el uso de RPC, siempre y cuando no se utilice C (o C++) para programar, tal vez no sea muy popular con muchos programadores.

Un cuarto problema se relaciona con el uso de las variables globales. Por lo general, los procedimientos invocador e invocado se pueden comunicar mediante el uso de variables globales, además de comunicarse a través de los parámetros. Pero si el procedimiento invocado se mueve a una máquina remota, el código fallará porque las variables globales ya no estarán compartidas.

Estos problemas no quieren decir que RPC no tenga mucho futuro. De hecho, se utiliza ampliamente, pero se necesitan algunas restricciones para hacerlo funcionar bien en la práctica.

En términos de protocolos de capa de transporte, UDP es una buena base sobre la cual se puede implementar la técnica RPC. Se pueden enviar tanto solicitudes como respuestas en un solo paquete UDP en el caso más simple; además la operación puede ser rápida. Sin embargo, una implementación debe incluir también otras herramientas. Como se puede llegar a perder la solicitud o la respuesta, el cliente debe mantener un temporizador para retransmitir la solicitud. Debemos tener en cuenta que una respuesta sirve como una confirmación de recepción explícita de una solicitud, por lo que no es necesario confirmar la recepción de la solicitud por separado. Algunas veces los parámetros o resultados pueden ser más grandes que el tamaño de paquete UDP máximo, en cuyo caso se requiere de algún protocolo para entregar mensajes grandes. Si existe la probabilidad de que varias solicitudes y respuestas se traslapen (como en el caso de la programación concurrente), se necesita un identificador para relacionar la solicitud con la respuesta.

Una preocupación de mayor nivel es que la operación tal vez no sea idempotente (es decir, que se pueda repetir en forma segura). El caso simple es el de las operaciones idempotentes como las solicitudes y respuestas de DNS. El cliente puede retransmitir en forma segura estas solicitudes una y otra vez, en caso de que no lleguen respuestas. No importa si el servidor nunca recibió la solicitud o si se perdió la respuesta. Cuando finalmente llegue la respuesta, será la misma (suponiendo que la base de datos de DNS no se actualice en el proceso). Sin embargo, no todas las operaciones son idempotentes porque algunas tienen importantes efectos colaterales; por ejemplo, el incremento a un contador. El uso de RPC para estas operaciones requiere de una semántica más sólida, de modo que cuando el programador llame a un procedimiento, éste no se ejecute varias veces. En este caso tal vez sea necesario establecer una conexión TCP y enviar la solicitud a través de ella, en vez de usar UDP.

### 6.4.3 Protocolos de transporte en tiempo real

El RPC cliente-servidor es un área en la que UDP se utiliza mucho. Otra área es la de las aplicaciones multimedia en tiempo real. En particular, conforme la radio en Internet, la telefonía en Internet, la música bajo demanda, las videoconferencias, el video bajo demanda y otras aplicaciones multimedia se volvían más comunes, las personas descubrieron que cada una de esas aplicaciones estaba reinventando más o menos el mismo protocolo de transporte de tiempo-real. Cada vez era más claro que tener un protocolo genérico de transporte en tiempo real para múltiples aplicaciones sería una excelente idea.

A raíz de esto fue que nació el **RTP** (**Protocolo de Transporte en Tiempo Real**, del inglés *Real-time Transport Protocol*). Se describe en el RFC 3550 y ahora se utiliza ampliamente para aplicaciones multimedia. A continuación describiremos dos aspectos del transporte en tiempo real. El primero es el protocolo RTP para transportar datos de audio y video en paquetes. El segundo es el procesamiento que se lleva a cabo, en su mayor parte en el receptor, para reproducir el audio y video en el momento correcto. Estas funciones se ajustan a la pila de protocolos según se muestra en la figura 6-30.

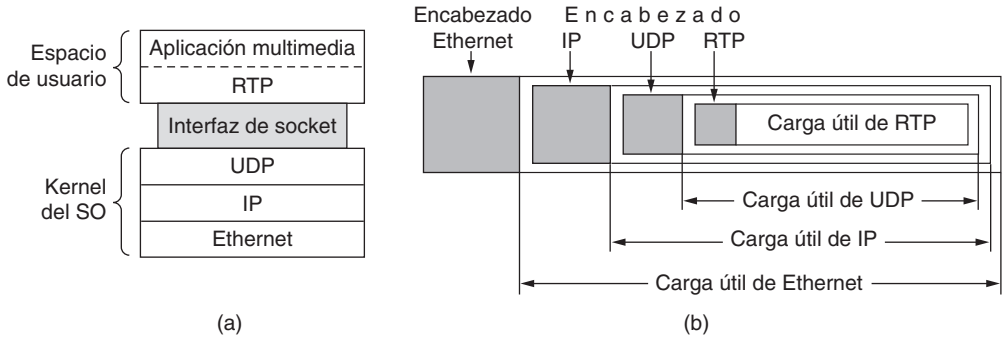


**Figura 6-30.** (a) La posición de RTP en la pila de protocolos. (b) Anidamiento de paquetes.

Por lo general, RTP se ejecuta en espacio de usuario sobre UDP (en el sistema operativo). Opera como se muestra a continuación. La aplicación multimedia consiste en múltiples flujos de audio, video, texto y quizás otros flujos. Éstos se colocan en la biblioteca RTP, la cual está en el espacio de usuario junto con la aplicación. Esta biblioteca multiplexa los flujos y los codifica en paquetes RTP, que después coloca en un socket. En el extremo del socket correspondiente al sistema operativo, se generan paquetes UDP para envolver los paquetes RTP y se entregan al IP para que los transmita a través de un enlace tal como Ethernet. En el receptor se lleva a cabo el proceso inverso. En un momento dado la aplicación multimedia recibirá datos multimedia de la biblioteca RTP. Es responsable de reproducir los medios. En la figura 6-30(a) se muestra la pila de protocolos para esta situación. En la figura 6-30(b) se muestra el anidamiento de paquetes.

Como consecuencia de este diseño, es un poco difícil saber en cuál capa está RTP. Debido a que se ejecuta en el espacio de usuario y está enlazado al programa de aplicación, sin duda luce como un protocolo de aplicación. Por otro lado, es un protocolo genérico, independiente de la aplicación que simplemente proporciona herramientas de transporte, por lo que se parece también a un protocolo de transporte. Tal vez la mejor descripción sea que es un protocolo de transporte que sólo está implementado en la capa de aplicación, razón por la cual lo estamos analizando en este capítulo.

### RTP: el Protocolo de Transporte en Tiempo Real

La función básica de RTP es multiplexar varios flujos de datos de tiempo real en un solo flujo de paquetes UDP. El flujo UDP se puede enviar a un solo destino (unidifusión) o a múltiples destinos (multidifusión). Debido a que RTP sólo utiliza UDP normal, los enrutadores no dan a sus paquetes un trato especial, a menos que se habiliten algunas características de calidad de servicio IP normales. En particular no hay garantías especiales acerca de la entrega, así que los paquetes se pueden perder, retrasar, corromper, etcétera.

El formato de RTP contiene varias características para ayudar a que los receptores trabajen con información multimedia. A cada paquete enviado en un flujo RTP se le da un número más grande que a su predecesor. Esta numeración permite al destino determinar si falta algún paquete en cuyo caso, la mejor acción a realizar queda a criterio de la aplicación. Tal vez esta acción sea omitir una trama de video si los paquetes transportan datos de video, o aproximar el valor faltante mediante la interpolación en caso de que los paquetes transporten datos de audio. La retransmisión no es una opción práctica debido a la probabilidad de que el paquete retransmitido llegue muy tarde como para ser útil. Como consecuencia, RTP no tiene confirmaciones de recepción ni ningún mecanismo para solicitar retransmisiones.

Cada carga útil de RTP podría contener múltiples muestras; éstas se pueden codificar de la forma en la que la aplicación desee. Para permitir la interconectividad, RTP define varios perfiles (por ejemplo, un solo flujo de audio), y para cada perfil se pueden permitir múltiples formatos de codificación. Por ejemplo, un solo flujo de audio se puede codificar como muestras de PCM de 8 bits a 8 kHz mediante codificación delta, codificación predictiva, codificación GSM, MP3, etc. RTP proporciona un campo de encabezado en el que el origen puede especificar la codificación, pero no se involucra de ninguna otra manera en la forma en que se realiza la codificación.

Las estampas de tiempo (*timestamping*) son otra herramienta que muchas de las aplicaciones en tiempo real necesitan. La idea aquí es permitir que la fuente asocie una estampa de tiempo con la primera muestra de cada paquete. Las estampas de tiempo son relativas al inicio del flujo, por lo que sólo son importantes las diferencias entre dichas estampas. Los valores absolutos no tienen significado. Como veremos en breve, este mecanismo permite que el destino haga un uso muy moderado del almacenamiento en búfer y reproduzca cada muestra el número exacto de milisegundos después del inicio del flujo, sin importar cuándo llegó el paquete que contiene la muestra.

La estampa de tiempo no sólo reduce los efectos de la variación en el retardo de la red, sino que también permite que múltiples flujos estén sincronizados entre sí. Por ejemplo, un programa de televisión digital podría tener un flujo de video y dos flujos de audio. Los dos flujos de audio podrían ser para difusiones en estéreo o para manejar películas con la banda sonora del idioma original y con una "traducción" al idioma local, para darle una opción al espectador. Cada flujo proviene de un dispositivo físico diferente, pero si tienen estampas de tiempo de un solo contador, se pueden reproducir de manera síncrona, incluso si los flujos se transmiten o reciben de manera irregular.

En la figura 6-31 se ilustra el encabezado RTP, que consiste de tres palabras de 32 bits y potencialmente de algunas extensiones. La primera palabra contiene el campo *Versión*, que actualmente es la 2. Esperemos que esta versión esté muy cerca de la última debido a que sólo quedó pendiente un punto del código (aunque se puede definir como 3 para indicar que la versión real estaba en una palabra de extensión).

El bit *P* indica que el paquete se ha rellenado para formar un múltiplo de 4 bytes. El último byte de relleno indica cuántos bytes se agregaron. El bit *X* indica que hay un encabezado de extensión. El formato y el significado de este encabezado no se definen. Lo único que se define es que la primera palabra de la extensión proporciona la longitud. Ésta es una puerta de escape para cualquier requerimiento imprevisto.

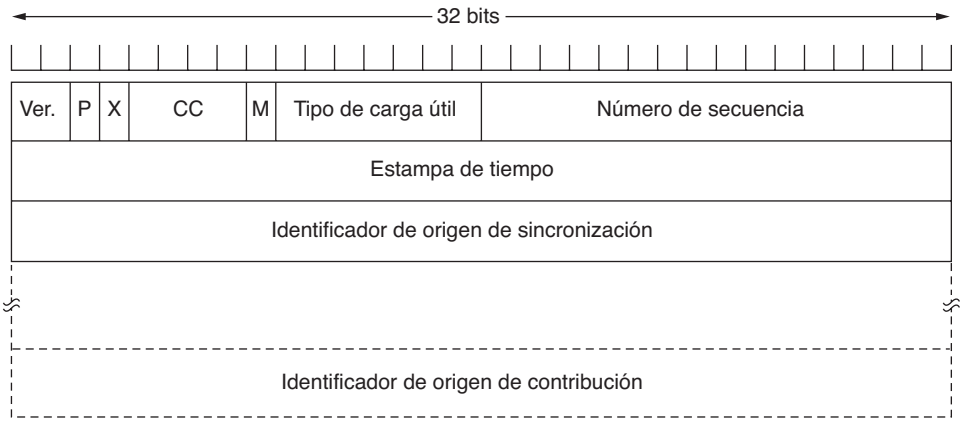


**Figura 6-31.** El encabezado RTP.

El campo *CC* indica cuántas fuentes de contribución están presentes, de 0 a 15 (vea abajo). El bit *M* es un bit marcador específico de la aplicación. Puede utilizarse para marcar el inicio de una trama de video, el inicio de una palabra en un canal de audio o algo más que la aplicación entienda. El campo *Tipo de carga útil* indica cuál algoritmo de codificación se utilizó (por ejemplo, audio de 8 bits sin compresión, MP3, etc.). Puesto que cada paquete lleva este campo, la codificación puede cambiar durante la transmisión. El *Número de secuencia* es simplemente un contador que se incrementa con cada paquete RTP enviado. Se utiliza para detectar paquetes perdidos.

La fuente del flujo produce la *Estampa de tiempo* para indicar cuándo se creó la primera muestra en el paquete. Este valor puede ayudar a reducir la variabilidad de la sincronización conocida como **variación del retardo** (***jitter***) en el receptor, al desacoplar la reproducción del tiempo de llegada del paquete. El *Identificador de origen de sincronización* indica a cuál flujo pertenece el paquete. Es el método utilizado para multiplexar y desmultiplexar varios flujos de datos en un solo flujo de paquetes UDP. Por último, los *Identificadores de origen de contribución*, en caso de que haya, se utilizan cuando hay mezcladoras en el estudio. En ese caso, la mezcladora es el origen de la sincronización y los flujos que se mezclan se listan aquí.

### RTCP: el Protocolo de Control de Transporte en Tiempo Real

RTP tiene un hermano pequeño llamado **RTCP** (**Protocolo de Control de Transporte en Tiempo Real**, del inglés *Real Transport Control Protocol*), el cual se define junto con RTP en el RFC 3550 y se encarga de la retroalimentación, la sincronización y la interfaz de usuario, pero no transporta muestras de medios.

La primera función se puede utilizar para proporcionar a las fuentes retroalimentación sobre el retardo, variación en el retardo o jitter, ancho de banda, congestión y otras propiedades de red. El proceso de codificación puede utilizar esta información para incrementar la tasa de datos (y para proporcionar mejor calidad) cuando la red está funcionando bien y para disminuir la tasa de datos cuando hay problemas en la red. Al proveer una retroalimentación continua, los algoritmos de codificación se pueden adaptar de manera continua para proporcionar la mejor calidad posible bajo las circunstancias actuales. Por ejemplo, si el ancho de banda aumenta o disminuye durante la transmisión, la codificación puede cambiar de MP3 a PCM de 8 bits o a codificación delta, según se requiera. El campo *Tipo de carga útil* se utiliza para indicar al destino cuál algoritmo de codificación se utiliza en el paquete actual, de modo que sea posible modificarlo a solicitud.

Un problema al proveer retroalimentación es que los informes de RTCP se envían a todos los participantes. Para una aplicación de multidifusión con un grupo extenso, el ancho de banda utilizado por RTCP

aumentaría con rapidez. Para evitar que esto ocurra, los emisores de RTCP reducen la tasa de sus informes para que en conjunto no consuman más de, por ejemplo, 5% del ancho de banda de los medios. Para ello, cada participante necesita que el emisor le dé a conocer el ancho de banda de los medios; también necesita conocer el número de participantes, lo cual estima escuchando los otros informes de RTCP.

RTCP también maneja la sincronización entre flujos. El problema es que distintos flujos pueden utilizar relojes diferentes, con distintas granularidades y distintas tasas de derivación. RTCP se puede utilizar para mantenerlos sincronizados.

Por último, RTCP proporciona una forma para nombrar las diferentes fuentes (por ejemplo, en texto ASCII). Esta información se puede desplegar en la pantalla del receptor para indicar quién está hablando en ese momento.

Para encontrar más información sobre RTP, consulte a Perkins (2003).

### Reproducción mediante búfer y control de variación de retardo

Una vez que la información de los medios llega al receptor, se debe reproducir en el momento adecuado. En general, éste no será el tiempo en el que llegó el paquete RTP al receptor, ya que los paquetes tardan cantidades un poco distintas de tiempo en transitar por la red. Incluso si los paquetes se inyectan con los intervalos correctos y exactos entre ellos desde el emisor, llegarán al receptor con distintos tiempos relativos. A esta variación en el retardo se le conoce como **jitter**. Incluso una pequeña cantidad de jitter en los paquetes puede provocar molestos artefactos en los medios, como tramas de video entrecortado y audio incomprensible, si los medios simplemente se reproducen a medida que vayan llegando.

La solución a este problema es colocar los paquetes en un **búfer** en el receptor antes de reproducirlos, para reducir el jitter. Como ejemplo, en la figura 6-32 podemos ver que se entrega un flujo de paquetes con una cantidad considerable de jitter. El paquete 1 se envía del servidor en el tiempo $t = 0$ segundos y llega al cliente en el tiempo $t = 1$ segundo. El paquete 2 sufre de un retardo mayor y tarda 2 segundos en llegar. A medida que llegan los paquetes, se colocan en un búfer en la máquina cliente.

Al llegar al tiempo $t = 10$ segundos, empieza la reproducción. En este momento los paquetes 1 a 6 se encuentran en el búfer, por lo que se pueden extraer del búfer a intervalos uniformes para una reproducción continua. En el caso general, no es necesario usar intervalos uniformes debido a que las estampas de tiempo del RTP indican cuándo se deben reproducir los medios.

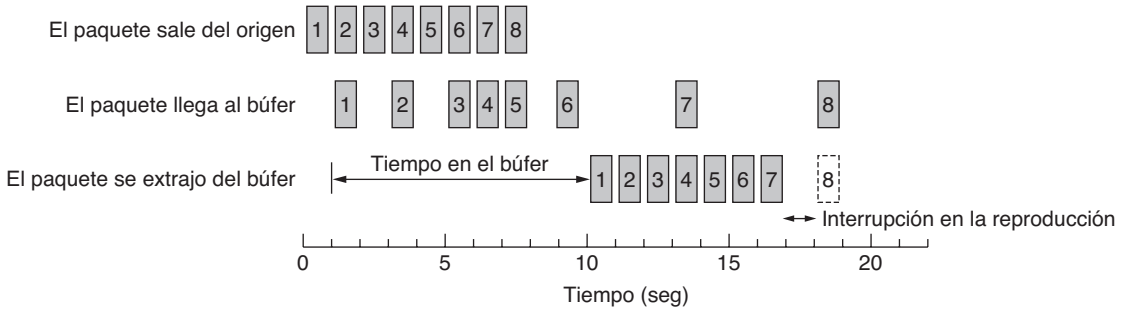


**Figura 6-32.** Hay que colocar los paquetes en un búfer para uniformar el flujo de salida.

Por desgracia, podemos ver que el paquete 8 se retrasó tanto que no está disponible cuando llega el momento de reproducirlo. Hay dos opciones. Podemos omitir el paquete 8 para que el reproductor avance a los paquetes subsecuentes. O también se puede detener la reproducción hasta que llegue el paquete 8, con lo

cual se crea una molesta interrupción en la música o película. En una aplicación de medios en vivo, como una llamada de voz sobre IP, por lo general se omite el paquete. Las aplicaciones en vivo no funcionan bien con la espera. En una aplicación de medios de flujo continuo el reproductor se podría detener. Podemos aligerar este problema si retardamos aún más el tiempo de inicio, mediante el uso de un búfer más grande. Para un reproductor de audio o video de flujo continuo, a menudo se usan búferes con cerca de 10 segundos para asegurar que el reproductor reciba todos los paquetes (que no se descarten en la red) a tiempo. Para las aplicaciones en vivo, como las videoconferencias, se necesitan búferes cortos para tener una capacidad de respuesta adecuada.

Una consideración clave para tener una reproducción uniforme es el **punto de reproducción**, o qué tanto tiempo esperar los medios en el receptor antes de reproducirlos. La decisión en cuanto al tiempo que debemos esperar depende del jitter. En la figura 6-33 se muestra la diferencia entre una conexión con un jitter bajo y otra con uno alto. El retardo promedio no puede diferir mucho entre las dos, pero si hay un jitter alto, tal vez el punto de reproducción tenga que estar mucho más adelante para poder capturar 99% de los paquetes, en comparación con un jitter bajo.

Para elegir un buen punto de reproducción, la aplicación puede medir el jitter si analiza la diferencia entre las estampas de tiempo RTP y el tiempo de llegada. Cada diferencia proporciona una muestra del retardo (más un desplazamiento fijo arbitrario). Sin embargo, el retardo puede cambiar con el tiempo debido al tráfico adicional que compite por el ancho de banda y a los cambios en las rutas. Para lidiar con este cambio, las aplicaciones pueden adaptar su punto de reproducción mientras se ejecutan. No obstante, si no hacen bien el cambio del punto de reproducción, el usuario puede llegar a percibir un problema técnico. Una manera de evitar este problema en el audio es adaptar el punto de reproducción entre las **ráfagas de voz** (*talkspurts*), en las interrupciones en la conversación. Nadie notará la diferencia entre un silencio corto y uno un poco más largo. RTP permite a las aplicaciones establecer el bit marcador *M* para indicar el inicio de una nueva ráfaga de voz para este fin.

Si el retardo absoluto hasta que se reproduzcan los medios es demasiado largo, las aplicaciones en vivo sufrirán. No se puede hacer nada para reducir el retardo de propagación si ya se está usando una trayectoria directa. Para retraer el punto de reproducción, simplemente hay que aceptar que habrá más paquetes que llegarán demasiado tarde como para reproducirlos. Si esto no es aceptable, la única forma de retraer el punto de reproducción es reducir el jitter mediante el uso de una mejor calidad de servicio; por ejemplo, el servicio diferenciado de reenvío expedito. Es decir, se necesita una mejor red.

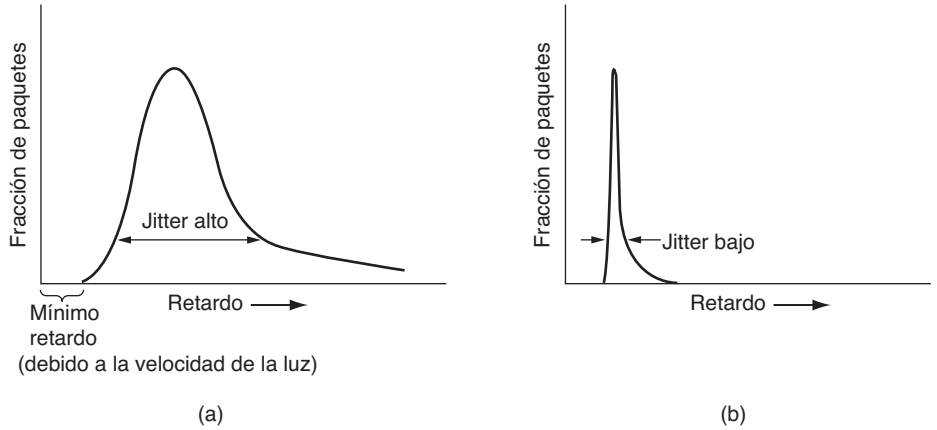


**Figura 6-33.** (a) Jitter alto. (b) Jitter bajo.

## 6.5 LOS PROTOCOLOS DE TRANSPORTE DE INTERNET: TCP

UDP es un protocolo simple y tiene algunos usos muy importantes, como las interacciones cliente-servidor y multimedia, pero para la mayoría de las aplicaciones de Internet se necesita una entrega en secuencia confiable. UDP no puede proporcionar esto, por lo que se requiere otro protocolo. Se llama TCP y es el más utilizado en Internet. A continuación lo estudiaremos con detalle.

### 6.5.1 Introducción a TCP

**TCP** (**Protocolo de Control de Transmisión**, del inglés *Transmission Control Protocol*) se diseñó específicamente para proporcionar un flujo de bytes confiable de extremo a extremo a través de una interred no confiable. Una interred difiere de una sola red debido a que sus diversas partes podrían tener diferentes topologías, anchos de banda, retardos, tamaños de paquete y otros parámetros. TCP se diseñó para adaptarse de manera dinámica a las propiedades de la interred y sobreponerse a muchos tipos de fallas.

TCP se definió formalmente en el RFC 793 en septiembre de 1981. Con el paso del tiempo se han realizado muchas mejoras y se han corregido varios errores e inconsistencias. Para que el lector tenga una idea de la magnitud de TCP, los RFC importantes son ahora el RFC 793 más: las aclaraciones y correcciones de errores en el RFC 1122; las extensiones para un alto desempeño en el RFC 1323; las confirmaciones de recepción selectivas en el RFC 2018; el control de congestión en el RFC 2581; la readaptación de los campos del encabezado para la calidad del servicio en el RFC 2873; los temporizadores de retransmisión mejorados en el RFC 2988 y la notificación explícita de congestión en el RFC 3168. La colección completa es todavía más grande, por lo cual se produjo una guía para los diversos documentos RFC, que por supuesto se publicó como otro documento RFC: el RFC 4614.

Cada máquina que soporta TCP tiene una entidad de transporte TCP, ya sea un procedimiento de biblioteca, un proceso de usuario o (lo más común) sea parte del kernel. En todos los casos, maneja flujos TCP e interactúa con la capa IP. Una entidad TCP acepta flujos de datos de usuario de procesos locales, los divide en fragmentos que no excedan los 64 KB (en la práctica, por lo general son 1460 bytes de datos para ajustarlos en una sola trama Ethernet con los encabezados IP y TCP), y envía cada pieza como un datagrama IP independiente. Cuando los datagramas que contienen datos TCP llegan a una máquina, se pasan a la entidad TCP, la cual reconstruye los flujos de bytes originales. Con el afán de simplificar, algunas veces utilizaremos sólo "TCP" para referirnos a la entidad de transporte TCP (una pieza de software) o al protocolo TCP (un conjunto de reglas). El contexto dejará claro a que nos referimos. Por ejemplo, en la frase "El usuario proporciona los datos a TCP", es claro que nos referimos a la entidad de transporte TCP.

La capa IP no ofrece ninguna garantía de que los datagramas se entregarán de manera apropiada, ni tampoco una indicación sobre qué tan rápido se pueden enviar los datagramas. Corresponde a TCP enviar los datagramas con la suficiente rapidez como para hacer uso de la capacidad sin provocar una congestión; también le corresponde terminar los temporizadores y retransmitir los datagramas que no se entreguen. Los datagramas que sí lleguen podrían hacerlo en el orden incorrecto; también corresponde a TCP reensamblarlos en mensajes con la secuencia apropiada. En resumen, TCP debe proporcionar un buen desempeño con la confiabilidad que la mayoría de las aplicaciones desean y que IP no proporciona.

### 6.5.2 El modelo del servicio TCP

El servicio TCP se obtiene al hacer que tanto el servidor como el receptor creen puntos terminales, llamados **sockets**, como se mencionó en la sección 6.1.3. Cada socket tiene un número (dirección) que consiste en la dirección IP del host y un número de 16 bits que es local para ese host, llamado **puerto**. Un puerto

es el nombre TCP para un TSAP. Para obtener el servicio TCP, hay que establecer de manera explícita una conexión entre un socket en una máquina y un socket en otra máquina. Las llamadas de socket se listan en la figura 6-5.

Podemos usar un socket para múltiples conexiones al mismo tiempo. En otras palabras, dos o más conexiones pueden terminar en el mismo socket. Las conexiones se identifican mediante los identificadores de socket de los dos extremos; esto es, (*socket1*, *socket2*). No se utilizan números de circuitos virtuales u otros identificadores.

Los números de puerto menores que 1024 están reservados para los servicios estándar que, por lo general, sólo los usuarios privilegiados pueden iniciar (por ejemplo, el usuario root en los sistemas UNIX). Éstos se llaman **puertos bien conocidos**. Por ejemplo, cualquier proceso que desee recuperar en forma remota el correo de un host se puede conectar con el puerto 143 del host de destino para contactarse con su demonio (daemon) IMAP. La lista de puertos bien conocidos se proporciona en www.iana.org. Se han asignado más de 700. En la figura 6-34 se listan algunos de los más conocidos.

| Puerto | Protocolo | Uso |
|---|---|---|
| 20, 21 | FTP | Transferencia de archivos. |
| 22 | SSH | Inicio de sesión remoto, reemplazo de Telnet. |
| 25 | SMTP | Correo electrónico. |
| 80 | HTTP | World Wide Web. |
| 110 | POP-3 | Acceso remoto al correo electrónico. |
| 143 | IMAP | Acceso remoto al correo electrónico. |
| 443 | HTTPS | Acceso seguro a web (HTTP sobre SSL/TLS). |
| 543 | RTSP | Control del reproductor de medios. |
| 631 | IPP | Compartición de impresoras. |

**Figura 6-34.** Algunos puertos asignados.

Se pueden registrar otros puertos del 1024 hasta el 49151 con la IANA para que los usuarios sin privilegios puedan usarlos, pero las aplicaciones pueden y seleccionan sus propios puertos. Por ejemplo, la aplicación BitTorrent para compartir archivos de igual a igual usa (de manera informal) los puertos 6881 a 6887, pero también puede operar en otros puertos.

Sin duda podría ser posible que el demonio FTP se conecte por sí solo al puerto 21 en tiempo de arranque, que el demonio SSH se conecte por sí solo al puerto 22 en tiempo de arranque, y así en lo sucesivo. Sin embargo, hacer lo anterior podría llenar la memoria con demonios que están inactivos la mayor parte del tiempo. En su lugar, lo que se hace por lo general es que un solo demonio, llamado *demonio de Internet* (*inetd*) en UNIX, se conecte por sí solo a múltiples puertos y espere la primera conexión entrante. Cuando eso ocurre, *inetd* bifurca un nuevo proceso y ejecuta el demonio apropiado en él, para dejar que ese demonio maneje la solicitud. De esta forma, los demonios distintos a *inetd* sólo están activos cuando hay trabajo para ellos. Inetd consulta un archivo de configuración para saber cuál puerto utilizar. En consecuencia, el administrador del sistema puede configurar el sistema para tener demonios permanentes en los puertos más ocupados (por ejemplo, el puerto 80) e *inetd* en los demás.

Todas las conexiones TCP son *full dúplex* y de punto a punto. *Full dúplex* significa que el tráfico puede ir en ambas direcciones al mismo tiempo. Punto a punto significa que cada conexión tiene exactamente dos puntos terminales. TCP no soporta la multidifusión ni la difusión.

Una conexión TCP es un flujo de bytes, no un flujo de mensajes. Los límites de los mensajes no se preservan de un extremo a otro. Por ejemplo, si el proceso emisor realiza cuatro escrituras de 512 bytes en un flujo TCP, tal vez estos datos se entreguen al proceso receptor como cuatro fragmentos de 512 bytes, dos fragmentos de 1024 bytes, uno de 2 048 bytes (vea la figura 6-35), o de alguna otra forma. No hay manera de que el receptor detecte la(s) unidad(es) en la(s) que se escribieron los datos, sin importar qué tanto se esfuerce.

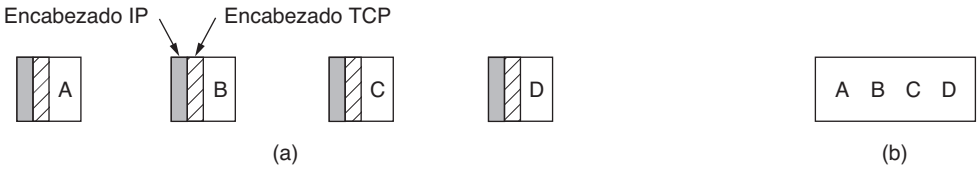


**Figura 6-35.** (a) Cuatro segmentos de 512 bytes que se envían como diagramas IP separados. (b) Los 2 048 bytes de datos que se entregan a la aplicación en una sola llamada READ.

Los archivos de UNIX también tienen esta propiedad. El lector de un archivo no puede indicar si éste se escribió un bloque a la vez, un byte a la vez o todo al mismo tiempo. Al igual que con un archivo de UNIX, el software TCP no tiene idea de lo que significan los bytes y no le interesa averiguarlo. Un byte es sólo un byte.

Cuando una aplicación pasa datos a TCP, éste decide entre enviarlos de inmediato o almacenarlos en el búfer (con el fin de recolectar una mayor cantidad y enviar todos los datos al mismo tiempo). Sin embargo, algunas veces la aplicación en realidad necesita que los datos se envíen de inmediato. Por ejemplo, suponga que el usuario de un juego interactivo quiere enviar un flujo continuo de actualizaciones. Es esencial que éstas se envíen de inmediato y no se almacenen en el búfer hasta que haya toda una colección. Para forzar la salida de los datos, TCP tiene la noción de una bandera PUSH que se transporta en los paquetes. La intención original era permitir que las aplicaciones indiquen a las implementaciones de TCP a través de la bandera PUSH que no retarden la transmisión. Sin embargo, las aplicaciones no pueden establecer literalmente la bandera PUSH cuando envían datos. En cambio, los distintos sistemas operativos han desarrollado distintas opciones para agilizar la transmisión (por ejemplo, TCP_NODELAY en Windows y Linux).

Para los arqueólogos de Internet, también mencionaremos una característica interesante del servicio TCP que permanece en el protocolo pero se usa muy raras veces: **datos urgentes**. Cuando una aplicación tiene datos de alta prioridad que se deben procesar de inmediato, por ejemplo cuando un usuario interactivo oprime las teclas CTRL-C para interrumpir un cálculo remoto que está en proceso, la aplicación emisora puede colocar cierta información de control en el flujo de datos y entregarla a TCP junto con la bandera URGENT. Este evento hace que TCP deje de acumular datos y transmita de inmediato todo lo que tiene para esa conexión.

Cuando los datos urgentes se reciben en el destino, la aplicación receptora se interrumpe (es decir, recibe una señal en términos de UNIX) para poder lo que estaba haciendo y leer el flujo de datos para encontrar los datos urgentes. El final de los datos urgentes se marca de modo que la aplicación sepa cuándo terminan. El inicio de los datos urgentes no se marca. Depende de la aplicación averiguarlo.

Este esquema proporciona básicamente un mecanismo de señalización simple y deja todo lo demás a la aplicación. Sin embargo, aunque los datos urgentes son útiles en potencia, no se encontró ninguna apli-

cación convincente para ellos desde un principio y dejaron de usarse. En la actualidad no se recomienda su uso debido a las diferencias de implementación, por lo que las aplicaciones tienen que manejar su propia señalización. Tal vez los futuros protocolos de transporte provean una mejor señalización.

### 6.5.3 El protocolo TCP

En esta sección daremos un repaso general del protocolo TCP. En la siguiente veremos el encabezado del protocolo, campo por campo.

Una característica clave de TCP, y una que domina el diseño del protocolo, es que cada byte de una conexión TCP tiene su propio número de secuencia de 32 bits. Cuando Internet comenzó, las líneas entre los enrutadores eran principalmente líneas rentadas de 56 kbps, por lo que un host que mandaba datos a toda velocidad tardaba una semana en recorrer los números de secuencia. A las velocidades de las redes modernas, los números de secuencia se pueden consumir con una rapidez alarmante, como veremos más adelante. Los números de secuencia separados de 32 bits se transmiten en paquetes para la posición de ventana deslizante en una dirección, y para las confirmaciones de recepción en la dirección opuesta, como veremos a continuación.

La entidad TCP emisora y receptora intercambian datos en forma de segmentos. Un **segmento TCP** consiste en un encabezado fijo de 20 bytes (más una parte opcional), seguido de cero o más bytes de datos. El software de TCP decide qué tan grandes deben ser los segmentos. Puede acumular datos de varias escrituras para formar un segmento, o dividir los datos de una escritura en varios segmentos. Hay dos límites que restringen el tamaño de segmento. Primero, cada segmento, incluido el encabezado TCP, debe caber en la carga útil de 65 515 bytes del IP. Segundo, cada enlace tiene una **MTU** (**Unidad Máxima de Transferencia**, del inglés *Maximum Transfer Unit*). Cada segmento debe caber en la MTU en el emisor y el receptor, de modo que se pueda enviar y recibir en un solo paquete sin fragmentar. En la práctica, la MTU es por lo general de 1500 bytes (el tamaño de la carga útil en Ethernet) y, por tanto, define el límite superior en el tamaño de segmento.

Sin embargo, de todas formas es posible que los paquetes IP que transportan segmentos TCP se fragmenten al pasar por una trayectoria de red en la que algún enlace tenga una MTU pequeña. Si esto ocurre, degradará el desempeño y provocará otros problemas (Kent y Mogul, 1987). En cambio, las implementaciones modernas de TCP realizan el **descubrimiento de MTU de la ruta** mediante el uso de la técnica descrita en el RF 1191 y que describimos en la sección 5.5.5. Esta técnica usa mensajes de error de ICMP para encontrar la MTU más pequeña para cualquier enlace en la ruta. Después, TCP ajusta el tamaño del segmento en forma descendente para evitar la fragmentación.

El protocolo básico que utilizan las entidades TCP es el protocolo de ventana deslizante con un tamaño dinámico de ventana. Cuando un emisor transmite un segmento, también inicia un temporizador. Cuando llega el segmento al destino, la entidad TCP receptora devuelve un segmento (con datos si existen, de otro modo sin ellos) que contiene un número de confirmación de recepción igual al siguiente número de secuencia que espera recibir, junto con el tamaño de la ventana remanente. Si el temporizador del emisor expira antes de recibir la confirmación de recepción, el emisor transmite de nuevo el segmento.

Aunque este protocolo suena sencillo, tiene muchos detalles prácticos que algunas veces son sutiles, los cuales explicaremos a continuación. Los segmentos pueden llegar fuera de orden, por lo que los bytes 3 072-4 095 pueden llegar pero tal vez sin que se envíe su respectiva confirmación de recepción, porque los bytes 2 048-3 071 no han aparecido aún. Además, los segmentos se pueden retardar tanto tiempo en tránsito que el temporizador del emisor expirará y este último retransmitirá los segmentos. Las retransmisiones podrían incluir rangos de bytes diferentes a los de la transmisión original, por lo cual se requiere una administración cuidadosa para llevar el control de los bytes que se han recibido de manera correcta en un momento determinado. Sin embargo, esto es factible ya que cada byte del flujo tiene su propio desplazamiento único.

El TCP debe estar preparado para manejar y resolver estos problemas de una manera eficiente. Se ha invertido una cantidad considerable de esfuerzo en la optimización del desempeño de los flujos TCP, incluso frente a problemas de red. A continuación estudiaremos varios de los algoritmos usados por muchas implementaciones de TCP.

### 6.5.4 El encabezado del segmento TCP

En la figura 6-36 se muestra la distribución de un segmento TCP. Cada segmento comienza con un encabezado de formato fijo de 20 bytes. El encabezado fijo puede ir seguido de encabezado de opciones. Después de las opciones, si las hay, pueden continuar hasta 65 535 − 20 − 20 = 65 495 bytes de datos, donde los primeros 20 se refieren al encabezado IP y los segundos al encabezado TCP. Los segmentos sin datos son legales y se usan por lo común para confirmaciones de recepción y mensajes de control.

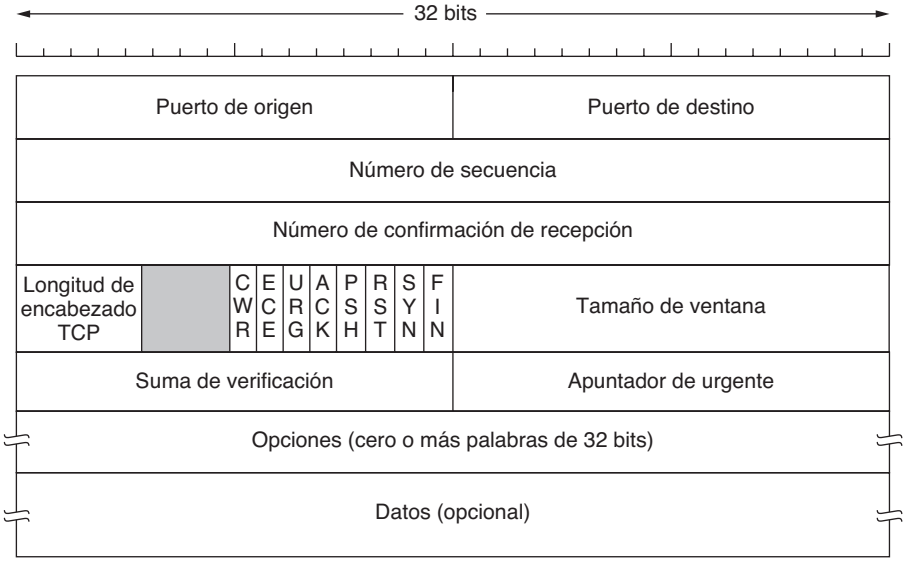


**Figura 6-36.** El encabezado TCP.

Ahora examinaremos el encabezado TCP campo por campo. Los campos *Puerto de origen* y *Puerto de destino* identifican los puntos terminales locales de la conexión. Un puerto TCP más la dirección IP de su host forman un punto terminal único de 48 bits. Los puntos terminales de origen y de destino en conjunto identifican la conexión. Este identificador de conexión se denomina **5-tupla,** ya que consiste en cinco piezas de información: el protocolo (TCP), IP de origen y puerto de origen, IP de destino y puerto de destino.

Los campos *Número de secuencia* y *Número de confirmación de recepción* desempeñan sus funciones normales. Cabe mencionar que el segundo especifica el siguiente byte en el orden esperado, no el último byte de manera correcta recibido. Es una **confirmación de recepción acumulativa** debido a que sintetiza los datos recibidos con un solo número. No va más allá de los datos perdidos. Ambos tienen 32 bits de longitud porque cada byte de datos está numerado en un flujo TCP.

La *Longitud del encabezado TCP* indica la cantidad de palabras de 32 bits contenidas en el encabezado TCP. Esta información es necesaria porque el campo *Opciones* es de longitud variable por lo que el encabezado también. Técnicamente, este campo en realidad indica el comienzo de los datos en el segmento,

medido en palabras de 32 bits, pero ese número es simplemente la longitud del encabezado en palabras, por lo que el efecto es el mismo.

A continuación viene un campo de 4 bits que no se usa. El hecho de que estos bits hayan permanecido sin ser utilizados durante 30 años (pues sólo se han reclamado 2 de los 6 bits reservados en un inicio) es testimonio de lo bien pensado que está l TCP. Protocolos inferiores habrían necesitado estos bits para corregir errores en el diseño original.

A continuación vienen ocho banderas de 1 bit. *CWR* y *ECE* se utilizan para indicar congestión cuando se usa ECN (Notificación Explícita de Congestión), como se especifica en el RFC 3168. *ECE* se establece para indicar una *Eco de ECN* (*ECN-Echo*) a un emisor TCP y decirle que reduzca su velocidad cuando el receptor TCP recibe una indicación de congestión de la red. *CWR* se establece para indicar una *Ventana de congestión reducida* del emisor TCP al receptor TCP, de modo que sepa que el emisor redujo su velocidad y puede dejar de enviar la *Repetición de ECN*. En la sección 6.5.10 analizaremos el papel de ECN en el control de congestión de TCP.

*URG* se establece en 1 si está en uso el *Apuntador urgente*. El *Apuntador urgente* se usa para indicar un desplazamiento en bytes a partir del número de secuencia actual en el que se deben encontrar los datos urgentes. Este recurso sustituye los mensajes de interrupción. Como se mencionó antes, este recurso es un mecanismo rudimentario para permitir que el emisor envíe una señal al receptor sin implicar a TCP en razón de la interrupción, pero se usa muy poco.

El bit *ACK* se establece en 1 para indicar que el *Número de confirmación de recepción* es válido. Éste es el caso para casi todos los paquetes. Si *ACK* es 0, el segmento no contiene una confirmación de recepción, por lo que se ignora el campo *Número de confirmación de recepción*.

El bit *PSH* indica datos que se deben transmitir de inmediato (*PUSHed data*). Por este medio se solicita atentamente al receptor que entregue los datos a la aplicación a su llegada y no los almacene en búfer hasta que se haya recibido un búfer completo (lo que podría hacer en otras circunstancias por razones de eficiencia).

El bit *RST* se usa para restablecer de manera repentina una conexión que se ha confundido debido a una falla de host o alguna otra razón. También se usa para rechazar un segmento no válido o un intento de abrir una conexión. Por lo general, si usted recibe un segmento con el bit *RST* encendido, tiene un problema entre manos.

El bit *SYN* se usa para establecer conexiones. La solicitud de conexión tiene $SYN = 1$ y $ACK = 0$ para indicar que el campo de confirmación de recepción superpuesto no está en uso. Pero la respuesta de conexión sí lleva una confirmación de recepción, por lo que tiene $SYN = 1$ y $ACK = 1$. En esencia, el bit *SYN* se usa para denotar CONNECTION REQUEST y CONNECTION ACCEPTED, y el bit *ACK* sirve para distinguir entre ambas posibilidades.

El bit *FIN* se usa para liberar una conexión y especifica que el emisor no tiene más datos que *transmitir*. Sin embargo, después de cerrar una conexión, el proceso encargado del cierre puede continuar *recibiendo* datos de manera indefinida. Ambos segmentos *SYN* y *FIN* tienen números de secuencia y, por tanto, se garantiza su procesamiento en el orden correcto.

El control de flujo en TCP se maneja mediante una ventana deslizante de tamaño variable. El campo *Tamaño de ventana* indica la cantidad de bytes que se pueden enviar, comenzando por el byte cuya recepción se ha confirmado. Un campo de *Tamaño de ventana* de 0 es válido e indica que se han recibido los bytes hasta *Número de confirmación de recepción* − 1, inclusive, pero que el receptor no ha tenido oportunidad de consumir los datos y ya no desea más datos por el momento. El receptor puede otorgar permiso más tarde para enviar mediante la transmisión de un segmento con el mismo *Número de confirmación de recepción* y un campo *Tamaño de ventana* distinto de cero.

En los protocolos del capítulo 3, las confirmaciones de recepción de las tramas recibidas y los permisos para enviar nuevas tramas estaban enlazados. Ésta fue una consecuencia de un tamaño de ventana fijo para cada protocolo. En TCP, las confirmaciones de recepción y los permisos para enviar datos adiciona-

les son por completo independientes. En efecto, un receptor puede decir: "He recibido bytes hasta *k*, pero por ahora no deseo más". Esta independencia (de hecho, una ventana de tamaño variable) proporciona una flexibilidad adicional. A continuación lo estudiaremos con más detalle.

También se proporciona una *Suma de verificación* para agregar confiabilidad. Realiza una suma de verificación del encabezado, los datos y un pseudoencabezado conceptual exactamente de la misma forma que UDP, sólo que el pseudoencabezado tiene el número de protocolo para TCP (6) y la suma de verificación es obligatoria. Consulte la sección 6.4.1 si desea más detalles.

El campo *Opciones* ofrece una forma de agregar las características adicionales que no están cubiertas por el encabezado normal. Se han definido muchas opciones, de las cuales varias se usan con frecuencia. Las opciones son de longitud variable, llenan un múltiplo de 32 bits mediante la técnica de relleno con ceros y se pueden extender hasta 40 bytes para dar cabida al encabezado TCP más largo que se pueda especificar. Algunas opciones se transmiten cuando se establece una conexión para negociar o informar al otro extremo sobre las capacidades disponibles. Otras opciones se transmiten en paquetes durante el tiempo de vida de la conexión. Cada opción tiene una codificación Tipo-Longitud-Valor (TLV).

Una opción muy utilizada es la que permite a cada host especificar el **MSS** (**Tamaño Máximo de Segmento**, del inglés *Maximum Segment Size*) que está dispuesto a aceptar. Es más eficiente usar segmentos grandes que segmentos pequeños, debido a que el encabezado de 20 bytes se puede amortizar entre más datos sean, pero los hosts pequeños tal vez no puedan manejar segmentos muy grandes. Durante el establecimiento de la conexión, cada lado puede anunciar su máximo y ver el de su compañero. Si un host no usa esta opción, tiene una carga útil predeterminada de 536 bytes. Se requiere que todos los hosts de Internet acepten segmentos TCP de 536 + 20 = 556 bytes. No es necesario que el tamaño máximo de segmento en ambas direcciones sea el mismo.

En las líneas con gran ancho de banda, alto retardo o ambas cosas, la ventana de 64 KB que corresponde a un campo de 16 bits es un problema. Por ejemplo, en una línea OC-12 (de al rededor de 600 Mbps), se requiere menos de 1 mseg para enviar una ventana de 64 KB completa. Si el retardo de propagación de ida y vuelta es de 50 mseg (lo cual es común para la fibra óptica transcontinental), el emisor estará inactivo por más de 98% del tiempo en espera de confirmaciones de recepción. Un tamaño de ventana más grande permitiría al emisor continuar enviando datos. La opción **escala de ventana** permite al emisor y al receptor negociar un factor de escala de ventana al inicio de una conexión. Ambos lados utilizan el factor de escala para desplazar el campo *Tamaño de ventana* hasta un máximo de 14 bits a la izquierda, con lo cual se permiten ventanas de hasta $2^{30}$ bytes. La mayoría de las implementaciones de TCP soportan esta opción.

La opción **estampa de tiempo** transmite una estampa de tiempo enviada por el emisor y repetida por el receptor. Se incluye en todos los paquetes una vez que se define su uso durante el establecimiento de la conexión, y se utiliza para calcular muestras del tiempo de ida y vuelta que se utilizan para estimar cuando se ha perdido un paquete. También se utiliza como extensión lógica del número de secuencia de 32 bits. En una conexión rápida, el número de secuencia se puede reiniciar muy rápido, lo cual puede llegar a provocar una confusión entre los datos viejos y los nuevos. El esquema **PAWS** (**Protección Contra el Reinicio de Números de Secuencia**, del inglés *Protection Against Wrapped Sequence numbers*) descarta los segmentos entrantes que tengan estampas de tiempo viejas para evitar este problema.

Por último, la opción **SACK** (**Confirmación de Recepción Selectiva**, del inglés *Selective ACKnowledgement*) permite a un receptor indicar al emisor los rangos de números de secuencia que ha recibido. Complementa el *Número de confirmación de recepción* y se utiliza después de haber perdido un paquete y de la llegada de los datos subsiguientes (o duplicados). Los nuevos datos no se reflejan mediante el campo *Número de confirmación de recepción* en el encabezado debido a que ese campo sólo proporciona el siguiente byte en el orden que se espera. Con SACK, el emisor está explícitamente consciente de los datos que tiene el receptor y, por ende, puede determinar qué datos se deben retransmitir. SACK se define

en el RFC 2108 y en el RFC 2883; su uso es cada vez más frecuente. En la sección 6.5.10 describiremos el uso de SACK junto con el control de congestión.

### 6.5.5 Establecimiento de una conexión TCP

En TCP las conexiones se establecen mediante el acuerdo de tres vías que estudiamos en la sección 6.2.2. Para establecer una conexión, uno de los lados (digamos que el servidor) espera en forma pasiva una conexión entrante mediante la ejecución de las primitivas LISTEN y ACCEPT en ese orden, ya sea que se especifique un origen determinado o a nadie en particular.

El otro lado (digamos que el cliente) ejecuta una primitiva CONNECT en la que especifica la dirección y el puerto con el que se desea conectar, el tamaño máximo de segmento TCP que está dispuesto a aceptar y de manera opcional algunos datos de usuario (por ejemplo, una contraseña). La primitiva CONNECT envía un segmento TCP con el bit *SYN* encendido y el bit *ACK* apagado, y espera una respuesta.

Cuando este segmento llega al destino, la entidad TCP de ahí revisa si hay un proceso que haya ejecutado una primitiva LISTEN en el puerto que se indica en el campo *Puerto de destino*. Si no lo hay, envía una respuesta con el bit *RST* encendido para rechazar la conexión.

Si algún proceso está escuchando en el puerto, ese proceso recibe el segmento TCP entrante y puede entonces aceptar o rechazar la conexión. Si la acepta, se devuelve un segmento de confirmación de recepción. La secuencia de segmentos TCP enviados en el caso normal se muestra en la figura 6-37(a). Observe que un segmento *SYN* consume 1 byte de espacio de secuencia, por lo que se puede reconocer sin ambigüedades.

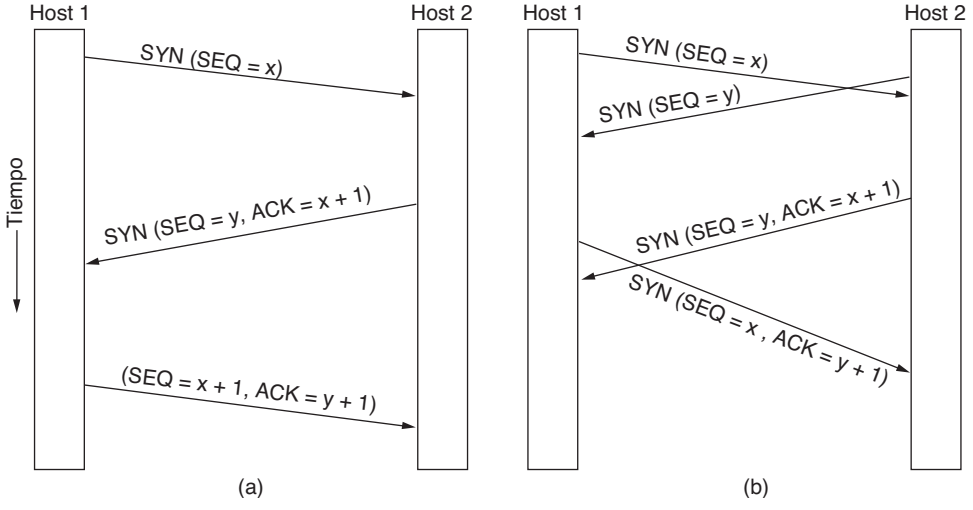


**Figura 6-37.** (a) Establecimiento de una conexión TCP en el caso normal. (b) Establecimiento de una conexión simultánea en ambos lados.

En el caso en que dos hosts intenten establecer al mismo tiempo una conexión entre los mismos dos sockets, la secuencia de eventos es como se ilustra en la figura 6-37(b). El resultado de estos eventos es que sólo se establece una conexión y no dos, pues las conexiones se identifican por sus puntos terminales. Si el primer establecimiento resulta en una conexión identificada por $(x, y)$ y el segundo también, sólo se hace una entrada de tabla; a saber, para $(x, y)$.

Recuerde que el número de secuencia inicial elegido por cada host se debe reiniciar con lentitud en vez de ser una constante tal como 0. Esta regla es para protegerse contra los paquetes duplicados retardados, como vimos en la sección 6.2.2. En un principio esto se lograba mediante un esquema basado en reloj, en donde éste emitía un pulso cada 4 μseg.

Sin embargo, una vulnerabilidad al implementar el acuerdo de tres vías es que el proceso de escucha debe recordar su número de secuencia tan pronto como responde con su propio segmento *SYN*. Esto significa que un emisor malicioso puede ocupar los recursos en un host si envía un flujo continuo de segmentos *SYN* y nunca les da seguimiento para completar la conexión. A este ataque se le conoce como **inundación SYN** y logró inutilizar varios servidores web en la década de 1990.

Una manera de defenderse contra este ataque es mediante el uso de la técnica **SYN cookies**. En vez de recordar el número de secuencia, un host selecciona un número de secuencia generado en forma criptográfica, lo coloca en el segmento de salida y se olvida de él. Si se completa el acuerdo de tres vías, este número de secuencia (más 1) se devolverá al host. Así, para regenerar el número de secuencia correcto hay que ejecutar la misma función criptográfica, siempre y cuando se conozcan las entradas para esa función; por ejemplo, la dirección IP y puerto del otro host, además de un secreto local. Este procedimiento permite al host verificar que un número de secuencia cuya recepción esté confirmada sea correcto sin tener que recordar el número de secuencia por separado. Existen algunos riesgos, como la incapacidad de manejar opciones TCP, por lo que la técnica SYN cookies sólo se puede usar cuando el host es sujeto de una inundación SYN. Sin embargo, son una interesante variante con relación al establecimiento de conexiones. Para obtener más información, consulte el RFC 4987 y a Lemon (2002).

### 6.5.6 Liberación de una conexión TCP

Aunque las conexiones TCP son *full dúplex*, para entender la manera en que se liberan las conexiones es mejor visualizarlas como un par de conexiones simplex. Cada conexión simplex se libera de manera independiente de su igual. Para liberar una conexión, cualquiera de las partes puede enviar un segmento TCP con el bit *FIN* establecido, lo que significa que no tiene más datos por transmitir. Al confirmarse la recepción de *FIN*, se apaga ese sentido para que no se transmitan nuevos datos. Sin embargo, lo datos pueden seguir fluyendo de manera indefinida por el otro sentido. Cuando se apagan ambos sentidos, se libera la conexión. Por lo general se requieren cuatro segmentos TCP para liberar una conexión: un *FIN* y un *ACK* para cada sentido. Sin embargo, es posible que el primer *ACK* y el segundo *FIN* estén contenidos en el mismo segmento, con lo cual se reduce la cuenta total a tres.

Al igual que con las llamadas telefónicas en las que ambas partes dicen adiós y cuelgan el teléfono al mismo tiempo, ambos extremos de una conexión TCP pueden enviar segmentos *FIN* al mismo tiempo. La recepción de ambos se confirma de la manera usual, y se apaga la conexión. De hecho, en esencia no hay diferencia entre la liberación secuencial o simultánea por parte de los hosts.

Para evitar el problema de los dos ejércitos (que vimos en la sección 6.2.3), se usan temporizadores. Si no llega una respuesta a un *FIN* en un máximo de dos tiempos de vida del paquete, el emisor del *FIN* libera la conexión. Tarde o temprano el otro lado notará que, al parecer, ya nadie lo está escuchando, y también expirará su temporizador. Aunque esta solución no es perfecta, dado el hecho de que teóricamente es imposible una solución perfecta, tendremos que conformarnos con ella. En la práctica, pocas veces ocurren problemas.

### 6.5.7 Modelado de administración de conexiones TCP

Los pasos requeridos para establecer y liberar conexiones se pueden representar en una máquina de estados finitos con los 11 estados que se listan en la figura 6-38. En cada estado son legales ciertos eventos. Al ocurrir un evento legal, debe emprenderse alguna acción. Si ocurre algún otro evento, se reporta un error.

Cada conexión comienza en el estado CLOSED (cerrado) y deja ese estado cuando hace una apertura pasiva (LISTEN) o una apertura activa (CONNECT). Si el otro lado realiza la acción opuesta, se establece una conexión y el estado se vuelve ESTABLISHED (establecida). La liberación de la conexión se puede iniciar desde cualquiera de los dos lados. Al completarse, el estado regresa a CLOSED.

| Estado | Descripción |
|---|---|
| CLOSED | No hay conexión activa o pendiente. |
| LISTEN | El servidor espera una llamada entrante. |
| SYN RCVD | Llegó una solicitud de conexión; espera ACK. |
| SYN SENT | La aplicación empezó a abrir una conexión. |
| ESTABLISHED | El estado normal de transferencia de datos. |
| FIN WAIT 1 | La aplicación notificó que ya terminó. |
| FIN WAIT 2 | El otro lado acordó liberar. |
| TIME WAIT | Espera a que todos los paquetes expiren. |
| CLOSING | Ambos lados intentaron cerrar al mismo tiempo. |
| CLOSE WAIT | El otro lado inició una liberación. |
| LAST ACK | Espera a que todos los paquetes expiren. |

**Figura 6-38.** Los estados utilizados en la máquina de estados finitos para administrar conexiones TCP.

La máquina de estados finitos se muestra en la figura 6-39. El caso común de un cliente que se conecta activamente a un servidor pasivo se indica con líneas gruesas (continuas para el cliente, punteadas para el servidor). Las líneas delgadas son secuencias de eventos no usuales. Cada línea de la figura 6-39 se marca mediante un par *evento/acción*. El evento puede ser una llamada de sistema iniciada por el usuario (CONNECT, LISTEN, SEND o CLOSE), la llegada de un segmento (*SYN*, *FIN*, *ACK* o *RST*) o, en un caso, una expiración de temporizador del doble del tiempo de vida máximo del paquete. La acción es el envío de un segmento de control (*SYN*, *FIN* o *RST*) o nada, lo cual se indica mediante (—). Los comentarios aparecen entre paréntesis.

Podemos entender mejor el diagrama si seguimos primero la trayectoria de un cliente (la línea continua gruesa) y luego la de un servidor (línea punteada gruesa). Cuando un programa de aplicación en la máquina cliente emite una solicitud CONNECT, la entidad TCP local crea un registro de conexión, lo marca para indicar que está en el estado *SYN SENT* y envía un segmento *SYN*. Cabe mencionar que muchas conexiones pueden estar abiertas (o en proceso de apertura) al mismo tiempo como parte de varias aplicaciones, por lo que el estado es por conexión y se graba en el registro de conexiones. Al llegar el *SYN + ACK*, TCP envía el último *ACK* del acuerdo de tres vías y cambia al estado *ESTABLISHED*. Ahora se pueden enviar y recibir datos.

Cuando una aplicación ha terminado, ejecuta una primitiva CLOSE; esto hace que la entidad TCP local envíe un segmento *FIN* y espere el *ACK* correspondiente (el recuadro punteado con la leyenda "cierre activo"). Al llegar el *ACK*, se hace una transición al estado *FIN WAIT 2* y se cierra un sentido de la conexión. Cuando cierra también el otro lado llega un *FIN*, para el cual se envía una confirmación de recepción. Ahora ambos lados están cerrados, pero el TCP espera un tiempo igual al doble del tiempo de vida máximo del paquete para garantizar que todos los paquetes de la conexión hayan expirado, sólo por

si acaso se perdió la confirmación de recepción. Al expirar el temporizador, TCP borra el registro de la conexión.

Examinemos ahora la administración de la conexión desde el punto de vista del servidor. El servidor emite una solicitud LISTEN y se detiene a esperar a que aparezca alguien. Cuando llega un *SYN*, se envía una confirmación de recepción y el servidor pasa al estado *SYN RCVD*. Cuando llega la confirmación de recepción del *SYN* del servidor, el acuerdo de tres vías se completa y el servidor pasa al estado *ESTABLISHED*. Ahora puede ocurrir la transferencia de datos.

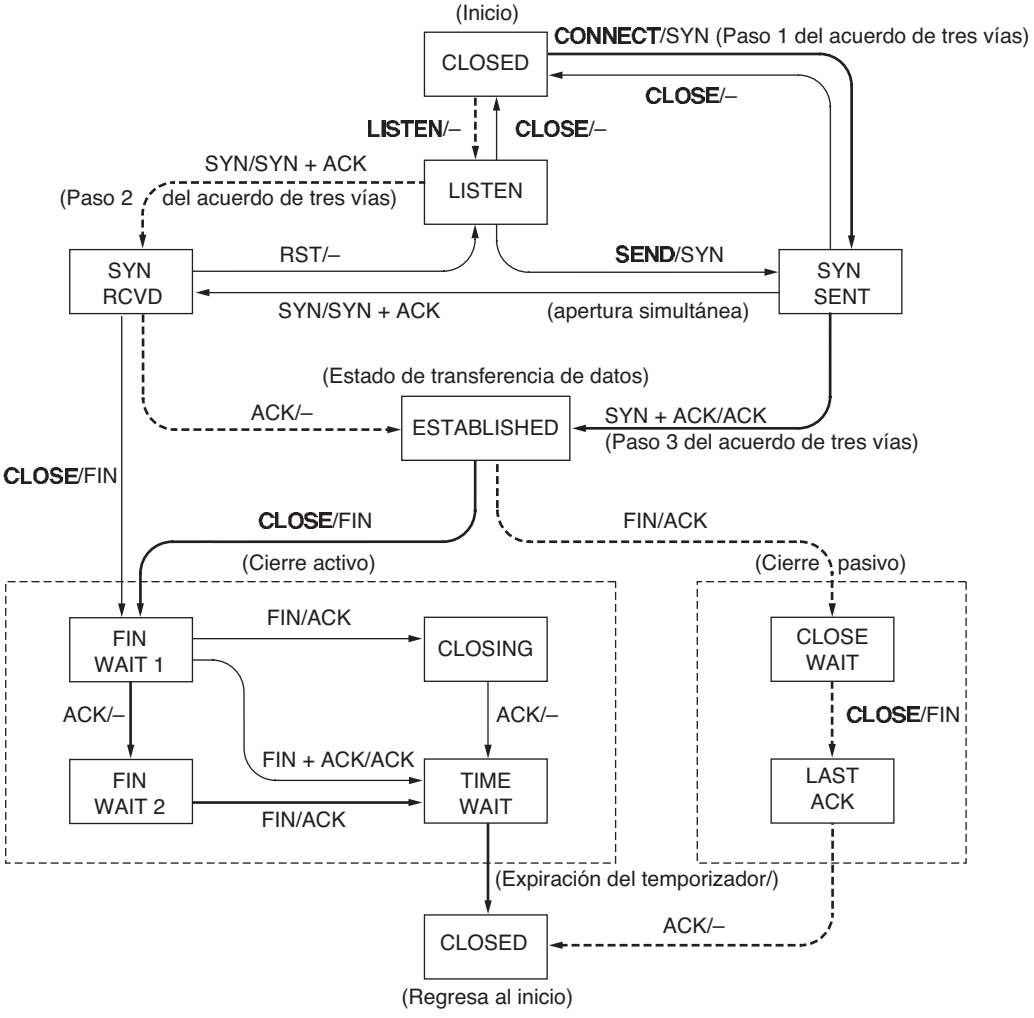


**Figura 6-39.** Máquina de estados finitos para administrar las conexiones TCP. La línea continua gruesa es la trayectoria normal para un cliente. La línea punteada gruesa es la trayectoria normal para un servidor. Las líneas delgadas son eventos no usuales. Cada transición se etiqueta con el evento que la ocasiona y la acción resultante, separada por una diagonal.

Cuando el cliente termina de transmitir sus datos, emite una solicitud CLOSE; esto provoca la llegada de un *FIN* al servidor (recuadro punteado con la leyenda "cierre pasivo"). Entonces se envía una señal al servidor. Cuando éste también emite una solicitud CLOSE, se envía un *FIN* al cliente. Al llegar la confirmación de recepción del cliente, el servidor libera la conexión y elimina el registro de conexión.

### 6.5.8 Ventana deslizante de TCP

Como ya vimos, la administración de ventanas en l TCP separa los aspectos de la confirmación de la recepción correcta de los segmentos y la asignación del búfer en el receptor. Por ejemplo, suponga que el receptor tiene un búfer de 4 096 bytes, como se muestra en la figura 6-40. Si el emisor transmite un segmento de 2 048 bytes que se recibe correctamente, el receptor enviará la confirmación de recepción del segmento. Sin embargo, dado que ahora sólo tiene 2 048 bytes de espacio de búfer (hasta que la aplicación retire algunos datos de éste), anunciará una ventana de 2048 comenzando con el siguiente byte esperado.

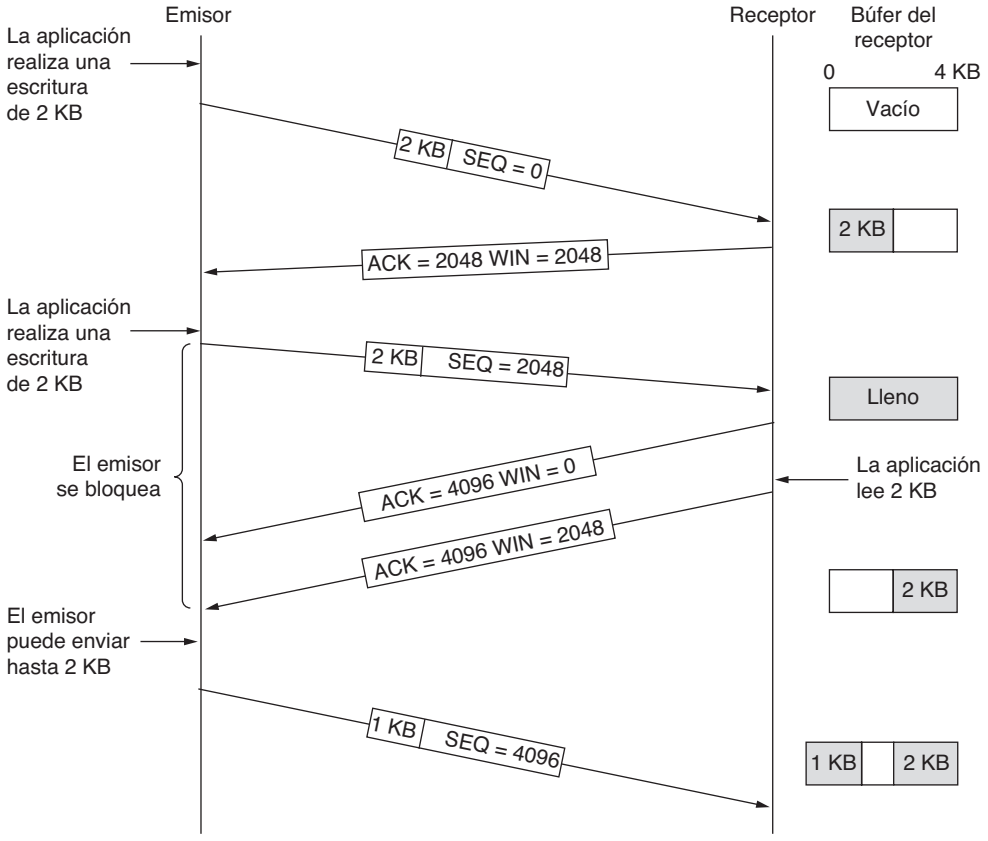


**Figura 6-40.** Administración de ventanas en TCP.

Ahora el emisor envía otros 2 048 bytes, para los cuales el receptor envía la confirmación de recepción, pero la ventana anunciada tiene un tamaño de 0. El emisor debe detenerse hasta que el proceso de aplicación en el host receptor retire algunos datos del búfer, momento en el que TCP podrá anunciar una ventana más grande y se podrán enviar más datos.

Cuando la ventana es de 0, el emisor por lo general no puede enviar segmentos, salvo en dos situaciones. En primer lugar, se pueden enviar datos urgentes (por ejemplo, para permitir que el usuario elimine el proceso en ejecución en la máquina remota). En segundo lugar, el emisor puede enviar un segmento de 1 byte para hacer que el receptor vuelva a anunciar el siguiente byte esperado y el tamaño de la ventana. Este paquete se denomina **sonda de ventana**. El estándar TCP proporciona explícitamente esta opción para evitar un interbloqueo si llega a perderse una actualización de ventana.

No se requiere que los emisores transmitan datos tan pronto como llegan de la aplicación. Tampoco se requiere que los receptores envíen confirmaciones de recepción tan pronto como sea posible. Por ejemplo, en la figura 6-40 cuando llegaron los primeros 2 KB de datos, TCP, a sabiendas que tenía disponible una ventana de 4 KB, hubiera actuado perfectamente bien si sólo almacenara en el búfer los datos hasta que llegaran otros 2 KB para transmitir un segmento con una carga útil de 4 KB. Podemos explotar esta libertad para mejorar el desempeño.

Considere una conexión a una terminal remota; por ejemplo, mediante el uso de SSH o telnet, que reacciona con cada pulso de tecla. En el peor de los casos, al llegar un carácter a la entidad TCP emisora, TCP crea un segmento TCP de 21 bytes que entrega al IP para que lo envíe como datagrama IP de 41 bytes. Del lado receptor, TCP envía de inmediato una confirmación de recepción de 40 bytes (20 bytes de encabezado TCP y 20 bytes de encabezado IP). Después, cuando la terminal remota lee el byte, TCP envía una actualización de ventana y recorre la ventana 1 byte hacia la derecha. Este paquete también es de 40 bytes. Por último, cuando la terminal remota procesa el carácter, lo retransmite para que se despliegue en forma local mediante un paquete de 41 bytes. En conjunto se usan 162 bytes de ancho de banda y se envían cuatro segmentos por cada carácter pulsado. Cuando el ancho de banda escasea, no es deseable este método de operación.

Un enfoque que usa muchas implementaciones de TCP para optimizar esta situación es el de las **confirmaciones de recepción con retardo**. La idea es retrasar las confirmaciones de recepción y las actualizaciones de ventana por hasta 500 mseg, con la esperanza de que lleguen algunos datos con los cuales se pueda viajar de manera gratuita. Suponiendo que la terminal hace eco en un lapso de 500 mseg, ahora el lado remoto sólo necesita enviar de vuelta un paquete de 41 bytes, con lo cual se recorta a la mitad la cuenta de paquetes y el uso de ancho de banda.

Aunque las confirmaciones de recepción con retardo reducen la carga impuesta en la red por el receptor, un emisor que envía varios paquetes cortos (por ejemplo, paquetes de 41 bytes que contengan 1 byte de datos) aún opera de manera ineficiente. El **algoritmo de Nagle** (Nagle, 1984) es una manera de reducir este uso. Lo que sugirió Nagle es sencillo: cuando llegan datos en pequeñas piezas al emisor, sólo se envía la primera pieza y el resto se almacena en búfer hasta que se confirma la recepción del byte pendiente. Después se envían todos los datos del búfer en un segmento TCP y nuevamente comienzan a almacenarse en búfer los datos hasta que se haya confirmado la recepción del siguiente segmento. Esto significa que sólo puede haber un paquete corto pendiente en cualquier momento dado. Si la aplicación envía muchas piezas de datos en el tiempo de ida y vuelta, el algoritmo de Nagle colocará todas las diversas piezas en un segmento, con lo cual se reducirá de manera considerable el ancho de banda utilizado. Además, el algoritmo establece que se debe enviar un nuevo segmento si se acumularon suficientes datos como para llenar un segmento máximo.

El algoritmo de Nagle se usa mucho en las implementaciones de TCP, pero hay veces en que es mejor deshabilitarlo. En particular, en los juegos interactivos que operan a través de Internet, por lo general los jugadores desean un flujo rápido de paquetes cortos de actualización. Si se acumulan las actualizaciones para enviarlas en ráfagas, el juego responderá de manera errática, lo cual provocará que los usuarios estén molestos. Un problema más sutil es que en ocasiones el algoritmo de Nagle puede interactuar con las confirmaciones de recepción con retardo para provocar un interbloqueo temporal: el receptor espera los datos sobre los cuales superponer una confirmación de recepción, y el emisor espera la confirmación de la recepción para enviar más datos. Esta interacción puede retardar las descargas de las páginas web. Debido a estos problemas, el algoritmo de Nagle se puede deshabilitar (lo cual se conoce como la opción *TCP_NODELAY*). Mogul y Minshall (2001) analizan ésta y otras soluciones.

Otro problema que puede arruinar el desempeño de TCP es el **síndrome de ventana tonta** (Clark, 1982). Este problema ocurre cuando se pasan datos a la entidad TCP emisora en bloques grandes, pero una aplicación interactiva del lado receptor lee datos sólo a razón de 1 byte a la vez. Para ver el problema, analice la figura 6-41. En un principio, el búfer TCP del lado receptor está lleno y el emisor lo sabe (es decir, tiene un

tamaño de ventana de 0). Entonces la aplicación interactiva lee un carácter del flujo TCP. Esta acción hace feliz al receptor TCP, por lo que envía una actualización de ventana al emisor para indicar que está bien que envíe 1 byte. El emisor accede y envía 1 byte. El búfer ahora está lleno, por lo que el receptor confirma la recepción del segmento de 1 byte y establece la ventana a 0. Este comportamiento puede continuar por siempre.

La solución de Clark es evitar que el receptor envíe una actualización de ventana para 1 byte. En cambio, se le obliga a esperar hasta tener disponible una cantidad decente de espacio, y luego lo anuncia. Específicamente, el receptor no debe enviar una actualización específica de ventana sino hasta que pueda manejar el tamaño máximo de segmento que anunció al establecerse la conexión, o hasta que su búfer quede a la mitad de capacidad, lo que sea más pequeño. Además, el emisor también puede ayudar al no enviar segmentos muy pequeños. En cambio, debe esperar hasta que pueda enviar un segmento completo, o por lo menos uno que contenga la mitad del tamaño del búfer del receptor.

El algoritmo de Nagle y la solución de Clark al síndrome de ventana tonta son complementarios. Nagle trataba de resolver el problema causado por la aplicación emisora que entregaba datos a TCP, 1 byte a la vez. Clark trataba de resolver el problema de que la aplicación receptora tomara los datos de TCP, 1 byte a la vez. Ambas soluciones son válidas y pueden operar juntas. El objetivo es que el emisor no envíe segmentos pequeños y que el receptor no los pida.

El receptor TCP también puede hacer más para mejorar el desempeño que sólo actualizar ventanas en unidades grandes. Al igual que el emisor TCP, tiene la capacidad de almacenar datos en el búfer, por lo que puede bloquear una solicitud READ de la aplicación hasta que pueda proporcionarle un bloque grande de datos. Al hacer esto se reduce la cantidad de llamadas a TCP (y la sobrecarga). También aumenta el tiempo de respuesta, pero en las aplicaciones no interactivas tales como la transferencia de archivos, la eficiencia puede ser más importante que el tiempo de respuesta a las solicitudes individuales.

Otro problema que el receptor debe solucionar es que los segmentos pueden llegar fuera de orden. El receptor colocará los datos en el búfer hasta que se puedan pasar en orden a la aplicación. En realidad no ocurriría nada malo si se descartaran los segmentos fuera de orden, ya que el emisor los retransmitiría en un momento dado, pero sería un desperdicio.

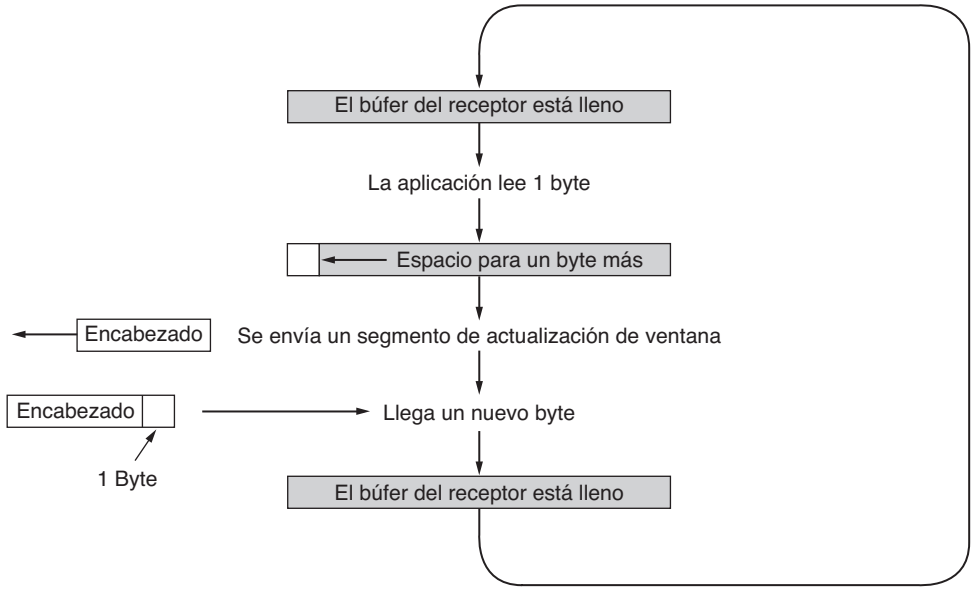


**Figura 6-41.** Síndrome de ventana tonta.

Las confirmaciones de recepción se pueden enviar sólo después de haber recibido todos los datos hasta el byte confirmado. A esto se le conoce como **confirmación de recepción acumulativa**. Si el receptor recibe los segmentos 0, 1, 2, 4, 5, 6 y 7, puede enviar una confirmación de recepción de todos los bytes hasta el último byte del segmento 2, inclusive. Al expirar el temporizador del emisor, éste retransmitirá el segmento 3. Como el receptor ha puesto en el búfer los segmentos 4 a 7, al recibir el segmento 3 puede enviar una confirmación de recepción de todos los bytes hasta el final del segmento 7.

### 6.5.9 Administración de temporizadores de TCP

TCP usa varios temporizadores (al menos de manera conceptual) para hacer su trabajo. El más importante de éstos es el **RTO** (**Temporizador de Retransmisión**, del inglés *Retransmission TimeOut*). Cuando se envía un segmento, se inicia un temporizador de retransmisiones. Si la confirmación de recepción del segmento llega antes de que expire el temporizador, éste se detiene. Por otro lado, si el temporizador termina antes de que llegue la confirmación de recepción, se retransmite el segmento (y se inicia de nuevo el temporizador). Surge entonces la pregunta: ¿qué tan grande debe ser el intervalo de expiración del temporizador?

Este problema es mucho más difícil en la capa de transporte que en los protocolos de enlace de datos tales como 802.11. En este último caso, el retardo esperado se mide en microsegundos y es muy predecible (es decir, tiene una varianza baja), por lo que el temporizador se puede establecer para expirar justo después del momento en que se esperaba la confirmación de recepción, como se muestra en la figura 6-42(a). Dado que las confirmaciones de recepción pocas veces se retardan en la capa de enlace de datos (debido a la falta de congestión), la ausencia de una confirmación de recepción en el momento esperado por lo general significa que se perdió la trama o la confirmación de recepción.

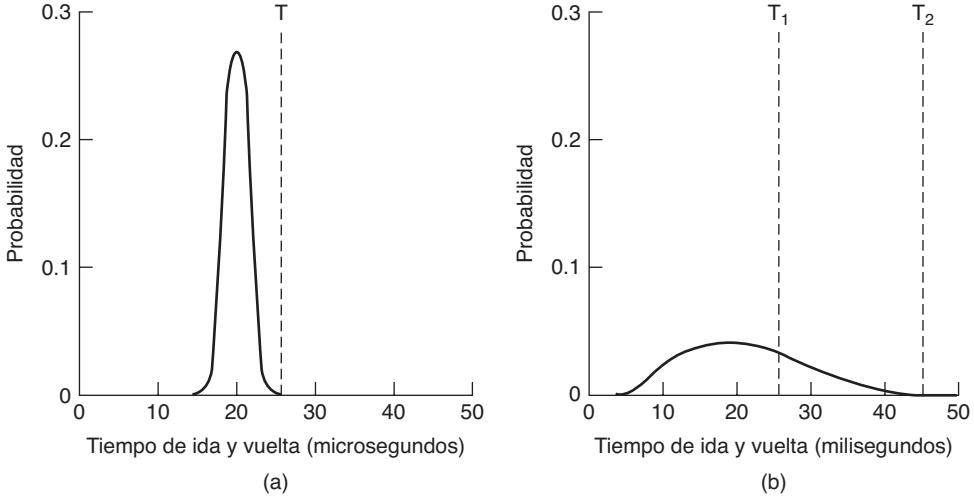


**Figura 6-42.** (a) Densidad de probabilidad de los tiempos de llegada de las confirmaciones de recepción en la capa de enlace de datos. (b) Densidad de probabilidad de los tiempos de llegada de las confirmaciones de recepción para TCP.

TCP se enfrenta a un entorno radicalmente distinto. La función de densidad de probabilidad del tiempo que tarda en regresar una confirmación de recepción TCP se parece más a la figura 6-42(b) que a la figura 6-42(a). Es más grande y variable. Es complicado determinar el tiempo de ida y vuelta al destino. Incluso cuando se conoce, es difícil decidir sobre el intervalo de expiración del temporizador. Si se establece demasiado corto, por decir $T_1$ en la figura 6-42(b), ocurrirán retransmisiones innecesarias e

Internet se llenará de paquetes inútiles. Si se establece demasiado largo (por ejemplo, $T_2$), el desempeño sufrirá debido al largo retardo de retransmisión de cada paquete perdido. Es más, la varianza y la media de la distribución de llegadas de confirmaciones de recepción pueden variar con rapidez en unos cuantos segundos, a medida que se generan y se resuelven congestionamientos.

La solución es usar un algoritmo dinámico que ajuste de manera constante el intervalo de expiración del temporizador, con base en mediciones continuas del desempeño de la red. El algoritmo utilizado por lo general por el TCP se lo debemos a Jacobson (1988) y funciona de la siguiente manera. Por cada conexión, TCP mantiene una variable llamada *SRTT* (Tiempo de Ida y Vuelta Suavizado), que es la mejor estimación actual del tiempo de ida y vuelta al destino en cuestión. Al enviarse un segmento, se inicia un temporizador, tanto para ver el tiempo que tarda la confirmación de recepción como para activar una retransmisión si se tarda demasiado. Si llega la confirmación de recepción antes de expirar el temporizador, TCP mide el tiempo que tardó la confirmación de recepción, por decir *R*. Luego actualiza el *SRTT* de acuerdo con la fórmula

$$SRTT = \alpha\, SRTT + (1 - \alpha)\, R$$

en donde $\alpha$ es un factor de suavizado que determina la rapidez con que se olvidan los valores anteriores. Por lo común, $\alpha = 7/8$. Este tipo de fórmula es un **EWMA** (**Promedio Móvil Ponderado Exponencialmente**, del inglés *Exponentially Weighted Moving Average*) o un filtro pasa bajas, que descarta el ruido en las muestras.

Incluso con un buen valor de *SRTT*, seleccionar la expiración adecuada del temporizador de retransmisión no es un asunto sencillo. Las primeras implementaciones de TCP usaban 2x*RTT*, pero la experiencia demostró que un valor constante era inflexible, puesto que no respondía cuando subía la varianza. En particular, los modelos de encolamiento de tráfico al azar (por ejemplo, Poisson) predicen que cuando la carga se acerca a la capacidad máxima, el retardo se hace grande y muy variable. Esto puede provocar que el temporizador de retransmisión se active y se retransmita una copia del paquete, aunque el paquete original aún esté transitando por la red. Es más probable que esto ocurra en condiciones de mucha carga, que viene siendo el peor momento para enviar paquetes adicionales por la red.

Para corregir este problema, Jacobson propuso hacer que el valor de expiración del temporizador fuera sensible a la diferencia en los tiempos de ida y vuelta, así como al tiempo de ida y vuelta suavizado. Para este cambio hay que llevar el registro de otra variable suavizada, *RTTVAR* (Variación de Tiempo de Ida y Vuelta) que se actualiza mediante la siguiente fórmula:

$$RTTVAR = \beta\, RTTVAR + (1 - \beta)\, |\, SRTT - R\, |$$

Ésta es también una fórmula EWMA; por lo general, $\beta = 3/4$. El tiempo de expiración de retransmisión, *RTO*, se establece así:

$$RTO = SRT + 4 \times RTTVAR$$

La elección del factor 4 es un tanto arbitraria, pero se puede hacer la multiplicación por 4 con un solo desplazamiento y menos de 1% de todos los paquetes llegan después de más de cuatro desviaciones estándar. Observe que *RTTVAR* no es exactamente lo mismo que la desviación estándar (en realidad es la desviación media), pero es lo bastante parecida en la práctica. La publicación de Jacobson está llena de trucos astutos para calcular los tiempos de expiración de los temporizadores con sólo usar sumas, restas y desplazamientos de enteros. Esta economía no es necesaria para los hosts modernos, pero se ha convertido en parte de la cultura que permite a TCP ejecutarse en todo tipo de dispositivos, desde supercomputadoras hasta pequeños dispositivos. Hasta ahora nadie ha logrado colocarlo dentro de un chip RFID, pero tal vez un día de éstos alguien lo logre.

En el RFC 2 988 se proporcionan más detalles acerca de cómo calcular este tiempo de expiración, incluyendo los valores iniciales de las variables. El temporizador de retransmisión también se mantiene en un mínimo de 1 segundo, sin importar las estimaciones. Éste es un valor conservador que se seleccionó para evitar retransmisiones espurias con base en las mediciones (Allman y Paxson, 1999).

Un problema que ocurre con la recopilación de las muestras, *R,* del tiempo de ida y vuelta es qué hacer cuando expira el temporizador de un segmento y se envía de nuevo. Cuando llega la confirmación de recepción, no está claro si ésta se refiere a la primera transmisión o a una posterior. Si adivinamos mal se puede contaminar seriamente el temporizador de retransmisión. Phil Karn descubrió este problema de la manera difícil. Él es un radioaficionado interesado en la transmisión de paquetes TCP/IP a través de la radio amateur, un medio notoriamente poco confiable. Karn hizo una propuesta sencilla: no actualizar las estimaciones sobre ninguno de los segmentos retransmitidos. Además, se duplicará el tiempo de expiración con cada retransmisión sucesiva hasta que los segmentos pasen a la primera. Esta corrección se conoce como **algoritmo de Karn** (Karn y Partridge, 1987). La mayoría de las implementaciones de TCP lo utilizan.

El temporizador de retransmisiones no es el único temporizador que TCP utiliza. El **temporizador de persistencia** es el segundo de ellos. Está diseñado para evitar el siguiente interbloqueo. El receptor envía una confirmación de recepción con un tamaño de ventana de 0 para indicar al emisor que espere. Después el receptor actualiza la ventana, pero se pierde el paquete con la actualización. Ahora, tanto el emisor como el receptor están esperando a que el otro haga algo. Cuando expira el temporizador de persistencia, el emisor transmite un sondeo al receptor. La respuesta al sondeo proporciona el tamaño de la ventana. Si aún es cero, se inicia el temporizador de persistencia una vez más y se repite el ciclo. Si es diferente de cero, ahora se pueden enviar datos.

Un tercer temporizador que utilizan algunas implementaciones es el **temporizador de seguir con vida** (*keepalive*). Cuando una conexión ha estado inactiva durante demasiado tiempo, el temporizador de seguir con vida puede expirar para ocasionar que un lado compruebe que el otro aún está ahí. Si no se recibe respuesta, se termina la conexión. Esta característica es controversial puesto que agrega sobrecarga y puede terminar una conexión saludable debido a una partición temporal de la red.

El último temporizador que se utiliza en cada conexión TCP es el que se usa en el estado TIME WAIT durante el cierre. Opera durante el doble del tiempo máximo de vida de paquete para asegurar que, al cerrarse una conexión, todos los paquetes creados por ella hayan desaparecido.

### 6.5.10 Control de congestión en TCP

Guardamos una de las funciones clave de TCP para lo último: el control de congestión. Cuando la carga ofrecida a cualquier red es mayor que la que puede manejar, se genera una congestión. Internet no es ninguna excepción. La capa de red detecta la congestión cuando las colas crecen demasiado en los enrutadores y trata de lidiar con este problema, aunque lo único que haga sea descartar paquetes. Es responsabilidad de la capa de transporte recibir la retroalimentación de congestión de la capa de red y reducir la tasa del tráfico que envía a la red. En Internet, TCP desempeña el papel principal en cuanto al control de la congestión, así como en el transporte confiable. Por esto se le considera un protocolo tan especial.

En la sección 6.3 vimos la situación general sobre el control de la congestión. Una conclusión clave fue la siguiente: un protocolo de transporte, que utiliza una ley de control AIMD (Incremento Aditivo/Decremento Multiplicativo) en respuesta a las señales de congestión binarias provenientes de la red, converge hacia una asignación de ancho de banda equitativa y eficiente. El control de congestión de TCP se basa en la implementación de esta metodología mediante el uso de una ventana y con la pérdida de paquetes como la señal binaria. Para ello, TCP mantiene una **ventana de congestión** cuyo tamaño es el número

de bytes que puede tener el emisor en la red en cualquier momento dado. La tasa correspondiente es el tamaño de ventana dividido entre el tiempo de viaje de ida y vuelta de la conexión. TCP ajusta el tamaño de la ventana, de acuerdo con la regla AIMD.

Recuerde que la ventana de congestión se mantiene *además* de la ventana de control de flujo, la cual especifica el número de bytes que el receptor puede colocar en el búfer. Ambas ventanas se rastrean en paralelo, y el número de bytes que se puede enviar es el número que sea menor de las dos ventanas. Por esto, la ventana efectiva es la cantidad más pequeña de lo que tanto el emisor como el receptor consideran que está bien. Se necesitan dos para bailar. TCP dejará de enviar datos si la ventana de congestión o la de control de flujo está temporalmente llena. Si el receptor dice "envía 64 KB" pero el emisor sabe que las ráfagas mayores de 32 KB obstruyen la red, enviará 32 KB. Por otro lado, si el receptor dice "envía 64 KB" y el emisor sabe que las ráfagas de hasta 128 KB pasan sin problema, enviará los 64 KB solicitados. Ya describimos antes la ventana de control de flujo, por lo que a continuación sólo describiremos la ventana de congestión.

El control de congestión moderno se agregó a TCP en gran parte gracias a los esfuerzos de Van Jacobson (1988). Es una historia fascinante. A partir de 1986, la creciente popularidad de Internet condujo a la primera ocurrencia de lo que más tarde se conoció como **colapso por congestión**, un periodo prolongado en donde el caudal útil se reducía en forma precipitada (es decir, por más de un factor de 100) debido a la congestión en la red. Jacobson (y muchos otros) buscaba comprender qué estaba ocurriendo para remediar la situación.

La corrección de alto nivel que implementó Jacobson era aproximar una ventana de congestión AIMD. La parte interesante, y una parte importante de la complejidad del control de congestión en TCP, es ver cómo Jacobson agregó esto a una implementación existente sin cambiar ninguno de los formatos de los mensajes, gracias a lo cual se podía implementar al instante. Para empezar, observó que la pérdida de paquetes es una señal adecuada de congestión. Esta señal llega un poco tarde (puesto que la red ya se encuentra congestionada) pero es bastante confiable. Después de todo, es difícil construir un enrutador que no descarte paquetes cuando está sobrecargado. No es muy probable que este hecho vaya a cambiar. Incluso cuando aparezcan memorias con capacidades de terabytes para colocar grandes cantidades de paquetes en el búfer, probablemente habrá redes de terabits/seg para llenar estas memorias.

Sin embargo, para usar la pérdida de paquetes como una señal de congestión es necesario que los errores de transmisión sean relativamente raros. Por lo general esto no es así para los enlaces inalámbricos como las redes 802.11, lo cual explica por qué incluyen su propio mecanismo de retransmisión en la capa de enlace. Debido a las retransmisiones inalámbricas, es común que la pérdida de paquetes en la capa de red debido a los errores de transmisión, se enmascare en las redes inalámbricas. También es algo raro en otros enlaces, ya que los cables y la fibra óptica por lo general tienen tasas bajas de error de bits.

Todos los algoritmos de TCP en Internet suponen que los paquetes perdidos se deben a la congestión, por lo cual monitorean las expiraciones de los temporizadores y buscan señales de problemas. Se requiere un buen temporizador de retransmisión para detectar las señales de pérdida de paquetes con precisión y en forma oportuna. Ya hemos visto cómo el temporizador de retransmisión de TCP incluye estimaciones de la media y la variación en los tiempos de ida y vuelta. Corregir este temporizador mediante la inclusión del factor de variación fue un paso importante en el trabajo realizado por Jacobson. Si se tiene un buen temporizador de retransmisión, el emisor de TCP puede rastrear el número pendiente de bytes que están cargando la red. Simplemente analiza la diferencia entre los números de secuencia que se transmiten y los números de secuencia cuya confirmación de recepción se ha enviado.

Ahora, tal parece que nuestra tarea es sencilla. Todo lo que tenemos que hacer es rastrear la ventana de congestión mediante el uso de los números de secuencia y los que ya se han confirmado, y ajustar la ventana de congestión mediante el uso de una regla AIMD. Pero como podríamos esperar, es más complicado que eso. Una de las primeras consideraciones es que la forma en que se envían los paquetes a la red, incluso a través de periodos cortos, debe coincidir con la trayectoria de red. En caso contrario, el

tráfico provocará una congestión. Por ejemplo, considere un host con una ventana de congestión de 64 KB conectado a una red Ethernet conmutada de 1 Gbps. Si el host envía toda la ventana a la vez, esta ráfaga de tráfico puede viajar a través de una línea ADSL lenta de 1 Mbps en un punto posterior en la trayectoria. La ráfaga que tardó sólo medio milisegundo en la línea de 1 Gbps obstruirá la línea de 1 Mbps durante medio segundo, lo cual perturbará por completo a los protocolos tales como el de voz sobre IP. Este comportamiento podría ser conveniente en un protocolo diseñado para provocar congestión, pero no en uno para controlarla.

Sin embargo, resulta ser que podemos usar pequeñas ráfagas de paquetes para nuestra ventaja. La figura 6-43 muestra qué ocurre cuando un emisor en una red rápida (el enlace de 1 Gbps) envía una pequeña ráfaga de cuatro paquetes a un receptor en una red lenta (el enlace de 1 Mbps), que constituye el cuello de botella o la parte más lenta de la trayectoria. En un principio los cuatro paquetes viajan a través del enlace lo más rápido que el emisor puede enviarlos. En el enrutador se ponen en cola mientras se envían, ya que es más tardado enviar un paquete a través del enlace lento que recibir el siguiente paquete a través del enlace rápido. Pero la cola no es grande debido a que sólo se envió un pequeño número de paquetes a la vez. Observe el aumento en la longitud de los paquetes en el enlace lento. El mismo paquete, por decir de 1 KB, ahora es más largo debido a que se requiere más tiempo para enviarlo por un enlace lento que por uno rápido.

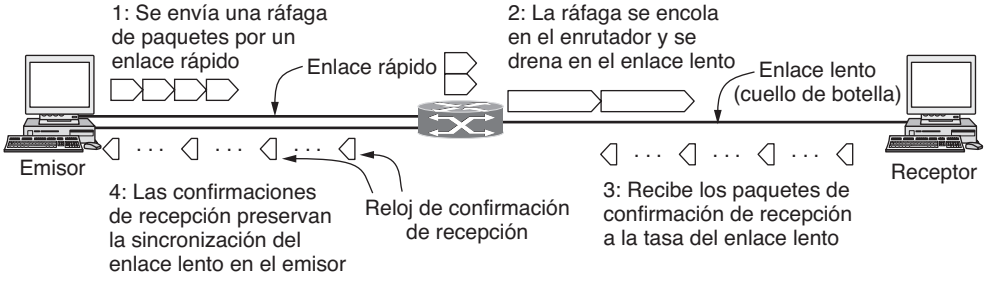


**Figura 6-43.** Una ráfaga de paquetes de un emisor y el reloj de confirmación de recepción devuelto.

En un momento dado, los paquetes llegan al receptor y se confirma su recepción. Los tiempos para las confirmaciones de recepción reflejan los tiempos en los que llegaron los paquetes al receptor después de atravesar el enlace lento. Están dispersos en comparación con los paquetes originales en el enlace rápido. Mientras estas confirmaciones de recepción viajan a través de la red y regresan al emisor, preservan esta sincronización.

La observación clave es ésta: las confirmaciones de recepción regresan al emisor aproximadamente a la misma tasa a la que se pueden enviar los paquetes a través del enlace más lento en la trayectoria. Ésta es la tasa que el emisor quiere usar. Si inyecta nuevos paquetes en la red a esta tasa, se enviarán tan rápido como lo permita el enlace lento, pero no se encolarán ni congestionarán ningún enrutador a lo largo de la trayectoria. Esta sincronización se conoce como **reloj de confirmación de recepción** (*ack clock*) y es una parte esencial de TCP. Mediante el uso de un reloj de confirmación de recepción, TCP regula el tráfico y evita las colas innecesarias en los enrutadores.

Una segunda consideración es que la regla AIMD tardará mucho tiempo para llegar a un buen punto de operación en las redes rápidas si la ventana de congestión se inicia a partir de un tamaño pequeño. Considere una trayectoria de red modesta que puede soportar 10 Mbps con un RTT de 100 milisegundos. La ventana de congestión apropiada es el producto ancho de banda-retardo, que es de 1 Mbit, o 100 paquetes de 1250 bytes cada uno. Si la ventana de congestión empieza en 1 paquete y se incrementa 1 paquete por cada RTT, transcurrirán 100 RTT o 10 segundos antes de que la conexión opere a la tasa correcta aproximada. Hay que esperar mucho tiempo sólo para llegar a la velocidad correcta para una transferencia.

Podríamos reducir este tiempo de inicio si empezáramos con una ventana inicial más grande, por decir de 50 paquetes. Pero esta ventana sería demasiado grande para los enlaces lentos o cortos. Provocaría congestión si se utilizara toda a la vez, como hemos descrito antes.

En cambio, la solución que Jacobson eligió para manejar ambas consideraciones es una mezcla de incremento lineal y de incremento multiplicativo. Al establecer una conexión, el emisor inicializa la ventana de congestión con un pequeño valor inicial, de cuando menos cuatro segmentos; los detalles se describen en el RFC 3 390 y el uso de cuatro segmentos es una mejoría respecto a un valor inicial anterior de un segmento, con base en la experiencia. A continuación el emisor envía la ventana inicial. Los paquetes tardarán un tiempo de ida y vuelta para que se confirme su recepción. Para cada segmento cuya recepción se confirme antes de que expire el temporizador de retransmisión, el emisor agrega a la ventana de congestión una cantidad de bytes equivalente a un segmento. Además, como ya se confirmó la recepción de ese segmento, ahora hay un segmento menos en la red. El resultado es que cada segmento confirmado permite enviar dos segmentos más. La ventana de congestión se duplica cada tiempo de ida y vuelta.

Este algoritmo se conoce como **inicio lento** (*slow start*), pero no es para nada lento (su crecimiento es exponencial), excepto en comparación con el algoritmo anterior que permitía enviar toda una ventana de control de flujo al mismo tiempo. El inicio lento se muestra en la figura 6-44. En el primer tiempo de ida y vuelta, el emisor inyecta un paquete a la red (y el receptor recibe un paquete). En el siguiente tiempo de ida y vuelta se envían dos paquetes, y en el tercer tiempo de ida y vuelta cuatro paquetes.

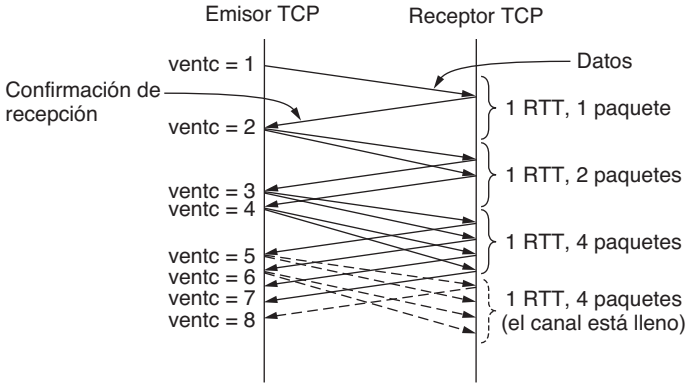


**Figura 6-44.** Inicio lento desde una ventana de congestión inicial de un segmento.

El inicio lento funciona bien sobre un rango de velocidades de enlaces y tiempos de ida y vuelta; además utiliza un reloj de confirmación de recepción para asociar la tasa de transmisiones del emisor con la trayectoria de red. En la figura 6-44 puede observar la forma en que las confirmaciones de recepción regresan del emisor al receptor. Cuando el emisor recibe una confirmación de recepción, incrementa en uno la ventana de congestión y envía de inmediato dos paquetes a la red (un paquete es el incremento en uno; el otro paquete es un reemplazo para el paquete cuya confirmación se recibió y salió de la red. La ventana de congestión proporciona en todo momento el número de paquetes sin confirmación de recepción). Sin embargo, estos dos paquetes no llegarán necesariamente al receptor con un espaciamiento tan estrecho como cuando se enviaron. Por ejemplo, suponga que el emisor se encuentra en una red Ethernet de 100 Mbps. Cada paquete de 1250 bytes tarda 100 μseg en enviarse. Así, el retardo entre los paquetes puede ser como mínimo de 100 μseg. La situación cambia si estos paquetes pasan a través de un enlace ADSL de 1 Mbps en cualquier parte a lo largo de la trayectoria. Ahora se requieren 10 mseg para enviar el mismo paquete. Esto significa que el espaciamiento mínimo entre los dos paquetes aumentó por un factor de 100.

A menos que los paquetes tengan que esperar juntos en una cola de un enlace posterior, el espaciamiento seguirá siendo grande.

En la figura 6-44 se muestra este efecto al imponer un espaciamiento mínimo entre los paquetes de datos que llegan al receptor. Se mantiene el mismo espaciamiento cuando el receptor envía confirmaciones de recepción, y también cuando el emisor recibe esas confirmaciones. Si la trayectoria de la red es lenta, las confirmaciones de recepción llegarán con lentitud (después de un retardo de un RTT). Si la trayectoria de la red es rápida, las confirmaciones de recepción llegarán con rapidez (de nuevo, después del RTT). Todo lo que tiene que hacer el emisor es seguir la sincronización del reloj de confirmación de recepción mientras inyecta nuevos paquetes; esto es lo que hace el inicio lento.

Como el inicio lento provoca un crecimiento exponencial, en un momento dado (y más pronto que tarde) enviará demasiados paquetes a la red con mucha rapidez. Cuando esto ocurra, las colas se acumularán en la red. Cuando las colas estén llenas se perderán uno o más paquetes. Una vez que ocurra esto, el temporizador del emisor TCP expirará cuando una confirmación de recepción no llegue a tiempo. Hay evidencia de que el inicio lento aumenta con demasiada rapidez en la figura 6-44. Después de tres RTT, hay cuatro paquetes en la red. Estos cuatro paquetes tardan todo un RTT en llegar al receptor. Es decir, una ventana de congestión de cuatro paquetes es el tamaño correcto para esta conexión. Sin embargo, a medida que se confirma la recepción de estos paquetes, el inicio lento continúa aumentando el tamaño de la ventana de congestión y llega a ocho paquetes en otro RTT. Sólo cuatro de estos paquetes pueden llegar al receptor en un RTT, sin importar cuántos se envíen. Es decir, el canal de la red está lleno. Los paquetes adicionales que el emisor coloque en la red se acumularán en las colas de los enrutadores, ya que no pueden entregarse al receptor con la suficiente rapidez. Pronto ocurrirá una congestión y se perderán paquetes.

Para mantener el inicio lento bajo control, el emisor mantiene un umbral para la conexión, conocido como **umbral de inicio lento**. En un principio este valor se establece en un nivel arbitrariamente alto, al tamaño de la ventana de control de flujo, de manera que no limite la conexión. TCP continúa incrementando la ventana de congestión en el inicio lento hasta que expira un temporizador o la ventana de congestión excede el umbral (o cuando se llena la ventana del receptor).

Por ejemplo, cada vez que se detecta la pérdida de un paquete debido a la expiración de un temporizador, el umbral de inicio lento se establece a la mitad de la ventana de congestión y se reinicia todo el proceso. La idea es que la ventana actual es demasiado grande, puesto que antes provocó una congestión que ahora sólo se puede detectar mediante la expiración de un temporizador. Quizá la mitad de la ventana, que se utilizó con éxito en un tiempo anterior, sea una mejor estimación para una ventana de congestión que está cerca de la capacidad de la trayectoria y no provocará pérdida. En nuestro ejemplo de la figura 6-44, si se aumenta el tamaño de la ventana de congestión a ocho paquetes se puede provocar una pérdida, mientras que la ventana de congestión de cuatro paquetes en el RTT anterior tenía el valor correcto. Así, la ventana de congestión se restablece a su valor pequeño inicial y se reanuda el inicio lento.

Cada vez que se atraviesa el umbral de inicio lento, TCP cambia del inicio lento al incremento aditivo. En este modo, la ventana de congestión se incrementa un segmento por cada tiempo de ida y vuelta. Al igual que el inicio lento, por lo general esto se implementa mediante un incremento por cada segmento cuya recepción se ha confirmado, en vez de usar un incremento una vez por cada RTT. Llamemos *ventc* a la ventana de congestión y *MSS* al tamaño de segmento máximo. Una aproximación común sería incrementar *ventc* mediante la fórmula (*MSS* $\times$ *MSS*)/*ventc* por cada uno de los *ventc*/*MSS* paquetes cuya recepción se pueda confirmar. Este incremento no necesita ser rápido. La idea general es que una conexión TCP pase mucho tiempo con su ventana de congestión cerca del valor óptimo: no tan pequeño como para que la tasa de transferencia real sea baja y no tan grande como para que ocurra una congestión.

El incremento aditivo se muestra en la figura 6-45 para la misma situación que el inicio lento. Al final de cada RTT, la ventana de congestión del emisor ha crecido lo suficiente como para inyectar un paquete

adicional a la red. En comparación con el inicio lento, la tasa lineal de crecimiento es mucho menor. No hay mucha diferencia para las ventanas de congestión pequeñas, como en este caso, sino una gran diferencia en el tiempo que se requiere para hacer crecer la ventana de congestión hasta 100 segmentos, por ejemplo.

También hay algo más que podemos hacer para mejorar el desempeño. El defecto en el esquema hasta ahora es esperar a que expire un temporizador. Los tiempos de expiración son algo largos debido a que deben ser conservadores. Después de perder un paquete, el receptor no puede confirmar la recepción más allá de ese paquete, por lo que el número de confirmación de recepción permanecerá fijo y el emisor no podrá enviar nuevos paquetes a la red, debido a que su ventana de congestión permanecerá llena. Esta condición puede continuar durante un periodo relativamente largo, hasta que el temporizador se dispare y se retransmita el paquete perdido. En ese punto, TCP vuelve a empezar con el inicio lento.

Hay una forma rápida de que el emisor reconozca que se ha perdido uno de sus paquetes. A medida que llegan al receptor los paquetes subsiguientes al perdido, activan confirmaciones de recepción que regresan al emisor. Estas confirmaciones de recepción contienen el mismo número de confirmación de recepción y, por ende, se denominan **confirmaciones de recepción duplicadas**. Cada vez que el emisor recibe una confirmación de recepción duplicada, es probable que haya llegado otro paquete al receptor y que el paquete perdido todavía no aparezca.

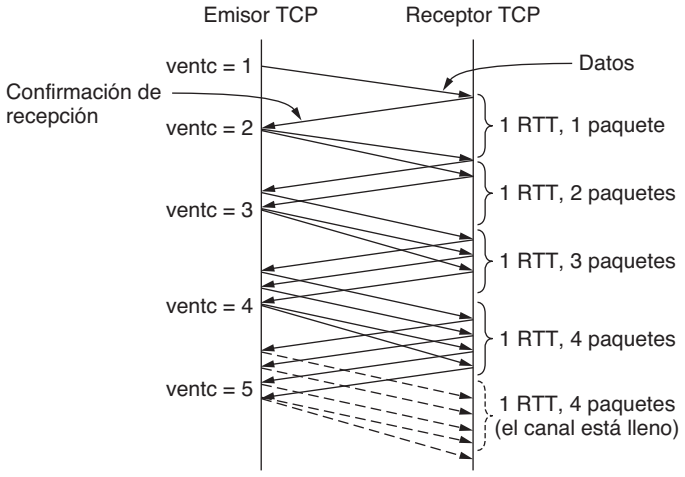


**Figura 6-45.** Incremento aditivo de una ventana de congestión inicial de un segmento.

Como los paquetes pueden tomar distintas trayectorias por la red, pueden llegar fuera de orden. Esto activará confirmaciones de recepción duplicadas, incluso aunque no se hayan perdido paquetes. Sin embargo, la mayor parte del tiempo esto es poco común en Internet. Cuando hay un reordenamiento a través de múltiples trayectorias, por lo general los paquetes recibidos no se reordenan demasiado. Así, TCP asume con cierta arbitrariedad que tres confirmaciones de recepción duplicadas indican que se ha perdido un paquete. La identidad del paquete perdido se puede inferir también del número de confirmación de recepción. Es el siguiente paquete inmediato en la secuencia. Entonces este paquete se puede retransmitir de inmediato, antes de que se dispare el temporizador de retransmisión.

Esta heurística se denomina **retransmisión rápida**. Una vez que se dispara, el umbral de inicio lento sigue establecido a la mitad de la ventana de congestión actual, justo como cuando expira un temporizador. Para volver a comenzar con el inicio lento, hay que establecer la ventana de congestión a un paquete. Con este tamaño de ventana, se enviará un nuevo paquete después del tiempo de ida y vuelta que se

requiere para confirmar la recepción del paquete retransmitido, junto con los datos que se habían enviado antes de detectar la pérdida.

En la figura 6-46 se muestra una ilustración del algoritmo de congestión que hemos construido hasta ahora. Esta versión de TCP se conoce como TCP Tahoe, en honor de la versión 4.2BSD Tahoe de 1988, en la que se incluyó. Aquí, el tamaño máximo de segmento es de 1 KB. En un principio la ventana de congestión era de 64 KB, pero ocurrió una expiración de temporizador, por lo que el umbral se estableció a 32 KB y la ventana de congestión a 1 KB para la transmisión 0. La ventana de congestión crece en forma exponencial hasta que llega al umbral (32 KB). La ventana se incrementa cada vez que llega una nueva confirmación de recepción, en vez de hacerlo en forma continua, lo cual provoca el patrón discreto de escalera. Una vez que se traspasa el umbral, la ventana crece en forma lineal. Se incrementa en un segmento por cada RTT.

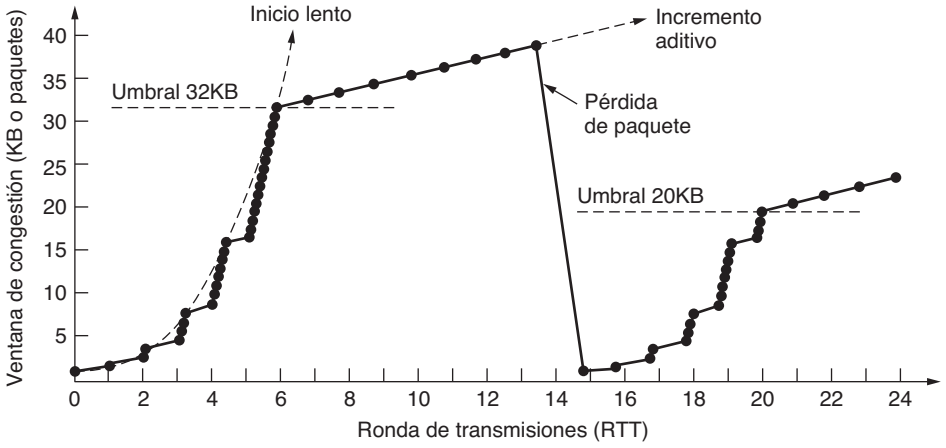


**Figura 6-46.** Inicio lento seguido de un incremento aditivo en el TCP Tahoe.

Las transmisiones en la ronda 13 son desafortunadas (deberían haberlo sabido), por lo que se pierde una de ellas en la red. Esto se detecta cuando llegan tres confirmaciones de recepción duplicadas. En ese momento se retransmite el paquete perdido, el umbral se establece a la mitad de la ventana actual (que para ahora es de 40 KB, por lo que la mitad es 20 KB) y se empieza de nuevo el proceso de inicio lento. Al reiniciar con una ventana de congestión de un paquete, se requiere un tiempo de ida y vuelta para que todos los datos transmitidos con anterioridad salgan de la red y se confirme su recepción, incluyendo el paquete retransmitido. La ventana de congestión crece con el inicio lento, como lo hizo antes, hasta que llega al nuevo umbral de 20 KB. En ese momento el crecimiento se vuelve lineal otra vez. Continuará de esta forma hasta que se detecte la pérdida de otro paquete mediante confirmaciones de recepción duplicadas o al expirar un temporizador (o cuando la ventana del receptor se convierta en el límite).

El TCP Tahoe (que incluía buenos temporizadores de retransmisión) ofrecía un algoritmo de control de congestión funcional que resolvió el problema del colapso por congestión. Jacobson descubrió que es posible hacerlo aún mejor. Al momento de la retransmisión rápida, la conexión está operando con una ventana de congestión demasiado grande, pero aún sigue operando con un reloj de confirmación de recepción funcional. Cada vez que llega otra confirmación de recepción duplicada, es probable que otro paquete haya dejado la red. Al usar las confirmaciones de recepción duplicadas para contabilizar los paquetes en la red, es posible dejar que algunos paquetes salgan de la red y continúen enviando uno nuevo para cada confirmación de recepción duplicada adicional.

La **recuperación rápida** es la heurística que implementa este comportamiento. Es un modo temporal que busca mantener el reloj de confirmación de recepción en operación con una ventana de congestión que es el nuevo umbral, o la mitad del valor de la ventana de congestión al momento de la retransmisión rápida. Para ello, se contabilizan las confirmaciones de recepción duplicadas (incluyendo las tres que desencadenaron la retransmisión rápida) hasta que el número de paquetes en la red disminuye a un valor equivalente al nuevo umbral. Esto requiere alrededor de medio tiempo de ida y vuelta. De ahí en adelante, se puede enviar un nuevo paquete por cada confirmación de recepción duplicada que se reciba. Un tiempo de ida y vuelta después de la retransmisión rápida se habrá confirmado la recepción del paquete perdido. En ese momento cesará el flujo de confirmaciones de recepción duplicadas y terminará el modo de recuperación rápida. Ahora la ventana de congestión se establecerá al nuevo umbral de inicio lento y crecerá mediante un incremento lineal.

El resultado de esta heurística es que TCP evita el inicio lento, excepto cuando la conexión se inicia por primera vez y cuando expira un temporizador. Esto último puede aún ocurrir cuando se pierde más de un paquete y la retransmisión rápida no recupera en forma adecuada. En vez de inicios lentos repetidos, la ventana de congestión de una conexión funcional sigue un patrón de **diente de sierra** de incremento aditivo (un segmento por cada RTT) y decremento multiplicativo (la mitad en un RTT). Ésta es exactamente la regla AIMD que buscábamos implementar.

En la figura 6-47 se muestra este comportamiento de diente de sierra. Se produce mediante el TCP Reno, nombrado en honor a la versión 4.3BSD Reno de 1990, en la cual se incluyó. En esencia, el TCP Reno es el TCP Tahoe más la recuperación rápida. Después de un inicio lento al principio, la ventana de congestión sube en forma lineal hasta que se detecta la pérdida de un paquete mediante confirmaciones de recepción duplicadas. El paquete perdido se retransmite y se utiliza la recuperación rápida para mantener el reloj de confirmación de recepción funcionando hasta que se confirma la recepción de la retransmisión. En ese momento, la ventana de congestión se reanuda a partir del nuevo umbral de inicio lento, en vez de empezar desde 1. Este comportamiento continúa en forma indefinida, y la conexión pasa la mayor parte del tiempo con su ventana de congestión cerca del valor óptimo del producto ancho de banda-retardo.

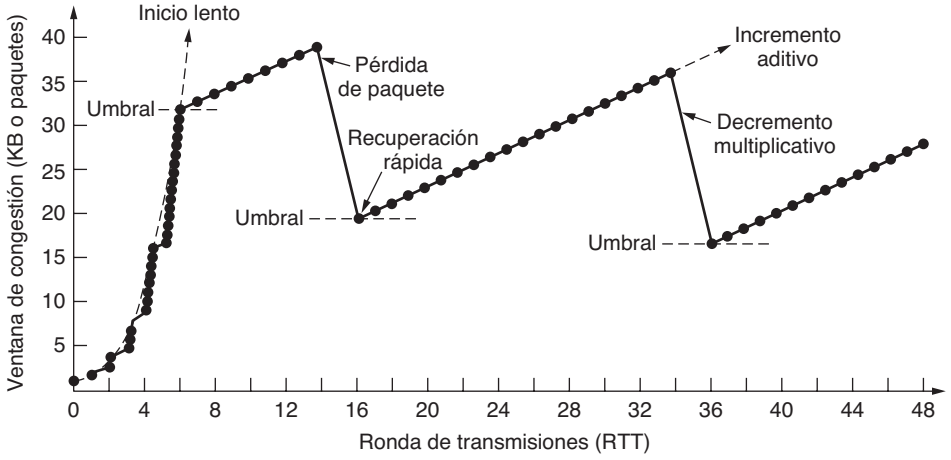


**Figura 6-47.** La recuperación rápida y el patrón de diente de sierra de TCP Reno.

Con sus mecanismos para ajustar la ventana de congestión, el TCP Reno ha formado la base para el control de congestión en TCP durante más de dos décadas. La mayoría de los cambios en los años intermedios han ajustado estos mecanismos en pequeñas formas; por ejemplo, al cambiar las opciones de la

ventana inicial y eliminar varias ambigüedades. Se han logrado algunas mejoras para recuperarse de dos o más pérdidas en una ventana de paquetes. Por ejemplo, la versión TCP NewReno usa un avance parcial del número de confirmación de recepción después de una retransmisión para encontrar y reparar otra pérdida (Hoe, 1996), según lo descrito en el RFC 3782. A mediados de la década de 1990 surgieron diversas variaciones que siguen los principios que hemos descrito, pero usan leyes de control un poco distintas. Por ejemplo, Linux usa una variante llamada CUBIC TCP (Ha y colaboradores, 2008), y Windows incluye una variante llamada Compound TCP (Tan y colaboradores, 2006).

También hay dos cambios más grandes que han afectado las implementaciones de TCP. En primer lugar, gran parte de la complejidad de TCP proviene de inferir mediante un flujo de confirmaciones de recepción duplicadas qué paquetes han llegado y cuáles se han perdido. El número de confirmación de recepción acumulativa no provee esta información. Una corrección simple es el uso de **SACK** (**Confirmaciones de Recepción Selectivas**, del inglés *Selective ACKnowledgements*), que lista hasta tres rangos de bytes que se hayan recibido. Con esta información, el emisor puede decidir de una manera más directa qué paquetes retransmitir, para así rastrear los paquetes en tránsito e implementar la ventana de congestión.

Cuando el emisor y el receptor establecen una conexión, cada uno envía la opción *SACK permitido* de TCP para indicar que comprenden las confirmaciones de recepción selectivas. Una vez que se habilita SACK para una conexión, funciona como se muestra en la figura 6-48. Un receptor usa el campo *Número de confirmación de recepción* de TCP de la manera usual, como una confirmación de recepción acumulativa del byte en orden más alto que se haya recibido. Cuando recibe el paquete 3 fuera de orden (debido a que se perdió el paquete 2), envía una *opción SACK* para los datos recibidos, junto con la confirmación de recepción acumulativa (duplicada) para el paquete 1. La *opción SACK* proporciona los rangos de bytes que se han recibido por encima del número proporcionado por la confirmación de recepción acumulativa. El primer rango es el paquete que desencadenó la confirmación de recepción duplicada. Los siguientes rangos, si están presentes, son bloques anteriores. Por lo común se utilizan hasta tres rangos. Para cuando se recibe el paquete 6, se utilizan dos rangos de bytes SACK para indicar que se recibieron el paquete 6 y los paquetes 3 a 4, además de todos los paquetes hasta el 1. De la información en cada *opción SACK* que recibe, el emisor puede decidir qué paquetes retransmitir. En este caso, sería una buena idea retransmitir los paquetes 2 y 5.

SACK es información estrictamente de asesoría. La detección real de la pérdida mediante el uso de confirmaciones de recepción duplicadas y ajustes a la ventana de congestión se lleva a cabo de la misma manera que antes. Sin embargo, mediante SACK, TCP se puede recuperar con más facilidad de las situaciones en las que se pierden varios paquetes casi al mismo tiempo, ya que el emisor TCP sabe qué paquetes no se han recibido. Ahora SACK se implementa ampliamente. Se describe en el RFC 2883, y el control de congestión en TCP mediante el uso de SACK se describe en el RFC 3517.

El segundo cambio es el uso de **ECN** (**Notificación Explícita de Congestión**, del inglés *Explicit Congestion Notification*) además de la pérdida de paquetes como señal de congestión. ECN es un mecanismo de la capa de IP para notificar a los hosts sobre la congestión, el cual describimos en la sección 5.3.4. Con este mecanismo, el receptor TCP puede recibir señales de congestión de IP.

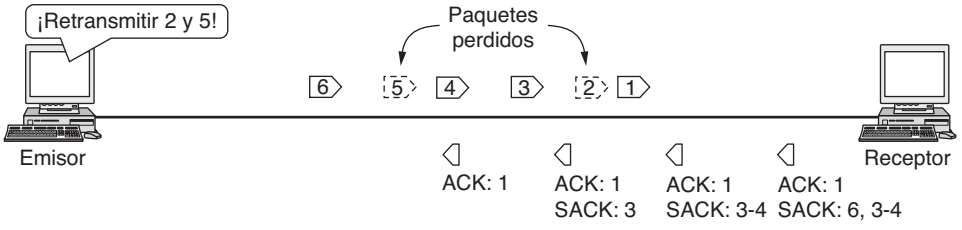


**Figura 6-48.** Confirmaciones de recepción selectivas.

El uso de ECN se habilita para una conexión TCP cuando tanto el emisor como el receptor indican que son capaces de usar ECN y activan los bits *ECE* y *CWR* al momento de establecer la conexión. Si se utiliza ECN, cada paquete que transporte un segmento TCP se señala con banderas en el encabezado IP para mostrar que puede transportar una señal ECN. Los enrutadores que soporten ECN establecerán una señal de congestión en los paquetes que puedan transportar banderas ECN cuando se aproxime la congestión, en vez de descartar esos paquetes después de que ocurra la congestión.

Se informa al receptor TCP si cualquier paquete que llegue transporta una señal de congestión de ECN. Luego el receptor utiliza la bandera *ECE* (*ECN Echo*) para indicar al emisor TCP que sus paquetes han experimentado una congestión. Para indicar al receptor que escuchó la señal, el emisor usa la bandera *CWR* (Ventana de Congestión Reducida).

El emisor TCP reacciona a estas notificaciones de congestión exactamente de la misma forma que reacciona con la pérdida de paquetes que se detecta mediante confirmaciones de recepción duplicadas. Sin embargo, la situación es mucho mejor. Se ha detectado la congestión y ningún paquete resultó dañado de ninguna forma. ECN se describe en el RFC 3168. Requiere soporte tanto del host como de los enrutadores, y todavía no se utiliza ampliamente en Internet.

Para obtener más información sobre el conjunto completo de los comportamientos de control de congestión que se implementan en TCP, consulte el RFC 5 681.

### 6.5.11 El futuro de TCP

Como caballo de batalla de Internet, TCP se ha utilizado para muchas aplicaciones y se ha extendido en el transcurso del tiempo para brindar un buen desempeño a través de un amplio rango de redes. Existen muchas versiones en uso con implementaciones un poco distintas de los algoritmos clásicos que hemos descrito, en especial para el control de la congestión y la robustez ante los ataques. Es probable que TCP continúe evolucionando con Internet. A continuación mencionaremos dos aspectos específicos.

El primero es que TCP no proporciona la semántica de transporte que desean todas las aplicaciones. Por ejemplo, algunas aplicaciones desean enviar mensajes o registros cuyos límites hay que preservar. Otras aplicaciones trabajan con un grupo de conversaciones relacionadas, como un navegador web que transfiere varios objetos del mismo servidor. Existen también otras aplicaciones que desean un mejor control sobre las trayectorias de red que utilizan. TCP con su interfaz de sockets estándar no cumple bien con esas necesidades. En esencia, la aplicación tiene la responsabilidad de lidiar con cualquier problema que TCP no resuelva. Esto ha conducido a propuestas de nuevos protocolos que proporcionen una interfaz ligeramente distinta. Dos ejemplos de esos protocolos son **SCTP** (**Protocolo de Control de Transmisión de Flujo**, del inglés *Stream Control Transmision Protocol*), que se define en el RFC 4 960, y **SST** (**Transporte Estructurado de Flujo**, del inglés *Structured Stream Transport*) (Ford, 2007). Sin embargo, cada vez que alguien propone cambiar algo que ha funcionado tan bien y por tanto tiempo, siempre hay una enorme batalla entre los "usuarios que exigen más características" y los que dicen "si no está roto, no hay que arreglarlo".

El segundo aspecto es el control de la congestión. Tal vez usted piense que éste es un problema resuelto después de nuestras deliberaciones y los mecanismos que se han desarrollado con el tiempo. Pero no es así. La forma de control de congestión en TCP que hemos descrito, y que se utiliza muchos, se basa en las pérdidas de paquetes como señal de congestión. Cuando Padhye y colaboradores (1998) modelaron la tasa de transferencia real de TCP con base en el patrón de diente de sierra, descubrieron que la tasa de pérdida de paquetes debe disminuir con rapidez con base en el aumento de velocidad. Para alcanzar una tasa de transferencia real de 1 Gbps con un tiempo de ida y vuelta de 100 ms y paquetes de 1500 bytes, se puede perder un paquete casi cada 10 minutos. Ésta es una tasa de pérdida de paquetes de $2 \times 10^{-8}$, un valor increíblemente pequeño. Es muy poco frecuente como para que sirva como una buena señal de

congestión, y cualquier otra fuente de pérdida (por ejemplo, tasas de errores de transmisión de paquetes de $10^{-7}$) puede dominarla con facilidad y, por ende, limitar la tasa de transferencia real.

Esta relación no ha representado un problema en el pasado, pero las redes se están volviendo cada vez más rápidas, lo cual ha orillado a muchas personas a revisar el control de la congestión. Una posibilidad es usar un control de congestión alternativo, en el que la señal no sea ningún tipo de pérdida de paquetes. En la sección 6.2 vimos varios ejemplos. La señal podría ser el tiempo de ida y vuelta, que aumenta cuando la red se congestiona, según se utiliza en FAST TCP (Wei y colaboradores, 2006). Existen también otras metodologías posibles; el tiempo dirá cuál es la mejor.

## 6.6 ASPECTOS DEL DESEMPEÑO

Los asuntos relacionados con el desempeño son muy importantes en las redes de computadoras. Cuando hay cientos o miles de computadoras conectadas entre sí, son comunes las interacciones complejas, con consecuencias imprevisibles. Con frecuencia, esta complejidad conduce a un desempeño pobre, sin que nadie sepa por qué. En las siguientes secciones examinaremos muchos aspectos relacionados con el desempeño de las redes para ver los tipos de problemas que existen y lo que podemos hacer para resolverlos.

Por desgracia, comprender el desempeño de las redes es más un arte que una ciencia. Existe muy poca teoría al respecto que tenga en realidad alguna utilidad en la práctica. Lo mejor que podemos hacer es dar algunas reglas empíricas derivadas de los tropiezos y ejemplos actuales tomados del mundo real. Hemos postergado de manera intencional este análisis hasta después de estudiar la capa de transporte, debido a que el desempeño que reciben las aplicaciones depende del desempeño combinado de las capas de transporte, de red y de enlace, además de la posibilidad de usar TCP como ejemplo en varios lugares.

En las siguientes secciones analizaremos seis aspectos del desempeño de las redes:

1. Problemas de desempeño.
2. Medición del desempeño de una red.
3. Diseño de hosts para redes rápidas.
4. Procesamiento rápido de los segmentos.
5. Compresión de encabezados.
6. Protocolos para redes de alto desempeño.

Estos aspectos consideran el desempeño de las redes tanto en el host como a través de la misma red, y a medida que las redes aumentan en velocidad y tamaño.

### 6.6.1 Problemas de desempeño en las redes de computadoras

Algunos problemas de desempeño, como la congestión, se deben a sobrecargas temporales de los recursos. Si repentinamente llega más tráfico a un enrutador del que puede manejar, ocurrirá una congestión y se reducirá el desempeño. Ya estudiamos la congestión con detalle en este capítulo y en el anterior.

El desempeño también se degrada cuando hay un desequilibrio estructural de los recursos. Por ejemplo, si una línea de comunicación de gigabits está conectada a una PC de bajo rendimiento, el pobre host no será capaz de procesar los paquetes de entrada con la suficiente rapidez y se perderán algunos. Tarde o temprano se retransmitirán estos paquetes; lo cual generará un retardo adicional, un desperdicio del ancho de banda y, en general, una reducción en el desempeño.

Las sobrecargas también se pueden desencadenar en forma sincrónica. Por ejemplo, si un segmento contiene un parámetro erróneo (por ejemplo, el puerto al que está destinado), en muchos casos el receptor