CAPÍTULO 3

La capa de transporte

Entre las capas de aplicación y de red se encuentra la capa de transporte, una pieza fundamental de la arquitectura de red en capas. Desempeña el papel crítico de proporcionar directamente servicios de comunicación a los procesos de aplicación que se ejecutan en hosts diferentes. El método didáctico que vamos a aplicar a lo largo de este capítulo va a consistir en alternar las explicaciones sobre los principios de la capa de transporte con explicaciones acerca de cómo estos principios se implementan en los protocolos existentes; como siempre, haremos un especial hincapié en los protocolos de Internet, en particular en los protocolos de transporte TCP y UDP.

Comenzaremos explicando la relación existente entre las capas de transporte y de red. Para ello, examinaremos la primera función crítica de la capa de transporte: ampliar el servicio de entrega de la capa de red entre dos sistemas terminales a un servicio de entrega entre dos procesos de la capa de aplicación que se ejecutan en los sistemas terminales. Ilustraremos esta función con el protocolo de transporte sin conexión de Internet, UDP.

A continuación, volveremos a los principios y afrontaremos uno de los problemas más importantes de las redes de computadoras: cómo dos entidades pueden comunicarse de forma fiable a través de un medio que puede perder o corromper los datos. A través de una serie de escenarios cada vez más complejos (¡y realistas!), construiremos un conjunto de técnicas que estos protocolos de transporte emplean para resolver este problema. Después, mostraremos cómo esos principios están integrados en TCP, el protocolo de transporte orientado a la conexión de Internet.

Luego pasaremos al segundo problema más importante de las redes: controlar la velocidad de transmisión de las entidades de la capa de transporte con el fin de evitar, o recuperarse de, las congestiones que tienen lugar dentro de la red. Consideraremos las causas y las consecuencias de la congestión, así como las técnicas de control de congestión más comúnmente utilizadas. Una vez que haya entendido los problemas que hay detrás de los mecanismos de control de congestión, estudiaremos el método que aplica TCP para controlar la congestión.

3.1 La capa de transporte y sus servicios

En los dos capítulos anteriores hemos visto por encima cuál es la función de la capa de transporte, así como los servicios que proporciona. Repasemos rápidamente lo que ya sabemos acerca de la capa de transporte.

Un protocolo de la capa de transporte proporciona una **comunicación lógica** entre procesos de aplicación que se ejecutan en hosts diferentes. Por *comunicación lógica* queremos decir que, desde la perspectiva de la aplicación, es como si los hosts que ejecutan los procesos estuvieran conectados directamente; en realidad, los hosts pueden encontrarse en puntos opuestos del planeta, conectados mediante numerosos routers y a través de un amplio rango de tipos de enlace. Los procesos de aplicación utilizan la comunicación lógica proporcionada por la capa de transporte para enviarse mensajes entre sí, sin preocuparse por los detalles de la infraestructura física utilizada para transportar estos mensajes. La Figura 3.1 ilustra el concepto de comunicación lógica.

Como se muestra en la Figura 3.1, los protocolos de la capa de transporte están implementados en los sistemas terminales, pero no en los routers de la red. En el lado emisor, la capa de transporte convierte los mensajes que recibe procedentes de un proceso de aplicación emisor en paquetes de la capa de transporte, conocidos como **segmentos** de la capa de transporte en la terminología de Internet. Muy posiblemente, esto se hace dividiendo los mensajes de la aplicación en fragmentos más pequeños y añadiendo una cabecera de la capa de transporte a cada fragmento, con el fin de crear el segmento de la capa de transporte. A continuación, la capa de transporte pasa el segmento a la capa de red del sistema terminal emisor, donde el segmento se encapsula dentro de un paquete de la capa de red (un datagrama) y se envía al destino. Es importante destacar que los routers de la red solo actúan sobre los campos correspondientes a la capa de red del datagrama; es decir, no examinan los campos del segmento de la capa de transporte encapsulado en el datagrama. En el lado receptor, la capa de red extrae el segmento de la capa de transporte del datagrama y lo sube a la capa de transporte. A continuación, esta capa procesa el segmento recibido, poniendo los datos del segmento a disposición de la aplicación de recepción.

Para las aplicaciones de red puede haber más de un protocolo de la capa de transporte disponible. Por ejemplo, Internet tiene dos protocolos: TCP y UDP. Cada uno de estos protocolos proporciona un conjunto diferente de servicios para la capa de transporte a la aplicación que lo haya invocado.

3.1.1 Relaciones entre las capas de transporte y de red

Recuerde que la capa de transporte se encuentra justo encima de la capa de red dentro de la pila de protocolos. Mientras que un protocolo de la capa de transporte proporciona una comunicación lógica entre *procesos* que se ejecutan en hosts diferentes, un protocolo de la capa de red proporciona una comunicación lógica entre *hosts*. Esta distinción es sutil, pero importante. Examinemos esta distinción con la ayuda de una analogía.

Considere dos viviendas, una situada en la costa este de Estados Unidos y otra en la costa oeste, y que en cada hogar viven una docena de niños. Los niños de la costa este son primos de los niños de la costa oeste. A todos los niños les gusta escribirse, y cada niño escribe a todos sus primos todas las semanas, enviando una carta a través del servicio postal ordinario para cada uno de ellos y empleando un sobre para cada uno. Así, cada casa envía 144 cartas a la otra casa cada semana (estos niños podrían ahorrar mucho dinero si utilizaran el correo electrónico). En cada uno de los hogares, hay un niño (Ann en la costa oeste y Bill en la costa este) responsable de recoger y distribuir el correo. Todas las semanas, Ann visita a sus hermanos y hermanas, les recoge el correo y lo entrega a la persona del servicio postal que pasa a diario por su casa. Cuando las cartas llegan al hogar de la costa oeste, Ann también se ocupa de distribuir el correo al resto de sus hermanos y hermanas. Bill hace el mismo trabajo que Ann en la costa este.

En este ejemplo, el servicio postal proporciona una comunicación lógica entre las dos casas (el servicio postal lleva el correo de una casa a la otra, no de una persona a otra). Por otro lado, Ann y Bill proporcionan una comunicación lógica entre los primos (recogen el correo y se lo entregan a sus

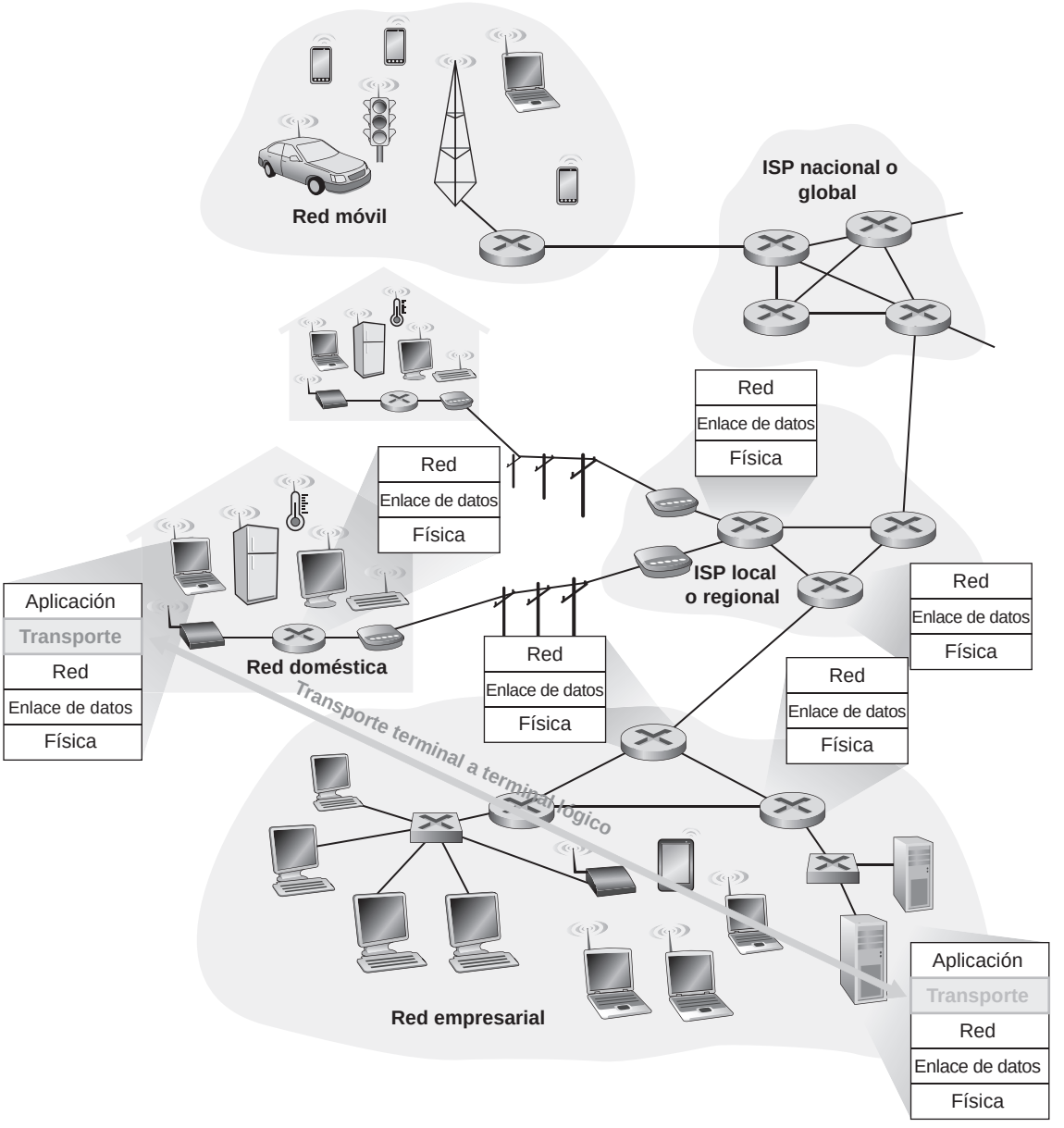


Figura 3.1 ♦ La capa de transporte proporciona una comunicación lógica en lugar de física entre los procesos de aplicación.

hermanos). Observe que desde la perspectiva de los primos, Ann y Bill *son* el servicio de correo, aun siendo ambos solamente una parte (el sistema terminal) del proceso de entrega terminal a terminal. Este ejemplo doméstico es una sencilla analogía que nos permite explicar cómo se relaciona la capa de transporte con la capa de red:

mensajes de la aplicación = las cartas introducidas en los sobres
procesos = los primos
hosts (también denominados sistemas terminales) = las casas
protocolo de la capa de transporte = Ann y Bill
protocolo de la capa de red = el servicio postal (incluyendo a los carteros)

Continuando con esta analogía, fíjese en que Ann y Bill hacen su trabajo dentro de sus respectivas casas; es decir, no están implicados, por ejemplo, en el proceso de ordenación del correo en una oficina de correos intermedia, ni tampoco trasladan el correo de una oficina a otra. De manera similar, los protocolos de la capa de transporte residen en los sistemas terminales. Dentro de un sistema terminal, el protocolo de transporte lleva los mensajes desde los procesos de la aplicación a la frontera de la red (es decir, a la capa de red) y viceversa, pero no tiene nada que ver con cómo se transmiten los mensajes dentro del núcleo de la red. De hecho, como se ilustra en la Figura 3.1, los routers intermedios ni actúan sobre la información que la capa de transporte pueda añadir a los mensajes de la aplicación ni tampoco la reconocen.

Suponga ahora que Ann y Bill se van de vacaciones y que otra pareja de primos, por ejemplo, Susan y Harvey, les sustituyen y son los que recogen y reparten el correo en sus respectivas casas. Lamentablemente para las dos familias, Susan y Harvey no recogen y reparten el correo de la misma forma que lo hacían Ann y Bill. Al ser más pequeños, Susan y Harvey recogen y entregan el correo menos frecuentemente y, ocasionalmente, pierden algunas cartas (que a veces se come el perro). Por tanto, la pareja de primos Susan y Harvey no proporcionan el mismo conjunto de servicios (es decir, el mismo modelo de servicio) que Ann y Bill. De forma análoga, una red de computadoras puede emplear distintos protocolos de transporte, ofreciendo cada uno de ellos un modelo de servicio distinto a las aplicaciones.

Los posibles servicios que Ann y Bill pueden proporcionar están evidentemente restringidos por los posibles servicios que el servicio postal suministra. Por ejemplo, si el servicio postal no especifica el tiempo máximo que se puede tardar en entregar el correo entre ambos hogares (por ejemplo, tres días), entonces Ann y Bill no tienen ninguna forma de garantizar un retardo máximo en la entrega del correo entre cualquier pareja de primos. Del mismo modo, los servicios que un protocolo de transporte puede proporcionar a menudo están restringidos por el modelo de servicio del protocolo de la capa de red subyacente. Si el protocolo de la capa de red no proporciona garantías acerca del retardo ni del ancho de banda para los segmentos de la capa de transporte enviados entre hosts, entonces el protocolo de la capa de transporte no puede proporcionar ninguna garantía acerca del retardo o del ancho de banda a los mensajes de aplicación enviados entre procesos.

No obstante, el protocolo de transporte *puede* ofrecer ciertos servicios incluso cuando el protocolo de red subyacente no ofrezca el servicio correspondiente en la capa de red. Por ejemplo, como veremos más adelante en el capítulo, un protocolo de transporte puede ofrecer un servicio de transferencia de datos fiable a una aplicación, incluso si el protocolo de red subyacente no es fiable; es decir, incluso si el protocolo de red pierde, altera o duplica paquetes. Otro ejemplo, que examinaremos en el Capítulo 8 cuando hablemos de la seguridad de la red, es que un protocolo de transporte puede emplear mecanismos de cifrado para garantizar que los mensajes de una aplicación no sean leídos por intrusos, incluso aunque la capa de red no pueda garantizar la privacidad de los segmentos de la capa de transporte.

3.1.2 La capa de transporte en Internet

Recuerde que Internet pone a disposición de la capa de aplicación dos protocolos de la capa de transporte diferentes. Uno de estos protocolos es el Protocolo de datagramas de usuario (**UDP**, *User Datagram Protocol*), que proporciona un servicio sin conexión no fiable a la aplicación que le invoca. El segundo de estos protocolos es el Protocolo de control de transmisión (**TCP**, *Transmission Control Protocol*), que proporciona a la aplicación que le invoca un servicio orientado a la conexión fiable. Al diseñar una aplicación de red, el desarrollador tiene que especificar el uso de uno de estos dos protocolos de transporte. Como hemos visto en las Sección 2.7, el desarrollador de la aplicación elige entre UDP y TCP cuando crea los sockets.

Para simplificar la terminología, nos referiremos a los paquetes de la capa de transporte como *segmentos*. No obstante, tenemos que decir que, en textos dedicados a Internet, como por ejemplo los RFC, también se emplea el término segmento para hacer referencia a los paquetes de la capa de transporte en el caso de TCP, pero a menudo a los paquetes de UDP se les denomina datagrama.

Pero resulta que estos mismos textos dedicados a Internet también utilizan el término *datagrama* para referirse a los paquetes de la capa de red. Por tanto, pensamos que en un libro de introducción a las redes de computadoras como este, resultará menos confuso hablar de segmentos tanto para referirse a los paquetes TCP como UDP, y reservar el término *datagrama* para los paquetes de la capa de red.

Antes de continuar con esta breve introducción a los protocolos UDP y TCP, nos será útil comentar algunas cosas acerca de la capa de red de Internet (en los Capítulos 4 y 5 examinaremos en detalle la capa de red). El protocolo de la capa de red de Internet es el protocolo IP (*Internet Protocol*). IP proporciona una comunicación lógica entre hosts. El modelo de servicio de IP es un **servicio de entrega de mejor esfuerzo (*best effort*)**. Esto quiere decir que IP hace todo lo que puede por entregar los segmentos entre los hosts que se están comunicando, *pero no garantiza la entrega*. En particular, no garantiza la entrega de los segmentos, no garantiza que los segmentos se entreguen en orden y no garantiza la integridad de los datos contenidos en los segmentos. Por estas razones, se dice que IP es un **servicio no fiable**. Además, sabemos que todos los hosts tienen al menos una dirección de capa de red, que se conoce como dirección IP. En el Capítulo 4 se estudia en detalle el direccionamiento IP, pero por el momento lo único que necesitamos saber es que *todo host tiene una dirección IP asociada*.

Después de haber echado una ojeada al modelo de servicio de IP, haremos un breve resumen de los modelos de servicio proporcionados por UDP y TCP. La responsabilidad principal de UDP y TCP es ampliar el servicio de entrega de IP entre dos sistemas terminales a un servicio de entrega entre dos procesos que estén ejecutándose en los sistemas terminales. Extender la entrega host a host a una entrega proceso a proceso es lo que se denomina **multiplexación** y **demultiplexación de la capa de transporte**, tema que desarrollaremos en la siguiente sección. UDP y TCP también proporcionan servicios de comprobación de la integridad de los datos al incluir campos de detección de errores en las cabeceras de sus segmentos. Estos dos servicios de la capa de transporte (entrega de datos proceso a proceso y comprobación de errores) son los dos únicos servicios que ofrece UDP. En particular, al igual que IP, UDP es un servicio no fiable, que no garantiza que los datos enviados por un proceso lleguen intactos (¡o que ni siquiera lleguen!) al proceso de destino. En la Sección 3.3 se aborda en detalle el protocolo UDP.

TCP, por el contrario, ofrece a las aplicaciones varios servicios adicionales. El primero y más importante es que proporciona una **transferencia de datos fiable**. Utilizando técnicas de control de flujo, números de secuencia, mensajes de reconocimiento y temporizadores (técnicas que estudiaremos en detalle en este capítulo), TCP garantiza que los datos transmitidos por el proceso emisor sean entregados al proceso receptor, correctamente y en orden. De este modo, TCP convierte el servicio no fiable de IP entre sistemas terminales en un servicio de transporte de datos fiable entre procesos. TCP también proporciona mecanismos de **control de congestión**. El control de congestión no es tanto un servicio proporcionado a la aplicación invocante, cuanto un servicio que se presta a Internet como un todo, un servicio para el bien común. En otras palabras, los mecanismos de control de congestión de TCP evitan que cualquier conexión TCP inunde con una cantidad de tráfico excesiva los enlaces y routers existentes entre los hosts que están comunicándose. TCP se esfuerza en proporcionar a cada conexión que atraviesa un enlace congestionado la misma cuota de ancho de banda del enlace. Esto se consigue regulando la velocidad a la que los lados emisores de las conexiones TCP pueden enviar tráfico a la red. El tráfico UDP, por el contrario, no está regulado. Una aplicación que emplee el protocolo de transporte UDP puede enviar los datos a la velocidad que le parezca, durante todo el tiempo que quiera.

Un protocolo que proporciona una transferencia de datos fiable y mecanismos de control de congestión necesariamente es un protocolo complejo. Vamos a necesitar varias secciones de este capítulo para examinar los principios de la transferencia de datos fiable y el control de congestión, además de otras secciones adicionales dedicadas a cubrir el propio protocolo TCP. Estos temas se tratan en las Secciones 3.4 a 3.8. El método de trabajo que hemos adoptado en este capítulo es el de alternar entre los principios básicos y el protocolo TCP. Por ejemplo, en primer lugar veremos en qué consiste en general una transferencia de datos fiable y luego veremos específicamente cómo TCP

proporciona ese servicio de transferencia de datos fiable. De forma similar, primero definiremos de forma general en qué consiste el mecanismo de control de congestión y luego veremos cómo lo hace TCP. Pero antes de entrar en estos temas, veamos qué es la multiplexación y la demultiplexación de la capa de transporte.

3.2 Multiplexación y demultiplexación

En esta sección vamos a estudiar la multiplexación y demultiplexación de la capa de transporte; es decir, la ampliación del servicio de entrega host a host proporcionado por la capa de red a un servicio de entrega proceso a proceso para las aplicaciones que se ejecutan en los hosts. Con el fin de centrar la explicación, vamos a ver este servicio básico de la capa de transporte en el contexto de Internet. Sin embargo, hay que destacar que un servicio de multiplexación/demultiplexación es necesario en todas las redes de computadoras.

En el host de destino, la capa de transporte recibe segmentos procedentes de la capa de red que tiene justo debajo. La capa de transporte tiene la responsabilidad de entregar los datos contenidos en estos segmentos al proceso de la aplicación apropiada que está ejecutándose en el host. Veamos un ejemplo. Suponga que está sentado frente a su computadora y que está descargando páginas web a la vez que ejecuta una sesión FTP y dos sesiones Telnet. Por tanto, tiene cuatro procesos de aplicación de red en ejecución: dos procesos Telnet, un proceso FTP y un proceso HTTP. Cuando la capa de transporte de su computadora recibe datos procedentes de la capa de red, tiene que dirigir los datos recibidos a uno de estos cuatro procesos. Veamos cómo hace esto.

En primer lugar, recordemos de la Sección 2.7 que un proceso (como parte de una aplicación de red) puede tener uno o más **sockets**, puertas por las que pasan los datos de la red al proceso, y viceversa. Por tanto, como se muestra en la Figura 3.2, la capa de transporte del host receptor realmente no entrega los datos directamente a un proceso, sino a un socket intermedio. Dado que en cualquier instante puede haber más de un socket en el host receptor, cada socket tiene asociado un identificador único. El formato de este identificador depende de si se trata de un socket UDP o de un socket TCP, como vamos a ver a continuación.

Veamos ahora cómo un host receptor dirige un segmento de entrada de la capa de transporte al socket apropiado. Cada segmento de la capa de transporte contiene un conjunto de campos destinados a este propósito. En el extremo receptor, la capa de transporte examina estos campos para identificar el socket receptor y, a continuación, envía el segmento a dicho socket. Esta tarea

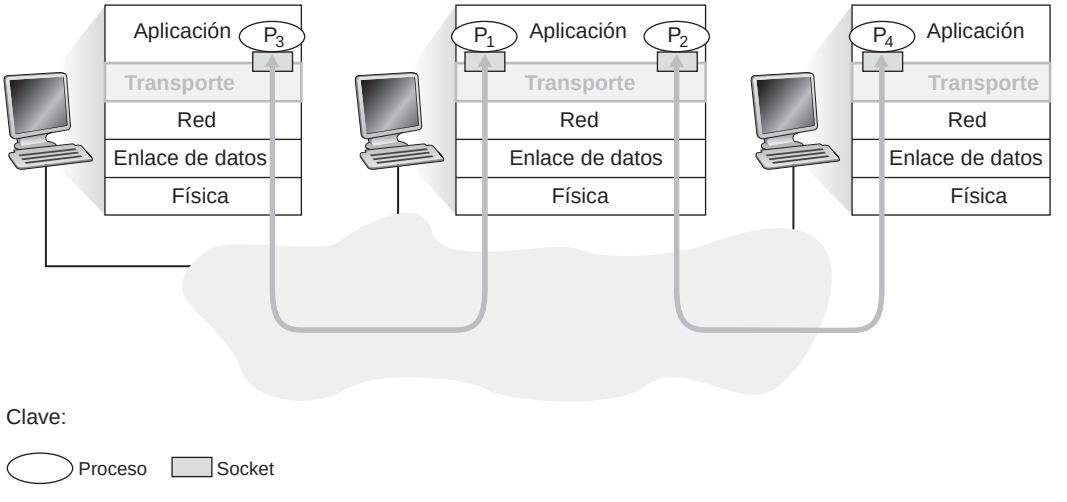


Figura 3.2 ♦ Multiplexación y demultiplexación en la capa de transporte.

de entregar los datos contenidos en un segmento de la capa de transporte al socket correcto es lo que se denomina **demultiplexación**. La tarea de reunir los fragmentos de datos en el host de origen desde los diferentes sockets, encapsulando cada fragmento de datos con la información de cabecera (la cual se utilizará después en el proceso de demultiplexación) para crear los segmentos y pasarlos a la capa de red es lo que se denomina **multiplexación**. Observe que la capa de transporte del host intermedio de la Figura 3.2 tiene que demultiplexar los segmentos que llegan de la capa de red inferior para entregárselos a los procesos P_1 o P_2 de la capa superior; esto se hace dirigiendo los datos del segmento entrante al correspondiente socket del proceso. La capa de transporte del host intermedio también tiene que recopilar los datos salientes desde estos sockets, construir los segmentos de la capa de transporte y pasar dichos segmentos a la capa de red. Aunque hemos presentado la multiplexación y la demultiplexación en el contexto de los protocolos de transporte de Internet, es importante darse cuenta de que esas técnicas son necesarias siempre que un único protocolo en una capa (en la capa de transporte o cualquier otra) sea utilizado por varios protocolos de la capa inmediatamente superior.

Para ilustrar la tarea de demultiplexación, recordemos la analogía doméstica de la sección anterior. Cada uno de los niños está identificado por su nombre. Cuando Bill recibe un lote de cartas del servicio de correos, lleva a cabo una operación de demultiplexación al determinar a quién van dirigidas las cartas y luego las entrega en mano a sus hermanos. Ann lleva a cabo una operación de multiplexación al recopilar las cartas de sus hermanos y entregar todas las cartas al cartero.

Ahora que ya entendemos las funciones de multiplexación y demultiplexación de la capa de transporte, vamos a examinar cómo se hace esto realmente en un host. Basándonos en las explicaciones anteriores, sabemos que la operación de multiplexación que se lleva a cabo en la capa de transporte requiere (1) que los sockets tengan identificadores únicos y (2) que cada segmento tenga campos especiales que indiquen el socket al que tiene que entregarse el segmento. Estos campos especiales, mostrados en la Figura 3.3, son el campo **número de puerto de origen** y el **campo número de puerto de destino**. (Los segmentos UDP y TCP contienen además otros campos, como veremos en las siguientes secciones del capítulo.) Cada número de puerto es un número de 16 bits comprendido en el rango de 0 a 65535. Los números de puerto pertenecientes al rango de 0 a 1023 se conocen como **números de puertos bien conocidos** y son restringidos, lo que significa que están reservados para ser empleados por los protocolos de aplicación bien conocidos, como por ejemplo HTTP (que utiliza el número de puerto 80) y FTP (que utiliza el número de puerto 21). Puede encontrar la lista de números de puerto bien conocidos en el documento RFC 1700 y su actualización en la dirección http://www.iana.org [RFC 3232]. Al desarrollar una nueva aplicación (como las aplicaciones desarrolladas en la Sección 2.7), es necesario asignar un número de puerto a la aplicación.

Ahora ya debería estar claro cómo la capa de transporte *podría* implementar el servicio de demultiplexación: cada socket del host se puede asignar a un número de puerto y, al llegar un

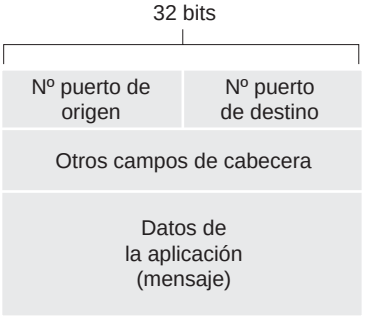


Figura 3.3 ♦ Los campos número de puerto de origen y de destino en un segmento de la capa de transporte.

segmento al host, la capa de transporte examina el número de puerto de destino contenido en el segmento y lo envía al socket correspondiente. A continuación, los datos del segmento pasan a través del socket y se entregan al proceso asociado. Como veremos, esto es básicamente lo que hace UDP. Sin embargo, también comprobaremos que la tarea de multiplexación/demultiplexación en TCP es más sutil.

Multiplexación y demultiplexación sin conexión

Recordemos de la Sección 2.7.1 que un programa Python que se ejecuta en un host puede crear un socket UDP mediante la línea de código:

```
clientSocket = socket(AF_INET, SOCK_DGRAM)
```

Cuando se crea un socket UDP de este modo, la capa de transporte asigna automáticamente un número de puerto al socket. En particular, la capa de transporte asigna un número de puerto comprendido en el rango de 1024 a 65535 que actualmente no esté siendo utilizado en ese host por ningún otro puerto UDP. Alternativamente, podemos añadir una línea a nuestro programa Python después de crear el socket para asociar un número de puerto específico (por ejemplo, 19157) a este socket UDP mediante el método `bind()` del socket:

```
clientSocket.bind(('', 19157))
```

Si el desarrollador de la aplicación que escribe el código estuviera implementando el lado del servidor de un "protocolo bien conocido", entonces tendría que asignar el correspondiente número de puerto bien conocido. Normalmente, el lado del cliente de la aplicación permite a la capa de transporte asignar de forma automática (y transparente) el número de puerto, mientras que el lado de servidor de la aplicación asigna un número de puerto específico.

Una vez asignados los números de puerto a los sockets UDP, podemos describir de forma precisa las tareas de multiplexación y demultiplexación en UDP. Suponga que un proceso del host A, con el puerto UDP 19157, desea enviar un fragmento de datos de una aplicación a un proceso con el puerto UDP 46428 en el host B. La capa de transporte del host A crea un segmento de la capa de transporte que incluye los datos de aplicación, el número de puerto de origen (19157), el número de puerto de destino (46428) y otros dos valores (que veremos más adelante, pero que por el momento no son importantes para el tema que nos ocupa). La capa de transporte pasa a continuación el segmento resultante a la capa de red. La capa de red encapsula el segmento en un datagrama IP y hace el máximo esfuerzo por entregar el segmento al host receptor. Si el segmento llega al host receptor B, la capa de transporte del mismo examina el número de puerto de destino especificado en el segmento (46428) y entrega el segmento a su socket identificado por el puerto 46428. Observe que el host B podría estar ejecutando varios procesos, cada uno de ellos con su propio socket UDP y número de puerto asociado. A medida que los segmentos UDP llegan de la red, el host B dirige (demultiplexa) cada segmento al socket apropiado examinando el número de puerto de destino del segmento.

Es importante observar que un socket UDP queda completamente identificado por una tupla que consta de una dirección IP de destino y un número de puerto de destino. En consecuencia, si dos segmentos UDP tienen diferentes direcciones IP y/o números de puerto de origen, pero la misma dirección IP de *destino* y el mismo número puerto de *destino*, entonces los dos segmentos se enviarán al mismo proceso de destino a través del mismo socket de destino.

Es posible que se esté preguntando en este momento cuál es el propósito del número de puerto de origen. Como se muestra en la Figura 3.4, en el segmento A a B, el número de puerto de origen forma parte de una "dirección de retorno"; es decir, si B desea devolver un segmento a A, el puerto de destino en el segmento de B a A tomará su valor del valor del puerto de origen del segmento de A a B. (La dirección de retorno completa es el número de puerto de origen y la dirección IP de A.) Por ejemplo, recuerde el programa de servidor UDP estudiado en la Sección 2.7. En `UDPServer.py`, el servidor utiliza el método `recvfrom()`para extraer el número de puerto (de origen) del lado

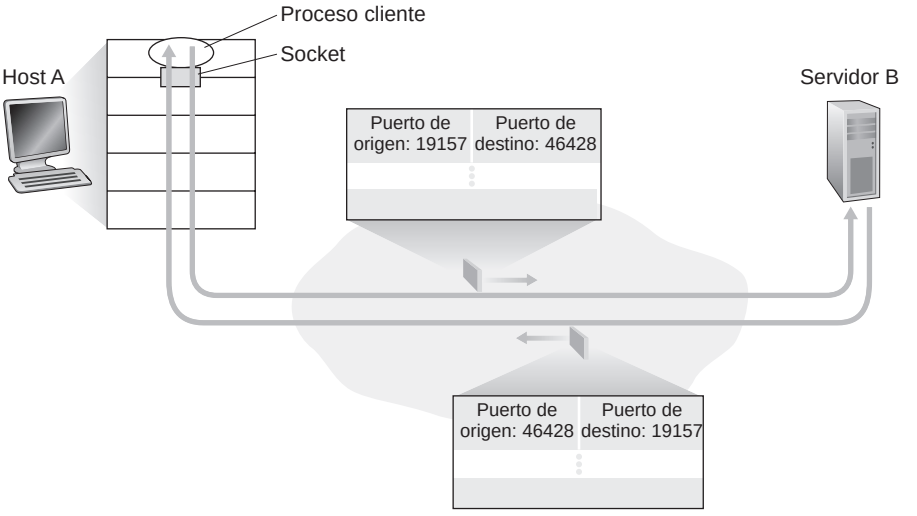


Figura 3.4 ♦ Inversión de los números de puerto de origen y de destino.

del cliente del segmento que recibe desde el cliente; a continuación, envía un segmento nuevo al cliente, utilizando como número de puerto de destino el número de puerto de origen extraido del segmento recibido.

Multiplexación y demultiplexación orientadas a la conexión

Para entender la demultiplexación TCP, tenemos que tener en cuenta los sockets TCP y el establecimiento de las conexiones TCP. Una sutil diferencia entre un socket TCP y un socket UDP es que el primero queda identificado por una tupla de cuatro elementos: dirección IP de origen, número de puerto de origen, dirección IP de destino, número de puerto de destino. Por tanto, cuando un segmento TCP llega a un host procedente de la red, el host emplea los cuatro valores para dirigir (demultiplexar) el segmento al socket apropiado. En particular, y al contrario de lo que ocurre con UDP, dos segmentos TCP entrantes con direcciones IP de origen o números de puerto de origen diferentes (con la excepción de un segmento TCP que transporte la solicitud original de establecimiento de conexión) serán dirigidos a dos sockets distintos. Con el fin de profundizar un poco, consideremos de nuevo el ejemplo de programación cliente-servidor TCP de la Sección 2.7.2:

- La aplicación de servidor TCP tiene un "socket de acogida", que está a la espera de las solicitudes de establecimiento de conexión procedentes de los clientes TCP en el puerto 12000 (véase la Figura 2.29).
- El cliente TCP crea un socket y envía un segmento de establecimiento de conexión con la líneas de código:

```
clientSocket = socket(AF_INET, SOCK_STREAM)
               clientSocket.connect((serverName,12000))
```

- Una solicitud de establecimiento de conexión no es nada más que un segmento TCP con el número de puerto de destino 12000 y un conjunto especial de bits de establecimiento de conexión en la cabecera TCP (que veremos en la Sección 3.5). El segmento también incluye un número de puerto de origen, que habrá sido seleccionado por el cliente.
- Cuando el sistema operativo del host que está ejecutando el proceso servidor recibe el segmento de entrada de solicitud de conexión con el puerto de destino 12000, localiza el proceso de

servidor que está esperando para aceptar una conexión en el número de puerto 12000. El proceso de servidor crea entonces un nuevo socket:

```
connectionSocket, addr = serverSocket.accept()
```

- Además, la capa de transporte en el servidor toma nota de los cuatro valores siguientes contenidos en el segmento de solicitud de conexión: (1) el número de puerto de origen en el segmento, (2) la dirección IP del host de origen, (3) el número de puerto de destino en el segmento y (4) su propia dirección IP. El socket de conexión recién creado queda identificado por estos cuatro valores; así, todos los segmentos que lleguen después y cuyo puerto de origen, dirección IP de origen, puerto de destino y dirección IP de destino se correspondan con estos cuatro valores serán enviados a este socket. Una vez establecida la conexión TCP, el cliente y el servidor podrán enviarse datos entre sí.

El host servidor puede dar soporte a muchos sockets TCP simultáneos, estando cada socket asociado a un proceso y con cada socket identificado por su tupla de cuatro elementos. Cuando un segmento TCP llega al host, los cuatro campos (dirección IP de origen, puerto de origen, dirección IP de destino y puerto de destino) se utilizan para dirigir (demultiplexar) el segmento al socket apropiado.

Esta situación se ilustra en la Figura 3.5, en la que el host C inicia dos sesiones HTTP con el servidor B y el host A inicia una sesión HTTP también con B. Los hosts A y C y el servidor B tienen sus propias direcciones IP únicas (A, C y B, respectivamente). El host C asigna dos números de puerto de origen diferentes (26145 y 7532) a sus dos conexiones HTTP. Dado que el host A está seleccionando los números de puerto de origen independientemente de C, también puede asignar el número de puerto de origen 26145 a su conexión HTTP. Pero esto no es un problema: el servidor B todavía podrá demultiplexar correctamente las dos conexiones con el mismo número de puerto de origen, ya que tienen direcciones IP de origen diferentes.

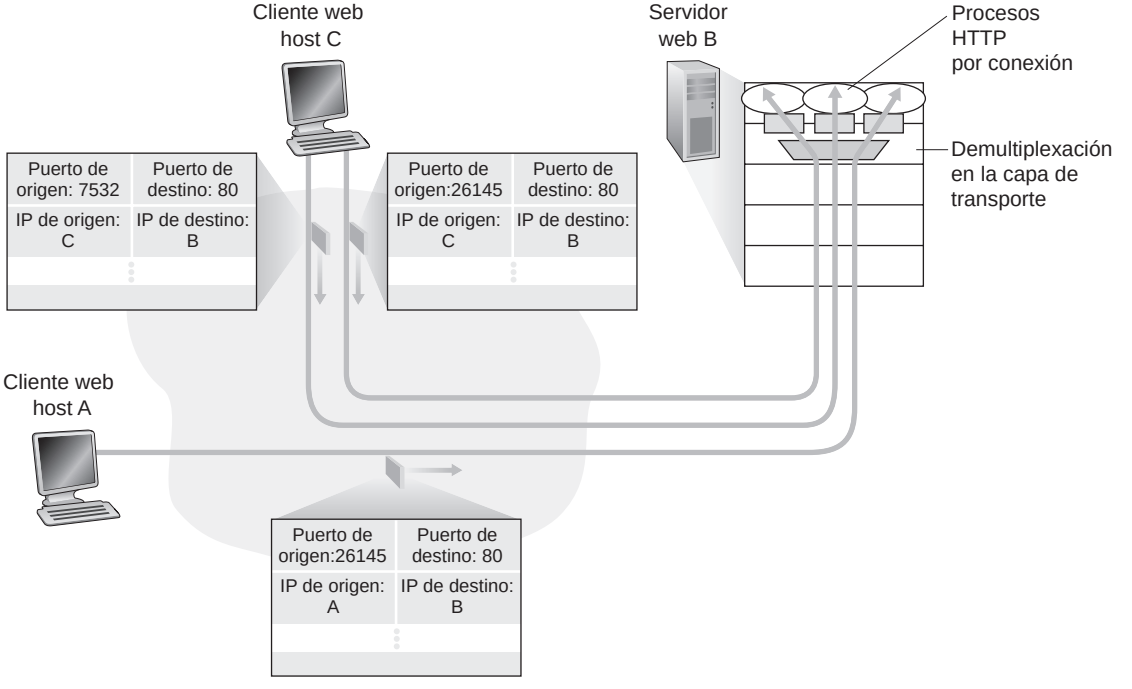


Figura 3.5 ♦ Dos clientes utilizando el mismo número de puerto de destino (80) para comunicarse con la misma aplicación de servidor web.

SEGURIDAD

EXPLORACIÓN DE PUERTOS

Hemos visto que un proceso servidor espera pacientemente en un puerto abierto a ser contactado por un cliente remoto. Algunos puertos están reservados para aplicaciones bien conocidas (como por ejemplo servidores web, FTP, DNS y SMTP); otros puertos, por convenio, son utilizados por aplicaciones populares (como por ejemplo Microsoft 2000 SQL Server que está a la escucha en el puerto UDP 1434). Por tanto, si determinamos que un puerto está abierto en un host, podemos ser capaces de hacer corresponder dicho puerto a una aplicación específica que se ejecute en el host. Esto es muy útil para los administradores de sistemas, ya que suelen estar interesados en saber qué aplicaciones de red están ejecutándose en los hosts de sus redes. Pero los atacantes, con el fin de detectar "las brechas en el muro", también desean saber qué puertos están abiertos en los hosts objetivo. Si se localiza un host que está ejecutando una aplicación con un defecto de seguridad conocido (por ejemplo, si un servidor SQL que está a la escucha en el puerto 1434 fuera objeto de un desbordamiento de buffer, permitiendo a un usuario remoto ejecutar código arbitrario en el host vulnerable, un defecto explotado por el gusano Slammer [CERT 2003-04]), entonces dicho host estaría en condiciones para recibir un ataque.

Determinar qué aplicaciones están a la escucha en qué puertos es una tarea relativamente fácil. Además, existen numerosos programas de dominio público, conocidos como escáneres de puertos, que realizan este trabajo. Quizá el más ampliamente utilizado de estos programas es nmap, que está disponible gratuitamente en http://nmap.org y se incluye en la mayor parte de las distribuciones de Linux. Para TCP, nmap explora secuencialmente los puertos buscando puertos que acepten conexiones TCP. Para UDP, nmap también explora secuencialmente los puertos buscando puertos UDP que respondan a segmentos UDP transmitidos. En ambos casos, nmap devuelve una lista de puertos abiertos, cerrados o inalcanzables. Un host que ejecute nmap puede tratar de explorar cualquier host objetivo situado *en cualquier lugar* de Internet. En la Sección 3.5.6 volveremos a hablar de nmap, al tratar la gestión de las conexiones TCP.

Servidores web y TCP

Antes de dar por terminada esta sección, es interesante comentar algunas cosas acerca de los servidores web y de cómo utilizan los números de puerto. Considere un host que está ejecutando un servidor web, como por ejemplo un servidor web Apache en el puerto 80. Cuando los clientes (por ejemplo, los navegadores) envían segmentos al servidor, *todos* los segmentos tendrán el puerto de destino 80. En particular, tanto los segmentos para el establecimiento de la conexión inicial como los segmentos que transportan los mensajes de solicitud HTTP utilizarán como puerto de destino el puerto 80. Como acabamos de describir, el servidor diferencia los segmentos procedentes de los distintos clientes mediante las direcciones IP de origen y los números de puerto de origen.

La Figura 3.5 nos muestra un servidor web que genera un nuevo proceso para cada conexión. Como se muestra en esta figura, cada uno de estos procesos tiene su propio socket de conexión a través del cual llegan las solicitudes HTTP y se envían las respuestas HTTP. Sin embargo, también hemos mencionado que no existe siempre una correspondencia uno a uno entre los sockets de conexión y los procesos. De hecho, los servidores web actuales de altas prestaciones a menudo solo utilizan un proceso y crean una nueva hebra con un nuevo socket de conexión para cada nueva conexión de un cliente (una hebra es una especie de subproceso de baja complejidad). Si realizó la primera tarea de programación del Capítulo 2, habrá creado un servidor web que hace exactamente esto. En un servidor así, en cualquier instante puede haber muchos sockets de conexión (con distintos identificadores) asociados al mismo proceso.

Si el cliente y el servidor están utilizando HTTP persistente, entonces mientras dure la conexión persistente el cliente y el servidor intercambiarán mensajes HTTP a través del mismo socket de

servidor. Sin embargo, si el cliente y el servidor emplean HTTP no persistente, entonces se creará y cerrará una nueva conexión TCP para cada pareja solicitud/respuesta y, por tanto, se creará (y luego se cerrará) un nuevo socket para cada solicitud/respuesta. Esta creación y cierre de sockets tan frecuente puede afectar seriamente al rendimiento de un servidor web ocupado (aunque se pueden utilizar una serie de trucos del sistema operativo para mitigar el problema). Los lectores interesados en los problemas que el uso de HTTP persistente y no persistente plantea a los sistemas operativos pueden consultar [Nielsen 1997; Nahum 2002].

Ahora que ya hemos visto en qué consiste la multiplexación y la demultiplexación en la capa de transporte, podemos pasar a tratar uno de los protocolos de transporte de Internet: UDP. En la siguiente sección veremos que UDP añade algo más al protocolo de la capa de red que un servicio de multiplexación/demultiplexación.

3.3 Transporte sin conexión: UDP

En esta sección vamos a ocuparnos de UDP, vamos a ver cómo funciona y qué hace. Le animamos a releer la Sección 2.1, en la que se incluye una introducción al modelo de servicio de UDP y la Sección 2.7.1, en la que se explica la programación de sockets con UDP.

Para iniciar nuestro análisis de UDP, suponga que queremos diseñar un protocolo de transporte simple y poco sofisticado. ¿Cómo haríamos esto? En primer lugar, podemos considerar la utilización de un protocolo de transporte vacuo. Es decir, en el lado emisor, tomará los mensajes del proceso de la aplicación y los pasará directamente a la capa de red; en el lado de recepción, tomará los mensajes procedentes de la capa de red y los pasará directamente al proceso de la aplicación. Pero, como hemos aprendido en la sección anterior, hay algunas cosas mínimas que tenemos que hacer. Como mínimo, la capa de transporte tiene que proporcionar un servicio de multiplexación/demultiplexación que permita transferir los datos entre la capa de red y el proceso de la capa de aplicación correcto.

UDP, definido en el documento [RFC 768], hace casi lo mínimo que un protocolo de transporte debe hacer. Además de la función de multiplexación/demultiplexación y de algún mecanismo de comprobación de errores, no añade nada a IP. De hecho, si el desarrollador de la aplicación elige UDP en lugar de TCP, entonces prácticamente es la aplicación la que se comunica directamente con IP. UDP toma los mensajes procedentes del proceso de la aplicación, asocia los campos correspondientes a los números de puerto de origen y de destino para proporcionar el servicio de multiplexación/demultiplexación, añade dos campos pequeños más y pasa el segmento resultante a la capa de red. La capa de red encapsula el segmento de la capa de transporte en un datagrama IP y luego hace el mejor esfuerzo por entregar el segmento al host receptor. Si el segmento llega al host receptor, UDP utiliza el número de puerto de destino para entregar los datos del segmento al proceso apropiado de la capa de aplicación. Observe que con UDP no tiene lugar una fase de establecimiento de la conexión entre las entidades de la capa de transporte emisora y receptora previa al envío del segmento. Por esto, se dice que UDP es un protocolo *sin conexión.*

DNS es un ejemplo de un protocolo de la capa de aplicación que habitualmente utiliza UDP. Cuando la aplicación DNS de un host desea realizar una consulta, construye un mensaje de consulta DNS y lo pasa a UDP. Sin llevar a cabo ningún proceso para establecer una conexión con la entidad UDP que se ejecuta en el sistema terminal de destino, el protocolo UDP del lado del host añade campos de cabecera al mensaje y pasa el segmento resultante a la capa de red, la cual encapsula el segmento UDP en un datagrama y lo envía a un servidor de nombres. La aplicación DNS que se ejecuta en el host que ha hecho la consulta espera entonces hasta recibir una respuesta a su consulta. Si no la recibe (posiblemente porque la red subyacente ha perdido la consulta o la respuesta), bien intenta enviar la consulta a otro servidor de nombres o bien informa a la aplicación invocante de que no puede obtener una respuesta.

Es posible que ahora se esté preguntando por qué un desarrollador de aplicaciones podría decidir crear una aplicación sobre UDP en lugar de sobre TCP. ¿No sería preferible emplear siempre TCP,

puesto que proporciona un servicio de transferencia de datos fiable y UDP no? La respuesta es no, ya que muchas aplicaciones están mejor adaptadas a UDP por las siguientes razones:

- *Mejor control en el nivel de aplicación sobre qué datos se envían y cuándo.* Con UDP, tan pronto como un proceso de la capa de aplicación pasa datos a UDP, UDP los empaqueta en un segmento UDP e inmediatamente entrega el segmento a la capa de red. Por el contrario, TCP dispone de un mecanismo de control de congestión que regula el flujo del emisor TCP de la capa de transporte cuando uno o más de los enlaces existentes entre los hosts de origen y de destino están excesivamente congestionados. TCP también continuará reenviando un segmento hasta que la recepción del mismo haya sido confirmada por el destino, independientemente de cuánto se tarde en llevar a cabo esta entrega fiable. Puesto que las aplicaciones en tiempo real suelen requerir una velocidad mínima de transmisión, no permiten un retardo excesivo en la transmisión de los segmentos y pueden tolerar algunas pérdidas de datos, el modelo de servicio de TCP no se adapta demasiado bien a las necesidades de este tipo de aplicaciones. Como veremos más adelante, estas aplicaciones pueden utilizar UDP e implementar, como parte de la aplicación, cualquier funcionalidad adicional que sea necesaria más allá del servicio básico de entrega de segmentos de UDP.
- *Sin establecimiento de la conexión*. Como se explicará más adelante, TCP lleva a cabo un proceso de establecimiento de la conexión en tres fases antes de iniciar la transferencia de datos. UDP inicia la transmisión sin formalidades preliminares. Por tanto, UDP no añade ningún retardo a causa del establecimiento de una conexión. Probablemente, esta es la razón principal por la que DNS opera sobre UDP y no sobre TCP (DNS sería mucho más lento si se ejecutara sobre TCP). HTTP utiliza TCP en lugar de UDP, ya que la fiabilidad es crítica para las páginas web con texto. Pero, como hemos comentado brevemente en la Sección 2.2, el retardo debido al establecimiento de la conexión TCP en HTTP contribuye de forma notable a los retardos asociados con la descarga de documentos web. De hecho, el protocolo QUIC (*Quick UDP Internet Connection*, [Iyengar 2015]), utilizado en el navegador Chrome de Google, utiliza UDP como su protocolo de transporte subyacente e implementa la fiabilidad en un protocolo de la capa de aplicación por encima de UDP.
- *Sin información del estado de la conexión*. TCP mantiene información acerca del estado de la conexión en los sistemas terminales. En el estado de la conexión se incluye información acerca de los buffers de recepción y envío, de los parámetros de control de congestión y de los parámetros relativos al número de secuencia y de reconocimiento. En la Sección 3.5 veremos que esta información de estado es necesaria para implementar el servicio fiable de transferencia de datos y proporcionar los mecanismos de control de congestión de TCP. Por el contrario, UDP no mantiene información del estado de la conexión y no controla ninguno de estos parámetros. Por esta razón, un servidor dedicado a una aplicación concreta suele poder soportar más clientes activos cuando la aplicación se ejecuta sobre UDP que cuando lo hace sobre TCP.
- *Poca sobrecarga debida a la cabecera de los paquetes*. Los segmentos TCP contienen 20 bytes en la cabecera de cada segmento, mientras que UDP solo requiere 8 bytes.

La tabla de la Figura 3.6 enumera aplicaciones de Internet muy populares junto con los protocolos de transporte que utilizan. Como era de esperar, el correo electrónico, el acceso remoto a terminales, la Web y la transferencia de archivos se ejecutan sobre TCP, ya que todas estas aplicaciones necesitan el servicio fiable de transferencia de datos de TCP. No obstante, muchas aplicaciones importantes se ejecutan sobre UDP en lugar de sobre TCP. Por ejemplo, UDP se utiliza para transmitir los datos de la administración de red (SNMP, véase la Sección 5.7). En este caso, UDP es preferible a TCP, ya que las aplicaciones de administración de la red a menudo se tienen que ejecutar cuando la red se encuentra en un estado de sobrecarga (precisamente en las situaciones en las que es difícil realizar transferencias de datos fiables y con control de la congestión). Además, como ya hemos mencionado anteriormente, DNS se ejecuta sobre UDP, evitando de este modo los retardos de establecimiento de la conexión TCP.

Aplicación	Protocolo de la capa de aplicación	Protocolo de transporte subyacente
Correo electrónico	SMTP	TCP
Acceso a terminales remotos	Telnet	TCP
Web	HTTP	TCP
Transferencia de archivos	FTP	TCP
Servidor de archivos remoto	NFS	Normalmente UDP
Flujos multimedia	Normalmente proprietario	UDP o TCP
Telefonía por Internet	Normalmente proprietario	UDP o TCP
Administración de red	SNMP	Normalmente UDP
Traducción de nombres	DNS	Normalmente UDP

Figura 3.6 ♦ Aplicaciones de Internet populares y sus protocolos de transporte subyacentes.

Como se indica en la Figura 3.6, actualmente tanto UDP como TCP se utilizan con aplicaciones multimedia, como la telefonía por Internet, las videoconferencias en tiempo real y la reproducción de flujos de vídeos y audios almacenados. En el Capítulo 9 nos ocuparemos de estas aplicaciones. Por el momento, basta con mencionar que todas estas aplicaciones toleran la pérdida de una pequeña cantidad de paquetes, por lo que una transferencia de datos fiable no es absolutamente crítica para que la aplicación funcione correctamente. Además, las aplicaciones en tiempo real, como la telefonía por Internet y la videoconferencia, responden muy mal a los mecanismos de control de congestión de TCP. Por estas razones, los desarrolladores de aplicaciones multimedia pueden elegir ejecutar sus aplicaciones sobre UDP y no sobre TCP. Cuando la tasa de pérdidas de paquetes es baja y teniendo en cuenta que algunas organizaciones bloquean el tráfico UDP por razones de seguridad (véase el Capítulo 8), TCP se está convirtiendo en un protocolo cada vez más atractivo para el transporte de flujos multimedia.

Aunque hoy en día es común emplear UDP para ejecutar las aplicaciones multimedia, continúa siendo un tema controvertido. Como ya hemos dicho, UDP no proporciona mecanismos de control de congestión, y estos mecanismos son necesarios para impedir que la red entre en un estado de congestión en el que se realice muy poco trabajo útil. Si todo el mundo deseara reproducir flujos de vídeo a alta velocidad sin utilizar ningún mecanismo de control de congestión, se produciría tal desbordamiento de paquetes en los routers que muy pocos paquetes UDP lograrían recorrer con éxito la ruta entre el origen y el destino. Además, las altas tasas de pérdidas inducidas por los emisores UDP no controlados harían que los emisores TCP (que, como veremos, reducen sus velocidades de transmisión para hacer frente a la congestión) disminuyeran dramáticamente sus velocidades. Por tanto, la ausencia de un mecanismo de control de congestión en UDP puede dar lugar a altas tasas de pérdidas entre un emisor y un receptor UDP y al estrangulamiento de las sesiones TCP, lo cual es un problema potencialmente serio [Floyd 1999]. Muchos investigadores han propuesto nuevos mecanismos para forzar a todas las fuentes de datos, incluyendo las de tipo UDP, a llevar a cabo un control adaptativo de la congestión [Mahdavi 1997; Floyd 2000; Kohler 2006; RFC 4340].

Antes de pasar a examinar la estructura de los segmentos UDP, tenemos que comentar que *es* posible que una aplicación disponga de un servicio fiable de transferencia de datos utilizando UDP. Esto puede conseguirse si las características de fiabilidad se incorporan a la propia aplicación (por ejemplo, añadiendo mecanismos de reconocimiento y retransmisión, tales como los que vamos

a ver en la siguiente sección). Como ya hemos mencionado antes, el protocolo QUIC [Iyengar 2015] utilizado en el navegador Chrome de Google implementa la fiabilidad en un protocolo de la capa de aplicación por encima de UDP. Pero se trata de una tarea nada sencilla que requiere una intensa tarea de depuración de las aplicaciones. No obstante, incorporar los mecanismos de fiabilidad directamente en la aplicación permite que ella misma "se lo guise y se lo coma". Es decir, los procesos de aplicación pueden comunicarse de forma fiable sin estar sujetos a las restricciones de la velocidad de transmisión impuestas por el mecanismo de control de congestión de TCP.

3.3.1 Estructura de los segmentos UDP

La estructura de los segmentos UDP, mostrada en la Figura 3.7, está definida en el documento RFC 768. Los datos de la aplicación ocupan el campo de datos del segmento UDP. Por ejemplo, para DNS, el campo de datos contiene un mensaje de consulta o un mensaje de respuesta. En el caso de una aplicación de flujo de audio, las muestras del audio llenarán el campo de datos. La cabecera UDP solo tiene cuatro campos, y cada uno de ellos tiene una longitud de dos bytes. Como se ha explicado en la sección anterior, los números de puerto permiten al host de destino pasar los datos de la aplicación al proceso apropiado que está ejecutándose en el sistema terminal de destino (es decir, realizar la función de demultiplexación). El campo de longitud especifica el número de bytes del segmento UDP (la cabecera más los datos). Es necesario un valor de longitud explícito ya que el tamaño del campo de datos puede variar de un segmento UDP al siguiente. El host receptor utiliza la suma de comprobación para detectar si se han introducido errores en el segmento. En realidad, la suma de comprobación también se calcula para unos pocos campos de la cabecera IP, además de para el segmento UDP. Pero por el momento no vamos a tener en cuenta este detalle para ver el bosque a través de los árboles. Veamos ahora cómo se calcula la suma de comprobación. En la Sección 6.2 se describen los principios básicos para la detección de errores. El campo longitud especifica la longitud del segmento UDP en bytes, incluyendo la cabecera.

3.3.2 Suma de comprobación de UDP

La suma de comprobación de UDP proporciona un mecanismo de detección de errores. Es decir, se utiliza para determinar si los bits contenidos en el segmento UDP han sido alterados según se desplazaban desde el origen hasta el destino (por ejemplo, a causa de la existencia de ruido en los enlaces o mientras estaban almacenados en un router). UDP en el lado del emisor calcula el complemento a 1 de la suma de todas las palabras de 16 bits del segmento, acarreando cualquier desbordamiento obtenido durante la operación de suma sobre el bit de menor peso. Este resultado se almacena en el campo suma de comprobación del segmento UDP. He aquí un ejemplo sencillo de cálculo de una suma de comprobación. Puede obtener información detallada acerca de una implementación eficiente del cálculo en el RFC 1071, así como de su rendimiento con datos reales en [Stone 1998; Stone 2000]. Por ejemplo, suponga que tenemos las siguientes tres palabras de 16 bits:

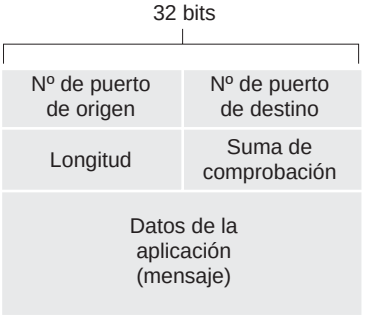


Figura 3.7 ♦ Estructura de un segmento UDP.

0110011001100000
0101010101010101
1000111100001100

La suma de las dos primeras palabras de 16 bits es:

0110011001100000
0101010101010101
1011101110110101

Sumando la tercera palabra a la suma anterior, obtenemos,

1011101110110101
1000111100001100
0100101011000010

Observe que en esta última suma se produce un desbordamiento, el cual se acarrea sobre el bit de menor peso. El complemento a 1 se obtiene convirtiendo todos los 0 en 1 y todos los 1 en 0. Por tanto, el complemento a 1 de la suma 0100101011000010 es 1011010100111101, que es la suma de comprobación. En el receptor, las cuatro palabras de 16 bits se suman, incluyendo la suma de comprobación. Si no se han introducido errores en el paquete, entonces la suma en el receptor tiene que ser 1111111111111111. Si uno de los bits es un 0, entonces sabemos que el paquete contiene errores.

Es posible que se esté preguntando por qué UDP proporciona una suma de comprobación, cuando muchos protocolos de la capa de enlace (incluyendo el popular protocolo Ethernet) también proporcionan mecanismos de comprobación de errores. La razón de ello es que no existe ninguna garantía de que todos los enlaces existentes entre el origen y el destino proporcionen un mecanismo de comprobación de errores; es decir, uno de los enlaces puede utilizar un protocolo de la capa de enlace que no proporcione comprobación de errores. Además, incluso si los segmentos se transfieren correctamente a través del enlace, es posible que se introduzcan errores de bit cuando un segmento se almacena en la memoria de un router. Dado que no están garantizadas ni la fiabilidad enlace a enlace, ni la detección de errores durante el almacenamiento en memoria, UDP tiene que proporcionar un mecanismo de detección de errores en la capa de transporte, *terminal a terminal*, si el servicio de transferencia de datos terminal a terminal ha de proporcionar la de detección de errores. Este es un ejemplo del famoso **principio terminal a terminal** del diseño de sistemas [Saltzer 1984], que establece que como cierta funcionalidad (en este caso, la detección de errores) debe implementarse terminal a terminal: "las funciones incluidas en los niveles inferiores pueden ser redundantes o escasamente útiles si se comparan con el coste de proporcionarlas en el nivel superior".

Dado que IP está pensado para ejecutarse sobre prácticamente cualquier protocolo de la capa 2, resulta útil para la capa de transporte proporcionar un mecanismo de comprobación de errores como medida de seguridad. Aunque UDP proporciona un mecanismo de comprobación de errores, no hace nada para recuperarse del error. Algunas implementaciones de UDP simplemente descartan el segmento dañado y otras lo pasan a la aplicación junto con una advertencia.

Hasta aquí llega nuestra exposición sobre UDP. Pronto veremos que TCP ofrece a las aplicaciones un servicio de transferencia de datos fiable, así como otros servicios que UDP no proporciona. Naturalmente, TCP también es más complejo que UDP. Sin embargo, antes de abordar TCP, nos resultará útil volver unos pasos atrás y ocuparnos primero de los principios que subyacen a una transferencia de datos fiable.

3.4 Principios de un servicio de transferencia de datos fiable

En esta sección vamos a considerar el problema de la transferencia de datos fiable en un contexto general. Este enfoque es conveniente porque el problema de implementar servicios de transferencia de datos fiables no solo aparece en la capa de transporte, sino también en la capa de enlace y en la capa de aplicación. El problema general tiene por tanto una gran relevancia en las redes de computadoras. En efecto, si tuviéramos que identificar la lista de los diez problemas más importantes que afectan a las redes, este sería un candidato a encabezar dicha lista. En la siguiente sección examinaremos TCP y, en concreto, mostraremos que TCP aplica muchos de los principios que vamos a describir.

La Figura 3.8 ilustra el marco de trabajo que vamos a emplear en nuestro estudio sobre la transferencia de datos fiable. La abstracción del servicio proporcionada a las entidades de la capa superior es la de un canal fiable a través del cual se pueden transferir datos. Disponiendo de un canal fiable, ninguno de los bits de datos transferidos está corrompido (cambia de 0 a 1, o viceversa) ni se pierde, y todos se entregan en el orden en que fueron enviados. Este es precisamente el modelo de servicio ofrecido por TCP a las aplicaciones de Internet que lo invocan.

Es la responsabilidad de un **protocolo de transferencia de datos fiable** implementar esta abstracción del servicio. Esta tarea es complicada por el hecho de que la capa que hay *por debajo* del protocolo de transferencia de datos puede ser no fiable. Por ejemplo, TCP es un protocolo de transferencia de datos fiable que se implementa encima de una capa de red terminal a terminal no fiable (IP). De forma más general, la capa que hay debajo de los dos puntos terminales que se

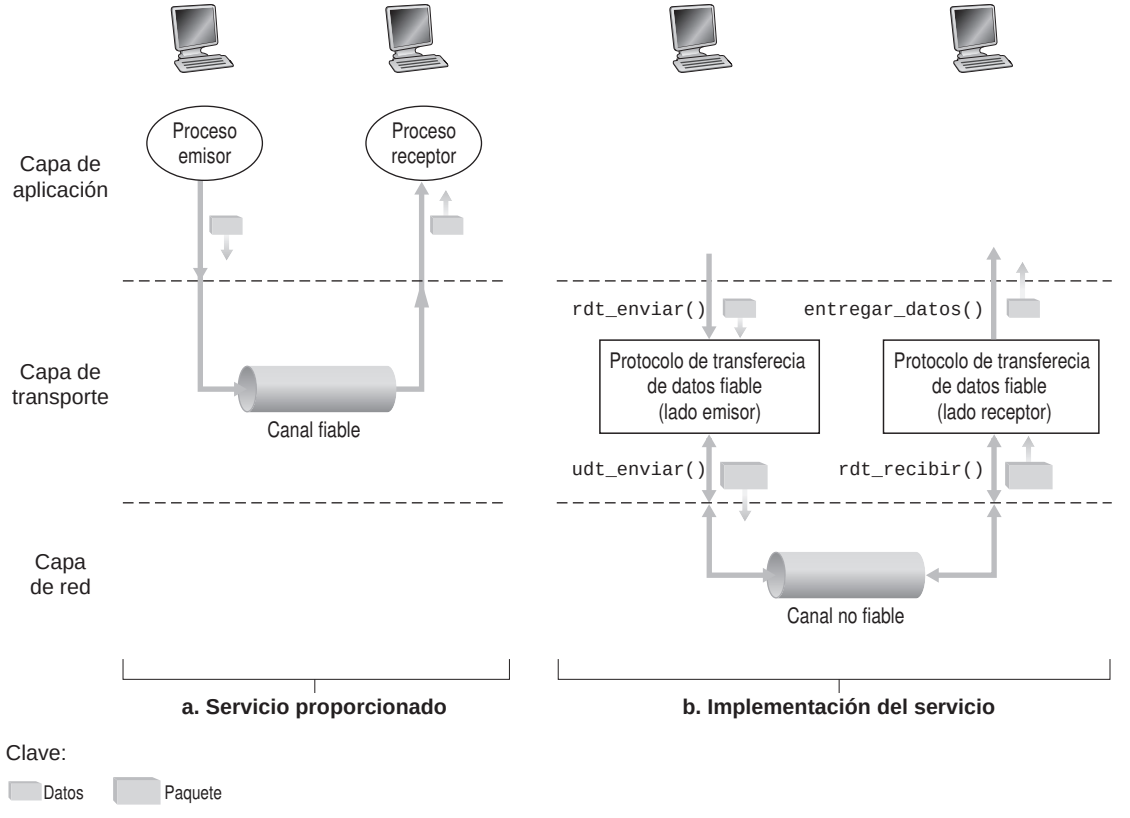


Figura 3.8 ♦ Transferencia de datos fiable: modelo del servicio e implementación del servicio.

comunican de forma fiable puede ser un único enlace físico (como en el caso de un protocolo de transferencia de datos a nivel de enlace) o una interred global (como en el caso de un protocolo del nivel de transporte). Para nuestros propósitos, sin embargo, podemos considerar esta capa inferior simplemente como un canal punto a punto no fiable.

En esta sección vamos a desarrollar de forma incremental los lados del emisor y del receptor de un protocolo de transferencia de datos fiable, considerando modelos cada vez más complejos del canal subyacente. Consideremos por ejemplo qué mecanismos del protocolo son necesarios cuando el canal subyacente puede corromper los bits o perder paquetes completos. Una suposición que vamos a adoptar a lo largo de esta exposición es que los paquetes se suministrarán en el orden en que fueron enviados, siendo posible que algunos paquetes se pierdan; es decir, es decir, el canal subyacente no reordena los paquetes. La Figura 3.8(b) ilustra las interfaces de nuestro protocolo de transferencia de datos. El lado emisor del protocolo de transferencia de datos será invocado desde la capa superior mediante una llamada a `rdt_enviar()`, que pasará los datos que haya que entregar a la capa superior en el lado receptor. Aquí `rdt` hace referencia al protocolo de transferencia de datos fiable (*reliable data transfer*) y `_enviar` indica que el lado emisor de `rdt` está siendo llamado. (¡El primer paso para desarrollar un buen protocolo es elegir un buen nombre!) En el lado receptor, `rdt_recibir()` será llamado cuando llegue un paquete desde el lado receptor del canal. Cuando el protocolo `rdt` desea suministrar datos a la capa superior, lo hará llamando a `entregar_datos()`. De aquí en adelante utilizaremos el término "paquete" en lugar de "segmento" de la capa de transporte. Puesto que la teoría desarrollada en esta sección se aplica a las redes de computadoras en general y no solo a la capa de transporte de Internet, quizá resulte más apropiado el término genérico "paquete".

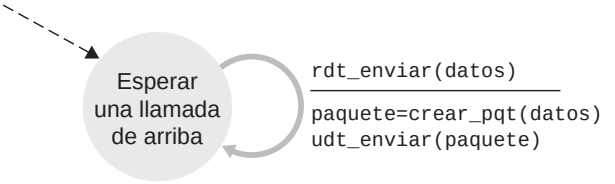
En esta sección únicamente consideraremos el caso de la **transferencia de datos unidireccional**, es decir, los datos se transfieren desde el lado emisor al receptor. El caso de la **transferencia de datos bidireccional** (es decir, *full-duplex*) conceptualmente no es más difícil, pero es considerablemente más tediosa de explicar. Aunque solo abordemos la transferencia de datos unidireccional, tendremos en cuenta que los lados emisor y receptor de nuestro protocolo necesitan transmitir paquetes en *ambas* direcciones, como se indica en la Figura 3.8. Veremos brevemente que, además de intercambiar paquetes que contengan los datos que van a transferirse, los lados emisor y receptor de `rdt` también intercambian paquetes de control de una parte a otra. Ambos lados, emisor y receptor, de `rdt` envían paquetes al otro lado haciendo una llamada a `udt_enviar()` (donde `udt` hace referencia a una *transferencia de datos no fiable* [*unreliable data transfer*]).

3.4.1 Construcción de un protocolo de transferencia de datos fiable

Ahora vamos a ver una serie de protocolos de complejidad creciente, hasta llegar a un protocolo de transferencia de datos fiable sin defectos.

Transferencia de datos fiable sobre un canal totalmente fiable: `rdt1.0`

En primer lugar consideremos el caso más simple, en el que el canal subyacente es completamente fiable. El protocolo en sí, que denominaremos `rdt1.0`, es trivial. En la Figura 3.9 se muestran las definiciones de las **máquinas de estados finitos** (**FSM*, Finite-State Machine*)** para el emisor y el receptor de `rdt1.0`. La máquina de estados finitos de la Figura 3.9(a) define el funcionamiento del emisor, mientras que la FSM de la Figura 3.9(b) define el funcionamiento del receptor. Es importante observar que existen máquinas de estados finitos *separadas* para el emisor y el receptor. Cada una de las máquinas de esta figura tiene solo un estado. Las flechas en la descripción de las FSM indican la transición del protocolo de un estado a otro. Puesto que en este caso cada una de las máquinas de la Figura 3.9 solo tiene un estado, necesariamente una transición es del estado a sí mismo (veremos diagramas más complicados enseguida). El suceso que provoca la transición se muestra encima de la línea horizontal que etiqueta la transición y las acciones que se toman cuando tiene lugar el suceso se indican debajo de la línea horizontal. Cuando no se lleve a cabo



a. rdt1.0: lado emisor

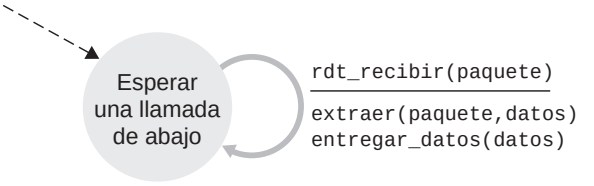


Figura 3.9 ♦ `rdt1.0` – Protocolo para un canal totalmente fiable.

ninguna acción al ocurrir un suceso, o cuando no se produzca un suceso y se realice una acción, utilizaremos el símbolo Λ por debajo o por encima de la horizontal, respectivamente, para indicar de manera explícita la ausencia de una acción o de un suceso. El estado inicial de la máquina FSM está indicado mediante la línea de puntos. Aunque las máquinas de estados finitos de la Figura 3.9 tienen un único estado, las que veremos a continuación tendrán múltiples, por lo que es importante identificar el estado inicial de cada máquina FSM.

El lado emisor de `rdt` simplemente acepta datos de la capa superior a través del suceso `rdt_enviar(datos)`, crea un paquete que contiene los datos (mediante la acción `crear_pqt(datos)`) y envía el paquete al canal. En la práctica, el suceso `rdt_enviar(datos)` resultaría de una llamada a procedimiento (por ejemplo, a `rdt_enviar()`) realizada por la aplicación de la capa superior.

En el lado receptor, `rdt` recibe un paquete del canal subyacente a través del suceso `rdt_recibir(paquete)`, extrae los datos del paquete (mediante la acción `extraer (paquete, datos)`) y pasa los datos a la capa superior (mediante la acción `entregar_datos(datos)`). En la práctica, el suceso `rdt_recibir(paquete)` resultaría de una llamada a procedimiento (por ejemplo, a `rdt_recibir()`) desde el protocolo de la capa inferior.

En este protocolo tan simple no existe ninguna diferencia entre una unidad de datos y un paquete. Además, todo el flujo de paquetes va del emisor al receptor, ya que disponiendo de un canal totalmente fiable no existe la necesidad en el lado receptor de proporcionar ninguna realimentación al emisor, puesto que no hay nada que pueda ser incorrecto. Observe que también hemos supuesto que el receptor podía recibir los datos tan rápido como el emisor los enviara. Luego tampoco existe la necesidad de que el receptor le pida al emisor que vaya más despacio.

Transferencia de datos fiable sobre un canal con errores de bit: `rdt2.0`

Un modelo más realista del canal subyacente sería uno en el que los bits de un paquete pudieran corromperse. Normalmente, tales errores de bit se producen en los componentes físicos de una red cuando un paquete se transmite, se propaga o accede a un buffer. Por el momento vamos a continuar suponiendo que todos los paquetes transmitidos son recibidos (aunque sus bits pueden estar corrompidos) en el orden en que se enviaron.

Antes de desarrollar un protocolo que permita una comunicación fiable a través de un canal así, vamos a ver cómo las personas se enfrentan a esta situación. Imagine cómo dictaría un mensaje

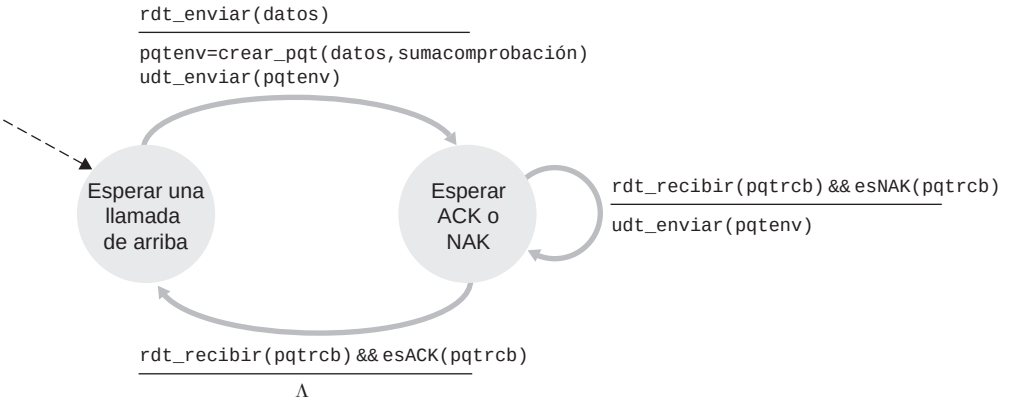
largo a través del teléfono. En un escenario típico, la persona que escucha el mensaje podría decir "De acuerdo" después de cada frase que escuche, comprenda y apunte. Si la persona que escucha el mensaje no oye una frase, le pedirá que la repita. Este protocolo de dictado de mensajes utiliza tanto **reconocimientos positivos** ("De acuerdo") como **reconocimientos negativos** ("Por favor, repita"). Estos mensajes de control permiten al receptor hacer saber al emisor qué es lo que ha recibido correctamente y qué ha recibido con errores y por tanto debe repetir. En una red de computadoras, los protocolos de transferencia de datos fiables basados en tales retransmisiones se conocen como **protocolos ARQ** (***Automatic Repeat reQuest,* Solicitud automática de repetición)**.

En los protocolos ARQ se requieren, fundamentalmente, tres capacidades de protocolo adicionales para gestionar la presencia de errores de bit:

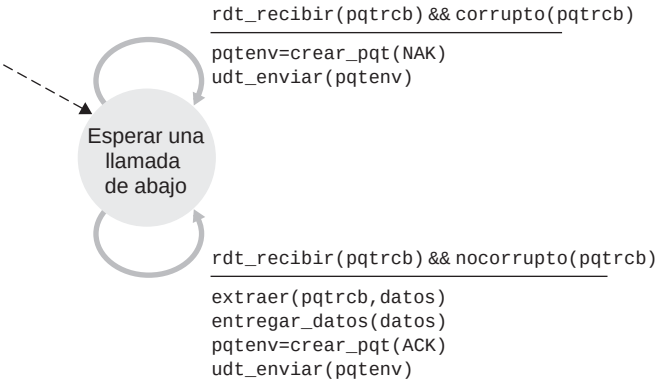
- *Detección de errores*. En primer lugar, se necesita un mecanismo que permita al receptor detectar que se han producido errores de bit. Recuerde de la sección anterior que UDP utiliza el campo de suma de comprobación de Internet precisamente para este propósito. En el Capítulo 6, examinaremos técnicas de detección y corrección de errores con más detalle; estas técnicas permiten al receptor detectar y, posiblemente, corregir los errores de bit en los paquetes. Por el momento, solo necesitamos saber que estas técnicas requieren que el emisor envíe al receptor bits adicionales (junto con los bits de los datos originales que se desean transferir) y dichos bits también se tendrán en cuenta para el cálculo de la suma de comprobación del paquete de datos `rdt2.0`.
- *Realimentación del receptor*. Dado que el emisor y el receptor normalmente se ejecutan en sistemas terminales diferentes, posiblemente separados por miles de kilómetros, la única forma de que el emisor sepa lo que ocurre en el receptor (en este caso, si un paquete ha sido recibido correctamente o no) es que el receptor envíe explícitamente información de realimentación al emisor. Las respuestas de acuse de recibo o reconocimiento positivo (ACK) y reconocimiento negativo (NAK) en el escenario del dictado de mensajes son ejemplos de esa realimentación. De forma similar, nuestro protocolo `rdt2.0` enviará paquetes ACK y NAK de vuelta desde el receptor al emisor. En principio, estos paquetes solo necesitan tener una longitud de un bit; por ejemplo, un valor 0 indicaría un reconocimiento negativo (NAK) y un valor 1 indicaría un reconocimiento positivo (ACK).
- *Retransmisión*. Un paquete que se recibe con errores en el receptor será retransmitido por el emisor.

La Figura 3.10 muestra la representación de la máquina de estados finitos para `rdt2.0`, un protocolo de transferencia de datos que dispone de mecanismos de detección de errores y paquetes de reconocimiento positivo y negativo.

El lado emisor de `rdt2.0` tiene dos estados. En el estado más a la izquierda, el protocolo del lado emisor está a la espera de datos procedentes de la capa superior. Cuando se produce el suceso `rdt_enviar(datos)`, el emisor creará un paquete (`pqtenv`) conteniendo los datos que van a ser transmitidos, junto con una suma de comprobación del paquete (como se ha visto en la Sección 3.3.2, por ejemplo, para el caso de un segmento UDP), y luego el paquete se envía mediante la operación `udt_enviar(pqtenv)`. En el estado más a la derecha, el protocolo del emisor está a la espera de un paquete de reconocimiento positivo ACK o negativo NAK procedente del receptor. Si se recibe un paquete ACK (la notación `rdt_recibir(pqtrcb) && esACK (pqtrcb)` mostrada en la Figura 3.10 corresponde a este suceso), el emisor sabe que el paquete más recientemente transmitido ha sido recibido correctamente y, por tanto, el protocolo vuelve al estado de espera de datos procedentes de la capa superior. Si se recibe un reconocimiento negativo NAK, el protocolo retransmite el último paquete y espera a recibir un mensaje ACK o NAK del receptor en respuesta al paquete de datos retransmitido. Es importante observar que cuando el emisor está en el estado "Esperar ACK o NAK", *no puede* obtener más datos de la capa superior; es decir, el suceso `rdt_enviar()` no puede ocurrir; esto solo ocurrirá después de que el emisor reciba un ACK y salga de ese estado. Por tanto, el emisor no enviará ningún nuevo fragmento de datos hasta estar seguro



a. rdt2.0: lado emisor



b. rdt2.0: lado receptor

Figura 3.10 ♦ `rdt2.0` – Protocolo para un canal con errores de bit.

de que el receptor ha recibido correctamente el paquete actual. Debido a este comportamiento, los protocolos como `rdt2.0` se conocen como **protocolos de parada y espera (*stop-and-wait protocol*)**.

El lado receptor de la máquina de estados finitos para `rdt2.0` tiene un único estado. Cuando llega un paquete, el receptor responde con un reconocimiento positivo ACK o negativo NAK, dependiendo de si el paquete recibido es correcto o está corrompido. En la Figura 3.10, la notación `rdt_recibir(pqtrcb) && corrupto(pqtrcb)` corresponde al suceso en el que se ha recibido un paquete y resulta ser erróneo.

Puede parecer que el protocolo `rdt2.0` funciona pero, lamentablemente, tiene un defecto fatal. ¡No hemos tenido en cuenta la posibilidad de que el paquete ACK o NAK pueda estar corrompido! (Antes de continuar, debería pararse a pensar en cómo se podría resolver este problema.) Lamentablemente, este descuido no es tan inocuo como puede parecer. Como mínimo, tendremos que añadir bits de suma de comprobación a los paquetes ACK/NAK para detectar tales errores. La cuestión más complicada es cómo puede recuperarse el protocolo de los errores en los paquetes ACK o NAK. La dificultad está en que si un paquete ACK o NAK está corrompido, el emisor no tiene forma de saber si el receptor ha recibido o no correctamente el último fragmento de datos transmitido.

Consideremos ahora tres posibilidades para gestionar los paquetes ACK o NAK corruptos:

- Para abordar la primera posibilidad, veamos lo que podría hacer una persona en el escenario del dictado de mensajes. Si la persona que está dictando el mensaje no entiende la respuesta "De acuerdo" o "Por favor, repita" del receptor, probablemente diría "¿Cómo dice?" (lo que introduce un nuevo tipo de paquete del emisor al receptor en nuestro protocolo). A continuación, el receptor repetiría la respuesta. ¿Pero qué ocurriría si el "Cómo dice" está corrompido? El receptor no tendría ni idea de si esa frase formaba parte del dictado o era una solicitud de que repitiera la última respuesta, por lo que probablemente respondería con un "¿Cómo dice usted?". Y, a su vez, por supuesto, dicha respuesta también podría verse alterada. Evidentemente, el problema se complica.
- Una segunda alternativa consistiría en añadir los suficientes bits de suma de comprobación como para permitir al emisor no solo detectar, sino también recuperarse de los errores de bit. De este modo se resuelve el problema inmediato de un canal que puede corromper los paquetes de datos, pero no perderlos.
- Un tercer método consistiría simplemente en que el emisor reenviara el paquete de datos actual al recibir un paquete ACK o NAK alterado. Sin embargo, este método introduce **paquetes duplicados** en el canal emisor-receptor. La principal dificultad con los paquetes duplicados es que el receptor no sabe si el último paquete ACK o NAK enviado fue recibido correctamente en el emisor. Por tanto, *a priori*, no puede saber si un paquete entrante contiene datos nuevos o se trata de una retransmisión.

Una solución sencilla a este nuevo problema (y que ha sido adoptada en prácticamente todos los protocolos de transferencia de datos existentes, incluido TCP) consiste en añadir un nuevo campo al paquete de datos, y hacer que el emisor numere sus paquetes de datos colocando un **número de secuencia** en este campo. Entonces bastará con que el receptor compruebe ese número de secuencia para determinar si el paquete recibido es o no una retransmisión. Para el caso de este protocolo de parada y espera simple, un número de secuencia de 1 bit será suficiente, ya que le permitirá al receptor saber si el emisor está retransmitiendo el paquete previamente transmitido (el número de secuencia del paquete recibido tiene el mismo número de secuencia que el paquete recibido más recientemente) o si se trata de un paquete nuevo (el número de secuencia es distinto, está desplazado "hacia adelante" en aritmética de módulo 2). Puesto que estamos suponiendo que disponemos de un canal que no pierde datos, los paquetes ACK y NAK no tienen que indicar el número de secuencia del paquete que están confirmando. El emisor sabe que un paquete ACK o NAK recibido (esté alterado o no) fue generado en respuesta a su paquete de datos transmitido más recientemente.

Las Figuras 3.11 y 3.12 muestran la descripción de la máquina de estados finitos para `rdt2.1`, nuestra versión revisada de `rdt2.0`. Ahora las máquinas de estados finitos de los lados emisor y receptor de `rdt2.1` tienen el doble de estados que antes. Esto se debe a que ahora el estado del protocolo tiene que reflejar si el paquete que está enviando actualmente el emisor o el que está esperando el receptor tiene que incluir un número de secuencia igual a 0 o a 1. Observe que las acciones en aquellos casos en los que un paquete con un número de secuencia de 0 está siendo enviado o es esperado son imágenes especulares de aquellos casos en los que el número de secuencia del paquete es 1; las únicas diferencias se encuentran en la gestión del número de secuencia.

En el protocolo `rdt2.1`, el receptor envía tanto respuestas de reconocimiento positivo como negativo al emisor. Cuando recibe un paquete fuera de secuencia, el receptor envía un paquete ACK para el paquete que ha recibido. Cuando recibe un paquete corrompido, el receptor envía una respuesta de reconocimiento negativo. Podemos conseguir el mismo efecto que con una respuesta NAK si, en lugar de enviar una NAK, enviamos una respuesta de reconocimiento positivo (ACK) para el último paquete recibido correctamente. Un emisor que recibe dos respuestas ACK para el mismo paquete (es decir, recibe **respuestas ACK duplicadas**) sabe que el receptor no ha recibido correctamente el paquete que sigue al que está siendo reconocido (respuesta ACK) dos veces. Nuestro protocolo de transferencia de datos fiable sin respuestas de tipo NAK para un canal con errores de bit es `rdt2.2`, el cual se ilustra en las Figuras 3.13 y 3.14. Una sutil variación entre los protocolos `rtdt2.1` y `rdt2.2` es que ahora el receptor tiene que incluir el número de secuencia

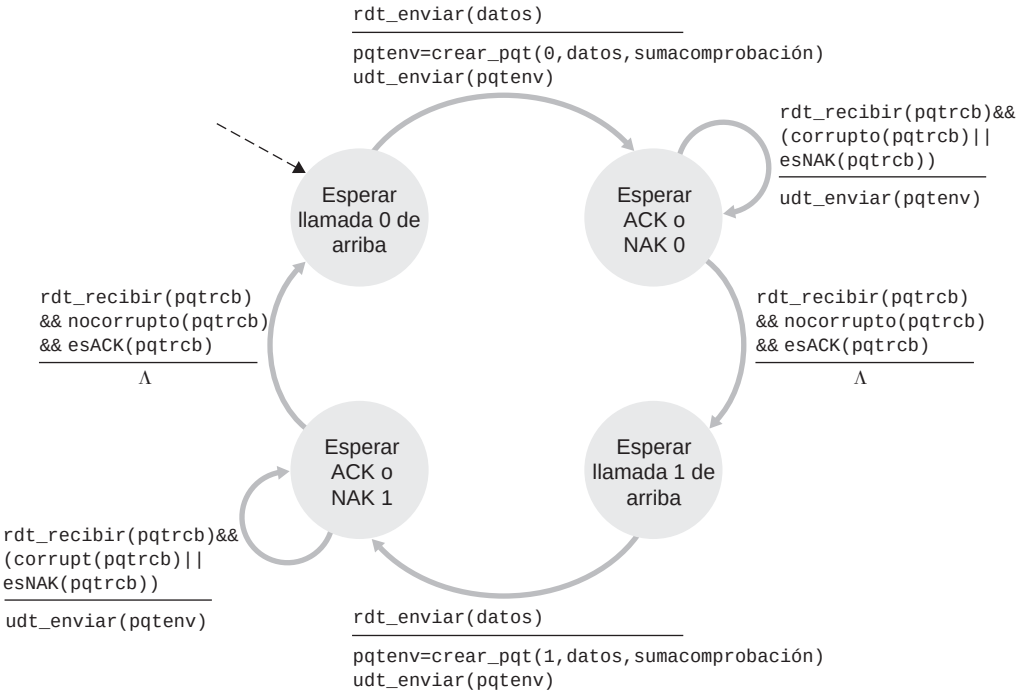


Figura 3.11 ♦ Lado emisor de `rdt2.1`

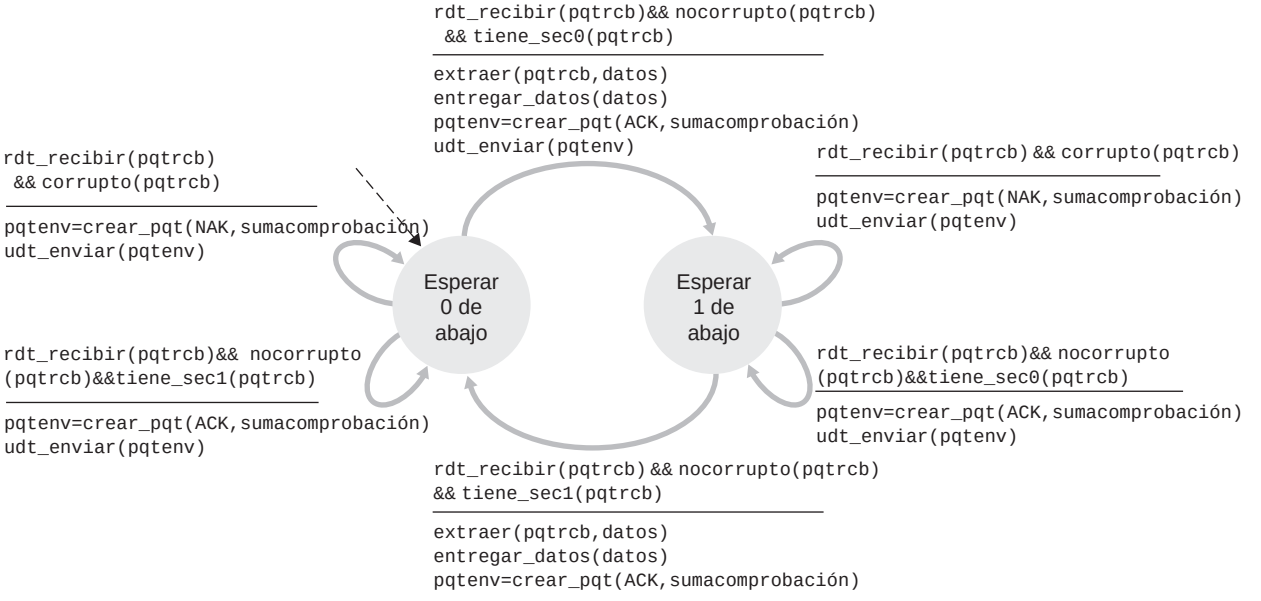


Figura 3.12 ♦ Lado receptor de `rdt2.1`

del paquete que está siendo confirmado mediante un mensaje ACK (lo que hace incluyendo el argumento `ACK,0` o `ACK,1` en `crear_pqt()` en la máquina de estados finitos de recepción), y el emisor tiene que comprobar el número de secuencia del paquete que está siendo confirmado por el mensaje ACK recibido (lo que se hace incluyendo el argumento `0` o `1` en `esACK()` en la máquina de estados finitos del emisor).

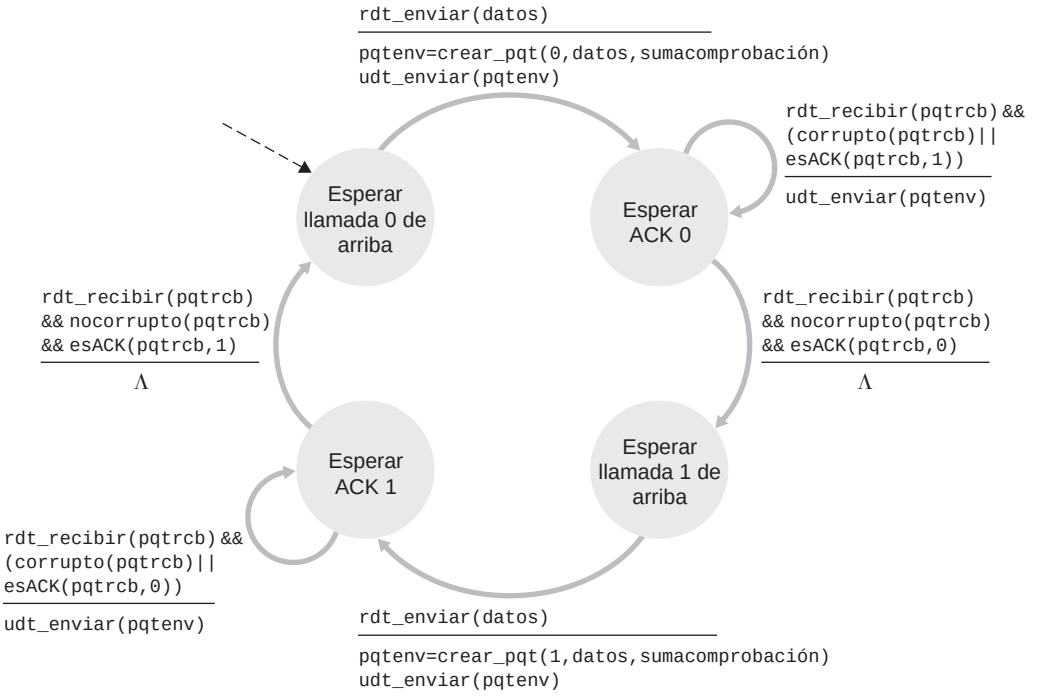


Figura 3.13 ♦ Lado emisor de `rdt2.2`.

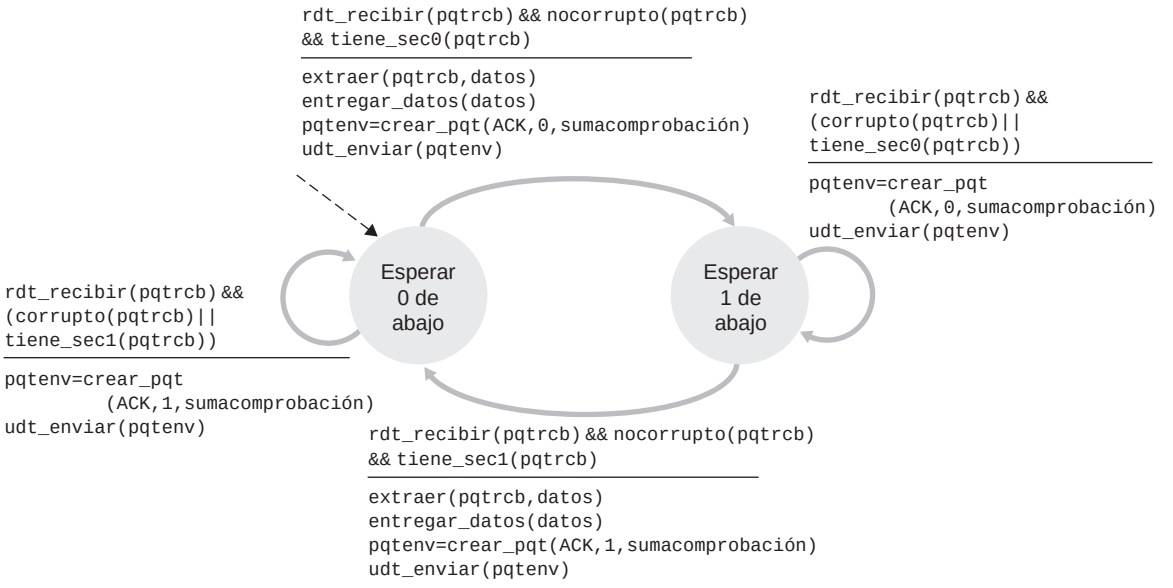


Figura 3.14 ♦ Lado receptor de `rdt2.2`

Transferencia de datos fiable sobre un canal con pérdidas y errores de bit: `rdt3.0`

Suponga ahora que además de bits corrompidos, el canal subyacente también puede *perder* paquetes, un suceso no desconocido en las redes de computadoras de hoy en día (incluyendo Internet). Por tanto, ahora el protocolo tiene que preocuparse por dos problemas más: cómo detectar la pérdida de paquetes y qué hacer cuando se pierde un paquete. El uso de la suma

de comprobación (*cheksum*), los números de secuencia, los paquetes ACK y la retransmisión de paquetes, técnicas que ya hemos desarrollado en el protocolo `rdt2.2`, nos van a permitir abordar este último problema, Para tratar el primero será necesario añadir un nuevo mecanismo de protocolo.

Hay disponibles muchas formas de abordar la pérdida de paquetes (varias de ellas se exploran en los ejercicios del final del capítulo). Veamos cómo el emisor puede detectar la pérdida de paquetes y cómo puede recuperarse de la misma. Suponga que el emisor transmite un paquete de datos y bien el propio paquete o el mensaje ACK del receptor para ese paquete se pierde. En cualquiera de estos dos casos, al emisor no le llega ninguna respuesta procedente del receptor. Si el emisor está dispuesto a esperar el tiempo suficiente como para *estar seguro* de que se ha perdido un paquete, simplemente puede retransmitirlo. Se puede comprobar de un modo sencillo que efectivamente este protocolo funciona.

Pero, ¿cuánto tiempo tiene que esperar el emisor para estar seguro de que se ha perdido un paquete? Es evidente que el emisor tiene que esperar al menos un tiempo igual al retardo de ida y vuelta entre el emisor y el receptor (lo que puede incluir el almacenamiento temporal en los buffers de los routers intermedios) más una cierta cantidad de tiempo que será necesaria para procesar un paquete en el receptor. En muchas redes, este retardo máximo del caso peor es muy difícil incluso de estimar y aún más de conocer con precisión. Además, idealmente, el protocolo debería recuperarse de la pérdida de paquetes tan pronto como fuera posible; pero si espera un tiempo igual al retardo en el caso peor, eso significa una larga espera hasta iniciar el mecanismo de recuperación de errores. Por tanto, el método que se adopta en la práctica es que el emisor seleccione juiciosamente un intervalo de tiempo tal que sea probable que un paquete se haya perdido, aunque no sea seguro que tal pérdida se haya producido. Si dentro de ese intervalo de tiempo no se ha recibido un ACK, el paquete se retransmite. Observe que si un paquete experimenta un retardo particularmente grande, el emisor puede retransmitirlo incluso aunque ni el paquete de datos ni su correspondiente ACK se hayan perdido. Esto introduce la posibilidad de que existan **paquetes de datos duplicados** en el canal emisor-receptor. Afortunadamente, el protocolo `rdt2.2` ya dispone de la funcionalidad (los números de secuencia) para afrontar la existencia de paquetes duplicados.

Desde el punto de vista del emisor, la retransmisión es la solución para todo. El emisor no sabe si se ha perdido un paquete de datos, se ha perdido un mensaje ACK, o simplemente el paquete o el ACK están retardados. En todos los casos, la acción es la misma: retransmitir. La implementación de un mecanismo de retransmisión basado en el tiempo requiere un **temporizador de cuenta atrás** que pueda interrumpir al emisor después de que haya transcurrido un determinado periodo de tiempo. Por tanto, el emisor necesitará poder (1) iniciar el temporizador cada vez que envíe un paquete (bien por primera vez o en una retransmisión), (2) responder a una interrupción del temporizador (ejecutando las acciones apropiadas) y (3) detener el temporizador.

La Figura 3.15 muestra la máquina de estados finitos del emisor para `rdt3.0`, un protocolo que transfiere datos de forma fiable a través de un canal que puede corromper o perder paquetes; en los problemas de repaso, se le pedirá que defina la máquina de estados finitos del receptor para `rdt3.0`. La Figura 3.16 muestra cómo opera el protocolo cuando no se pierden ni se retardan los paquetes y cómo gestiona la pérdida de paquetes de datos. En la Figura 3.16, el tiempo va avanzando desde la parte superior del diagrama hacia la parte inferior del mismo; observe que el instante de recepción de un paquete es necesariamente posterior al instante de envío de un paquete como consecuencia de los retardos de transmisión y de propagación. En las Figuras 3.16(b)–(d), los corchetes en el lado del emisor indican los instantes de inicio y fin del temporizador. Algunos de los aspectos más sutiles de este protocolo se verán en los ejercicios incluidos al final del capítulo. Dado que los números de secuencia de los paquetes alternan entre 0 y 1, el protocolo `rdt3.0` se denomina en ocasiones **protocolo de bit alternante**.

Nota de vídeo

Desarrollo de un protocolo y representación de una FSM para un protocolo de la capa de aplicación simple.

Hasta aquí hemos ensamblado los elementos clave de un protocolo de transferencia de datos. Las sumas de comprobación, los números de secuencia, los temporizadores y los paquetes de reconocimiento positivo y negativo desempeñan un papel fundamental y necesario en el funcionamiento del protocolo. A continuación vamos a trabajar con un protocolo de transferencia de datos fiable.

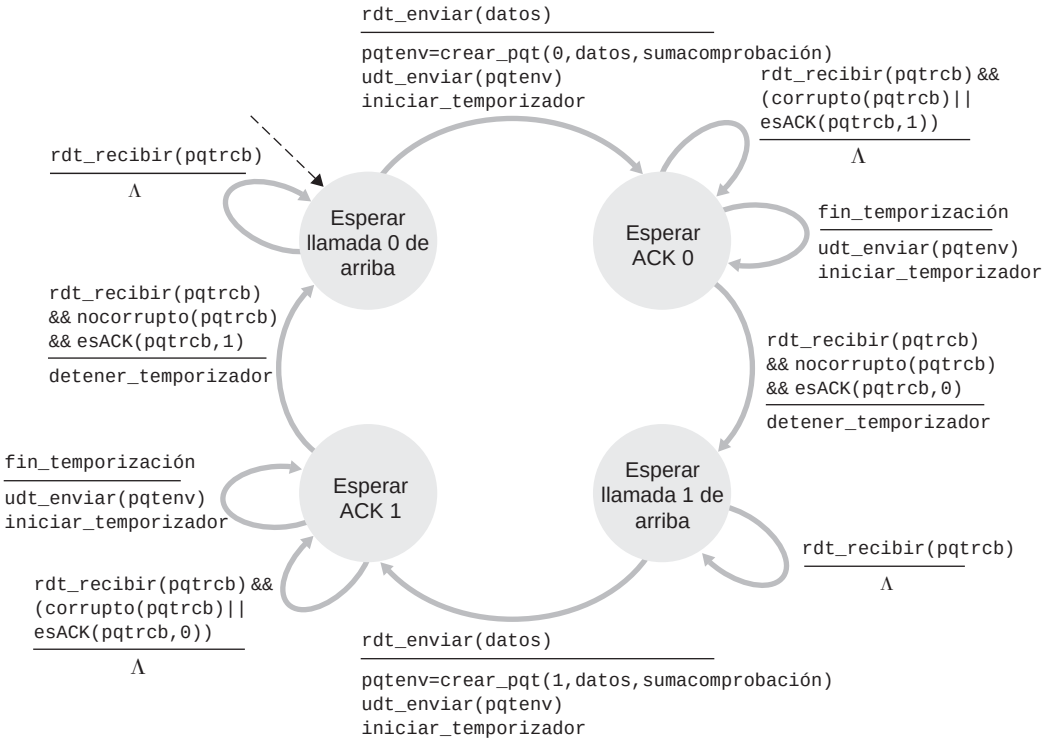


Figura 3.15 ♦ Lado emisor de `rdt3.0`

3.4.2 Protocolo de transferencia de datos fiable con procesamiento en cadena

El protocolo `rdt3.0` es un protocolo funcionalmente correcto, pero es muy improbable que haya alguien a quien satisfaga su rendimiento, especialmente en las redes de ata velocidad actuales. La base del problema del rendimiento de `rdt3.0` se encuentra en el hecho de que es un protocolo de parada y espera.

Para entender el impacto sobre el rendimiento de este comportamiento de parada y espera, vamos a considerar un caso ideal de dos hosts, uno localizado en la costa oeste de Estados Unidos y el otro en la costa este, como se muestra en la Figura 3.17. El retardo de propagación de ida y vuelta, RTT, a la velocidad de la luz entre estos dos sistemas terminales es aproximadamente igual a 30 milisegundos. Suponga que están conectados mediante un canal cuya velocidad de transmisión, R, es de 1 Gbps (10^9 bits por segundo). Con un tamaño de paquete, L, de 1.000 bytes (8.000 bits) por paquete, incluyendo los campos de cabecera y los datos, el tiempo necesario para transmitir el paquete por un enlace de 1 Gbps es:

$$d_{trans} = \frac{L}{R} = \frac{8000 \text{ bits/paquete}}{10^9 \text{ bits/segundo}} = 8 \text{ microsegundos}$$

La Figura 3.18(a) muestra que, con nuestro protocolo de parada y espera, si el emisor comienza a transmitir el paquete en el instante $t = 0$, entonces en el instante $t = L/R = 8$ microsegundos el último bit entra en el canal en el lado del emisor. El paquete entonces tarda 15 milisegundos en atravesar el país, emergiendo el último bit del paquete en el lado del receptor en el instante $t = RTT/2 + L/R = 15{,}008$ milisegundos. Con el fin de simplificar, vamos a suponer que los

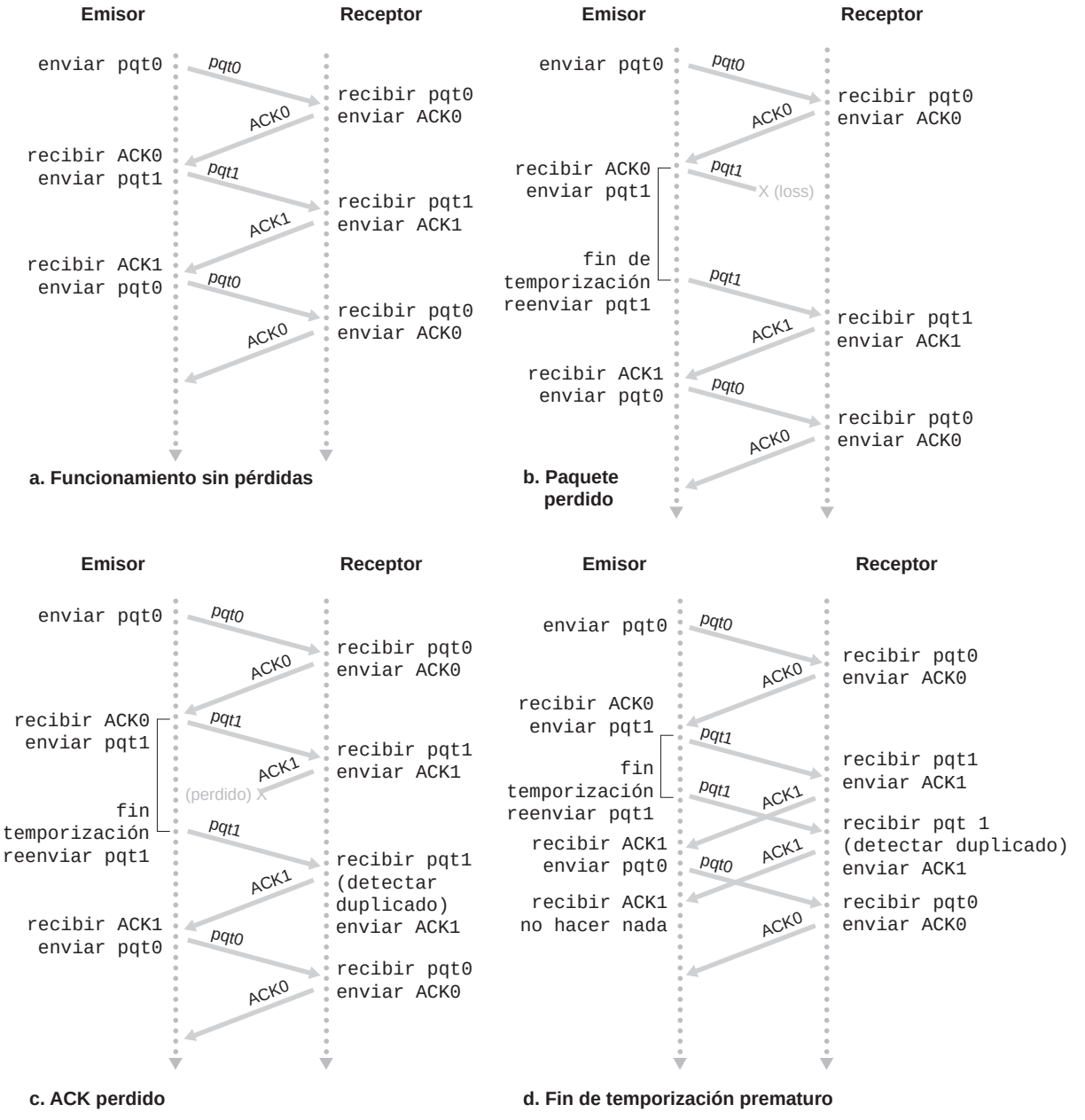
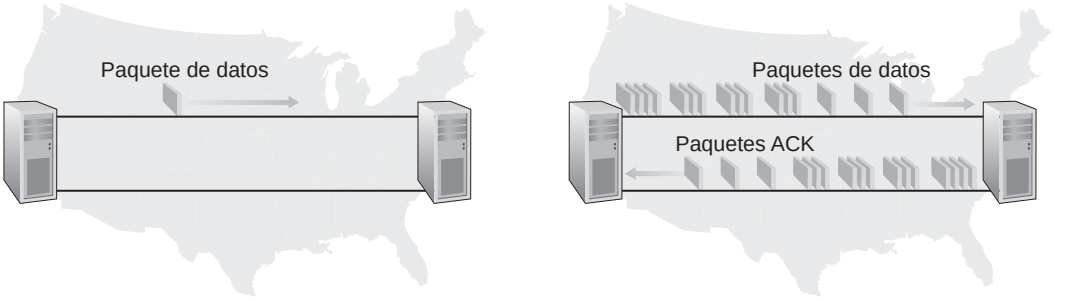


Figura 3.16 ♦ Funcionamiento de `rdt3.0`, el protocolo de bit alternante.

paquetes de reconocimiento ACK son extremadamente pequeños (por lo que podemos ignorar su tiempo de transmisión) y que el receptor puede enviar un ACK tan pronto como ha recibido el último bit de un paquete de datos, llegando dicho ACK al emisor en $t = RTT + L/R = 30{,}008$ ms. En esta situación, ahora el emisor puede transmitir el siguiente mensaje. Por tanto, en 30,008 milisegundos, el emisor ha estado transmitiendo durante solo 0,008 milisegundos. Si definimos la tasa de **utilización** del emisor (o del canal) como la fracción de tiempo que el emisor está realmente ocupado enviando bits al canal, el análisis de la Figura 3.18(a) muestra que el protocolo de parada y espera tiene una tasa de utilización del emisor, U_{emisor}, bastante mala de



a. Funcionamiento de un protocolo de parada y espera

b. Funcionamiento de un protocolo con procesamiento en cadena

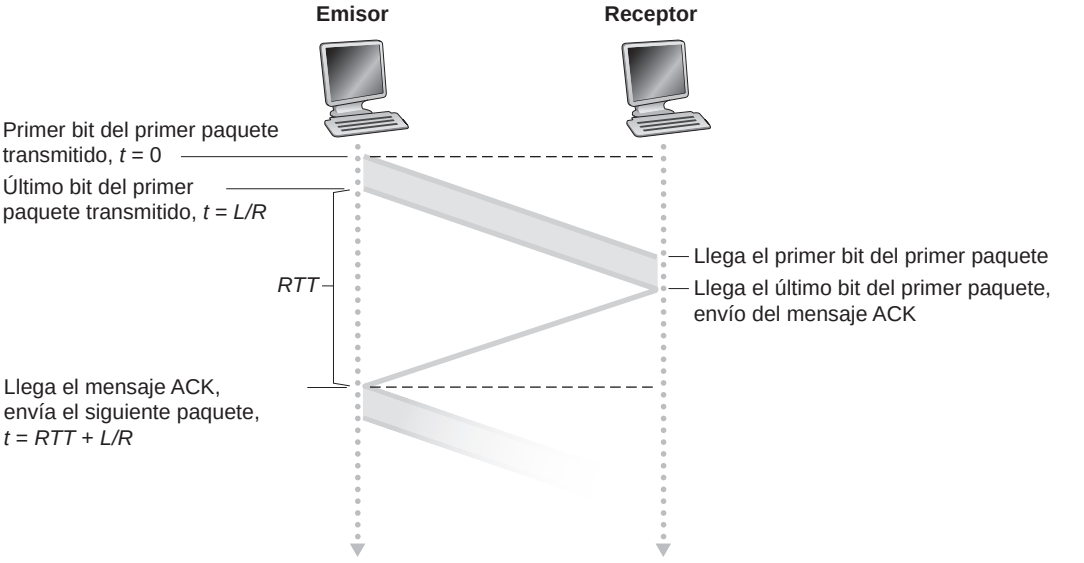
Figura 3.17 ♦ Protocolo de parada y espera y protocolo con procesamiento en cadena.

$$U_{emisor} = \frac{L / R}{RTT + L / R} = \frac{0,008}{30,008} = 0,00027$$

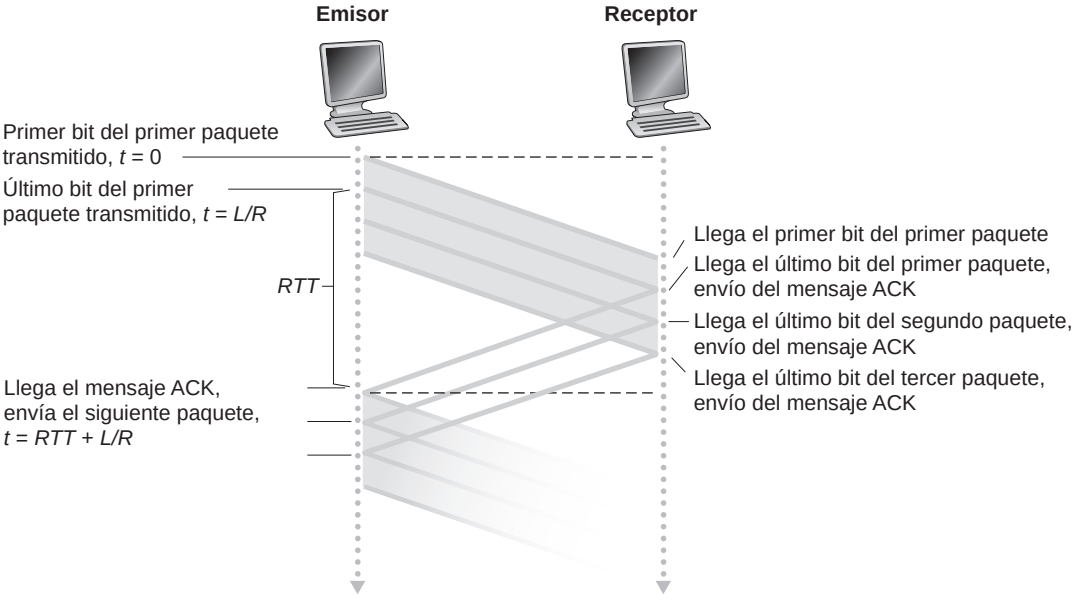
Es decir, ¡el emisor solo está ocupado 2,7 diezmilésimas del tiempo! En otras palabras, el emisor solo ha podido enviar 1.000 bytes en 30,008 milisegundos, una tasa de transferencia efectiva de solo 267 kbps, incluso disponiendo de un enlace a 1 Gbps. Imagine al infeliz administrador de la red que ha pagado una fortuna por un enlace con una capacidad de un gigabit para obtener una tasa de transferencia de únicamente 267 kilobits por segundo. Este es un ejemplo gráfico de cómo los protocolos de red pueden limitar las capacidades proporcionadas por el hardware de red subyacente. Además, hemos despreciado los tiempos de procesamiento del protocolo de la capa inferior tanto en el emisor como en el receptor, así como los retardos de procesamiento y de cola que pueden tener lugar en cualquiera de los routers intermedios existentes entre el emisor y el receptor. La inclusión de estos efectos solo serviría para incrementar el retardo y acentuar más su pésimo rendimiento.

La solución a este problema de rendimiento concreto es simple: en lugar de operar en el modo parada y espera, el emisor podría enviar varios paquetes sin esperar a los mensajes de reconocimiento, como se ilustra en la Figura 3.17(b). La Figura 3.18(b) muestra que si el emisor transmite tres paquetes antes de tener que esperar a los paquetes de reconocimiento, la utilización del emisor prácticamente se triplica. Dado que los muchos paquetes que están en tránsito entre el emisor y el receptor pueden visualizarse como el relleno de un conducto (*pipeline*), esta técnica se conoce como ***pipelining*** o **procesamiento en cadena**. El procesamiento en cadena tiene las siguientes consecuencias en los protocolos de transferencia de datos fiables:

- El rango de los números de secuencia tiene que incrementarse, dado que cada paquete en tránsito (sin contar las retransmisiones) tiene que tener un número de secuencia único y pueden coexistir múltiples paquetes en tránsito que no hayan sido confirmados mediante un reconocimiento.
- Los lados emisor y receptor de los protocolos pueden tener que almacenar en buffer más de un paquete. Como mínimo, el emisor tendrá en el buffer los paquetes que han sido transmitidos pero que todavía no han sido reconocidos. Como veremos enseguida, también puede ser necesario almacenar en el buffer del receptor los paquetes recibidos correctamente.
- El rango necesario de los números de secuencia y los requisitos de buffer dependerán de la forma en que un protocolo de transferencia de datos responda a la pérdida de paquetes y a los paquetes corrompidos o excesivamente retardados. Hay disponibles dos métodos básicos que permiten la recuperación de errores mediante procesamiento en cadena: **Retroceder N** y la **repetición selectiva**.



a. Funcionamiento de un protocolo de parada y espera



b. Funcionamiento con procesamiento en cadena

Figura 3.18 ♦ Proceso de transmisión con un protocolo de parada y espera y un protocolo con procesamiento en cadena.

3.4.3 Retroceder N (GBN)

En un **protocolo GBN (*Go-Back-N*, Retroceder N)**, el emisor puede transmitir varios paquetes (si están disponibles) sin tener que esperar a que sean reconocidos, pero está restringido a no tener más de un número máximo permitido, *N*, de paquetes no reconocidos en el canal. En esta sección vamos a describir el protocolo GBN con cierto detalle. Pero antes de seguir leyendo, le animamos a practicar con el applet GBN (¡un applet impresionante!) disponible en el sitio web del libro.

La Figura 3.19 muestra la parte correspondiente al emisor del rango de los números de secuencia en un protocolo GBN. Si definimos la `base` como el número de secuencia del paquete no reconocido más antiguo y `signumsec` como el número de secuencia más pequeño no utilizado (es decir, el número de secuencia del siguiente paquete que se va a enviar), entonces se pueden identificar los cuatro intervalos en rango de los números de secuencia. Los números de secuencia pertenecientes al intervalo `[0,base-1]` corresponden a paquetes que ya han sido transmitidos y reconocidos. El intervalo `[base,signumsec-1]` corresponde a paquetes que ya han sido enviados pero todavía no se han reconocido. Los números de secuencia del intervalo `[signumsec,base+N-1]` se pueden emplear para los paquetes que pueden ser enviados de forma inmediata, en caso de que lleguen datos procedentes de la capa superior. Y, por último, los números de secuencia mayores o iguales que `base+N` no pueden ser utilizados hasta que un paquete no reconocido que se encuentre actualmente en el canal sea reconocido (específicamente, el paquete cuyo número de secuencia sea igual a `base`).

Como sugiere la Figura 3.19, el rango de los números de secuencia permitidos para los paquetes transmitidos pero todavía no reconocidos puede visualizarse como una ventana de tamaño *N* sobre el rango de los números de secuencia. Cuando el protocolo opera, esta ventana se desplaza hacia adelante sobre el espacio de los números de secuencia. Por esta razón, *N* suele denominarse **tamaño de ventana** y el propio protocolo GBN se dice que es un **protocolo de ventana deslizante**. Es posible que se esté preguntando, para empezar, por qué debemos limitar a *N* el número de paquetes no reconocidos en circulación. ¿Por qué no permitir un número ilimitado de tales paquetes? En la Sección 3.5 veremos que el control de flujo es una de las razones para imponer un límite en el emisor. En la Sección 3.7 examinaremos otra razón al estudiar el mecanismo de control de congestión de TCP.

En la práctica, el número de secuencia de un paquete se incluye en un campo de longitud fija de la cabecera del paquete. Si k es el número de bits contenido en el campo que especifica el número de secuencia del paquete, el rango de los números de secuencia será $[0,2^k - 1]$. Con un rango finito de números de secuencia, todas las operaciones aritméticas que impliquen a los números de secuencia tendrán que efectuarse utilizando aritmética en módulo 2^k. (Es decir, el espacio de números de secuencia puede interpretarse como un anillo de tamaño 2^k, donde el número de secuencia $2^k - 1$ va seguido por el número de secuencia 0.) Recuerde que el protocolo `rdt3.0` dispone de un número de secuencia de 1 bit y de un rango de números de secuencia de [0,1]. Algunos de los problemas incluidos al final del capítulo exploran las consecuencias de disponer de un rango finito de números de secuencia. En la Sección 3.5 veremos que TCP utiliza un campo para el número de secuencia de 32 bits, donde los números de secuencia de TCP cuentan los bytes del flujo de datos, en lugar de los paquetes.

Las Figuras 3.20 y 3.21 proporcionan una descripción ampliada de la máquina de estados finitos de los lados emisor y receptor de un protocolo GBN basado en paquetes de reconocimiento ACK y que no emplea paquetes de reconocimiento NAK. Nos referiremos a esta descripción de una FSM como *FSM ampliada*, ya que hemos añadido variables (similares a las variables de un lenguaje de programación) para `base` y `signumsec`, y operaciones sobre dichas variables y acciones condicionales que las implican. Observe que la especificación de una FSM ampliada empieza a asemejarse a una especificación de un lenguaje de programación. [Bochman 1984]

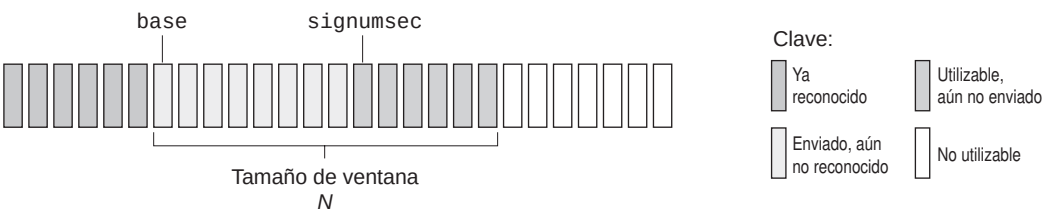


Figura 3.19 ♦ Números de secuencia en el emisor en el protocolo Retroceder N.

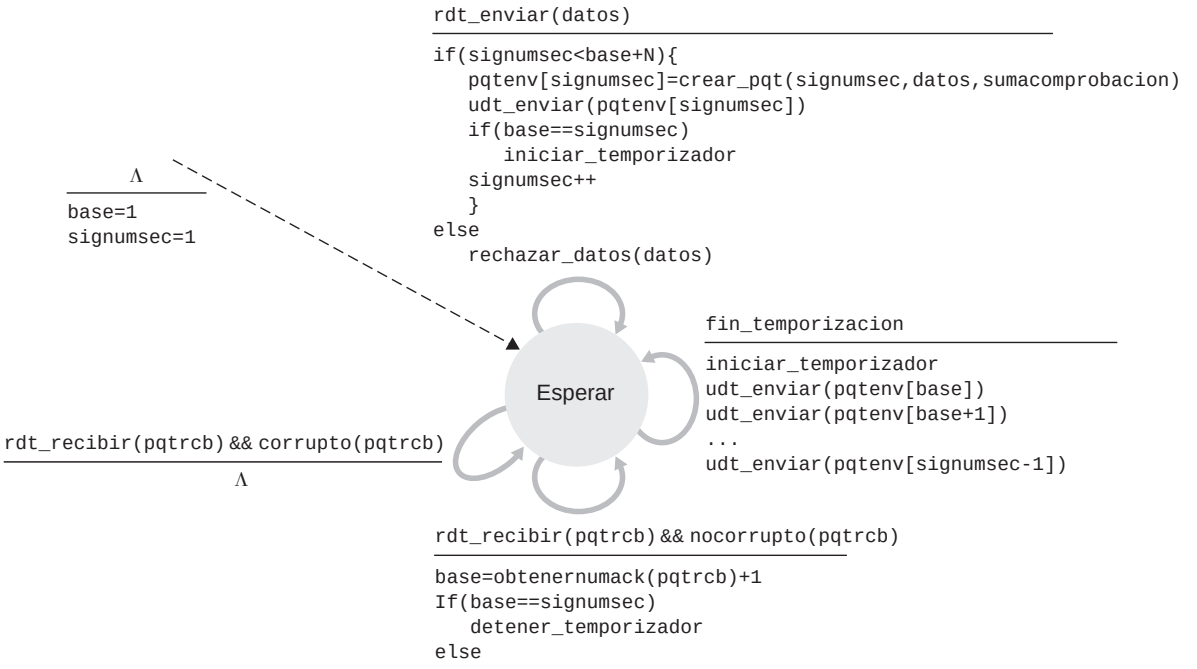


Figura 3.20 ♦ Descripción de la FSM ampliada del lado emisor de GBN.

proporciona un excelente repaso de otras ampliaciones de las técnicas de máquinas de estados finitos, así como de técnicas basadas en lenguajes de programación para la especificación de protocolos.

El emisor del protocolo GBN tiene que responder a tres tipos de sucesos:

- *Invocación desde la capa superior*. Cuando se llama a `rdt_enviar()` desde la capa superior, lo primero que hace el emisor es ver si la ventana está llena; es decir, si hay *N* paquetes no reconocidos en circulación. Si la ventana no está llena, se crea y se envía un paquete y se actualizan las variables de la forma apropiada. Si la ventana está llena, el emisor simplemente devuelve los datos a la capa superior, indicando de forma implícita que la ventana está llena. Probablemente entonces la capa superior volverá a intentarlo más tarde. En una implementación real, muy posiblemente el emisor almacenaría en el buffer estos datos (pero no los enviaría de forma inmediata) o dispondría de un mecanismo de sincronización (por ejemplo, un semáforo o un indicador) que permitiría a la capa superior llamar a `rdt_enviar()` solo cuando la ventana no estuviera llena.
- *Recepción de un mensaje de reconocimiento ACK*. En nuestro protocolo GBN, un reconocimiento de un paquete con un número de secuencia *n* implica un **reconocimiento acumulativo**, lo que indica que todos los paquetes con un número de secuencia menor o igual que *n* han sido correctamente recibidos por el receptor. Volveremos sobre este tema enseguida al examinar el lado receptor de GBN.
- *Un suceso de fin de temporización*. El nombre de este protocolo, "Retroceder N", se deriva del comportamiento del emisor en presencia de paquetes perdidos o muy retardados. Como en los protocolos de parada y espera, se empleará un temporizador para recuperarse de la pérdida de paquetes de datos o de reconocimiento de paquetes. Si se produce un fin de temporización, el emisor reenvía *todos* los paquetes que haya transmitido anteriormente y que todavía no hayan sido reconocidos. El emisor de la Figura 3.20 utiliza un único temporizador, que puede interpretarse como un temporizador para los paquetes transmitidos pero todavía no reconocidos. Si

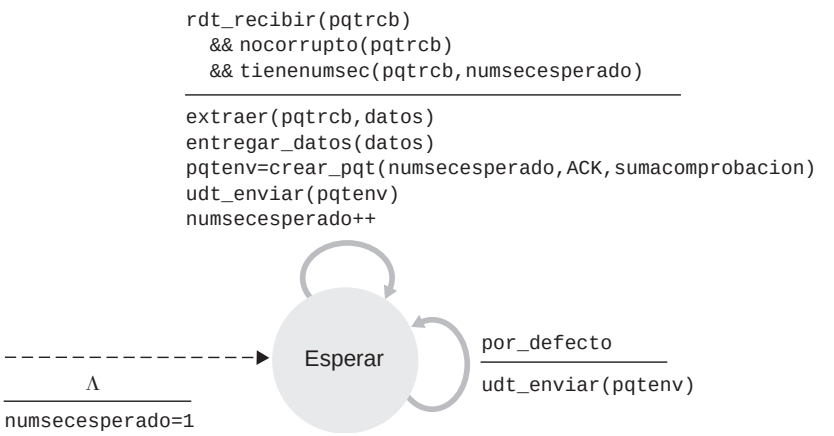


Figura 3.21 ♦ Descripción de la FSM ampliada del lado receptor de GBN.

se recibe un paquete ACK pero existen más paquetes transmitidos adicionales no reconocidos, entonces se reinicia el temporizador. Si no hay paquetes no reconocidos en circulación, el temporizador se detiene.

Las acciones del receptor en el protocolo GBN también son simples. Si un paquete con un número de secuencia *n* se recibe correctamente y en orden (es decir, los últimos datos entregados a la capa superior proceden de un paquete con el número de secuencia $n - 1$), el receptor envía un paquete ACK para el paquete *n* y entrega la parte de los datos del paquete a la capa superior. En todos los restantes casos, el receptor descarta el paquete y reenvía un mensaje ACK para el paquete recibido en orden más recientemente. Observe que dado que los paquetes se entregan a la capa superior de uno en uno, si el paquete *k* ha sido recibido y entregado, entonces todos los paquetes con un número de secuencia menor que *k* también han sido entregados. Por tanto, el uso de confirmaciones acumulativas es una opción natural del protocolo GBN.

En nuestro protocolo GBN, el receptor descarta los paquetes que no están en orden. Aunque puede parecer algo tonto y una pérdida de tiempo descartar un paquete recibido correctamente (pero desordenado), existe una justificación para hacerlo. Recuerde que el receptor debe entregar los datos en orden a la capa superior. Suponga ahora que se espera el paquete *n*, pero llega el paquete $n + 1$. Puesto que los datos tienen que ser entregados en orden, el receptor *podría* guardar en el buffer el paquete $n + 1$ y luego entregar ese paquete a la capa superior después de haber recibido y entregado el paquete *n*. Sin embargo, si se pierde el paquete *n*, tanto él como el paquete $n + 1$ serán retransmitidos como resultado de la regla de retransmisión del protocolo GBN en el lado de emisión. Por tanto, el receptor puede simplemente descartar el paquete $n + 1$. La ventaja de este método es la simplicidad del almacenamiento en el buffer del receptor (el receptor no necesita almacenar en el buffer *ninguno* de los paquetes entregados desordenados. Por tanto, mientras el emisor tiene que mantener los límites inferior y superior de su ventana y la posición de `signumsec` dentro de esa ventana, el único fragmento de información que el receptor debe mantener es el número de secuencia del siguiente paquete en orden. Este valor se almacena en la variable `numsecesperado`, como se muestra en la máquina de estados finitos del receptor de la Figura 3.21. Por supuesto, la desventaja de descartar un paquete correctamente recibido es que la subsiguiente retransmisión de dicho paquete puede perderse o alterarse y, por tanto, ser necesarias aún más retransmisiones.

La Figura 3.22 muestra el funcionamiento del protocolo GBN para el caso de un tamaño de ventana de cuatro paquetes. A causa de esta limitación del tamaño de ventana, el emisor transmite los paquetes 0 a 3, pero tiene que esperar a que uno o más de estos paquetes sean reconocidos antes de continuar. A medida que se reciben los sucesivos paquetes ACK (por ejemplo, `ACK0` y `ACK1`), la

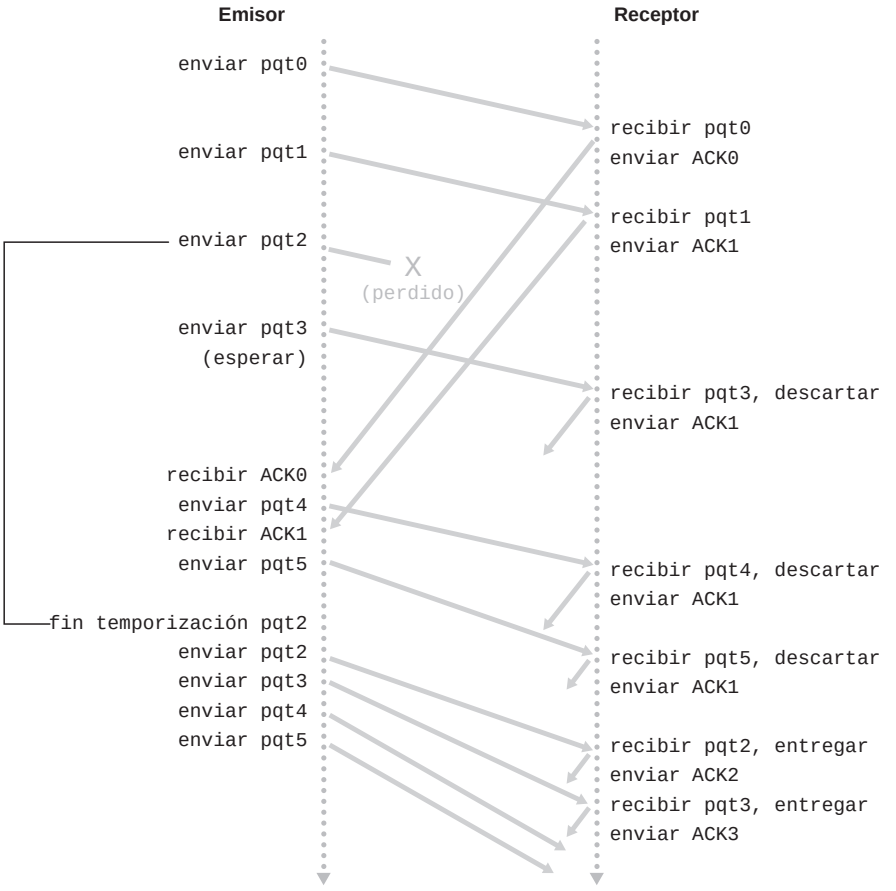


Figura 3.22 ♦ Funcionamiento del protocolo GBN (retroceder N).

ventana se desplaza hacia adelante y el emisor puede transmitir un nuevo paquete (`pqt4` y `pqt5`, respectivamente). En el lado receptor se pierde el paquete 2 y, por tanto, los paquetes 3, 4 y 5 no se reciben en el orden correcto y, lógicamente, se descartan.

Antes de terminar esta exposición acerca de GBN, merece la pena destacar que una implementación de este protocolo en una pila de protocolos tendría probablemente una estructura similar a la de la máquina de estados finitos ampliada de la Figura 3.20. Posiblemente, la implementación se realizaría mediante varios procedimientos que implementasen las acciones que habría que realizar en respuesta a los distintos sucesos que pueden producirse. En una **programación basada en sucesos**, los diversos procedimientos son llamados (invocados) por otros procedimientos de la pila de protocolos, o como resultado de una interrupción. En el emisor, estos sucesos podrían ser (1) una llamada de una entidad de la capa superior para invocar `rdt_enviar()`, (2) una interrupción del temporizador y (3) una llamada de la capa inferior para invocar `rdt_recibir()` cuando llega un paquete. Los ejercicios sobre programación incluidos al final del capítulo le darán la oportunidad de implementar en la práctica estas rutinas en una red simulada, pero realista.

Observe que el protocolo GBN incorpora casi todas las técnicas que veremos en la Sección 3.5 al estudiar los componentes del servicio de transferencia de datos fiable de TCP. Estas técnicas incluyen el uso de números de secuencia, reconocimientos acumulativos, sumas de comprobación y una operación de fin de temporización/retransmisión.

3.4.4 Repetición selectiva (SR)

El protocolo GBN permite al emisor, en teoría, "llenar el conducto" mostrado en la Figura 3.17 con paquetes, evitando así los problemas de utilización del canal que hemos visto con los protocolos de parada y espera. Sin embargo, hay algunos escenarios en los que el propio GBN presenta problemas de rendimiento. En particular, cuando el tamaño de la ventana y el producto ancho de banda-retardo son grandes puede haber muchos paquetes en el canal. En este caso, un único paquete erróneo podría hacer que el protocolo GBN retransmitiera una gran cantidad de paquetes, muchos de ellos de forma innecesaria. A medida que la probabilidad de errores en el canal aumenta, este puede comenzar a llenarse con estas retransmisiones innecesarias. En nuestro escenario del dictado de mensajes, imagine que si cada vez que se altera una palabra, las 1.000 palabras que la rodean (por ejemplo, con un tamaño de ventana de 1.000 palabras) tuvieran que ser repetidas, el dictado se ralentizaría a causa de la repetición de palabras.

Como su nombre sugiere, los protocolos de repetición selectiva evitan las retransmisiones innecesarias haciendo que el emisor únicamente retransmita aquellos paquetes que se sospeche que llegaron al receptor con error (es decir, que se perdieron o estaban corrompidos). Esta retransmisión individualizada y necesaria requerirá que el receptor confirme *individualmente* qué paquetes ha recibido correctamente. De nuevo, utilizaremos una ventana de tamaño *N* para limitar el número de paquetes no reconocidos y en circulación en el canal. Sin embargo, a diferencia de GBN, el emisor ya habrá recibido mensajes ACK para algunos de los paquetes de la ventana. En la Figura 3.23 se muestra el espacio de números de secuencia del lado de emisión del protocolo SR. La Figura 3.24 detalla las distintas acciones que lleva a cabo el emisor del protocolo SR.

El receptor de SR confirmará que un paquete se ha recibido correctamente tanto si se ha recibido en el orden correcto como si no. Los paquetes no recibidos en orden se almacenarán en el buffer hasta que se reciban los paquetes que faltan (es decir, los paquetes con números de secuencia menores), momento en el que un lote de paquetes puede entregarse en orden a la capa superior. En la Figura 3.25 se enumeran las acciones tomadas por el receptor de SR. La Figura 3.26 muestra un

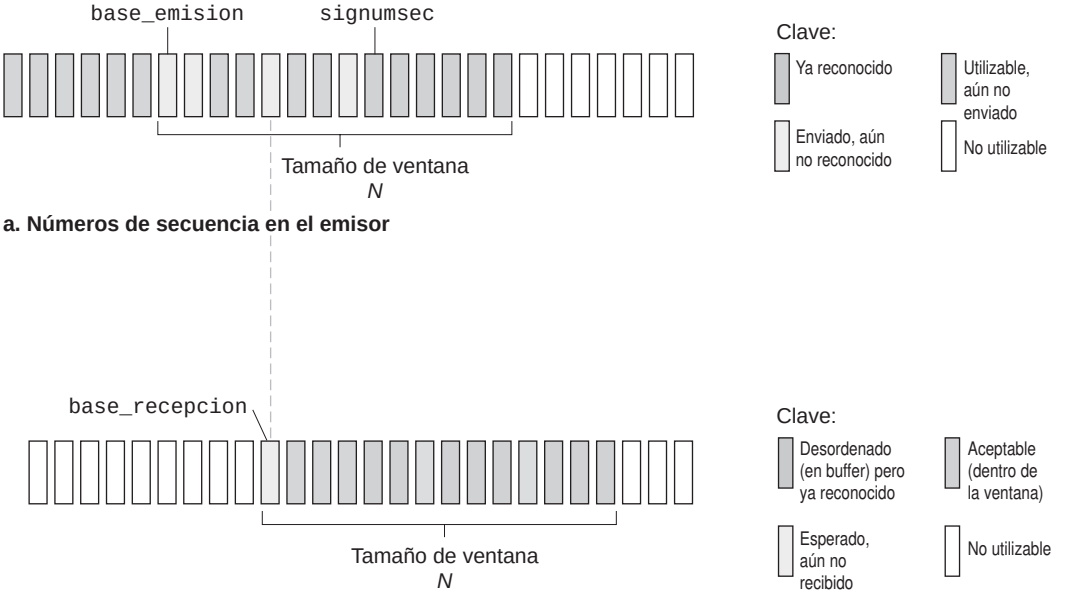


Figura 3.23 ♦ Espacios de números de secuencia del lado emisor y del lado receptor en el protocolo de repetición selectiva (SR).

1. *Datos recibidos de la capa superior*. Cuando se reciben datos de la capa superior, el emisor de SR comprueba el siguiente número de secuencia disponible para el paquete. Si el número de secuencia se encuentra dentro de la ventana del emisor, los datos se empaquetan y se envían; en caso contrario, bien se almacenan en el buffer o bien se devuelven a la capa superior para ser transmitidos más tarde, como en el caso del protocolo GBN.
2. *Fin de temporización*. De nuevo, se emplean temporizadores contra la pérdida de paquetes. Sin embargo, ahora, cada paquete debe tener su propio temporizador lógico, ya que solo se transmitirá un paquete al producirse el fin de la temporización. Se puede utilizar un mismo temporizador hardware para imitar el funcionamiento de varios temporizadores lógicos [Varghese 1997].
3. *ACK recibido*. Si se ha recibido un mensaje ACK, el emisor de SR marca dicho paquete como que ha sido recibido, siempre que esté dentro de la ventana. Si el número de secuencia del paquete es igual a `base_emision`, se hace avanzar la base de la ventana, situándola en el paquete no reconocido que tenga el número de secuencia más bajo. Si la ventana se desplaza y hay paquetes que no han sido transmitidos con números de secuencia que ahora caen dentro de la ventana, entonces esos paquetes se transmiten.

Figura 3.24 ♦ Sucesos y acciones en el lado emisor del protocolo SR.

1. *Se ha recibido correctamente un paquete cuyo número de secuencia pertenece al intervalo* `[base_recepcion, base_recepcion+N-1]`. En este caso, el paquete recibido cae dentro de la ventana del receptor y se devuelve al emisor un paquete ACK selectivo. Si el paquete no ha sido recibido con anterioridad, se almacena en el buffer. Si este paquete tiene un número de secuencia igual a la base de la ventana de recepción (`base_recepcion` en la Figura 3.22), entonces este paquete y cualquier paquete anteriormente almacenado en el buffer y numerado consecutivamente (comenzando por `base_recepcion`) se entregan a la capa superior. La ventana de recepción avanza entonces el número de paquetes suministrados entregados a la capa superior. Por ejemplo, en la Figura 3.26, cuando se recibe un paquete con el número de secuencia `base_recepcion = 2`, este y los paquetes 3, 4 y 5 pueden entregarse a la capa superior.
2. *Se ha recibido correctamente un paquete cuyo número de secuencia pertenece al intervalo* `[base_recepcion-N, base_recepcion-1]`. En este caso, se tiene que generar un mensaje ACK, incluso aunque ese paquete haya sido reconocido anteriormente por el receptor.
3. *En cualquier otro caso*. Ignorar el paquete.

Figura 3.25 ♦ Sucesos y acciones en el lado receptor del protocolo SR.

ejemplo del funcionamiento de SR en presencia de paquetes perdidos. Observe que, en esta figura, inicialmente el receptor almacena en el buffer los paquetes 3, 4 y 5, y luego los entrega, junto con el paquete 2, a la capa superior una vez que se ha recibido dicho paquete 2.

Es importante observar en el Paso 2 de la Figura 3.25 que el receptor vuelve a reconocer (en lugar de ignorar) los paquetes ya recibidos con determinados números de secuencia *inferiores* al número base actual de la ventana. Puede comprobar fácilmente que este doble reconocimiento es necesario. Por ejemplo, dados los espacios de números de secuencia del emisor y del receptor

de la Figura 3.23, si no hay ningún paquete ACK para el paquete `base_emision` propagándose desde el receptor al emisor, finalmente el emisor retransmitirá el paquete `base_emision`, incluso aunque esté claro (¡para nosotros, pero no para el emisor!) que el receptor ya ha recibido dicho paquete. Si el receptor no hubiera confirmado este paquete, la ventana del emisor nunca avanzaría. Este ejemplo ilustra un aspecto importante de los protocolos SR (y de otros muchos protocolos). El emisor y el receptor no siempre tienen una visión idéntica de lo que se ha recibido correctamente y de lo que no. En los protocolos SR, esto significa que las ventanas del emisor y del receptor no siempre coinciden.

La falta de sincronización entre las ventanas del emisor y del receptor tiene consecuencias importantes cuando nos enfrentamos con la realidad de un rango finito de números de secuencia. Por ejemplo, imagine lo que ocurriría con un rango de cuatro números de secuencia de paquete, 0, 1, 2, 3, y un tamaño de ventana de tres. Suponga que los paquetes 0 a 2 se transmiten y son recibidos correctamente y reconocidos por el receptor. En esta situación, la ventana del receptor se encontraría sobre los paquetes cuarto, quinto y sexto, que tienen los números de secuencia 3, 0 y 1, respectivamente. Ahora consideremos dos escenarios. En el primero, mostrado en la Figura 3.27(a), los mensajes ACK para los tres primeros paquetes se pierden y el emisor los retransmite. A continuación, el receptor recibe un paquete con el número de secuencia 0, una copia del primer paquete enviado.

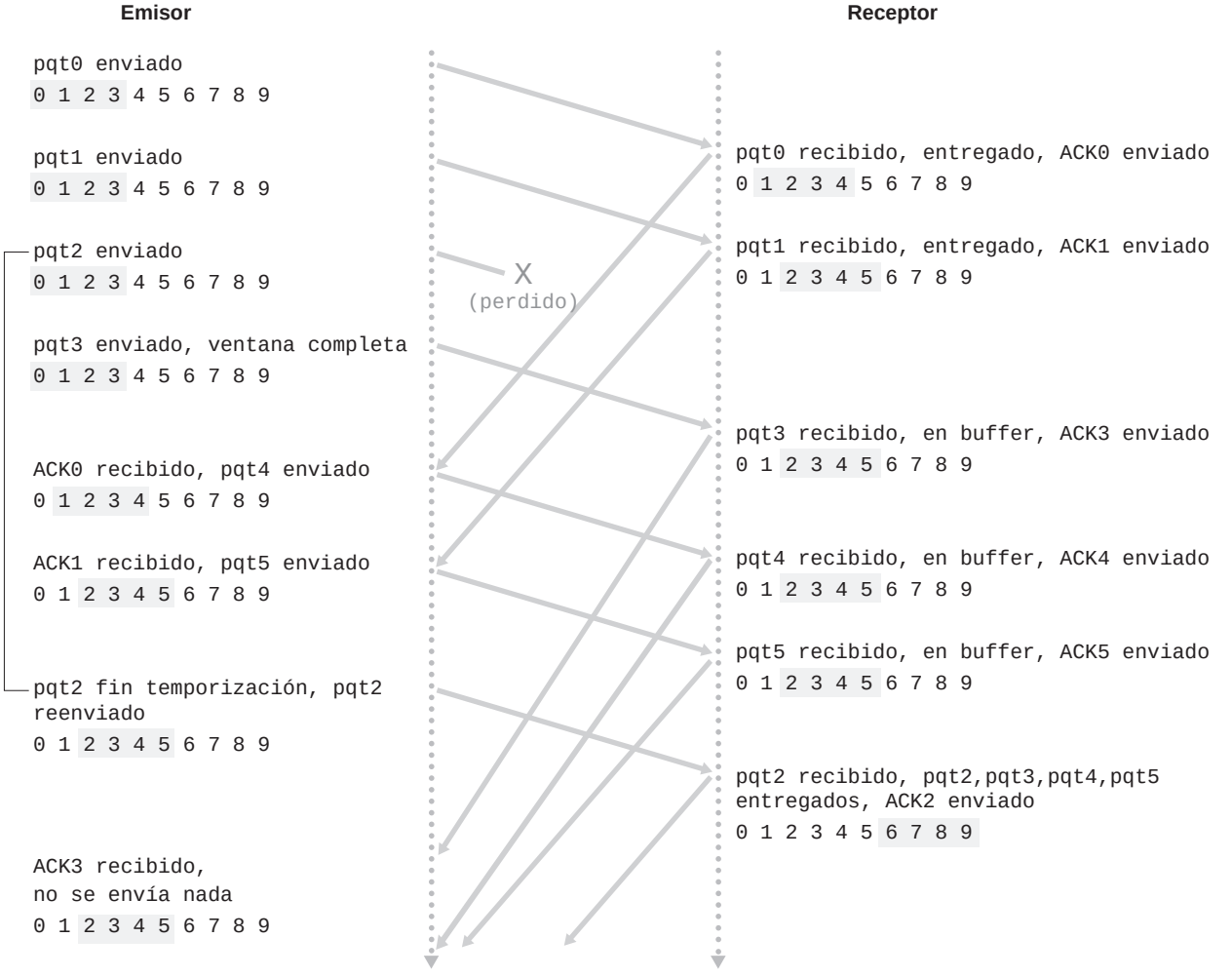
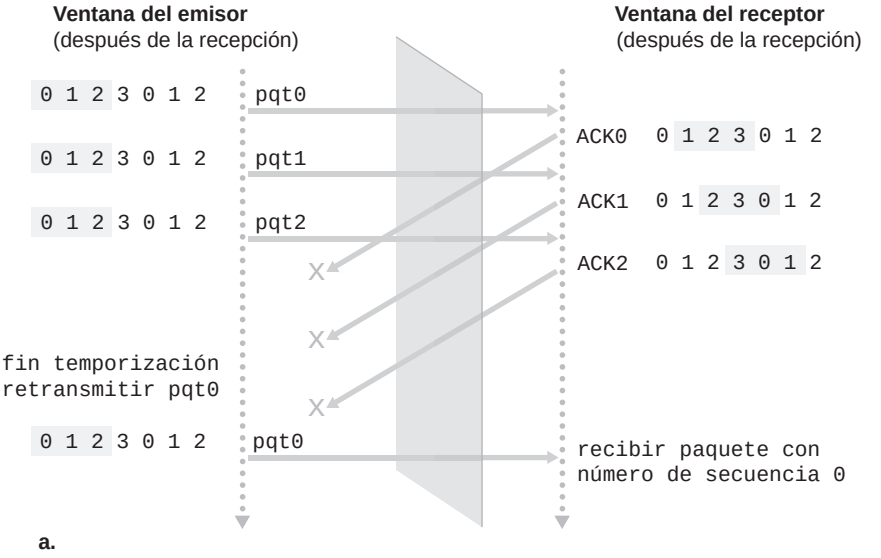


Figura 3.26 ♦ Funcionamiento del protocolo SR.

En el segundo escenario, mostrado en la Figura 3.27(b), los mensajes ACK correspondientes a los tres primeros paquetes se entregan correctamente. El emisor hace avanzar su ventana y envía los paquetes cuarto, quinto y sexto, con los número de secuencia 3, 0 y 1, respectivamente. El paquete con el número de secuencia 3 se pierde pero llega el paquete con el número de secuencia 0, un paquete que contiene datos *nuevos*.

Examinemos ahora el punto de vista del receptor en la Figura 3.27, el cual tiene delante una cortina imaginaria entre el emisor y el receptor, ya que el receptor no puede "ver" las acciones que lleva a cabo el emisor. Todo lo que ve el receptor es la secuencia de mensajes que recibe del canal y que él envía al mismo. En lo que respecta al receptor, los dos escenarios de la Figura 3.27 son *idénticos*. No hay forma de diferenciar la retransmisión del primer paquete de la transmisión inicial



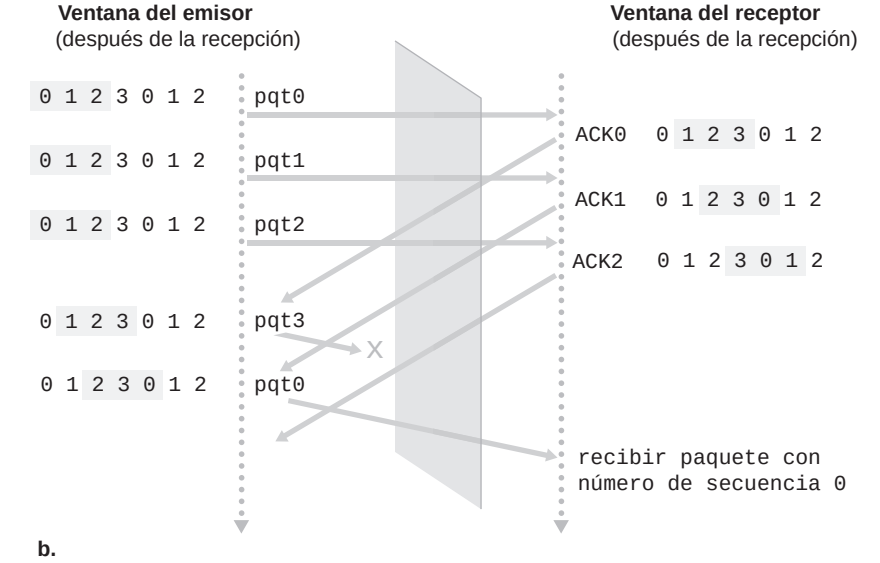


Figura 3.27 ♦ Dilema del receptor de los protocolos SR con ventanas demasiado grandes: ¿un nuevo paquete o una retransmisión?

del quinto paquete. Evidentemente, un tamaño de ventana que sea una unidad menor que el tamaño del espacio de números de secuencia no puede funcionar. Entonces, ¿cuál tiene que ser el tamaño de la ventana? Uno de los problemas incluidos al final del capítulo le pedirá que demuestre que el tamaño de la ventana tiene que ser menor o igual que la mitad del tamaño del espacio de números de secuencia en los protocolos SR.

En el sitio web de acompañamiento encontrará un applet que simula el funcionamiento del protocolo SR. Intente llevar a cabo los mismos experimentos que realizó con el applet de GBN. ¿Coinciden los resultados con los que cabría esperar?

Con esto hemos terminado con nuestra exposición sobre los protocolos fiables de transferencia de datos. Hemos visto *muchos* de los numerosos mecanismos básicos que contribuyen a proporcionar una transferencia de datos fiable. La Tabla 3.1 resume estos mecanismos. Ahora que ya hemos visto todos estos mecanismos en funcionamiento y que hemos adquirido una "visión de conjunto", le animamos a que repase esta sección con el fin de ver cómo estos mecanismos se fueron añadiendo para cubrir los modelos cada vez más complejos (y realistas) del canal que conecta al emisor y el receptor, o para mejorar el rendimiento de los protocolos.

Concluimos esta exposición sobre los protocolos de transferencia de datos fiables haciendo una suposición más sobre el modelo del canal subyacente. Recuerde que hemos supuesto que los paquetes no se pueden reordenar dentro del canal existente entre el emisor y el receptor. Generalmente, esta suposición es razonable cuando el emisor y el receptor están conectados simplemente mediante un cable físico. Sin embargo, cuando el "canal" que los conecta es una red puede tener lugar la reordenación de paquetes. Una manifestación de la reordenación de

Mecanismo	Uso, Comentarios
Suma de comprobación (Checksum)	Utilizada para detectar errores de bit en un paquete transmitido.
Temporizador	Se emplea para detectar el fin de temporización y retransmitir un paquete, posiblemente porque el paquete (o su mensaje ACK correspondiente) se ha perdido en el canal. Puesto que se puede producir un fin de temporización si un paquete está retardado pero no perdido (fin de temporización prematura), o si el receptor ha recibido un paquete pero se ha perdido el correspondiente ACK del receptor al emisor, puede ocurrir que el receptor reciba copias duplicadas de un paquete.
Número de secuencia	Se emplea para numerar secuencialmente los paquetes de datos que fluyen del emisor hacia el receptor. Los saltos en los números de secuencia de los paquetes recibidos permiten al receptor detectar que se ha perdido un paquete. Los paquetes con números de secuencia duplicados permiten al receptor detectar copias duplicadas de un paquete.
Reconocimiento (ACK)	El receptor utiliza estos paquetes para indicar al emisor que un paquete o un conjunto de paquetes ha sido recibido correctamente. Los mensajes de reconocimiento suelen contener el número de secuencia del paquete o los paquetes que están confirmando. Dependiendo del protocolo, los mensajes de reconocimiento pueden ser individuales o acumulativos.
Reconocimiento negativo (NAK)	El receptor utiliza estos paquetes para indicar al emisor que un paquete no ha sido recibido (NAK) correctamente. Normalmente, los mensajes de reconocimiento negativo contienen el número de secuencia de dicho paquete erróneo.
Ventana, procesamiento en cadena	El emisor puede estar restringido para enviar únicamente paquetes cuyo número de secuencia caiga dentro de un rango determinado. Permitiendo que se transmitan varios paquetes aunque no estén todavía reconocidos, se puede incrementar la tasa de utilización del emisor respecto al modo de operación de los protocolos de parada y espera. Veremos brevemente que el tamaño de la ventana se puede establecer basándose en la capacidad del receptor para recibir y almacenar en buffer los mensajes, o en el nivel de congestión de la red, o en ambos parámetros.

Tabla 3.1 ♦ Resumen de los mecanismos para la transferencia de datos fiable y su uso.

paquetes es que pueden aparecer copias antiguas de un paquete con un número de secuencia o de reconocimiento *x*, incluso aunque ni la ventana del emisor ni la del receptor contengan *x*. Con la reordenación de paquetes, puede pensarse en el canal como en un buffer que almacena paquetes y que espontáneamente puede transmitirlos en *cualquier* instante futuro. Puesto que los números de secuencia pueden reutilizarse, hay que tener cuidado para prevenir la aparición de esos paquetes duplicados. En la práctica, lo que se hace es asegurarse de que no se reutilice un número de secuencia hasta que el emisor esté "seguro" de que los paquetes enviados anteriormente con el número de secuencia *x* ya no se encuentran en la red. Esto se hace suponiendo que un paquete no puede "vivir" en la red durante más tiempo que un cierto periodo temporal máximo fijo. En las ampliaciones de TCP para redes de alta velocidad se supone un tiempo de vida máximo de paquete de aproximadamente tres minutos [RFC 1323]. [Sunshine 1978] describe un método para utilizar números de secuencia tales que los problemas de reordenación pueden ser eliminados por completo.

3.5 Transporte orientado a la conexión: TCP

Ahora que ya hemos visto los principios básicos de la transferencia de datos fiable, vamos a centrarnos en TCP, un protocolo de la capa de transporte de Internet, fiable y orientado a la conexión. En esta sección veremos que para proporcionar una transferencia de datos fiable, TCP confía en muchos de los principios básicos expuestos en la sección anterior, incluyendo los mecanismos de detección de errores, las retransmisiones, los reconocimientos acumulativos, los temporizadores y los campos de cabecera para los números de secuencia y de reconocimiento. El protocolo TCP está definido en los documentos RFC 793, RFC 1122, RFC 1323, RFC 2018 y RFC 2581.

3.5.1 La conexión TCP

Se dice que TCP está **orientado a la conexión** porque antes de que un proceso de la capa aplicación pueda comenzar a enviar datos a otro, los dos procesos deben primero "establecer una comunicación" entre ellos; es decir, tienen que enviarse ciertos segmentos preliminares para definir los parámetros de la transferencia de datos que van a llevar a cabo a continuación. Como parte del proceso de establecimiento de la conexión TCP, ambos lados de la misma iniciarán muchas variables de estado TCP (muchas de las cuales se verán en esta sección y en la Sección 3.7) asociadas con la conexión TCP.

La "conexión" TCP no es un circuito terminal a terminal con multiplexación TDM o FDM como lo es una red de conmutación de circuitos. En su lugar, la "conexión" es una conexión lógica, con un estado común que reside solo en los niveles TCP de los dos sistemas terminales que se comunican. Dado que el protocolo TCP se ejecuta únicamente en los sistemas terminales y no en los elementos intermedios de la red (routers y switches de la capa de enlace), los elementos intermedios de la red no mantienen el estado de la conexión TCP. De hecho, los routers intermedios son completamente inconscientes de las conexiones TCP; los routers ven los datagramas, no las conexiones.

Una conexión TCP proporciona un **servicio full-duplex**: si existe una conexión TCP entre el proceso A que se ejecuta en un host y el proceso B que se ejecuta en otro host, entonces los datos de la capa de aplicación pueden fluir desde el proceso A al proceso B en el mismo instante que los datos de la capa de aplicación fluyen del proceso B al proceso A. Una conexión TCP casi siempre es una conexión **punto a punto**, es decir, entre un único emisor y un único receptor. La "multidifusión" (véase el material disponible en línea para este texto), la transferencia de datos desde un emisor a muchos receptores en una única operación de envío, no es posible con TCP. Con TCP, dos hosts son compañía y tres multitud.

Veamos ahora cómo se establece una conexión TCP. Suponga que un proceso que se está ejecutando en un host desea iniciar una conexión con otro proceso que se ejecuta en otro host.

Recuerde que el proceso que inicia la conexión es el *proceso cliente*, y el otro proceso es el *proceso servidor*. El proceso de la aplicación cliente informa en primer lugar a la capa de transporte del cliente que desea establecer una conexión con un proceso del servidor. Recuerde que en la Sección 2.7.2, hemos visto un programa cliente en Python que hacía esto ejecutando el comando:

```
clientSocket.connect((serverName,serverPort))
```

donde `serverName` es el nombre del servidor y `serverPort` identifica al proceso del servidor. El proceso TCP en el cliente procede entonces a establecer una conexión TCP con el TCP del servidor. Al finalizar esta sección veremos en detalle el procedimiento de establecimiento de la conexión. Por el momento, nos basta con saber que el cliente primero envía un segmento TCP especial; el servidor responde con un segundo segmento TCP especial y, por último, el cliente responde de nuevo con un tercer segmento especial. Los dos primeros segmentos no transportan ninguna carga útil; es decir, no transportan datos de la capa de aplicación; el tercero de estos segmentos es el que puede llevar la carga útil. Puesto que los tres segmentos son intercambiados entre dos hosts, este procedimiento de establecimiento de la conexión suele denominarse **acuerdo en tres fases**.

Una vez que se ha establecido una conexión TCP, los dos procesos de aplicación pueden enviarse datos entre sí. Consideremos la transmisión de datos desde el proceso cliente al proceso servidor. El proceso cliente pasa un flujo de datos a través del socket (la puerta del proceso), como se ha descrito en la Sección 2.7. Una vez que los datos atraviesan la puerta, se encuentran en manos del protocolo TCP que se ejecuta en el cliente. Como se muestra en la Figura 3.28, TCP dirige estos datos al **buffer de emisión** de la conexión, que es uno de los buffers que se definen durante el proceso inicial del acuerdo en tres fases. De vez en cuando, TCP tomará fragmentos de datos del buffer de emisión y los pasa a la capa de red. La especificación de TCP [RFC 793] es bastante vaga en lo que respecta a especificar cuándo TCP debe realmente enviar los datos almacenados en el buffer, enunciando que TCP "debe transmitir esos datos en segmentos según su propia conveniencia". La cantidad máxima de datos que pueden cogerse y colocarse en un segmento está limitada por el **tamaño máximo de segmento** (**MSS, *Maximum Segment Size*)**. Normalmente,

HISTORIA

VINTON CERF, ROBERT KAHN Y TCP/IP

A principios de la década de 1970 comenzaron a proliferar las redes de conmutación de paquetes, siendo ARPAnet (la red precursora de Internet) solo una más de muchas redes. Cada una de estas redes utilizaba su propio protocolo. Dos investigadores, Vinton Cerf y Robert Kahn, se dieron cuenta de la importancia de interconectar estas redes e inventaron un protocolo interred denominado TCP/IP (*Transmission Control Protocol/Internet Protocol*). Aunque Cerf y Kahn comenzaron viendo el protocolo como una sola entidad, más tarde la dividieron en dos partes, TCP e IP, que operaban por separado. Cerf y Kahn publicaron un estudio sobre TCP/IP en mayo de 1974 en *IEEE Transactions on Communications Technology* [Cerf 1974].

El protocolo TCP/IP, que es la base de la Internet actual, fue diseñado antes que los PC, las estaciones de trabajo, los smartphones y las tabletas, antes de la proliferación de las tecnologías de las redes Ethernet, por cable, DSL, WiFi y otras tecnologías de red, y antes que la Web, las redes sociales y los flujos multimedia. Cerf y Kahn vieron la necesidad que existía de un protocolo de red que, por un lado, proporcionara un amplio soporte para las aplicaciones que ya estaban definidas y que, por otro lado, permitiera interoperar a los hosts y los protocolos de la capa de enlace.

En 2004, Cerf y Kahn recibieron el premio Turing Award de ACM, que está considerado como el "Premio Nobel de la Informática" por su trabajo pionero sobre los procesos de comunicación entre redes, incluyendo el diseño y la implementación de los protocolos básicos de comunicación de Internet (TCP/IP) y por su liderazgo en el mundo de las redes.

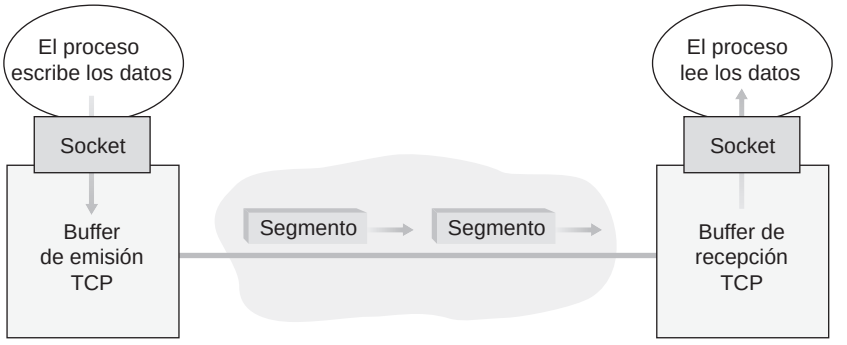


Figura 3.28 ♦ Buffers de emisión y recepción de TCP.

el MSS queda determinado en primer lugar por la longitud de la trama más larga de la capa de enlace que el host emisor local puede enviar [que es la **unidad máxima de transmisión**, (**MTU, *Maximum Transmission Unit***)], y luego el MSS se establece de manera que se garantice que un segmento TCP (cuando se encapsula en un datagrama IP) más la longitud de la cabecera TCP/IP (normalmente 40 bytes) se ajuste a una única trama de la capa de enlace. Los protocolos de la capa de enlace Ethernet y PPP tienen una MTU de 1.500 bytes. Un valor común de MTU es 1.460 bytes. También se han propuesto métodos para descubrir la MTU de la ruta (la trama más larga de la capa de enlace que puede enviarse a través de todos los enlaces desde el origen hasta el destino) [RFC 1191] y establecer el MSS basándose en el valor de la MTU de la ruta. Observe que el MSS es la cantidad máxima de datos de la capa de aplicación en el segmento, no el tamaño máximo del segmento TCP incluyendo las cabeceras. Esta terminología resulta confusa, pero tenemos que convivir con ella, porque está muy extendida.

TCP empareja cada fragmento de datos del cliente con una cabecera TCP, formando **segmentos TCP**. Los segmentos se pasan a la capa de red, donde son encapsulados por separado dentro de datagramas IP de la capa de red. Los datagramas IP se envían entonces a la red. Cuando TCP recibe un segmento en el otro extremo, los datos del mismo se colocan en el buffer de recepción de la conexión TCP, como se muestra en la Figura Figure 3.28. La aplicación lee el flujo de datos de este buffer. Cada lado de la conexión tiene su propio buffer de emisión y su propio buffer de recepción (puede ver el applet de control de flujo en línea en http://www.awl.com/kurose-ross, que proporciona una animación de los buffers de emisión y de recepción).

Por tanto, una conexión TCP consta de buffers, variables y un socket de conexión a un proceso en un host, y otro conjunto de buffers, variables y un socket de conexión a un proceso en otro host. Como hemos mencionado anteriormente, no se asignan ni buffers ni variables a la conexión dentro de los elementos de red (routers, switches y repetidores) existentes entre los hosts.

3.5.2 Estructura del segmento TCP

Después de haber visto de forma breve la conexión TCP, examinemos la estructura de un segmento TCP. El segmento TCP consta de campos de cabecera y un campo de datos. El campo de datos contiene un fragmento de los datos de la aplicación. Como hemos mencionado anteriormente, el MSS limita el tamaño máximo del campo de datos de un segmento. Cuando TCP envía un archivo grande, como por ejemplo una imagen como parte de una página web, normalmente divide el archivo en fragmentos de tamaño MSS (excepto el último fragmento, que normalmente será más pequeño que MSS). Sin embargo, las aplicaciones interactivas suelen transmitir fragmentos de datos que son más pequeños que el MSS; por ejemplo, en las aplicaciones de inicio de sesión remoto (*remote login*) como Telnet, el campo de datos del segmento TCP suele tener únicamente un byte. Puesto que habitualmente la cabecera de TCP tiene 20 bytes (12 bytes más que la cabecera de UDP), los segmentos enviados mediante Telnet solo pueden tener una longitud de 21 bytes.

La Figura 3.29 muestra la estructura del segmento TCP. Al igual que con UDP, la cabecera incluye los **números de puerto de origen y de destino**, que se utilizan para multiplexar y demultiplexar los datos de y para las aplicaciones de la capa superior. También, al igual que UDP, la cabecera incluye un **campo de suma de comprobación**. La cabecera de un segmento TCP también contiene los siguientes campos:

- El **campo Número de secuencia** de 32 bits y el **campo Número de reconocimiento** también de 32 bits son utilizados por el emisor y el receptor de TCP para implementar un servicio de transferencia de datos fiable, como se explica más adelante.
- El campo **Ventana de recepción** de 16 bits se utiliza para el control de flujo. Veremos en breve que se emplea para indicar el número de bytes que un receptor está dispuesto a aceptar.
- El **campo Longitud de cabecera** de 4 bits especifica la longitud de la cabecera TCP en palabras de 32 bits. La cabecera TCP puede tener una longitud variable a causa del campo opciones de TCP (normalmente, este campo está vacío, por lo que la longitud de una cabecera TCP típica es de 20 bytes).
- El **campo Opciones** es opcional y de longitud variable. Se utiliza cuando un emisor y un receptor negocian el tamaño máximo de segmento (MSS) o como un factor de escala de la ventana en las redes de alta velocidad. También se define una opción de marca temporal. Consulte los documentos RFC 854 y RFC 1323 para conocer detalles adicionales.
- El **campo Indicador** tiene 6 bits. El **bit ACK** se utiliza para indicar que el valor transportado en el campo de reconocimiento es válido; es decir, el segmento contiene un reconocimiento para un segmento que ha sido recibido correctamente. Los bits **RST**, **SYN** y **FIN** se utilizan para el establecimiento y cierre de conexiones, como veremos al final de esta sección. Los bits CWR y ECE se emplean en la notificación de congestión explícita, como veremos en la Sección 3.7.2. La activación del bit **PSH** indica que el receptor deberá pasar los datos a la capa superior de forma inmediata. Por último, el bit **URG** se utiliza para indicar que hay datos en este segmento que la entidad de la capa superior del lado emisor ha marcado como "urgentes". La posición de este último byte de estos datos urgentes se indica mediante el **campo puntero de datos urgentes** de 16 bits. TCP tiene que informar a la entidad de la capa superior del lado receptor si existen datos

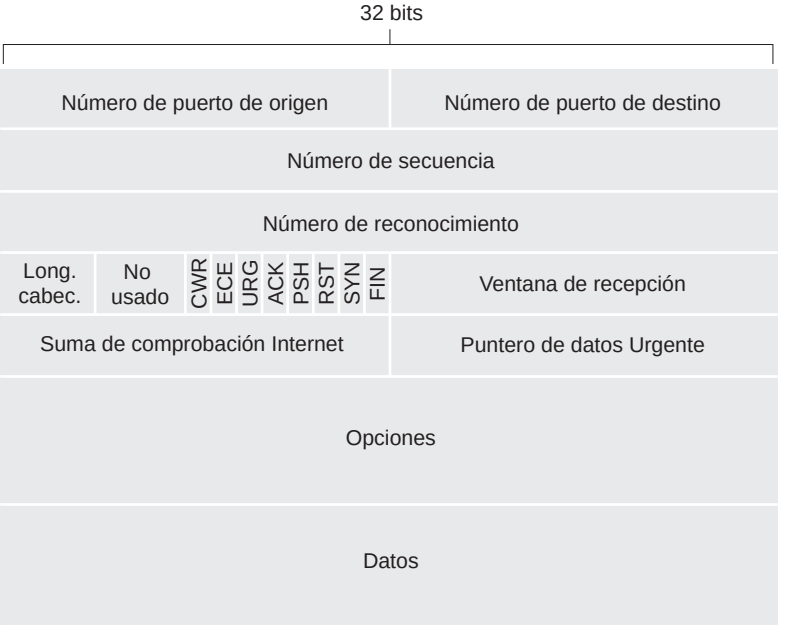


Figura 3.29 ♦ Estructura del segmento TCP.

urgentes y pasarle un puntero a la posición donde finalizan los datos urgentes. En la práctica, PSH, URG y el puntero a datos urgentes no se utilizan. Sin embargo, hemos hablado de estos campos con el fin de proporcionar al lector la información completa.

Nuestra experiencia como profesores es que nuestros estudiantes, en ocasiones, encuentran las exposiciones sobre los formatos de paquetes bastante áridas y quizá algo aburridas. Puede leer una exposición más divertida y amena sobre los campos de cabecera TCP, especialmente si le gustan los Legos™ como a nosotros, en [Pomeranz 2010].

Números de secuencia y números de reconocimiento

Dos de los campos más importantes de la cabecera de un segmento TCP son el campo número de secuencia y el campo número de reconocimiento. Estos campos son una parte crítica del servicio de transferencia de datos fiable de TCP. Pero antes de ver cómo se utilizan estos campos para proporcionar una transferencia de datos fiable, explicaremos en primer lugar lo que pone exactamente TCP en esos campos.

TCP percibe los datos como un flujo de bytes no estructurado pero ordenado. El uso que hace TCP de los números de secuencia refleja este punto de vista, en el sentido de que los números de secuencia hacen referencia al flujo de bytes transmitido y *no* a la serie de segmentos transmitidos. El **número de secuencia de un segmento** es por tanto el número del primer byte del segmento dentro del flujo de bytes. Veamos un ejemplo. Suponga que un proceso del host A desea enviar un flujo de datos a un proceso del host B a través de una conexión TCP. El protocolo TCP en el host A numerará implícitamente cada byte del flujo de datos. Suponga también que el flujo de datos consta de un archivo de 500.000 bytes, que el tamaño MSS es de 1.000 bytes y que el primer byte del flujo de datos está numerado como 0. Como se muestra en la Figura 3.30, TCP construye 500 segmentos a partir del flujo de datos. El primer segmento tiene asignado el número de secuencia 0, el segundo segmento tiene asignado el número de secuencia 1.000, el tercero tendrá asignado el número de secuencia 2.000, etc. Cada número de secuencia se inserta en el campo número de secuencia de la cabecera del segmento TCP apropiado.

Consideremos ahora los números de reconocimiento, que son algo más complicados que los números de secuencia. Recuerde que TCP es una conexión full-duplex, de modo que el host A puede estar recibiendo datos del host B mientras envía datos al host B (como parte de la misma conexión TCP). Todos los segmentos que llegan procedentes del host B tienen un número de secuencia para los datos que fluyen de B a A. *El número de reconocimiento que el host A incluye en su segmento es el número de secuencia del siguiente byte que el host A está esperando del host B.* Veamos algunos ejemplos para comprender qué es lo que ocurre aquí. Suponga que el host A ha recibido todos los bytes numerados de 0 a 535 procedentes de B y suponga también que está enviando un segmento al host B. El host A está esperando al byte 536 y todos los bytes que le siguen del flujo de datos del host B. Por tanto, el host A incluye 536 en el campo número de reconocimiento del segmento que envía a B.

Otro ejemplo: suponga que el host A ha recibido un segmento del host B que contiene los bytes de 0 a 535 y otro segmento que contiene los bytes de 900 a 1.000. Por alguna razón, el host A no ha

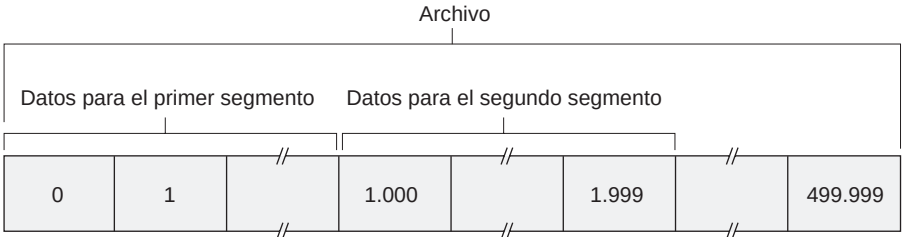


Figura 3.30 ♦ División de los datos del archivos en segmentos TCP.

recibido todavía los bytes de 536 a 899. En este ejemplo, el host A está esperando el byte 536 (y los que le siguen) para volver a crear el flujo de datos de B. Por tanto, el siguiente segmento de A a B contendrá el número 536 en el campo de número de reconocimiento. Dado que TCP solo confirma los bytes hasta el primer byte que falta en el flujo, se dice que TCP proporciona **reconocimientos acumulativos**.

Este último ejemplo plantea también un problema importante aunque sutil. El host A ha recibido el tercer segmento (bytes 900 a 1.000) antes de recibir el segundo segmento (bytes 536 a 899). Por tanto, el tercer segmento no ha llegado en orden. El sutil problema es el siguiente: ¿qué hace un host cuando no recibe los segmentos en orden a través de una conexión TCP? Curiosamente, los RFC dedicados a TCP no imponen ninguna regla y dejan la decisión a las personas que programan las implementaciones de TCP. Básicamente, tenemos dos opciones: (1) el receptor descarta de forma inmediata los segmentos que no han llegado en orden (lo que, como hemos anteriormente, puede simplificar el diseño del receptor) o (2) el receptor mantiene los bytes no ordenados y espera a que lleguen los bytes que faltan con el fin de rellenar los huecos. Evidentemente, esta última opción es más eficiente en términos de ancho de banda de la red, y es el método que se utiliza en la práctica.

En la Figura 3.30 hemos supuesto que el número de secuencia inicial era cero. En la práctica, ambos lados de una conexión TCP eligen aleatoriamente un número de secuencia inicial. Esto se hace con el fin de minimizar la posibilidad de que un segmento que todavía está presente en la red a causa de una conexión anterior que ya ha terminado entre dos hosts pueda ser confundido con un segmento válido de una conexión posterior entre esos dos mismos hosts (que también estén usando los mismos números de puerto que la conexión antigua) [Sunshine 1978].

Telnet: caso de estudio de los números de secuencia y de reconocimiento

Telnet, definido en el documento RFC 854, es un popular protocolo de la capa de aplicación utilizado para los inicios de sesión remotos. Se ejecuta sobre TCP y está diseñado para trabajar entre cualquier pareja de hosts. A diferencia de las aplicaciones de transferencia masiva de datos vistas en el Capítulo 2, Telnet es una aplicación interactiva. Vamos a ver aquí un ejemplo, ya que ilustra muy bien los números de secuencia y de reconocimiento de TCP. Debemos mencionar que actualmente muchos usuarios prefieren utilizar el protocolo SSH en lugar de Telnet, porque los datos enviados a través de una conexión Telnet (¡incluidas las contraseñas!) no están cifrados, lo que hace que Telnet sea vulnerable a los ataques de personas que quieran escuchar la conexión (como veremos en la Sección 8.7).

Supongamos que el host A inicia una sesión Telnet con el host B. Puesto que el host A inicia la sesión, se etiqueta como el cliente y el host B como el servidor. Cada carácter escrito por el usuario (en el cliente) se enviará al host remoto; el host remoto devolverá una copia de cada carácter, que será mostrada en la pantalla del usuario Telnet. Este “eco” se emplea para garantizar que los caracteres vistos por el usuario Telnet ya han sido recibidos y procesados en el sitio remoto. Por tanto, cada carácter atraviesa la red dos veces entre el instante en el que usuario pulsa una tecla y el instante en el que el carácter se muestra en el monitor del usuario.

Suponga ahora que el usuario escribe una única letra, la ‘C’, y luego se va a por un café. Examinemos los segmentos TCP que están siendo enviados entre el cliente y el servidor. Como se muestra en la Figura 3.31, suponemos que los números de secuencia iniciales para el cliente y el servidor son, respectivamente, 42 y 79. Recuerde que el número de secuencia de un segmento es el número de secuencia del primer byte del campo de datos. Por tanto, el primer segmento enviado por el cliente tendrá el número de secuencia 42 y el primer segmento enviado desde el servidor tendrá el número de secuencia 79. Recuerde que el número de reconocimiento es el número de secuencia del siguiente byte de datos que el host está esperando. Cuando ya se ha establecido una conexión TCP pero todavía no se enviado ningún dato, el cliente está esperando la llegada del byte 79 y el servidor está esperando al byte 42.

Como se muestra en la Figura 3.31, se envían tres segmentos. El primer segmento se transmite desde el cliente al servidor, conteniendo la representación ASCII de 1 byte de la letra 'C' en su campo de datos. Este primer segmento también contiene el número 42 en su campo número de secuencia, como ya hemos descrito. Además, dado que el cliente todavía no ha recibido ningún dato procedente del servidor, este primer segmento contendrá el número 79 en su campo número de reconocimiento.

El segundo segmento se envía desde el servidor al cliente y además sirve a un doble propósito. En primer lugar, proporciona un reconocimiento de los datos que ha recibido el servidor. Al incluir el número 43 en el campo número de reconocimiento, el servidor está diciendo al cliente que ha recibido correctamente todo hasta el byte 42 y ahora está esperando los bytes 43 y posteriores. El segundo propósito de este segmento es devolver el eco de la letra 'C'. Por tanto, el segundo segmento contiene la representación ASCII de la letra 'C' en su campo de datos. Este segundo segmento tiene el número de secuencia 79, el número de secuencia inicial del flujo de datos servidor-cliente de esta conexión TCP, ya que se trata del primer byte de datos que el servidor está enviando. Observe que la confirmación de los datos enviados desde el cliente al servidor se transporta en un segmento que contiene los datos enviados del servidor al cliente; se dice que este reconocimiento está **superpuesto** al segmento de datos enviado por el servidor al cliente.

El tercer segmento se envía desde el cliente al servidor. Su único propósito es confirmar los datos que ha recibido procedentes del servidor (recuerde que el segundo segmento contenía datos, la letra 'C', del servidor para el cliente). Este segmento tiene un campo de datos vacío (es decir, el paquete de reconocimiento no va superpuesto a los datos que van del cliente al servidor). El segmento contiene el número 80 en el campo número de reconocimiento porque el cliente ha recibido el flujo de bytes hasta el byte con el número de secuencia 79 y ahora está esperando los bytes 80 y subsiguientes. Es posible que le parezca extraño que este segmento también tenga un número de secuencia aunque no contenga datos pero, dado que los segmentos TCP disponen de un campo número de secuencia, es necesario incluir siempre en ese campo un valor.

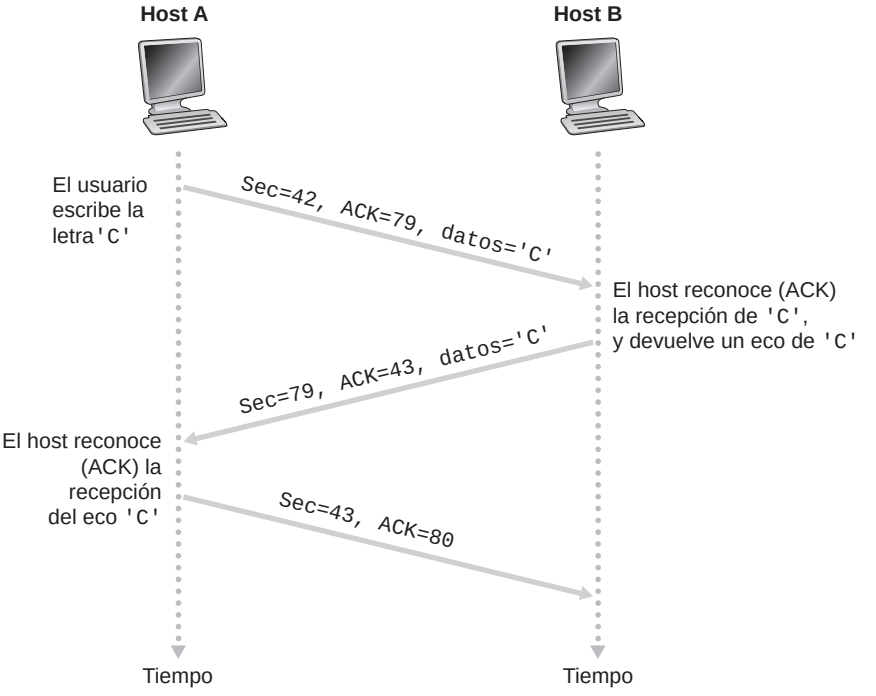


Figura 3.31 ♦ Números de secuencia y de reconocimiento en una aplicación Telnet simple sobre TCP.

3.5.3 Estimación del tiempo de ida y vuelta y fin de temporización

TCP, al igual que nuestro protocolo `rdt` de la Sección 3.4, utiliza un mecanismo de fin de temporización/retransmisión para recuperarse de la pérdida de segmentos. Aunque conceptualmente esto es muy simple, surgen muchos problemas sutiles al implementar dicho mecanismo en un protocolo real como, por ejemplo, TCP. Quizá la cuestión más obvia es la longitud de los intervalos de fin de temporización. Evidentemente, el intervalo de fin de temporización debería ser mayor que el tiempo de ida y vuelta (RTT) de la conexión; es decir, mayor que el tiempo que transcurre desde que se envía un segmento hasta que se recibe su reconocimiento. Si fuera de otra manera, se enviarían retransmisiones innecesarias. Pero, ¿cuánto mayor? y, ¿cómo se debería estimar el RTT por primera vez? ¿Debería asociarse un temporizador con cada uno de los segmentos no reconocidos? ¡Demasiadas preguntas! En esta sección vamos a basar nuestra exposición en el trabajo sobre TCP de [Jacobson 1988] y en las recomendaciones actuales del IETF para gestionar los temporizadores TCP [RFC 6298].

Estimación del tiempo de ida y vuelta

Comencemos nuestro estudio de la gestión del temporizador TCP viendo cómo estima TCP el tiempo de ida y vuelta entre el emisor y el receptor. Esto se lleva a cabo de la siguiente manera: el RTT de muestra, expresado como `RTTMuestra`, para un segmento es la cantidad de tiempo que transcurre desde que se envía el segmento (es decir, se pasa a IP) hasta que se recibe el correspondiente paquete de reconocimiento del segmento. En lugar de medir `RTTMuestra` para cada segmento transmitido, la mayor parte de las implementaciones TCP toman solo una medida de `RTTMuestra` cada vez. Es decir, en cualquier instante, `RTTMuestra` se estima a partir de uno solo de los segmentos transmitidos pero todavía no reconocidos, lo que nos proporciona un nuevo valor de `RTTMuestra` aproximadamente cada RTT segundos. Además, TCP nunca calcula `RTTMuestra` para un segmento que haya sido retransmitido; solo mide este valor para los segmentos que han sido transmitidos una vez [Karn 1987]. En uno de los problemas incluidos al final del capítulo se le pedirá que explique el por qué de este comportamiento.

Obviamente, los valores de `RTTMuestra` fluctuarán de un segmento a otro a causa de la congestión en los routers y a la variación de la carga en los sistemas terminales. A causa de esta fluctuación, cualquier valor de `RTTMuestra` dado puede ser atípico. Con el fin de estimar un RTT típico, es natural por tanto calcular algún tipo de promedio de los valores de `RTTMuestra`. TCP mantiene un valor promedio, denominado `RTTEstimado`, de los valores `RTTMuestra`. Para obtener un nuevo valor de `RTTMuestra`, TCP actualiza `RTTEstimado` según la fórmula siguiente:

$$\texttt{RTTEstimado} = (1 - \alpha) \cdot \texttt{RTTEstimado} + \alpha \cdot \texttt{RTTMuestra}$$

Hemos escrito esta fórmula como una instrucción de un lenguaje de programación (el nuevo valor de `RTTEstimado` es una combinación ponderada del valor anterior de `RTTEstimado` y del nuevo valor de `RTTMuestra`). El valor recomendado para α es $\alpha = 0{,}125$ (es decir, 1/8) [RFC 6298], en cuyo caso la fórmula anterior se expresaría como sigue:

$$\texttt{RTTEstimado} = 0{,}875 \cdot \texttt{RTTEstimado} + 0{,}125 \cdot \texttt{RTTMuestra}$$

Observe que `RTTEstimado` es una media ponderada de los valores de `RTTMuestra`. Como se examina en uno de los problemas de repaso incluidos al final del capítulo, esta media ponderada asigna un mayor peso a las muestras recientes que a las más antiguas. Esto es lógico, ya que las muestras más recientes reflejan mejor la congestión que existe actualmente en la red. En estadística, una media como esta se denomina **media móvil exponencialmente ponderada** (**EWMA, *Exponential Weighted Moving Average***). El término "exponencial" aparece en EWMA porque el peso de un valor dado de `RTTMuestra` disminuye exponencialmente tan rápido como tienen lugar

EN LA PRÁCTICA

TCP proporciona un servicio de transferencia de datos fiable utilizando mensajes de reconocimiento positivos y temporizadores, de forma muy similar a como hemos visto en la Sección 3.4. TCP confirma los datos que ha recibido correctamente y retransmite segmentos cuando estos o sus correspondientes reconocimientos se piensa que se han perdido o se han corrompido. Ciertas versiones de TCP también disponen de un mecanismo NAK implícito con un mecanismo rápido de retransmisión TCP: la recepción de tres ACK duplicados para un determinado segmento sirve como un NAK implícito para el siguiente segmento, provocando la retransmisión de dicho segmento antes del fin de la temporización. TCP utiliza secuencias de números para permitir al receptor identificar los segmentos perdidos o duplicados. Al igual que en el caso de nuestro protocolo de transferencia de datos fiable, `rdt3.0`, TCP no puede por sí mismo saber si un cierto segmento, o su correspondiente ACK, se ha perdido, está corrompido o se ha retardado demasiado. En el emisor, la respuesta TCP será la misma: retransmitir el segmento en cuestión.

TCP también utiliza el procesamiento en cadena, permitiendo al emisor tener múltiples segmentos transmitidos pero aun no reconocidos en cualquier instante. Anteriormente hemos visto que el procesamiento en cadena puede mejorar enormemente la tasa de transferencia de una sesión cuando la relación entre el tamaño del segmento y el retardo de ida y vuelta es pequeña. El número específico de segmentos pendientes no reconocidos que un emisor puede tener se determina mediante los mecanismos de control de congestión y de control de flujo de TCP. El control de flujo de TCP se examina al final de esta sección y el mecanismo de control de congestión en la Sección 3.7. Por el momento, basta con que seamos conscientes de que el emisor TCP utiliza el procesamiento en cadena.

las actualizaciones. En los problemas de repaso se le pedirá que deduzca el término exponencial en `RTTEstimado`.

La Figura 3.32 muestra los valores `RTTMuestra` y `RTTEstimado` para $\alpha = 1/8$ en una conexión TCP entre `gaia.cs.umass.edu` (en Amherst, Massachusetts) y `fantasia.eurecom.fr` (en el sur de Francia). Evidentemente, las variaciones en `RTTMuestra` se suavizan en el cálculo de `RTTEstimado`.

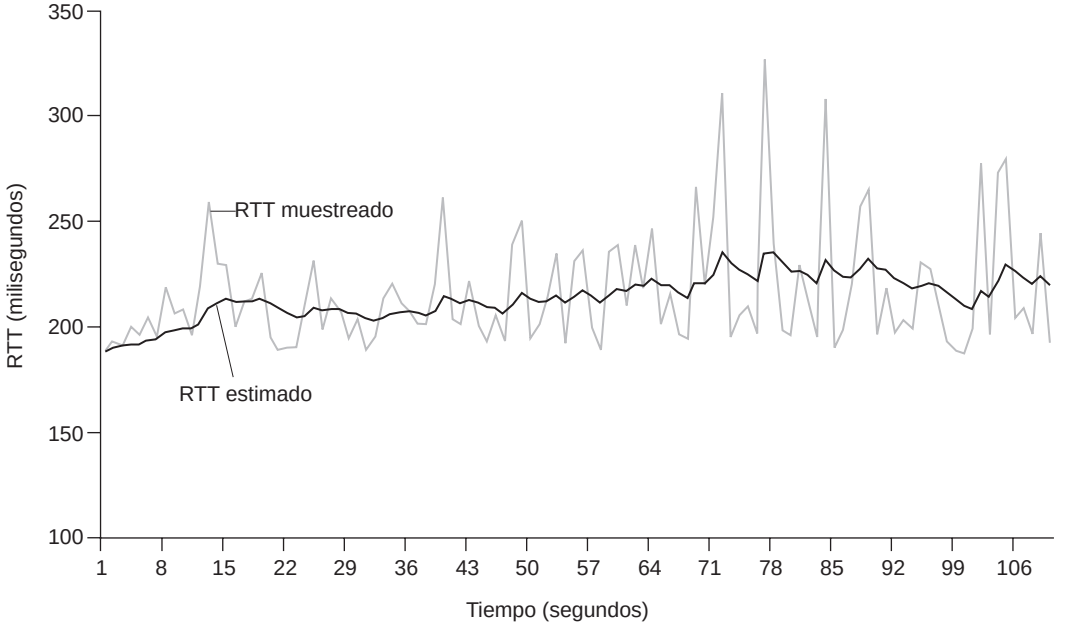


Figura 3.32 ♦ Muestreo de RTT y estimación de RTT.

Además de tener un estimado de RTT, también es importante disponer de una medida de la variabilidad de RTT. [RFC 6298] define la variación de RTT, `RTTDesv`, como una estimación de cuánto se desvía típicamente `RTTMuestra` de `RTTEstimado`:

$$\texttt{RTTDesv} = (1 - \beta) \cdot \texttt{RTTDesv} + \beta \cdot |\ \texttt{RTTMuestra} - \texttt{RTTEstimado}\ |$$

Observe que `RTTDesv` es una media EWMA de la diferencia entre `RTTMuestra` y `RTTEstimado`. Si los valores de `RTTMuestra` presentan una pequeña fluctuación, entonces `RTTDesv` será pequeño; por el contrario, si existe una gran fluctuación, `RTTDesv` será grande. El valor recomendado para β es 0,25.

Definición y gestión del intervalo de fin de temporización para la retransmisión

Dados los valores de `RTTEstimado` y `RTTDesv`, ¿qué valor debería utilizarse para el intervalo de fin de temporización de TCP? Evidentemente, el intervalo tendrá que ser mayor o igual que `RTTEstimado`, o se producirán retransmisiones innecesarias. Pero el intervalo de fin de temporización no debería ser mucho mayor que `RTTEstimado`; de otra manera, si un segmento se pierde, TCP no retransmitirá rápidamente el segmento, provocando retardos muy largos en la transferencia de datos. Por tanto, es deseable hacer el intervalo de fin de temporización igual a `RTTEstimado` más un cierto margen. El margen deberá ser más grande cuando la fluctuación en los valores de `RTTMuestra` sea grande y más pequeño cuando la fluctuación sea pequeña. El valor de `RTTDesv` deberá entonces tenerse en cuenta. Todas estas consideraciones se tienen en cuenta en el método TCP para determinar el intervalo de fin de temporización de las retransmisiones:

$$\texttt{IntervaloFinTemporizacion} = \texttt{RTTEstimado} + 4 \cdot \texttt{RTTDesv}$$

[RFC 6298] recomienda un valor inicial para `IntervaloFinTemporización` de 1 segundo. Además, cuando se produce un fin de temporización, el valor de `IntervaloFinTemporización` se duplica con el fin de evitar un fin de temporización prematuro para el segmento subsiguiente que enseguida será reconocido. Sin embargo, tan pronto como se recibe un segmento y se actualiza el valor de `RTTEstimado`, el `IntervaloFinTemporización` se calcula de nuevo utilizando la fórmula anterior.

3.5.4 Transferencia de datos fiable

Recuerde que el servicio de la capa de red de Internet (el servicio IP) no es fiable. IP no garantiza la entrega de los datagramas, no garantiza la entrega en orden de los datagramas y no garantiza la integridad de los datos contenidos en los datagramas. Con el servicio IP, los datagramas pueden desbordar los buffers de los routers y no llegar nunca a su destino, pueden llegar desordenados y los bits de un datagrama pueden corromperse (bascular de 0 a 1, y viceversa). Puesto que los segmentos de la capa de transporte son transportados a través de la red por los datagramas IP, estos segmentos de la capa de transporte pueden sufrir también estos problemas.

TCP crea un **servicio de transferencia de datos fiable** sobre el servicio de mejor esfuerzo pero no fiable de IP. El servicio de transferencia de datos fiable de TCP garantiza que el flujo de datos que un proceso extrae de su buffer de recepción TCP no está corrompido, no contiene huecos, ni duplicados y está en orden; es decir, el flujo de bytes es exactamente el mismo flujo que fue enviado por el sistema terminal existente en el otro extremo de la conexión. La forma en que TCP proporciona una transferencia de datos fiable implica muchos de los principios que hemos estudiado en la Sección 3.4.

En el anterior desarrollo que hemos realizado sobre las técnicas que proporcionan una transferencia de datos fiable, conceptualmente era fácil suponer que cada segmento transmitido pero aun no reconocido tenía asociado un temporizador individual. Aunque esto está bien en teoría, la gestión del temporizador puede requerir una sobrecarga considerable. Por tanto, los procedimientos

de gestión del temporizador TCP recomendados [RFC 6298] utilizan un *único* temporizador de retransmisión, incluso aunque haya varios segmentos transmitidos y aún no reconocidos. El protocolo TCP descrito en esta sección sigue esta recomendación de emplear un único temporizador.

Ahora vamos a ver cómo proporciona TCP el servicio de transferencia de datos fiable en dos pasos incrementales. En primer lugar, vamos a presentar una descripción extremadamente simplificada de un emisor TCP que solo emplea los fines de temporización para recuperarse de las pérdidas de segmentos; a continuación, veremos una descripción más completa que utiliza los mensajes de reconocimiento duplicados además de los fines de temporización. En la siguiente exposición suponemos que solo se están enviando datos en una única dirección, del host A al host B, y que el host A está enviando un archivo grande.

La Figura 3.33 presenta una descripción simplificada de un emisor TCP. Vemos que hay tres sucesos importantes relacionados con la transmisión y la retransmisión de datos en el emisor TCP: los datos recibidos desde la aplicación; el fin de temporización del temporizador y la recepción de paquetes ACK. Al producirse el primero de los sucesos más importantes, TCP recibe datos de la aplicación, encapsula los datos en un segmento y pasa el segmento a IP. Observe que cada segmento incluye un número de secuencia que es el número del primer byte de datos del segmento, dentro del flujo de datos, como se ha descrito en la Sección 3.5.2. Fíjese también en que si el temporizador no está ya funcionando para algún otro segmento, TCP lo inicia en el momento de pasar el segmento a IP (resulta útil pensar en el temporizador como si estuviera asociado con el segmento no reconocido más antiguo). El intervalo de caducidad para este temporizador es `IntervaloFinTemporizacion`, que se calcula a partir de `RTTEstimado` y `RTTDesv`, como se ha descrito en la Sección 3.5.3.

```
/* Suponga que el emisor no está restringido por los mecanismos de control de flujo ni de
control de congestión de TCP, que el tamaño de los datos procedentes de la capa superior es
menor que el MSS y que la transferencia de datos tiene lugar en un único sentido.*/

SigNumSec=NumeroSecuenciaInicial
BaseEmision=NumeroSecuenciaInicial

loop (siempre) {
    switch(suceso)

        suceso: datos recibidos de la aplicación de la capa superior
            crear segmento TCP con número de secuencia SigNumSec
            if (el temporizador no se está ejecutando actualmente)
                iniciar temporizador
            pasar segmento a IP
            SigNumSec=SigNumSec+longitud(datos)
            break;

        suceso: fin de temporización del temporizador
            retransmitir el segmento aún no reconocido con el
                número de secuencia más pequeño
            iniciar temporizador
            break;

        suceso: ACK recibido, con valor de campo ACK igual a y
            if (y > BaseEmision) {
                BaseEmision=y
                if (existen actualmente segmentos aún no reconocidos)
                    iniciar temporizador
                }
            break;

    } /* fin del bucle siempre */
```

Figura 3.33 ♦ Emisor TCP simplificado.

El segundo suceso importante es el fin de temporización. TCP responde a este suceso retransmitiendo el segmento que ha causado el fin de la temporización y, a continuación, reinicia el temporizador.

El tercer suceso importante que tiene que gestionar el emisor TCP es la llegada de un segmento de reconocimiento (ACK) procedente del receptor (más específicamente, un segmento que contenga un valor de campo ACK válido). Al ocurrir este suceso, TCP compara el valor ACK y con su variable `BaseEmision`. La variable de estado TCP `BaseEmision` es el número de secuencia del byte de reconocimiento más antiguo. (Por tanto, `BaseEmision-1` es el número de secuencia del último byte que se sabe que ha sido recibido correctamente y en orden en el receptor). Como hemos indicado anteriormente, TCP utiliza reconocimientos acumulativos, de modo que y confirma la recepción de todos los bytes anteriores al número de byte y. Si `y>BaseEmision`, entonces el ACK está confirmando uno o más de los segmentos no reconocidos anteriores. Así, el emisor actualiza su variable `BaseEmision` y reinicia el temporizador si actualmente aun existen segmentos no reconocidos.

Algunos escenarios interesantes

Acabamos de describir una versión enormemente simplificada de cómo TCP proporciona un servicio de transferencia de datos fiable. Pero incluso esta versión simplificada tiene sus sutilezas. Con el fin de clarificar cómo funciona este protocolo, vamos a analizar algunos escenarios sencillos. La Figura 3.34 describe el primero de ellos, en el que el host A envía un segmento al host B. Suponga que ese segmento tiene el número de secuencia 92 y contiene 8 bytes de datos. Después de enviar este segmento, el host A espera un segmento procedente de B con un número de reconocimiento de 100. Aunque el segmento de A se recibe en B, el paquete de reconocimiento de B a A se pierde. En este caso, se produce un suceso de fin de temporización y el host A retransmite el mismo segmento. Por supuesto, cuando el host B recibe la retransmisión, comprueba que de acuerdo con el número de secuencia el

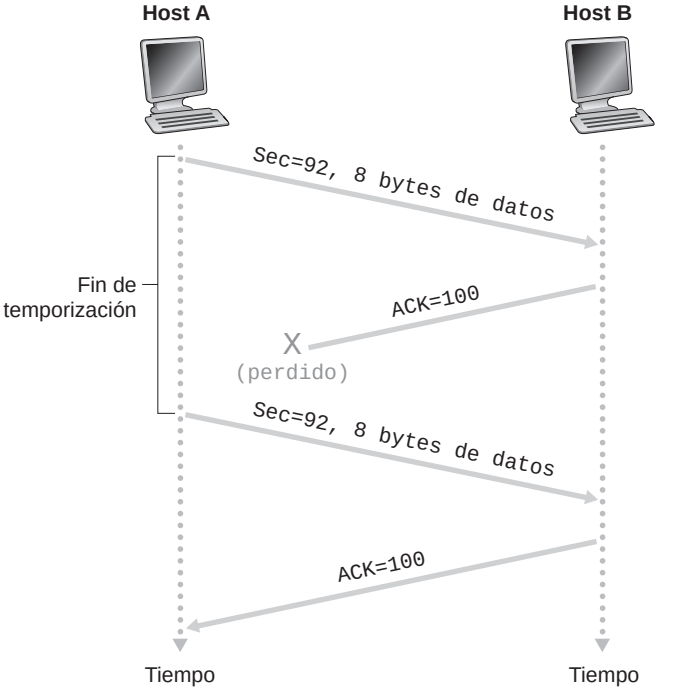


Figura 3.34 ♦ Retransmisión debida a la pérdida de un paquete de reconocimiento ACK.

segmento contiene datos que ya habían sido recibidos. Por tanto, TCP en el host B descartará los bytes del segmento retransmitido.

En el segundo escenario, mostrado en la Figura 3.35, el host A envía dos segmentos seguidos. El primer segmento tiene el número de secuencia 92 y 8 bytes de datos, y el segundo segmento tiene el número de secuencia 100 y 20 bytes de datos. Suponga que ambos segmentos llegan intactos a B, y que B envía dos mensajes de reconocimiento separados para cada uno de estos segmentos. El primero de estos mensajes tiene el número de reconocimiento 100 y el segundo el número 120. Suponga ahora que ninguno de estos mensajes llega al host A antes de que tenga lugar el fin de la temporización. Cuando se produce el fin de temporización, el host A reenvía el primer segmento con el número de secuencia 92 y reinicia el temporizador. Siempre y cuando el ACK correspondiente al segundo segmento llegue antes de que tenga lugar un nuevo fin de temporización, el segundo segmento no se retransmitirá.

En el tercer y último escenario, suponemos que el host A envía los dos segmentos del mismo modo que en el segundo ejemplo. El paquete de reconocimiento del primer segmento se pierde en la red, pero justo antes de que se produzca el fin de la temporización, el host A recibe un paquete de reconocimiento con el número de reconocimiento 120. Por tanto, el host A sabe que el host B ha recibido *todo* hasta el byte 119; por tanto, el host A no reenvía ninguno de los dos segmentos. Este escenario se ilustra en la Figura 3.36.

Duplicación del intervalo de fin de temporización

Examinemos ahora algunas de las modificaciones que aplican la mayor parte de las implementaciones TCP. La primera de ellas está relacionada con la duración del intervalo de fin de temporización después de que el temporizador ha caducado. En esta modificación, cuando tiene lugar un suceso de fin de temporización TCP retransmite el segmento aún no reconocido con el número de secuencia más pequeño, como se ha descrito anteriormente. Pero cada vez que TCP retransmite, define el

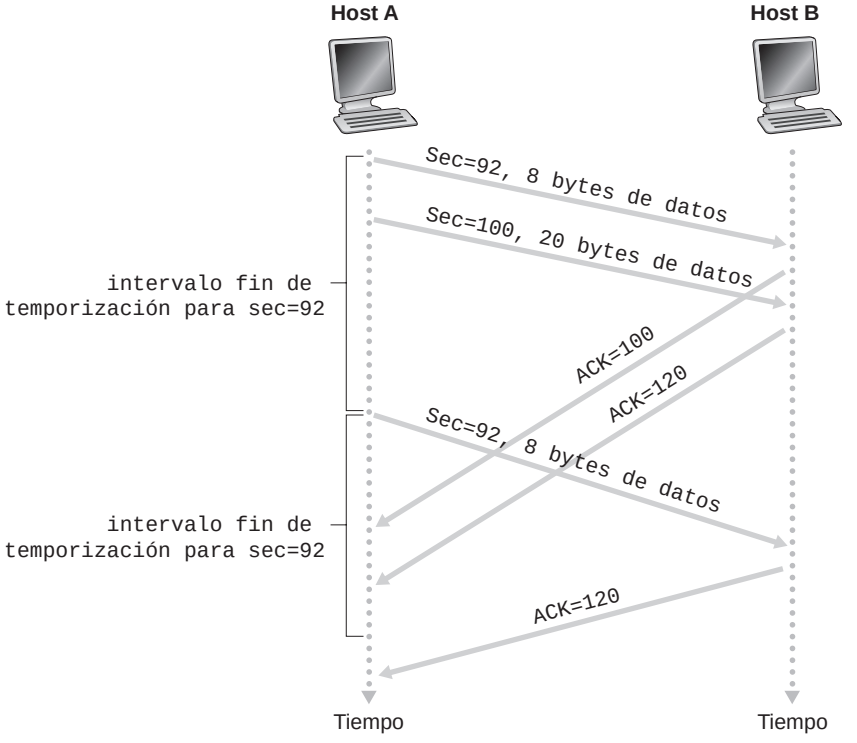


Figura 3.35 ♦ Segmento 100 no retransmitido.

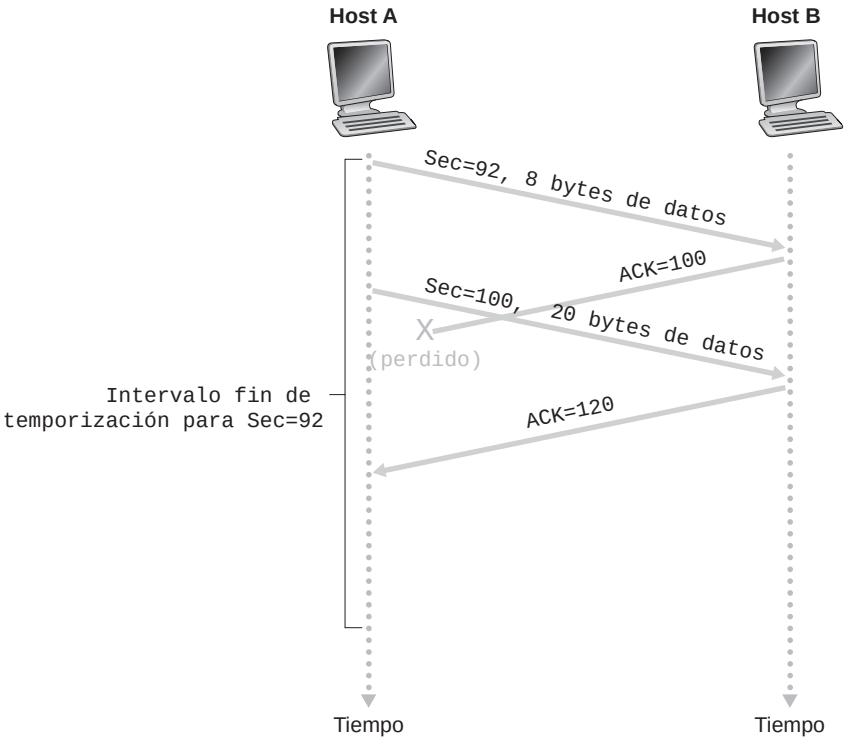


Figura 3.36 ♦ Un reconocimiento acumulativo evita la retransmisión del primer segmento.

siguiente intervalo de fin de temporización como dos veces el valor anterior, en lugar de obtenerlo a partir de los últimos valores de `RTTEstimado` y `RTTDesv` (como se ha descrito en la Sección 3.5.3). Por ejemplo, suponga que el `IntervaloFinTemporización` asociado con el segmento no reconocido más antiguo es 0,75 segundos cuando el temporizador caduca por primera vez. Entonces TCP retransmitirá este segmento y establecerá el nuevo intervalo en 1,5 segundos. Si el temporizador caduca de nuevo 1,5 segundos más tarde, TCP volverá a retransmitir este segmento, estableciendo el intervalo de caducidad en 3,0 segundos. Por tanto, los intervalos crecen exponencialmente después de cada retransmisión. Sin embargo, cuando el temporizador se inicia después de cualquiera de los otros dos sucesos (es decir, datos recibidos de la aplicación y recepción de un ACK), el `IntervaloFinTemporización` se obtiene a partir de los valores más recientes de `RTTEstimado` y `RTTDesv`.

Esta modificación proporciona una forma limitada de control de congestión (en la Sección 3.7 estudiaremos formas más exhaustivas de realizar el control de congestión en TCP). La caducidad del temporizador está causada muy probablemente por la congestión en la red, es decir, llegan demasiados paquetes a una (o más) colas de router a lo largo de la ruta entre el origen y el destino, haciendo que los paquetes sean descartados y/o sufran largos retardos de cola. Cuando existe congestión, si los orígenes continúan retransmitiendo paquetes persistentemente, la congestión puede empeorar. En lugar de ello, TCP actúa de forma más diplomática, haciendo que los emisores retransmitan después de intervalos cada vez más grandes. Cuando estudiemos CSMA/CD en el Capítulo 5, veremos que una idea similar se emplea en Ethernet .

Retransmisión rápida

Uno de los problemas con las retransmisiones generadas por los sucesos de fin de temporización es que el periodo de fin de temporización puede ser relativamente largo. Cuando se pierde un segmento, un periodo de fin de temporización largo fuerza al emisor a retardar el reenvío del paquete

perdido, aumentando el retardo terminal a terminal. Afortunadamente, el emisor puede a menudo detectar la pérdida de paquetes antes de que tenga lugar el suceso de fin de temporización, observando los ACK duplicados. Un **ACK duplicado** es un ACK que vuelve a reconocer un segmento para el que el emisor ya ha recibido un reconocimiento anterior. Para comprender la respuesta del emisor a un ACK duplicado, tenemos que entender en primer lugar por qué el receptor envía un ACK duplicado. La Tabla 3.2 resume la política de generación de mensajes ACK en el receptor TCP [RFC 5681]. Cuando un receptor TCP recibe un segmento con un número de secuencia que es mayor que el siguiente número de secuencia en orden esperado, detecta un hueco en el flujo de datos, es decir, detecta que falta un segmento. Este hueco podría ser el resultado de segmentos perdidos o reordenados dentro de la red. Dado que TCP no utiliza paquetes NAK, el receptor no puede devolver al emisor un mensaje de reconocimiento negativo explícito. En su lugar, simplemente vuelve a reconocer (es decir, genera un ACK duplicado) al último byte de datos en orden que ha recibido. Observe que la Tabla 3.2 contempla el caso de que el receptor no descarte los segmentos que no llegan en orden.

Puesto que un emisor suele enviar una gran número de segmentos seguidos, si se pierde uno de ellos, probablemente habrá muchos ACK duplicados seguidos. Si el emisor TCP recibe tres ACK duplicados para los mismos datos, toma esto como una indicación de que el segmento que sigue al segmento que ha sido reconocido tres veces se ha perdido (en los problemas de repaso, consideraremos la cuestión de por qué el emisor espera tres ACK duplicados, en lugar de un único duplicado). En el caso de que se reciban tres ACK duplicados, el emisor TCP realiza una **retransmisión rápida** [RFC 5681], reenviando el segmento que falta *antes* de que caduque el temporizador de dicho segmento. Esto se muestra en la Figura 3.37, en la que se pierde el segundo segmento y luego se retransmite antes de que caduque su temporizador. Para TCP con retransmisión rápida, el siguiente fragmento de código reemplaza al suceso de recepción de un ACK de la Figura 3.33:

```
suceso: ACK recibido, con un valor de campo ACK  de y
             if (y > BaseEmision) {
             BaseEmision=y
             if (existen segmentos pendientes de reconocimiento)
                iniciar temporizador
                }
             else {/* un ACK duplicado para un segmento
                    ya reconocido */
                incrementar el nº de mensajes ACK duplicados
                    recibidos para y
                if (número de mensajes ACK duplicados recibidos
                    para y==3)
                    /* TCP con retransmisión rápida */
                    reenviar el segmento con nº de secuencia y
                }
            break;
```

Anteriormente hemos mencionado que surgen muchos problemas sutiles cuando se implementa un mecanismo de fin de temporización/retransmisión en un protocolo real tal como TCP. Los procedimientos anteriores, que se han desarrollado como resultado de más de 20 años de experiencia con los temporizadores TCP, deberían convencerle de hasta que punto son sutiles esos problemas.

Retroceder N o Repetición selectiva

Profundicemos algo más en nuestro estudio del mecanismo de recuperación de errores de TCP considerando la siguiente cuestión: ¿Es TCP un protocolo GBN o un protocolo SR? Recuerde que los reconocimientos de TCP son acumulativos y que los segmentos correctamente recibidos pero no en orden no son reconocidos individualmente por el receptor. En consecuencia, como se muestra en la Figura 3.33 (véase también la Figura 3.19), el emisor TCP solo necesita mantener

Suceso	Acción del receptor TCP
Llegada de un segmento en orden con el número de secuencia esperado. Todos los datos hasta el número de secuencia esperado ya han sido reconocidos.	ACK retardado. Esperar hasta durante 500 milisegundos la llegada de otro segmento en orden. Si el siguiente segmento en orden no llega en este intervalo, enviar un ACK.
Llegada de un segmento en orden con el número de secuencia esperado. Hay otro segmento en orden esperando la transmisión de un ACK.	Enviar inmediatamente un único ACK acumulativo, reconociendo ambos segmentos ordenados.
Llegada de un segmento desordenado con un número de secuencia más alto que el esperado. Se detecta un hueco.	Enviar inmediatamente un ACK duplicado, indicando el número de secuencia del siguiente byte esperado (que es el límite inferior del hueco).
Llegada de un segmento que completa parcial o completamente el hueco existente en los datos recibidos.	Enviar inmediatamente un ACK, suponiendo que el segmento comienza en el límite inferior del hueco.

Tabla 3.2 ♦ Recomendación para la generación de mensajes ACK en TCP [RFC 5681].

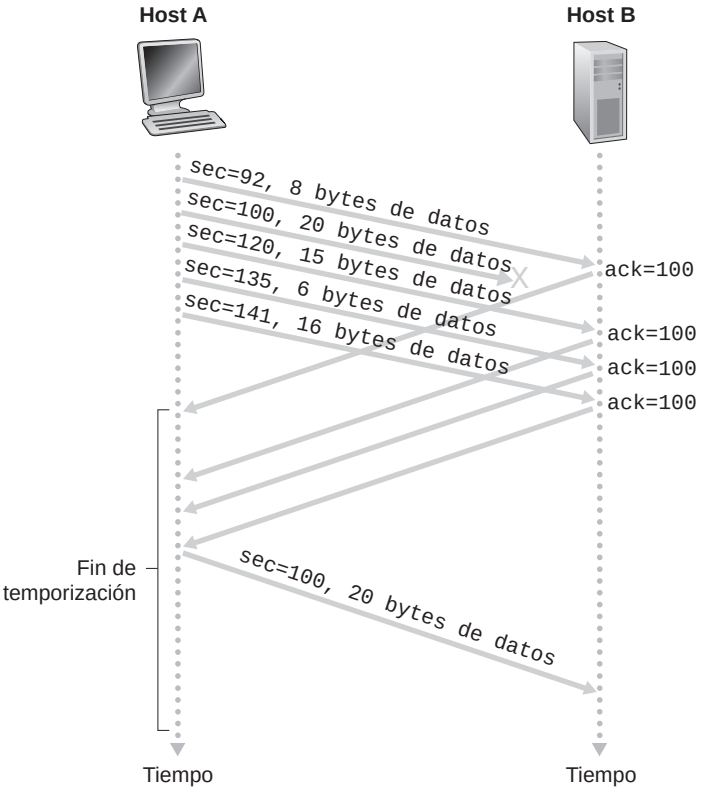


Figura 3.37 ♦ Retransmisión rápida: retransmisión del segmento que falta antes de que caduque el temporizador del segmento

el número de secuencia más pequeño de un byte transmitido pero no reconocido (`BaseEmision`) y el número de secuencia del siguiente byte que va a enviar (`SigNumSec`). En este sentido, TCP se parece mucho a un protocolo de tipo GBN. No obstante, existen algunas diferencia entre TCP y Retroceder N. Muchas implementaciones de TCP almacenan en buffer los segmentos recibidos correctamente pero no en orden [Stevens 1994]. Considere también lo que ocurre cuando el emisor envía una secuencia de segmentos 1, 2, . . . , N, y todos ellos llegan en orden y sin errores al receptor. Suponga además que el paquete de reconocimiento para el paquete $n < N$ se pierde, pero los $N - 1$ paquetes de reconocimiento restantes llegan al emisor antes de que tengan lugar sus

respectivos fines de temporización. En este ejemplo, GBN retransmitiría no solo el paquete *n*, sino también todos los paquetes subsiguientes $n + 1, n + 2, \ldots, N$. Por otro lado, TCP retransmitiría como mucho un segmento, el segmento *n*. Además, TCP no retransmitiría ni siquiera el segmento *n* si el reconocimiento para el segmento $n + 1$ llega antes del fin de temporización correspondiente al segmento *n*.

Una modificación propuesta para TCP es lo que se denomina **reconocimiento selectivo** [RFC 2018], que permite a un receptor TCP reconocer segmentos no ordenados de forma selectiva, en lugar de solo hacer reconocimientos acumulativos del último segmento recibido correctamente y en orden. Cuando se combina con la retransmisión selectiva (saltándose la retransmisión de segmentos que ya han sido reconocidos de forma selectiva por el receptor), TCP se comporta como nuestro protocolo SR selectivo. Por tanto, el mecanismo de recuperación de errores de TCP probablemente es mejor considerarlo como un híbrido de los protocolos GBN y SR.

3.5.5 Control de flujo

Recuerde que los hosts situados a cada lado de una conexión TCP disponen de un buffer de recepción para la conexión. Cuando la conexión TCP recibe bytes que son correctos y en secuencia, coloca los datos en el buffer de recepción. El proceso de aplicación asociado leerá los datos de este buffer, pero no necesariamente en el instante en que llegan. De hecho, la aplicación receptora puede estar ocupada con alguna otra tarea y puede incluso no leer los datos hasta mucho tiempo después de que estos hayan llegado. Si la aplicación es relativamente lenta en lo que respecta a la lectura de los datos, el emisor puede fácilmente desbordar el buffer de recepción de la conexión enviando muchos datos demasiado rápidamente.

TCP proporciona un **servicio de control de flujo** a sus aplicaciones para eliminar la posibilidad de que el emisor desborde el buffer del receptor. El control de flujo es por tanto un servicio de adaptación de velocidades (adapta la velocidad a la que el emisor está transmitiendo frente a la velocidad a la que la aplicación receptora está leyendo). Como hemos mencionado anteriormente, un emisor TCP también puede atascarse debido a la congestión de la red IP; esta forma de control del emisor se define como un mecanismo de **control de congestión**, un tema que exploraremos en detalle en las Secciones 3.6 y 3.7. Aunque las acciones tomadas por los controles de flujo y de congestión son similares (regular el flujo del emisor), obviamente se toman por razones diferentes. Lamentablemente, muchos autores utilizan los términos de forma indistinta, por lo que los lectores conocedores del tema deberían tratar de diferenciarlos. Examinemos ahora cómo proporciona TCP su servicio de control de flujo. En esta sección vamos a suponer que la implementación de TCP es tal que el receptor TCP descarta los segmentos que no llegan en orden.

TCP proporciona un servicio de control de flujo teniendo que mantiene el *emisor* una variable conocida como **ventana de recepción**. Informalmente, la ventana de recepción se emplea para proporcionar al emisor una idea de cuánto espacio libre hay disponible en el buffer del receptor. Puesto que TCP es una conexión full-duplex, el emisor de cada lado de la conexión mantiene una ventana de recepción diferente. Estudiemos la ventana de recepción en el contexto de una operación de transferencia de un archivo. Suponga que el host A está enviando un archivo grande al host B a través de una conexión TCP. El host B asigna un buffer de recepción a esta conexión; designamos al tamaño de este buffer como `BufferRecepcion`. De vez en cuando, el proceso de aplicación del host B lee el contenido del buffer. Definimos las siguientes variables:

- `UltimoByteLeido`: el número del último byte del flujo de datos del buffer leído por el proceso de aplicación del host B.
- `UltimoByteRecibido`: el número del último byte del flujo de datos que ha llegado procedente de la red y que se ha almacenado en el buffer de recepción del host B.

Puesto que en TCP no está permitido desbordar el buffer asignado, tenemos que:

$$\texttt{UltimoByteRecibido} - \texttt{UltimoByteLeido} \leq \texttt{BufferRecepcion}$$

La ventana de recepción, llamada `VentRecepcion`, se hace igual a la cantidad de espacio libre disponible en el buffer:

```
VentRecepcion = BufferRecepcion – [UltimoByteRecibido – UltimoByteLeido]
```

Dado que el espacio libre varía con el tiempo, `VentRecepcion` es una variable dinámica, la cual se ilustra en la Figura 3.38.

¿Cómo utiliza la conexión la variable `VentRecepcion` para proporcionar el servicio de control de flujo? El host B dice al host A la cantidad de espacio disponible que hay en el buffer de la conexión almacenando el valor actual de `VentRecepcion` en el campo ventana de recepción de cada segmento que envía a A. Inicialmente, el host B establece que `VentRecepcion = BufferRecepcion`. Observe que para poder implementar este mecanismo, el host B tiene que controlar diversas variables específicas de la conexión.

A su vez, el host A controla dos variables, `UltimoByteEnviado` y `UltimoByteReconocido`, cuyos significados son obvios. Observe que la diferencia entre estas dos variables, `UltimoByteEnviado – UltimoByteReconocido`, es la cantidad de datos no reconocidos que el host A ha enviado a través de la conexión. Haciendo que el número de datos no reconocidos sea inferior al valor de `VentRecepcion`, el host A podrá asegurarse de no estar desbordando el buffer de recepción en el host B. Por tanto, el host A se asegura, a todo lo largo del tiempo de vida de la conexión, de que:

```
UltimoByteEnviado – UltimoByteReconocido ≤ VentRecepcion
```

Existe un pequeño problema técnico con este esquema. Para ver de qué se trata, suponga que el buffer de recepción del host B está lleno, de manera que `VentRecepcion = 0`. Después de anunciar al host A que `VentRecepcion = 0`, suponga también que B no tiene *nada* que enviar a A. Veamos qué es lo que ocurre. A medida que el proceso de aplicación del host B vacía el buffer, TCP no envía nuevos segmentos con valores nuevos `VentRecepcion` al host A; por supuesto, TCP envía un segmento al host A solo si tiene datos que enviar o si tiene que enviar un paquete de reconocimiento. Por tanto, el host A nunca es informado de que hay algo de espacio en el buffer de recepción del host B (el host A está bloqueado y no puede transmitir más datos). Para resolver este problema, la especificación TCP requiere al host A que continúe enviando segmentos con un byte de datos cuando la longitud de la ventana de recepción de B es cero. Estos segmentos serán reconocidos por el receptor. Finalmente, el buffer comenzará a vaciarse y los ACK contendrán un valor de `VentRecepcion` distinto de cero.

El sitio web del libro en http://www.awl.com/kurose-ross proporciona un applet Java interactivo que ilustra el funcionamiento de la ventana de recepción de TCP.

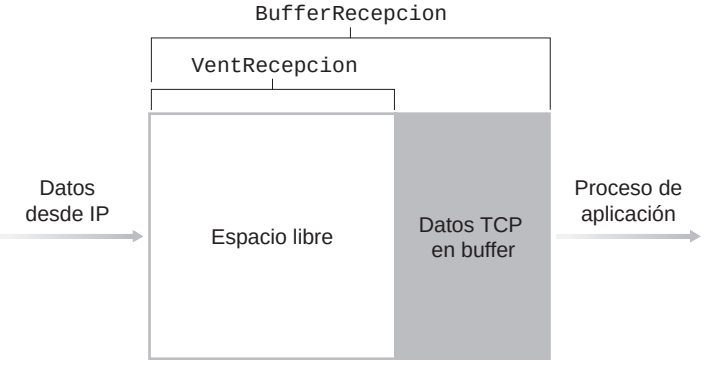


Figura 3.38 ♦ La ventana de recepción (`VentRecepcion`) y el buffer de recepción (`BufferRecepcion`).

Debemos comentar, una vez que hemos descrito el servicio de control de flujo de TCP, que UDP no proporciona ningún mecanismo de control de flujo y, en consecuencia, los segmentos pueden perderse en el receptor a causa del desbordamiento del buffer. Por ejemplo, considere la transmisión de una serie de segmentos UDP desde un proceso que se ejecuta en el host A a un proceso que se ejecuta en el host B. En una implementación típica de UDP, el protocolo UDP almacenará los segmentos en un buffer de tamaño finito que "precede" al correspondiente socket (es decir, la puerta de entrada al proceso). El proceso lee un segmento completo del buffer cada vez. Si el proceso no lee los segmentos del buffer lo suficientemente rápido, este se desbordará y los segmentos serán descartados.

3.5.6 Gestión de la conexión TCP

En esta subsección vamos a ver cómo se establece y termina una conexión TCP. Aunque este tema no parece particularmente emocionante, tiene una gran importancia, porque el establecimiento de una conexión TCP puede aumentar significativamente el retardo percibido (por ejemplo, cuando se navega por la Web). Además, muchos de los ataques de red más comunes, incluyendo el increíblemente popular ataque por inundación SYN, explotan las vulnerabilidades de la gestión de una conexión TCP. En primer lugar, veamos cómo se establece una conexión TCP. Suponga que hay un proceso en ejecución en un host (cliente) que desea iniciar una conexión con otro proceso que se ejecuta en otro host (servidor). El proceso de aplicación cliente informa en primer lugar al cliente TCP que desea establecer una conexión con un proceso del servidor. A continuación, el protocolo TCP en el cliente establece una conexión TCP con el protocolo TCP en el servidor de la siguiente manera:

- *Paso 1.* En primer lugar, TCP del lado del cliente envía un segmento TCP especial al TCP del lado servidor. Este segmento especial no contiene datos de la capa de aplicación. Pero uno de los bits indicadores de la cabecera del segmento (véase la Figura 3.29), el bit SYN, se pone a 1. Por esta razón, este segmento especial se referencia como un segmento SYN. Además, el cliente selecciona de forma aleatoria un número de secuencia inicial (`cliente_nsi`) y lo coloca en el campo número de secuencia del segmento TCP inicial SYN. Este segmento se encapsula dentro de un datagrama IP y se envía al servidor. Es importante que esta elección aleatoria del valor de `cliente_nsi` se haga apropiadamente con el fin de evitar ciertos ataques de seguridad [CERT 2001-09].
- *Paso 2.* Una vez que el datagrama IP que contiene el segmento SYN TCP llega al host servidor (¡suponiendo que llega!), el servidor extrae dicho segmento SYN del datagrama, asigna los buffers y variables TCP a la conexión y envía un segmento de conexión concedida al cliente TCP. (Veremos en el Capítulo 8 que la asignación de estos buffers y variables antes de completar el tercer paso del proceso de acuerdo en tres fases hace que TCP sea vulnerable a un ataque de denegación de servicio, conocido como ataque por inundación SYN.) Este segmento de conexión concedida tampoco contiene datos de la capa de aplicación. Sin embargo, contiene tres fragmentos de información importantes de la cabecera del segmento. El primero, el bit SYN se pone a 1. El segundo, el campo reconocimiento de la cabecera del segmento TCP se hace igual a `cliente_nsi+1`. Por último, el servidor elige su propio número de secuencia inicial (`servidor_nsi`) y almacena este valor en el campo número de secuencia de la cabecera del segmento TCP. Este segmento de conexión concedida está diciendo, en efecto, "He recibido tu paquete SYN para iniciar una conexión con tu número de secuencia inicial, `cliente_nsi`. Estoy de acuerdo con establecer esta conexión. Mi número de secuencia inicial es `servidor_nsi`". El segmento de conexión concedida se conoce como **segmento SYNACK**.
- *Paso 3.* Al recibir el segmento SYNACK, el cliente también asigna buffers y variables a la conexión. El host cliente envía entonces al servidor otro segmento; este último segmento confirma el segmento de conexión concedida del servidor (el cliente hace esto almacenando el valor `servidor_nsi+1` en el campo de reconocimiento de la cabecera del segmento TCP). El

bit SYN se pone a cero, ya que la conexión está establecida. Esta tercera etapa del proceso de acuerdo en tres fases puede transportar datos del cliente al servidor dentro de la carga útil del segmento.

Una vez completados estos tres pasos, los hosts cliente y servidor pueden enviarse entre sí segmentos que contengan datos. En cada uno de estos segmentos futuros, el valor del bit SYN será cero. Observe que con el fin de establecer la conexión se envían tres paquetes entre los dos hosts, como se ilustra en la Figura 3.39. Por ello, este procedimiento de establecimiento de la conexión suele denominarse proceso de **acuerdo en tres fases**. En los problemas de repaso se exploran varios aspectos de este proceso de TCP (¿Por qué se necesitan los números de secuencia iniciales? ¿Por qué se necesita un proceso de acuerdo en tres fases en lugar de uno en dos fases?). Es interesante observar que un escalador y la persona que le asegura (que se encuentra por debajo del escalador y cuya tarea es sostener la cuerda de seguridad del mismo) utilizan un protocolo de comunicaciones con un proceso de acuerdo en tres fases que es idéntico al de TCP, para garantizar que ambas partes estén preparadas antes de que el escalador inicie el ascenso.

Todo lo bueno se termina y esto es aplicable también a una conexión TCP. Cualquiera de los dos procesos participantes en una conexión TCP pueden dar por teminada dicha conexión. Cuando una conexión se termina, los "recursos" (es decir, los buffers y las variables) de los hosts se liberan. Por ejemplo, suponga que el cliente decide cerrar la conexión, como se muestra en la Figura 3.40. El proceso de la aplicación cliente ejecuta un comando de cierre. Esto hace que el cliente TCP envíe un segmento especial TCP al proceso servidor. Este segmento especial contiene un bit indicador en la cabecera del segmento, el bit FIN (véase la Figura 3.29), puesto a 1. Cuando el servidor recibe este segmento, devuelve al cliente un segmento de reconocimiento. El servidor entonces envía su propio segmento de desconexión, que tiene el bit FIN puesto a 1. Por último, el cliente reconoce el segmento de desconexión del servidor. En esta situación, los recursos de ambos hosts quedan liberados.

Mientras se mantiene una conexión TCP, el protocolo TCP que se ejecuta en cada host realiza transiciones a través de los diversos **estados TCP**. La Figura 3.41 ilustra una secuencia típica de

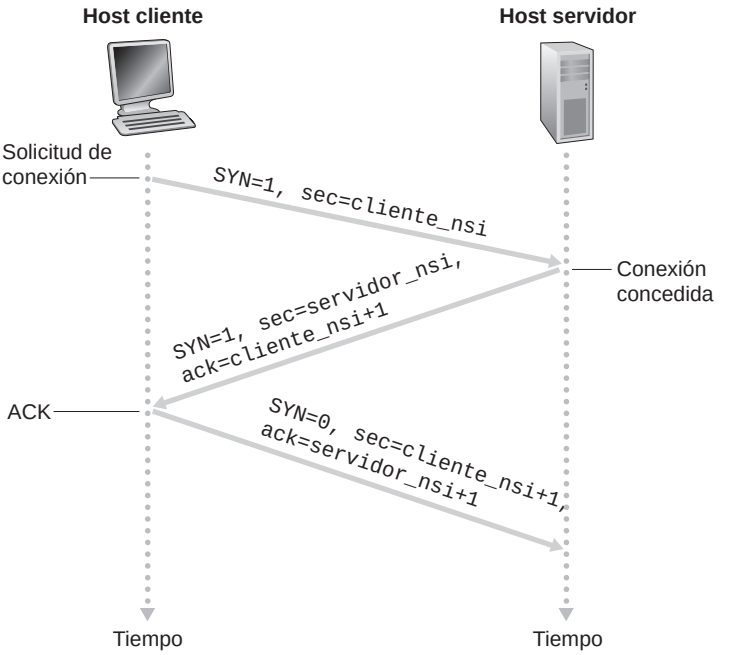


Figura 3.39 ♦ El proceso de acuerdo en tres fases de TCP: intercambio de segmentos.

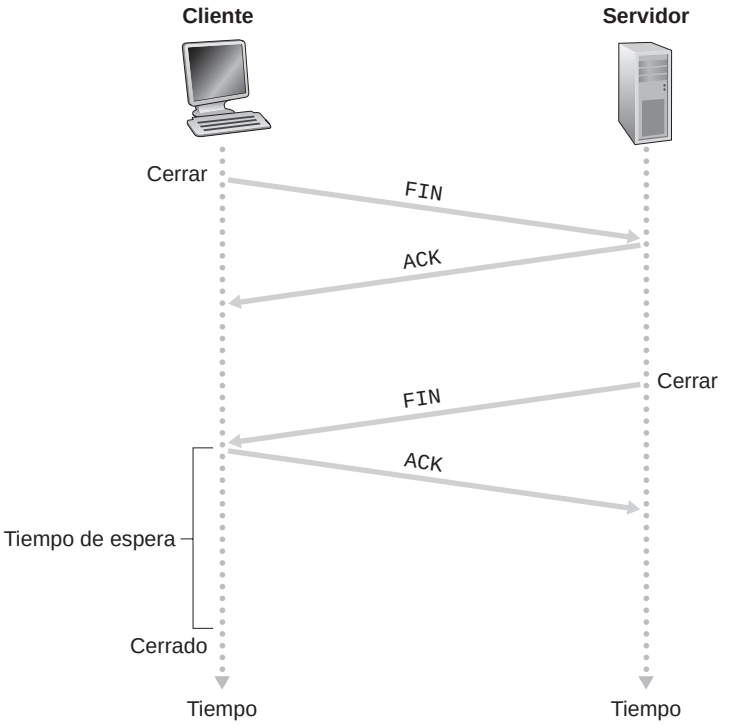


Figura 3.40 ♦ Cierre de una conexión TCP.

los estados TCP visitados por el *cliente* TCP, el cual se inicia en el estado CLOSED (cerrado). La aplicación en el lado del cliente inicia una nueva conexión (creando un objeto Socket en nuestros ejemplos de Java, así como en los ejemplos de Python del Capítulo 2). Esto hace que TCP en el cliente envíe un segmento SYN a TCP en el servidor. Después de haber enviado el segmento SYN, el cliente TCP entra en el estado SYN_SENT (SYN_enviado). Mientras se encuentra en este estado, el cliente TCP espera un segmento procedente del TCP servidor que incluya un reconocimiento del segmento anterior del cliente y que tenga el bit SYN puesto a 1. Una vez recibido ese segmento, el cliente TCP entra en el estado ESTABLISHED (establecido). Mientras está en este último estado, el cliente TCP puede enviar y recibir segmentos TCP que contengan datos de carga útil (es decir, datos generados por la aplicación).

Suponga que la aplicación cliente decide cerrar la conexión (tenga en cuenta que el servidor también podría decidir cerrar la conexión). Así, el cliente TCP envía un segmento TCP con el bit FIN puesto a 1 y entra en el estado FIN_WAIT_1 (FIN_espera_1). En este estado, el cliente TCP queda a la espera de un segmento TCP procedente del servidor que contenga un mensaje de reconocimiento. Cuando lo recibe, el cliente TCP pasa al estado FIN_WAIT_2. En este estado, el cliente espera a recibir otro segmento del servidor con el bit FIN puesto a 1; una vez recibido, el cliente TCP reconoce el segmento del servidor y pasa al estado TIME_WAIT, en el que puede reenviar al cliente TCP el reconocimiento final en caso de que el paquete ACK se pierda. El tiempo invertido en el estado TIME_WAIT es dependiente de la implementación, siendo sus valores típicos 30 segundos, 1 minuto y 2 minutos. Después de este tiempo de espera, la conexión se cierra y todos los recursos del lado del cliente (incluyendo los números de puerto) son liberados.

La Figura 3.42 ilustra la serie de estados que visita normalmente el TCP del lado del servidor, suponiendo que el cliente inicia el cierre de la conexión. Las transiciones se explican por sí mismas. En estos dos diagramas de transiciones de estados únicamente hemos mostrado cómo se establece y termina normalmente una conexión TCP. No hemos descrito lo que ocurre en determinados escenarios problemáticos; por ejemplo, cuando ambos lados de una conexión desean iniciar o

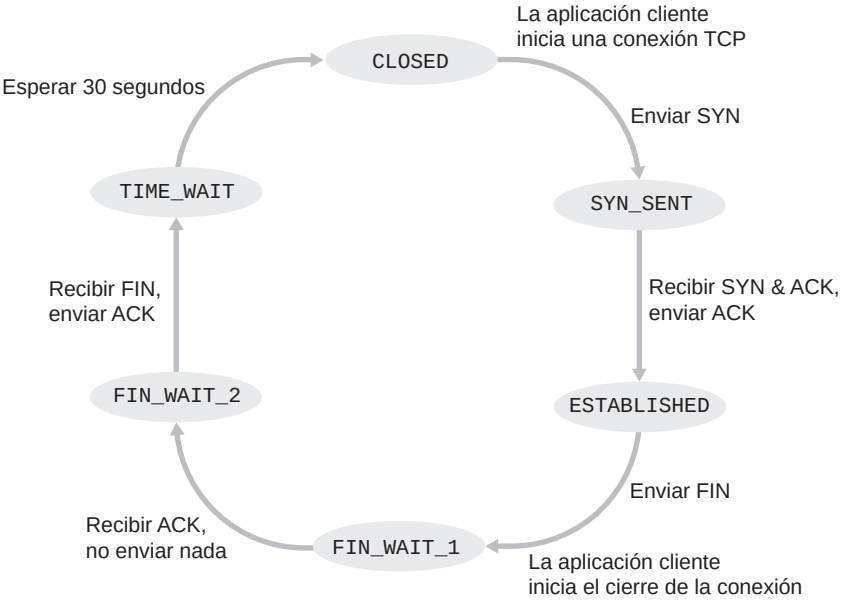


Figura 3.41 ♦ Secuencia típica de estados TCP visitados por un cliente TCP.

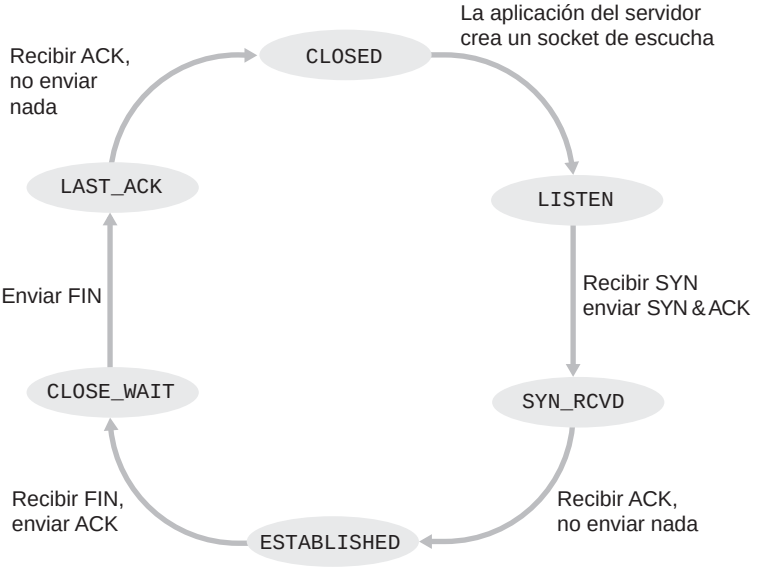


Figura 3.42 ♦ Secuencia típica de estados TCP visitados por un servidor TCP.

terminar al mismo tiempo la conexión. Si está interesado en este tema y en otros algo más avanzados sobre TCP, le animamos a consultar el exhaustivo libro de [Stevens 1994].

Hasta aquí, hemos supuesto que tanto el cliente como el servidor están preparados para comunicarse; es decir, que el servidor está escuchando en el puerto al que el cliente envía su segmento SYN. Consideremos lo que ocurre cuando un host recibe un segmento TCP cuyo número de puerto o cuya dirección IP de origen no se corresponde con ninguno de los sockets activos en el host. Por ejemplo, suponga que un host recibe un paquete TCP SYN cuyo puerto de destino es el número 80, pero el host no está aceptando conexiones en dicho puerto (es decir, no está ejecutando un servidor web en el puerto 80). Entonces, el host enviará al origen un segmento especial de

SEGURIDAD

EL ATAQUE POR INUNDACIÓN SYN

Hemos visto el proceso de acuerdo en tres fases de TCP en el que un servidor asigna e inicializa los buffers y variables de una conexión en respuesta a la recepción de un segmento SYN. A continuación, el servidor envía como respuesta un segmento SYNACK y espera al correspondiente segmento de reconocimiento (ACK) del cliente. Si el cliente no envía un segmento ACK para completar el tercero de los pasos del proceso de acuerdo en tres fases, al final (a menudo después de un minuto o más) el servidor terminará la conexión semiabierta y reclamará los recursos asignados.

Este protocolo de gestión de la conexión TCP establece la base para un ataque DoS clásico, conocido como **ataque por inundación SYN**. En este ataque, el atacante o atacantes envían un gran número de segmentos SYN TCP, sin completar el tercer paso del proceso de acuerdo. Con esta gran cantidad de segmentos SYN, los recursos de conexión del servidor pueden agotarse rápidamente a medida que se van asignando (¡aunque nunca se utilizan!) a conexiones semiabiertas. Con los recursos del servidor agotados, se niega el servicio a los clientes legítimos. Estos ataques por inundación SYN se encuentran entre los primeros ataques DoS documentados [CERT SYN 1996]. Afortunadamente, existe una defensa efectiva, denominada **cookies SYN** [RFC 4987], actualmente implantada en la mayoría de los principales sistemas operativos. Las cookies SYN funcionan del siguiente modo:

- Cuando el servidor recibe un segmento SYN, no sabe si ese segmento procede de un usuario legítimo o forma parte de un ataque por inundación SYN. Por tanto, el servidor en lugar de crear una conexión semiabierta TCP para ese segmento SYN, crea un número de secuencia TCP inicial que es una función compleja (una función hash) de las direcciones IP de origen y de destino y los números de puerto del segmento SYN, así como de un número secreto que únicamente conoce el servidor. Este número de secuencia inicial cuidadosamente confeccionado se denomina "cookie". El servidor tiene entonces que enviar al cliente un paquete SYNACK con este número de secuencia inicial especial. *Es importante que el servidor no recuerde la cookie ni ninguna otra información de estado correspondiente al paquete SYN.*
- Si el cliente es legítimo, entonces devolverá un segmento ACK. El servidor, al recibir este ACK, tiene que verificar que corresponde a algún SYN enviado anteriormente. ¿Cómo se hace esto si el servidor no mantiene memoria acerca de los segmentos SYN? Como ya habrá imaginado, se hace utilizando la cookie. Recuerde que para un segmento ACK legítimo, el valor contenido en el campo de reconocimiento es igual al número de secuencia inicial del segmento SYNACK (el valor de la cookie en este caso) más uno (véase la Figura 3.39). A continuación, el servidor puede ejecutar la misma función hash utilizando las direcciones IP de origen y de destino, los números de puerto en el segmento SYNACK (los cuales son los mismos que en el segmento SYN original) y el número secreto. Si el resultado de la función más uno es igual que el número de reconocimiento (cookie) del SYNACK del cliente, el servidor concluye que el ACK se corresponde con un segmento SYN anterior y, por tanto, lo valida. A continuación, el servidor crea una conexión completamente abierta ¡unto con un socket.
- Por el contrario, si el cliente no devuelve un segmento ACK, entonces el segmento SYN original no causa ningún daño al servidor, ya que este no le ha asignado ningún recurso.

reinicio. Este segmento TCP tiene el bit indicador RST (véase la Sección 3.5.2) puesto a 1. Por tanto, cuando un host envía un segmento de reinicio, le está diciendo al emisor "No tengo un socket para ese segmento. Por favor, no reenvies el segmento." Cuando un host recibe un paquete UDP en el que el número de puerto de destino no se corresponde con un socket UDP activo, el host envía un datagrama ICMP especial, como se explica en el Capítulo 5.

Ahora que ya tenemos unos buenos conocimientos sobre cómo se gestiona una conexión TCP, vamos a volver sobre la herramienta de exploración de puertos nmap y vamos a ver más

detalladamente cómo funciona. Para explorar un puerto TCP específico, por ejemplo, el puerto 6789, en un host objetivo, nmap enviará un segmento TCP SYN con el puerto de destino 6789 a dicho host. Pueden obtenerse entonces tres posibles resultados:

- *El host de origen recibe un segmento TCP SYNACK del host objetivo*. Dado que esto significa que hay una aplicación ejecutándose con TCP en el puerto 6789 del host objetivo, nmap devuelve "open" (puerto abierto).
- *El host de origen recibe un segmento TCP RST procedente del host objetivo*. Esto significa que el segmento SYN ha alcanzado el host objetivo, pero este no está ejecutando una aplicación con TCP en el puerto 6789. Pero el atacante sabrá como mínimo que el segmento destinado al puerto 6789 del host no está bloqueado por ningún cortafuegos existente en la ruta entre los hosts de origen y de destino. (Los cortafuegos se estudian en el Capítulo 8.)
- *El origen no recibe nada*. Probablemente, esto significa que el segmento SYN fue bloqueado por un cortafuegos intermedio y nunca llegó al host objetivo.

nmap es una potente herramienta que puede detectar "las brechas en el muro", no solo en lo relativo a los puertos TCP abiertos, sino también a los puertos UDP abiertos, a los cortafuegos y sus configuraciones e incluso a las versiones de aplicaciones y sistemas operativos. La mayor parte de esta tarea se lleva a cabo manipulando los segmentos de gestión de la conexión TCP [Skoudis 2006]. Puede descargar nmap en la dirección www.nmap.org.

Con esto completamos nuestra introducción a los mecanismos de control de errores y de control de flujo de TCP. En la Sección 3.7, volveremos sobre TCP y estudiaremos más detalladamente el control de congestión de TCP. Sin embargo, antes de eso, vamos a dar un paso atrás y vamos a examinar los problemas de control de congestión en un contexto más amplio.

3.6 Principios del control de congestión

En la sección anterior hemos examinado tanto los principios generales como los específicos de los mecanismos de TCP utilizados para proporcionar un servicio de transferencia de datos fiable en lo que se refiere a la pérdida de paquetes. Anteriormente habíamos mencionado que, en la práctica, tales pérdidas son normalmente el resultado de un desbordamiento de los buffers de los routers a medida que la red se va congestionando. La retransmisión de paquetes por tanto se ocupa de un síntoma de la congestión de red (la pérdida de un segmento específico de la capa de transporte) pero no se ocupa de la causa de esa congestión de la red (demasiados emisores intentando transmitir datos a una velocidad demasiado alta). Para tratar la causa de la congestión de la red son necesarios mecanismos que regulen el flujo de los emisores en cuanto la congestión de red aparezca.

En esta sección consideraremos el problema del control de congestión en un contexto general, con el fin de comprender por qué la congestión es algo negativo, cómo la congestión de la red se manifiesta en el rendimiento ofrecido a las aplicaciones de la capa superior y cómo pueden aplicarse diversos métodos para evitar la congestión de la red o reaccionar ante la misma. Este estudio de carácter general del control de la congestión es apropiado porque, puesto que junto con la transferencia de datos fiable, se encuentra al principio de la lista de los diez problemas más importantes de las redes. En la siguiente sección se lleva a cabo un estudio detallado sobre el algoritmo de control de congestión de TCP.

3.6.1 Las causas y los costes de la congestión

Iniciemos este estudio de carácter general sobre el control de congestión examinando tres escenarios con una complejidad creciente en los que se produce congestión. En cada caso, veremos en primer lugar por qué se produce la congestión y el coste de la misma (en términos de recursos no utilizados

por completo y del bajo rendimiento ofrecido a los sistemas terminales). Todavía no vamos a entrar a ver cómo reaccionar ante una congestión, o cómo evitarla, simplemente vamos a poner el foco sobre la cuestión más simple de comprender: qué ocurre cuando los hosts incrementan su velocidad de transmisión y la red comienza a congestionarse.

Escenario 1: dos emisores, un router con buffers de capacidad ilimitada

Veamos el escenario de congestión más simple posible: dos hosts (A y B), cada uno de los cuales dispone de una conexión que comparte un único salto entre el origen y el destino, como se muestra en la Figura 3.43.

Supongamos que la aplicación del host A está enviando datos a la conexión (por ejemplo, está pasando datos al protocolo de la capa de transporte a través de un socket) a una velocidad media de λ_{in} bytes/segundo. Estos datos son originales en el sentido de que cada unidad de datos se envía solo una vez al socket. El protocolo del nivel de transporte subyacente es un protocolo simple. Los datos se encapsulan y se envían; no existe un mecanismo de recuperación de errores (como por ejemplo, las retransmisiones), ni se realiza un control de flujo ni un control de congestión. Ignorando la sobrecarga adicional debida a la adición de la información de cabecera de las capas de transporte e inferiores, la velocidad a la que el host A entrega tráfico al router en este primer escenario es por tanto λ_{in} bytes/segundo. El host B opera de forma similar, y suponemos, con el fin de simplificar, que también está enviando datos a una velocidad de λ_{in} bytes/segundo. Los paquetes que salen de los hosts A y B atraviesan un router y un enlace de salida compartido de capacidad *R*. El router tiene buffers que le permiten almacenar paquetes entrantes cuando la tasa de llegada de paquetes excede la capacidad del enlace de salida. En este primer escenario, suponemos que el router tiene un buffer con una cantidad de espacio infinita.

La Figura 3.44 muestra el rendimiento de la conexión del host A en este primer escenario. La gráfica de la izquierda muestra **la tasa de transferencia por conexión** (número de bytes por segundo en el receptor) como una función de la velocidad de transmisión de la conexión. Para una velocidad de transmisión comprendida entre 0 y *R*/2, la tasa de transferencia en el receptor es igual a la velocidad de transmisión en el emisor (todo lo que envía el emisor es recibido en el receptor con un retardo finito). Sin embargo, cuando la velocidad de transmisión es mayor que *R*/2, la tasa de transferencia es de solo *R*/2. Este límite superior de la tasa de transferencia es una consecuencia de compartir entre dos conexiones la capacidad del enlace. El enlace simplemente no puede proporcionar paquetes a un receptor a una velocidad de régimen permanente que sea mayor que *R*/2. Independientemente de lo altas que sean las velocidades de transmisión de los hosts A y B, nunca verán una tasa de transferencia mayor que *R*/2.

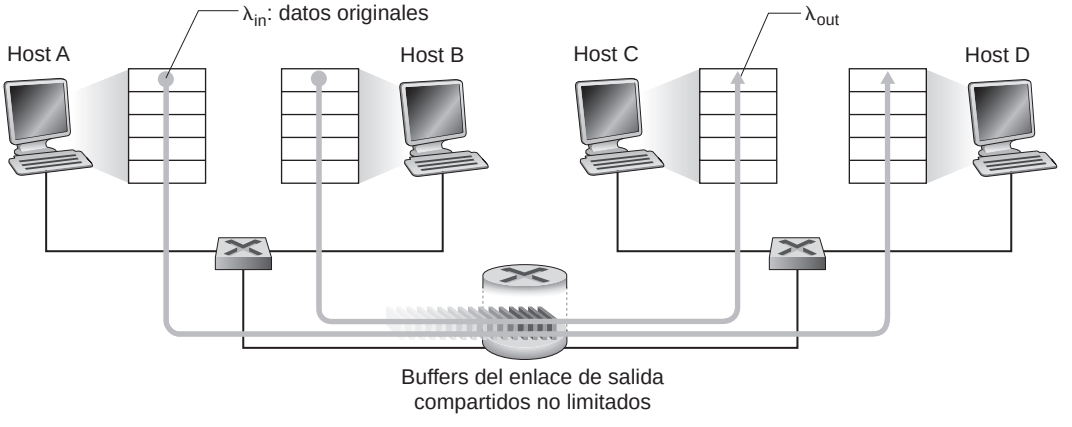


Figura 3.43 ♦ Escenario de congestión 1: dos conexiones que comparten un único salto con buffers de capacidad ilimitada.

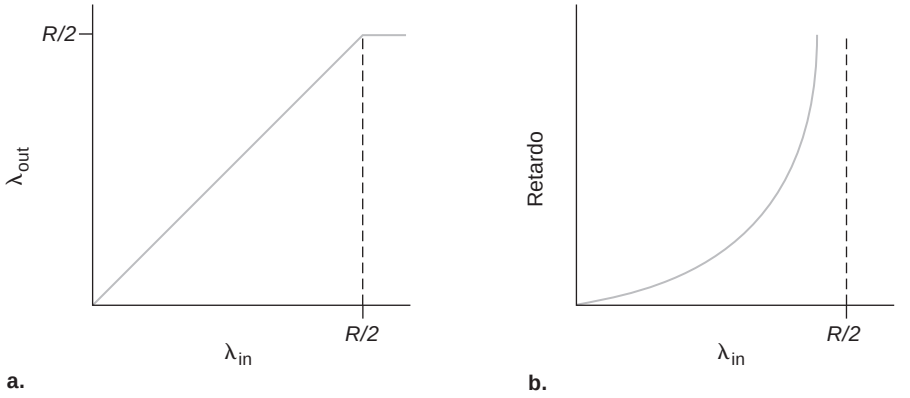


Figura 3.44 ♦ Escenario de congestión 1: tasa de transferencia y retardo en función de la velocidad de transmisión del host.

Alcanzar una tasa de transferencia por conexión de *R*/2 podría realmente parecer algo positivo, porque el enlace se utiliza completamente en suministrar paquetes a sus destinos. Sin embargo, la gráfica de la derecha de la Figura 3.44 muestra la consecuencia de operar cerca de la capacidad del enlace. A medida que la velocidad de transmisión se aproxima a *R*/2 (desde la izquierda), el retardo medio se hace cada vez más grande. Cuando la velocidad de transmisión excede de *R*/2, el número medio de paquetes en cola en el router no está limitado y el retardo medio entre el origen y el destino se hace infinito (suponiendo que las conexiones operan a esas velocidades de transmisión durante un periodo de tiempo infinito y que existe una cantidad infinita de espacio disponible en el buffer). Por tanto, aunque operar a una tasa de transferencia agregada próxima a *R* puede ser ideal desde el punto de vista de la tasa de transferencia, no es ideal en absoluto desde el punto de vista del retardo. *Incluso en este escenario (extremadamente) idealizado, ya hemos encontrado uno de los costes de una red congestionada: los grandes retardos de cola experimentados cuando la tasa de llegada de los paquetes se aproxima a la capacidad del enlace.*

Escenario 2: dos emisores y un router con buffers finitos

Ahora vamos a modificar ligeramente el escenario 1 de dos formas (véase la Figura 3.45). En primer lugar, suponemos que el espacio disponible en los buffers del router es finito. Una consecuencia de esta suposición aplicable en la práctica es que los paquetes serán descartados cuando lleguen a un buffer que ya esté lleno. En segundo lugar, suponemos que cada conexión es fiable. Si un paquete que contiene un segmento de la capa de transporte se descarta en el router, el emisor tendrá que retransmitirlo. Dado que los paquetes pueden retransmitirse, ahora tenemos que ser más cuidadosos al utilizar el término *velocidad de transmisión*. Específicamente, vamos a designar la velocidad a la que la aplicación envía los datos originales al socket como λ_{in} bytes/segundo. La velocidad a la que la capa de transporte envía segmentos (que contienen los datos originales *y* los datos retransmitidos) a la red la denotaremos como λ'_{in} bytes/segundo. En ocasiones, λ'_{in} se denomina **carga ofrecida** a la red.

El rendimiento de este segundo escenario dependerá en gran medida de cómo se realicen las retransmisiones. En primer lugar, considere el caso no realista en el que el host A es capaz de determinar de alguna manera (mágicamente) si el buffer en el router está libre o lleno y enviar entonces un paquete solo cuando el buffer esté libre. En este caso no se produciría ninguna pérdida, λ_{in} sería igual a λ'_{in} y la tasa de transferencia de la conexión sería igual a λ_{in}. Este caso se muestra en la Figura 3.46(a). Desde el punto de vista de la tasa de transferencia, el rendimiento es ideal: todo lo que se envía, se recibe. Observe que, en este escenario, la velocidad media de transmisión de host no puede ser mayor que *R*/2, ya que se supone que nunca tiene lugar una pérdida de paquetes.

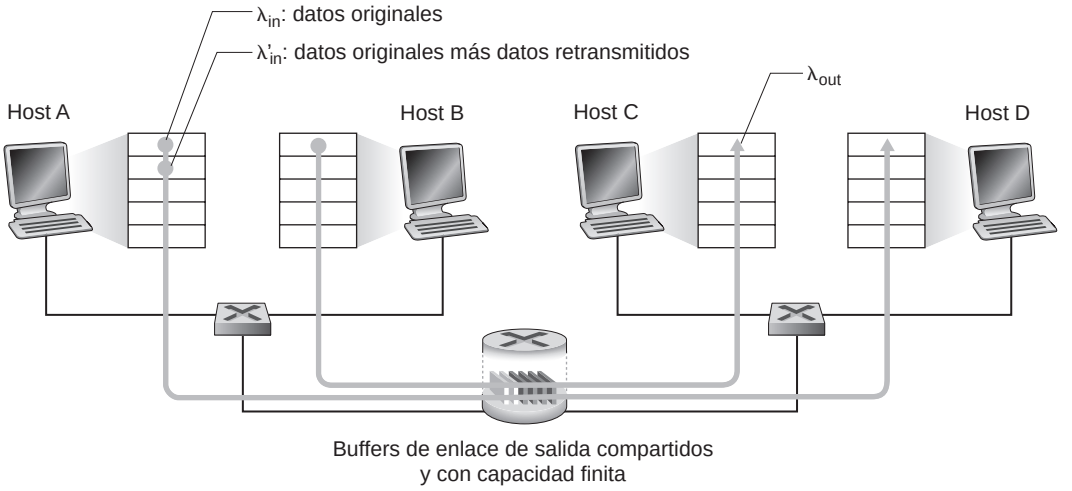


Figura 3.45 ♦ Escenario 2: dos hosts (con retransmisión) y un router con buffers con capacidad finita.

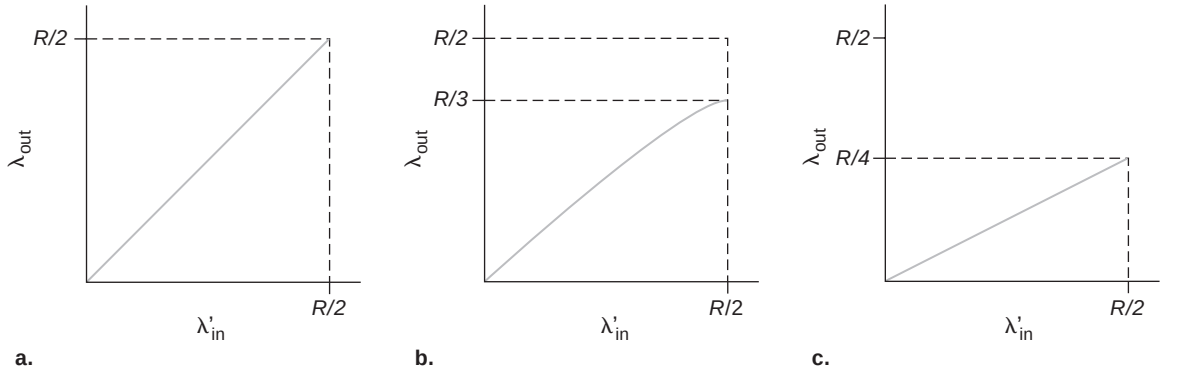


Figura 3.46 ♦ Escenario 2: rendimiento con buffers de capacidad finita.

Consideremos ahora un caso algo más realista en el que el emisor solo retransmite cuando sabe con seguridad que un paquete se ha perdido. De nuevo, esta es una suposición un poco exagerada; sin embargo, es posible que el host emisor tenga fijado su intervalo de fin de temporización en un valor lo suficientemente grande como para garantizar que un paquete que no ha sido reconocido es un paquete que se ha perdido. En este caso, el rendimiento puede ser similar al mostrado en la Figura 3.46(b). Para apreciar lo que está ocurriendo aquí, considere el caso en el que la carga ofrecida, λ'_{in} (la velocidad de transmisión de los datos originales más la de las retransmisiones), es igual a $R/2$. Según la Figura 3.46(b), para este valor de la carga ofrecida la velocidad a la que los datos son suministrados a la aplicación del receptor es $R/3$. Por tanto, de las 0,5R unidades de datos transmitidos, 0,333R bytes/segundo (como media) son datos originales y 0,166R bytes/segundo (como media) son datos retransmitidos. *Tenemos aquí por tanto otro de los costes de una red congestionada: el emisor tiene que realizar retransmisiones para poder compensar los paquetes descartados (perdidos) a causa de un desbordamiento de buffer.*

Por último, considere el caso en el que el emisor puede alcanzar el fin de la temporización de forma prematura y retransmitir un paquete que ha sido retardado en la cola pero que todavía no se ha perdido. En este caso, tanto el paquete de datos original como la retransmisión pueden llegar al receptor. Por supuesto, el receptor necesitará solo una copia de este paquete y descartará

la retransmisión. En este caso, el trabajo realizado por el router al reenviar la copia retransmitida del paquete original se desperdicia, ya que el receptor ya había recibido la copia original de ese paquete. El router podría haber hecho un mejor uso de la capacidad de transmisión del enlace enviando en su lugar un paquete distinto. *Por tanto, tenemos aquí otro de los costes de una red congestionada: las retransmisiones innecesarias del emisor causadas por retardos largos pueden llevar a que un router utilice el ancho de banda del enlace para reenviar copias innecesarias de un paquete.* La Figura 3.46(c) muestra la tasa de transferencia en función de la carga ofrecida cuando se supone que el router reenvía cada paquete dos veces (como media). Dado que cada paquete se reenvía dos veces, la tasa de transferencia tendrá un valor asintótico de *R*/4 cuando la carga ofrecida se aproxime a *R*/2.

Escenario 3: cuatro emisores, routers con buffers de capacidad finita y rutas con múltiples saltos

En este último escenario dedicado a la congestión de red tenemos cuatro hosts que transmiten paquetes a través de rutas solapadas con dos saltos, como se muestra en la Figura 3.47. De nuevo suponemos que cada host utiliza un mecanismo de fin de temporización/retransmisión para implementar un servicio de transferencia de datos fiable, que todos los hosts tienen el mismo valor de λ_{in} y que todos los enlaces de router tienen una capacidad de *R* bytes/segundo.

Consideremos la conexión del host A al host C pasando a través de los routers R1 y R2. La conexión A–C comparte el router R1 con la conexión D–B y comparte el router R2 con la conexión B–D. Para valores extremadamente pequeños de λ_{in}, es raro que el buffer se desborde (como en los dos escenarios de congestión anteriores), y la tasa de transferencia es aproximadamente igual a

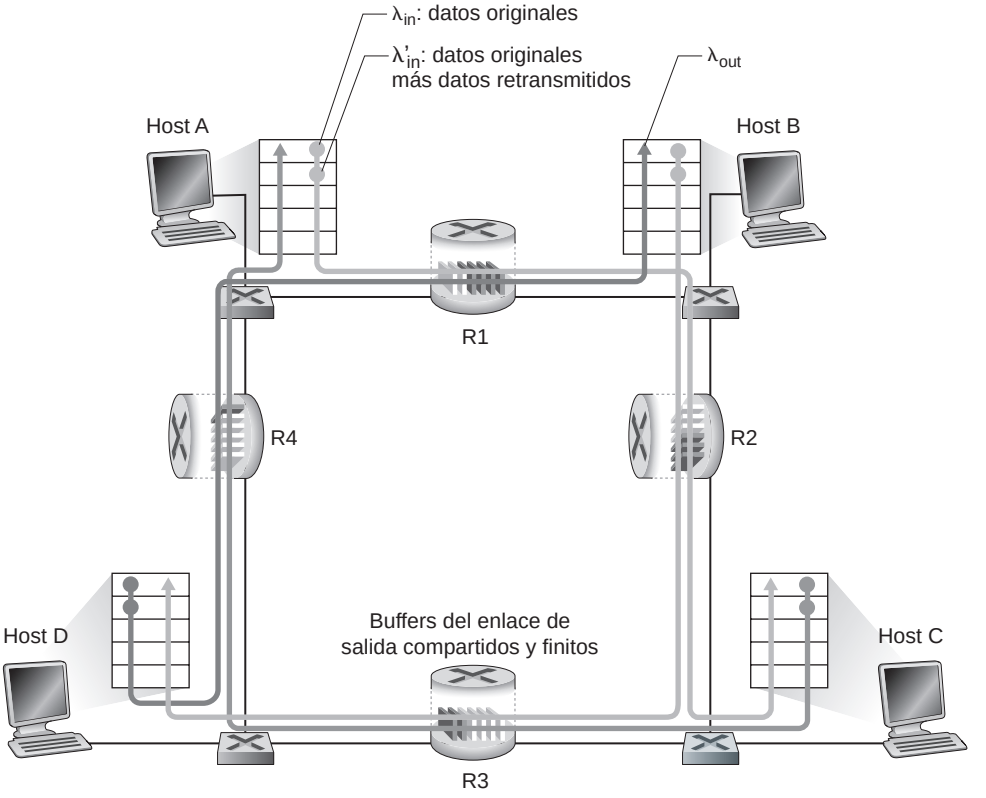


Figura 3.47 ♦ Cuatro emisores, routers con buffers finitos y rutas con varios saltos.

la carga ofrecida. Para valores ligeramente más grandes de λ_{in}, la correspondiente tasa de transferencia es también más grande, ya que se están transmitiendo más datos originales por la red y entregándose en su destino, y los desbordamientos siguen siendo raros. Por tanto, para valores pequeños de λ_{in}, un incremento de λ_{in} da lugar a un incremento de λ_{out}.

Una vez considerado el caso de tráfico extremadamente bajo, pasemos a examinar el caso en que λ_{in} (y por tanto λ'_{in}) es extremadamente grande. Sea el router R2. El tráfico de A–C que llega al router R2 (el que llega a R2 después de ser reenviado desde R1) puede tener una velocidad de llegada a R2 que es como máximo *R*, la capacidad del enlace de R1 a R2, independientemente del valor de λ_{in}. Si λ'_{in} es extremadamente grande en todas las conexiones (incluyendo la conexión B–D), entonces la velocidad de llegada del tráfico de B–D a R2 puede ser mucho mayor que la del tráfico de A–C. Puesto que los tráficos de A–C y B–D tienen que competir en el router R2 por el espacio limitado disponible en el buffer, la cantidad de tráfico de A–C que consiga atravesar con éxito R2 (es decir, que no se pierda por desbordamiento del buffer) será cada vez menor a medida que la carga ofrecida por la conexión B–D aumente. En el límite, a medida que la carga ofrecida se aproxima a infinito, un buffer vacío de R2 es llenado de forma inmediata por un paquete de B–D y la tasa de transferencia de la conexión A–C en R2 tiende a cero. Esto, a su vez, *implica que la tasa de transferencia terminal a terminal de A–C tiende a cero* en el límite correspondiente a una situación de tráfico intenso. Estas consideraciones dan lugar a la relación de compromiso entre la carga ofrecida y la tasa de transferencia que se muestra en la Figura 3.48.

La razón del eventual decrecimiento de la tasa de transferencia al aumentar la carga ofrecida es evidente cuando se considera la cantidad de trabajo desperdiciado realizado por la red. En el escenario descrito anteriormente en el que había una gran cantidad de tráfico, cuando un paquete se descartaba en un router de segundo salto, el trabajo realizado por el router del primer salto al encaminar un paquete al router del segundo salto terminaba siendo "desperdiciado". Para eso, hubiera dado igual que el primer router simplemente hubiera descartado dicho paquete y hubiera permanecido inactivo, porque de todos modos el paquete no llegaría a su destino. Aún más, la capacidad de transmisión utilizada en el primer router para reenviar el paquete al segundo router podría haber sido mejor aprovechada si se hubiera empleado para transmitir un paquete diferente (por ejemplo, al seleccionar un paquete para transmitirlo, puede resultar mejor para un router dar la prioridad a los paquetes que ya hayan pasado por varios routers anteriormente). *Por tanto, aquí nos encontramos con otro de los costes de descartar un paquete a causa de la congestión de la red: cuando un paquete se descarta a lo largo de una ruta, la capacidad de transmisión empleada en cada uno de los enlaces anteriores para encaminar dicho paquete hasta el punto en el que se ha descartado termina por desperdiciarse.*

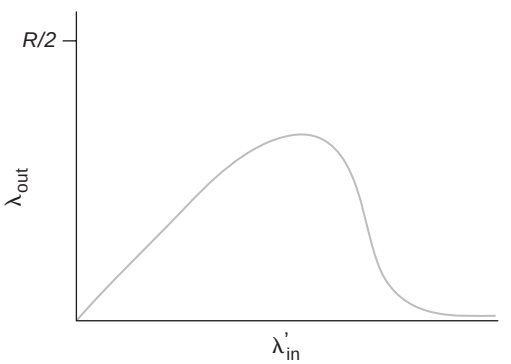


Figura 3.48 ♦ Escenario 3: rendimiento con buffers finitos y rutas con varios saltos.

3.6.2 Métodos para controlar la congestión

En la Sección 3.7 examinaremos en gran detalle el método específico de TCP para controlar la congestión. Ahora, vamos a identificar los dos métodos más comunes de control de congestión que se utilizan en la práctica y abordaremos las arquitecturas de red específicas y los protocolos de control de congestión que se integran en estos métodos.

En el nivel más general, podemos diferenciar entre las técnicas de control de congestión basándonos en si la capa de red proporciona alguna ayuda explícita a la capa de transporte con propósitos de controlar la congestión:

- *Control de congestión terminal a terminal*. En este método, la capa de red no proporciona soporte explícito a la capa de transporte para propósitos de control de congestión. Incluso la presencia de congestión en la red tiene que ser inferida por los sistemas terminales basándose únicamente en el comportamiento observado de la red (por ejemplo, la pérdida de paquetes y los retardos). En la Sección 3.7.1 veremos que TCP tiene que aplicar este método de control de congestión terminal a terminal, ya que la capa IP no proporciona ninguna realimentación a los hosts relativa a la congestión de la red. La pérdida de segmentos TCP (indicada por un fin de temporización o por la recepción de un triple paquete ACK duplicado) se toma como indicación de que existe congestión en la red, por lo que TCP reduce el tamaño de su ventana en consecuencia. También veremos una propuesta más reciente para abordar el control de congestión de TCP que utiliza valores de retardo de ida y vuelta crecientes como indicadores de que existe una mayor congestión de red.
- *Control de congestión asistido por la red*. En este método de control de congestión, los routers proporcionan una realimentación explícita al emisor y/o receptor informando del estado de congestión en la red. Esta realimentación puede ser tan simple como un único bit que indica que existe congestión en un enlace. Este método se aplicó en las tempranas arquitecturas de red SNA de IBM [Schwartz 1982] y DECnet de DEC [Jain 1989; Ramakrishnan 1990] y ATM [Black 1995]. También es posible proporcionar una realimentación de red más sofisticada. Por ejemplo, una forma del mecanismo de control de congestión de **ABR (*Available Bite Rate*)** en las redes **ATM** permite a un router informar al emisor de la velocidad máxima de transmisión de host que el router puede soportar en un enlace saliente. Como hemos mencionado anteriormente, las versiones predeterminadas de Internet de IP y TCP adoptan un método terminal a terminal para llevar a cabo el control de congestión. Sin embargo, veremos en la Sección 3.7.2 que, recientemente, IP y TCP pueden implementar de forma opcional un mecanismo de control de congestión asistido por la red.

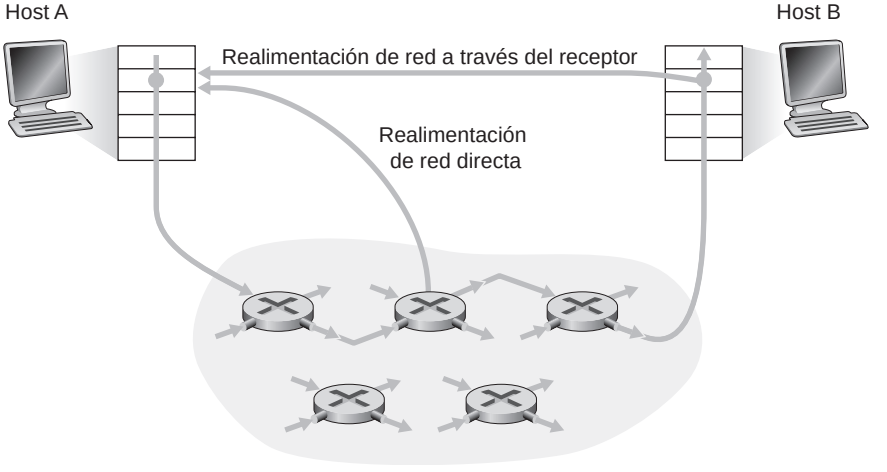


Figura 3.49 ♦ Dos rutas de realimentación de la información de la congestión asistida por la red.

En el mecanismo de control de congestión asistido por la red, la información acerca de la congestión suele ser realimentada de la red al emisor de una de dos formas, como se ve en la Figura 3.49. La realimentación directa puede hacerse desde un router de la red al emisor. Esta forma de notificación, normalmente, toma la forma de un paquete de asfixia o bloqueo (*choke packet*) (que esencialmente dice "¡Estoy congestionada!"). La segunda forma de notificación, más común, tiene lugar cuando un router marca/actualiza un campo de un paquete que se transmite del emisor al receptor para indicar que existe congestión. Después de recibir un paquete marcado, el receptor notifica al emisor la existencia de congestión. Observe que esta última forma de notificación tarda al menos un periodo igual al tiempo de ida y vuelta completo.

3.7 Mecanismo de control de congestión de TCP

En esta sección vamos a continuar con nuestro estudio de TCP. Como hemos visto en la Sección 3.5, TCP proporciona un servicio de transporte fiable entre dos procesos que se ejecutan en hosts diferentes. Otro componente clave de TCP es su mecanismo de control de congestión. Como hemos mencionado en la sección anterior, TCP tiene que utilizar un control de congestión terminal a terminal en lugar de un control de congestión asistido por la red, ya que la capa IP no proporciona una realimentación explícita a los sistemas terminales en lo tocante a la congestión de la red.

El método empleado por TCP consiste en que cada emisor limite la velocidad a la que transmite el tráfico a través de su conexión en función de la congestión de red percibida. Si un emisor TCP percibe que en la ruta entre él mismo y el destino apenas existe congestión, entonces incrementará su velocidad de transmisión; por el contrario, si el emisor percibe que existe congestión a lo largo de la ruta, entonces reducirá su velocidad de transmisión. Pero este método plantea tres cuestiones. En primer lugar, ¿cómo limita el emisor TCP la velocidad a la que envía el tráfico a través de su conexión? En segundo lugar, ¿cómo percibe el emisor TCP que existe congestión en la ruta entre él mismo y el destino? Y, tercero, ¿qué algoritmo deberá emplear el emisor para variar su velocidad de transmisión en función de la congestión percibida terminal a terminal?

Examinemos en primer lugar cómo un emisor TCP limita la velocidad a la que envía el tráfico a través de su conexión. En la Sección 3.5, hemos visto que cada lado de una conexión TCP consta de un buffer de recepción, un buffer de transmisión y varias variables (`UltimoByteLeido`, `VentRecepcion`, etc.). El mecanismo de control de congestión de TCP que opera en el emisor hace un seguimiento de una variable adicional, la **ventana de congestión**. Esta ventana, indicada como `VentCongestion`, impone una restricción sobre la velocidad a la que el emisor TCP puede enviar tráfico a la red. Específicamente, la cantidad de datos no reconocidos en un emisor no puede exceder el mínimo de entre `VentCongestion` y `VentRecepcion`, es decir:

```
UltimoByteEnviado – UltimoByteReconocido ≤ min{VentCongestion, VentRecepcion}
```

Con el fin de centrarnos en el mecanismo de control de congestión (en oposición al control de flujo), vamos a suponer que el buffer de recepción TCP es tan grande que la restricción de la ventana de recepción puede ignorarse; por tanto, la cantidad de datos no reconocidos por el emisor queda limitada únicamente por `VentCongestion`. Supondremos también que el emisor siempre tiene datos que enviar; es decir, todos los segmentos de la ventana de congestión son enviados.

La restricción anterior limita la cantidad de datos no reconocidos por el emisor y, por tanto, limita de manera indirecta la velocidad de transmisión del emisor. Para comprender esto imagine una conexión en la que tanto la pérdida de paquetes como los retardos de transmisión sean despreciables. En esta situación, lo que ocurre a grandes rasgos es lo siguiente: al inicio de cada periodo RTT, la restricción permite al emisor enviar `VentCongestion` bytes de datos a través de la conexión; al final del periodo RTT, el emisor recibe los paquetes ACK correspondientes a los datos. *Por tanto, la velocidad de transmisión del emisor es aproximadamente igual a VentCongestion/RTT bytes/*

segundo. Ajustando el valor de la ventana de congestión, el emisor puede ajustar la velocidad a la que transmite los datos a través de su conexión.

Veamos ahora cómo percibe un emisor TCP que existe congestión en la ruta entre él mismo y el destino. Definamos un "suceso de pérdida" en un emisor TCP como el hecho de que se produzca un fin de temporización o se reciban tres paquetes ACK duplicados procedentes del receptor (recuerde la exposición de la Sección 3.5.4 sobre el suceso de fin de temporización mostrado en la Figura 3.33 y la subsiguiente modificación para incluir la retransmisión rápida a causa de la recepción de tres paquetes ACK duplicados). Cuando existe una congestión severa, entonces uno o más de los buffers de los routers existentes a lo largo de la ruta pueden desbordarse, dando lugar a que un datagrama (que contenga un segmento TCP) sea descartado. A su vez, el datagrama descartado da lugar a un suceso de pérdida en el emisor (debido a un fin de temporización o a la recepción de tres paquetes ACK duplicados), el cual lo interpreta como una indicación de que existe congestión en la ruta entre el emisor y el receptor.

Ahora que ya hemos visto cómo se detecta la existencia de congestión en la red, vamos a considerar el mejor de los casos, cuando no existe congestión en la red, es decir, cuando no se producen pérdidas de paquetes. En este caso, el emisor TCP recibirá los paquetes de reconocimiento ACK correspondientes a los segmentos anteriormente no reconocidos. Como veremos, TCP interpretará la llegada de estos paquetes ACK como una indicación de que todo está bien (es decir, que los segmentos que están siendo transmitidos a través de la red están siendo entregados correctamente al destino) y empleará esos paquetes de reconocimiento para incrementar el tamaño de la ventana de congestión (y, por tanto, la velocidad de transmisión). Observe que si la velocidad de llegada de los paquetes ACK es lenta, porque por ejemplo la ruta terminal a terminal presenta un retardo grande o contiene un enlace con un ancho de banda pequeño, entonces el tamaño de la ventana de congestión se incrementará a una velocidad relativamente lenta. Por el contrario, si la velocidad de llegada de los paquetes de reconocimiento es alta, entonces el tamaño de la ventana de congestión se incrementará más rápidamente. Puesto que TCP utiliza los paquetes de reconocimiento para provocar (o temporizar) sus incrementos del tamaño de la ventana de congestión, se dice que TCP es **auto-temporizado**.

Conocido el *mecanismo* de ajuste del valor de `VentCongestion` para controlar la velocidad de transmisión, la cuestión crítica que nos queda es: ¿*cómo* debería un emisor TCP determinar su velocidad de transmisión? Si los emisores TCP de forma colectiva transmiten a velocidades demasiado altas pueden congestionar la red, llevándola al tipo de colapso de congestión que hemos visto en la Figura 3.48. De hecho, la versión de TCP que vamos a estudiar a continuación fue desarrollada en respuesta al colapso de congestión observado en Internet [Jacobson 1988] en las versiones anteriores de TCP. Sin embargo, si los emisores TCP son demasiado cautelosos y transmiten la información muy lentamente, podrían infrautilizar el ancho de banda de la red; es decir, los emisores TCP podrían transmitir a velocidades más altas sin llegar a congestionar la red. Entonces, ¿cómo pueden determinar los emisores TCP sus velocidades de transmisión de manera que no congestionen la red a la vez que hacen uso del todo el ancho de banda disponible? ¿Están los emisores TCP explícitamente coordinados, o existe un método distribuido en el que dichos emisores TCP puedan establecer sus velocidades de transmisión basándose únicamente en la información local? TCP responde a estas preguntas basándose en los siguientes principios:

- *Un segmento perdido implica congestión y, por tanto, la velocidad del emisor TCP debe reducirse cuando se pierde un segmento.* Recuerde que en la Sección 3.5.4 hemos visto que un suceso de fin de temporización o la recepción de cuatro paquetes de reconocimiento para un segmento dado (el paquete ACK original y los tres duplicados) se interpreta como una indicación implícita de que se ha producido un "suceso de pérdida" del segmento que sigue al segmento que ha sido reconocido cuatro veces, activando el proceso de retransmisión del segmento perdido. Desde el punto de vista del mecanismo de control de congestión, la cuestión es cómo el emisor TCP debe disminuir el tamaño de su ventana de congestión y, por tanto, su velocidad de transmisión, en respuesta a este suceso de pérdida inferido.

- *Un segmento que ha sido reconocido indica que la red está entregando los segmentos del emisor al receptor y, por tanto, la velocidad de transmisión del emisor puede incrementarse cuando llega un paquete ACK correspondiente a un segmento que todavía no había sido reconocido.* La llegada de paquetes de reconocimiento se interpreta como una indicación implícita de que todo funciona bien (los segmentos están siendo entregados correctamente del emisor al receptor y la red por tanto no está congestionada). Luego se puede aumentar el tamaño de la ventana de congestión.
- *Tanteo del ancho de banda*. Puesto que los paquetes ACK indican que la ruta entre el origen y el destino está libre de congestión y la pérdida de paquetes indica que hay una ruta congestionada, la estrategia de TCP para ajustar su velocidad de transmisión consiste en incrementar su velocidad en respuesta a la llegada de paquetes ACK hasta que se produce una pérdida, momento en el que reduce la velocidad de transmisión. El emisor TCP incrementa entonces su velocidad de transmisión para tantear la velocidad a la que de nuevo aparece congestión, retrocede a partir de ese punto y comienza de nuevo a tantear para ver si ha variado la velocidad a la que comienza de nuevo a producirse congestión. El comportamiento del emisor TCP es quizá similar a la del niño que pide (y consigue) una y otra vez golosinas hasta que se le dice "¡No!", momento en el que da marcha atrás, pero enseguida comienza otra vez a pedir más golosinas. Observe que la red no proporciona una indicación explícita del estado de congestión (los paquetes ACK y las pérdidas se utilizan como indicadores implícitos) y que cada emisor TCP reacciona a la información local de forma asíncrona con respecto a otros emisores TCP.

Ahora que ya conocemos los fundamentos del mecanismo de control de congestión de TCP, estamos en condiciones de pasar a estudiar los detalles del famoso **algoritmo de control de congestión de TCP**, que fue descrito por primera vez en el libro de [Jacobson 1988] y que está estandarizado en el documento [RFC 5681]. El algoritmo consta de tres componentes principales: (1) arranque lento (*slow start*), (2) evitación de la congestión (*congestion avoidance*) y (3) recuperación rápida (*fast recovery*). Los dos primeros componentes son obligatorios en TCP, diferenciándose en la forma en que aumentan el tamaño de la ventana de congestión en respuesta a los paquetes ACK recibidos. Vamos a ver brevemente que el arranque lento incrementa el tamaño de la ventana de congestión más rápidamente (¡contrariamente a lo que indica su nombre!) que la evitación de la congestión. El componente recuperación rápida es recomendable, aunque no obligatorio, para los emisores TCP.

Arranque lento

Cuando se inicia una conexión TCP, el valor de la ventana de congestión (`VentCongestion`) normalmente se inicializa con un valor pequeño igual a 1 MSS (tamaño máximo de segmento) [RFC 3390], que da como resultado una velocidad de transmisión inicial aproximadamente igual a MSS/ RTT. Por ejemplo, si MSS = 500 bytes y RTT = 200 milisegundos, la velocidad de transmisión inicial será aproximadamente de 20 kbps. Puesto que el ancho de banda disponible para el emisor TCP puede ser mucho más grande que el valor de MSS/ RTT, al emisor TCP le gustaría poder determinar rápidamente la cantidad de ancho de banda disponible. Por tanto, en el estado de **arranque lento**, el valor de `VentCongestion` se establece en 1 MSS y se incrementa 1 MSS cada vez que se produce el primer reconocimiento de un segmento transmitido. En el ejemplo de la Figura 3.50, TCP envía el primer segmento a la red y espera el correspondiente paquete ACK. Cuando llega dicho paquete de reconocimiento, el emisor TCP incrementa el tamaño de la ventana de congestión en 1 MSS y transmite dos segmentos de tamaño máximo. Estos segmentos son entonces reconocidos y el emisor incrementa el tamaño de la ventana de congestión en 1 MSS por cada uno de los segmentos de reconocimiento, generando así una ventana de congestión de 4 MSS, etc. Este proceso hace que la velocidad de transmisión se duplique en cada periodo RTT. Por tanto, la velocidad de transmisión inicial de TCP es baja, pero crece exponencialmente durante esa fase de arranque lento.

Pero, ¿cuándo debe finalizar este crecimiento exponencial? El algoritmo del arranque lento proporciona varias respuestas a esta cuestión. En primer lugar, si se produce un suceso de pérdida de

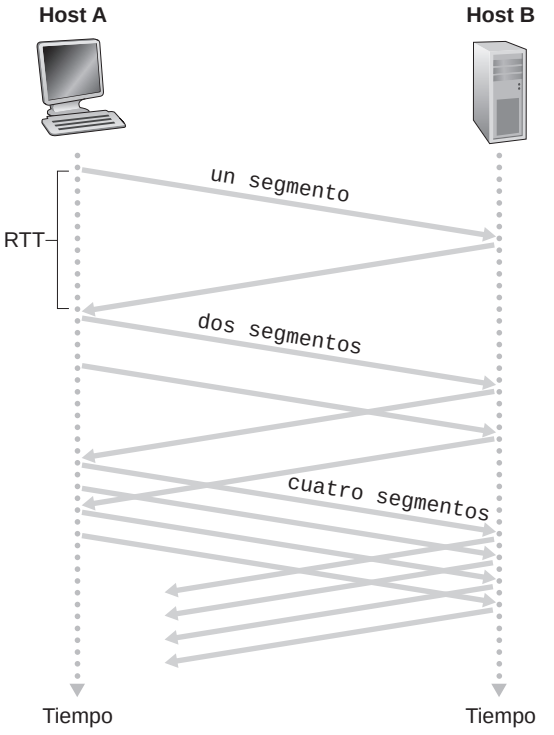


Figura 3.50 ♦ Fase de arranque lento de TCP.

paquete (es decir, si hay congestión) señalado por un fin de temporización, el emisor TCP establece el valor de `VentCongestion` en 1 e inicia de nuevo un proceso de arranque lento. También define el valor de una segunda variable de estado, `umbralAL`, que establece el umbral del arranque lento en `VentCongestion/2`, la mitad del valor del tamaño de la ventana de congestión cuando se ha detectado que existe congestión. La segunda forma en la que la fase de arranque lento puede terminar está directamente ligada al valor de `umbralAL`. Dado que `umbralAL` es igual a la mitad del valor que `VentCongestion` tenía cuando se detectó congestión por última vez, puede resultar algo imprudente continuar duplicando el valor de `VentCongestion` cuando se alcanza o sobrepasa el valor de umbral. Por tanto, cuando el valor de `VentCongestion` es igual a `umbralAL`, la fase de arranque lento termina y las transacciones TCP pasan al modo de evitación de la congestión. Como veremos, TCP incrementa con más cautela el valor de `VentCongestion` cuando está en el modo de evitación de la congestión. La última forma en la que puede terminar la fase de arranque lento es si se detectan tres paquetes ACK duplicados, en cuyo caso TCP realiza una retransmisión rápida (véase la Sección 3.5.4) y entra en el estado de recuperación rápida, que veremos más adelante. El comportamiento de TCP en la fase de arranque lento se resume en la descripción de la FSM del control de congestión de TCP de la Figura 3.51. El algoritmo de arranque lento tiene sus raíces en [Jacobson 1988];en [Jain 1986] se proponía, de forma independiente, un método similar al algoritmo de arranque lento.

Evitación de la congestión

Al entrar en el estado de evitación de la congestión, el valor de `VentCongestion` es aproximadamente igual a la mitad de su valor en el momento en que se detectó congestión por última vez (podemos estar, por tanto, al borde de la congestión). En consecuencia, en lugar de duplicar el valor de `VentCongestion` para cada RTT, TCP adopta un método más conservador e incrementa el valor de `VentCongestion` solamente en un MSS cada RTT [RFC 5681]. Esto puede llevarse

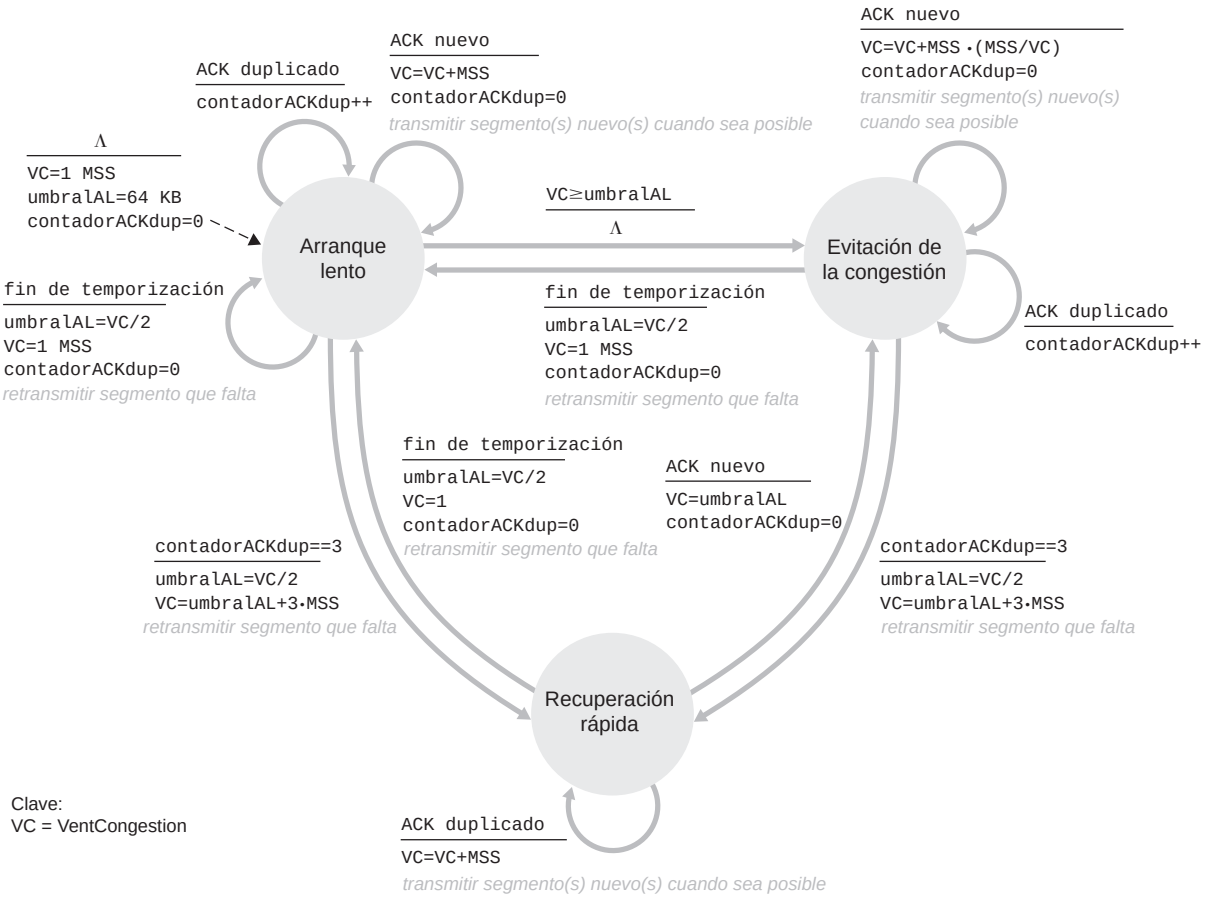


Figura 3.51 ♦ Descripción de la máquina de estados finitos del mecanismo de control de congestión de TCP.

a cabo de varias maneras. Un método habitual consiste en que un emisor TCP aumenta el valor de `VentCongestion` en MSS bytes (MSS/`VentCongestion`) cuando llega un nuevo paquete de reconocimiento. Por ejemplo, si MSS es igual a 1.460 bytes y `VentCongestion` es igual a 14.600 bytes, entonces se enviarán 10 segmentos en un RTT. Cada ACK que llega (suponiendo un ACK por segmento) incrementa el tamaño de la ventana de congestión en 1/10 MSS y, por tanto, el valor del tamaño de la ventana de congestión habrá aumentado en un MSS después de los ACK correspondientes a los 10 segmentos que hayan sido recibidos.

Pero, ¿en qué momento debería detenerse el crecimiento lineal (1 MSS por RTT) en el modo de evitación de la congestión? El algoritmo de evitación de la congestión de TCP se comporta del mismo modo que cuando tiene lugar un fin de temporización. Como en el caso del modo de arranque lento, el valor de `VentCongestion` se fija en 1 MSS y el valor de `umbralAL` se actualiza haciéndose igual a la mitad del valor de `VentCongestion` cuando se produce un suceso de pérdida de paquete. Sin embargo, recuerde que también puede detectarse una pérdida de paquete a causa de la llegada de tres ACK duplicados. En este caso, la red continúa entregando segmentos del emisor al receptor (como señala la recepción de paquetes ACK duplicados). Por tanto, el comportamiento de TCP ante este tipo de pérdida debería ser menos drástico que ante una pérdida de paquete indicada por un fin de temporización: TCP divide entre dos el valor de `VentCongestion` (añadiendo 3 MSS como forma de tener en cuenta los tres ACK duplicados que se han recibido) y configura el valor de `umbralAL`

para que sea igual a la mitad del valor que `VentCongestion` tenía cuando se recibieron los tres ACK duplicados. A continuación, se entra en el estado de recuperación rápida.

Nota de vídeo

Examen del comportamiento de TCP

Recuperación rápida

En la fase de recuperación rápida, el valor de `VentCongestion` se incrementa en 1 MSS por cada ACK duplicado recibido correspondiente al segmento que falta y que ha causado que TCP entre en el estado de recuperación rápida. Cuando llega un ACK para el segmento que falta, TCP entra de nuevo en el estado de evitación de la congestión después de disminuir el valor de `VentCongestion`. Si se produce un fin de temporización, el mecanismo de recuperación rápida efectúa una transición al estado de arranque lento después de realizar las mismas acciones que en los modos de arranque lento y de evitación de la congestión: el valor de `VentCongestion` se establece en 1 MSS

EN LA PRÁCTICA

DIVISIÓN TCP: OPTIMIZACIÓN DEL RENDIMIENTO DE LOS SERVICIOS EN LA NUBE

Para servicios en la nube como los de búsqueda, correo electrónico y redes sociales, resulta deseable proporcionar un alto nivel de capacidad de respuesta, idealmente proporcionando a los usuarios la ilusión de que esos servicios se están ejecutando dentro de sus propios sistemas terminales (incluyendo sus teléfonos inteligentes). Esto puede constituir todo un desafío, ya que los usuarios están ubicados a menudo a mucha distancia de los centros de datos responsables de servir el contenido dinámico asociado con los servicios en la nube. De hecho, si el sistema terminal está lejos de un centro de datos, entonces el RTT será grande, lo que podría dar lugar a un tiempo de respuesta excesivo, debido al arranque lento de TCP.

Como caso de estudio, considere el retardo a la hora de recibir la respuesta a una consulta de búsqueda. Normalmente, el servidor necesita tres ventanas TCP durante el arranque lento para suministrar la respuesta [Pathak 2010]. Por tanto, el tiempo desde que un sistema terminal inicia una conexión TCP, hasta que recibe el último paquete de la respuesta, es de aproximadamente 4 · RTT (un RTT para establecer la conexión TCP y tres RTT para las tres ventanas de datos), más el tiempo de procesamiento en el centro de datos. Estos retardos debidos al RTT pueden dar como resultado un retraso perceptible a la hora de devolver los resultados de las búsquedas, para un porcentaje significativo de las consultas. Además, puede haber un significativo porcentaje de pérdidas de paquetes en las redes de acceso, lo que provocaría retransmisiones TCP y retardos aún más grandes.

Una forma de mitigar este problema y mejorar la capacidad de respuesta percibida por el usuario consiste en (1) desplegar los servidores de *front-end* más cerca de los usuarios y (2) utilizar la **división TCP** (*TCP splitting*), descomponiendo la conexión TCP en el servidor de *front-end*. Con la división TCP, el cliente establece una conexión TCP con el *front-end* cercano, y el *front-end*mantiene una conexión TCP persistente con el centro de datos, con una ventana de congestión TCP muy grande [Tariq 2008, Pathak 2010, Chen 2011]. Con esta técnica, el tiempo de respuesta pasa a ser, aproximadamente, $4 \cdot RTT_{FE} + RTT_{BE}$ + tiempo de procesamiento, donde RTT_{FE} es el tiempo de ida y vuelta entre el cliente y el servidor de *front-end* y RTT_{BE} es el tiempo de ida y vuelta entre el servidor de *front-end* y el centro de datos (servidor de *back-end*). Si el servidor de *front-end* está próximo al cliente, entonces este tiempo de respuesta es aproximadamente igual a RTT más el tiempo de procesamiento, ya que RTT_{FE} es despreciable y RTT_{BE} es aproximadamente RTT. Resumiendo, la técnica de división TCP permite reducir el retardo de red, aproximadamente, de 4· RTT a RTT, mejorando significativamente la capacidad de respuesta percibida por el usuario, en especial para aquellos usuarios que estén lejos de su centro de datos más próximo. La división TCP también ayuda a reducir los retardos de retransmisión TCP provocados por las pérdidas de paquetes en las redes de acceso. Google y Akamai han hecho un amplio uso de sus servidores CDN en las redes de acceso (recuerde nuestra exposición en la Sección 2.6) para llevar a cabo la división TCP para los servicios en la nube que soportan [Chen 2011].

y el valor de `umbralAL` se hace igual a la mitad del valor que tenía `VentCongestion` cuando tuvo lugar el suceso de pérdida.

El mecanismo de recuperación rápida es un componente de TCP recomendado, aunque no obligatorio [RFC 5681]. Es interesante resaltar que una versión anterior de TCP, conocida como **TCP Tahoe**, establece incondicionalmente el tamaño de la ventana de congestión en 1 MSS y entra en el estado de arranque lento después de un suceso de pérdida indicado por un fin de temporización o por la recepción de tres ACK duplicados. La versión más reciente de TCP, **TCP Reno**, incorpora la recuperación rápida.

La Figura 3.52 ilustra la evolución de la ventana de congestión de TCP para Reno y Tahoe. En esta figura, inicialmente el umbral es igual a 8 MSS. Durante los ocho primeros ciclos de transmisión, Tahoe y Reno realizan acciones idénticas. La ventana de congestión crece rápidamente de forma exponencial durante la fase de arranque lento y alcanza el umbral en el cuarto ciclo de transmisión. A continuación, la ventana de congestión crece linealmente hasta que se produce un suceso de tres ACK duplicados, justo después del octavo ciclo de transmisión. Observe que el tamaño de la ventana de congestión es igual a $12 \cdot MSS$ cuando se produce el suceso de pérdida. El valor de `umbralAL` se hace entonces igual a $0{,}5 \cdot$ `VentCongestion` $= 6 \cdot$ MSS. En TCP Reno, el tamaño de la ventana de congestión es puesto a `VentCongestion` $= 9 \cdot$ MSS y luego crece linealmente. En TCP Tahoe, la ventana de congestión es igual a 1 MSS y crece exponencialmente hasta que alcanza el valor de `umbralAL`, punto a partir del cual crece linealmente.

La Figura 3.51 presenta la descripción completa de la máquina de estados finitos de los algoritmos del mecanismo de control de congestión de TCP: arranque lento, evitación de la congestión y recuperación rápida. En la figura también se indica dónde pueden producirse transmisiones de nuevos segmentos y dónde retransmisiones de segmentos. Aunque es importante diferenciar entre las retransmisiones/control de errores de TCP y el control de congestión de TCP, también lo es apreciar cómo estos dos aspectos de TCP están estrechamente vinculados.

Control de congestión de TCP: retrospectiva

Una vez vistos los detalles de las fases de arranque lento, de evitación de la congestión y de recuperación rápida, merece la pena retroceder un poco para clarificar las cosas. Ignorando la fase inicial de arranque lento en la que se establece la conexión y suponiendo que las pérdidas están indicadas por la recepción de tres ACK duplicados en lugar de por fines de temporización, el control de congestión de TCP consiste en un crecimiento lineal (aditivo) de `VentCongestion` a razón de 1 MSS por RTT, seguido de un decrecimiento multiplicativo (división entre dos) del tamaño de la ventana, `VentCongestion`, cuando se reciben tres ACK duplicados. Por esta razón, suele decirse que el control de congestión de TCP es una forma de **crecimiento aditivo y decrecimiento multiplicativo**

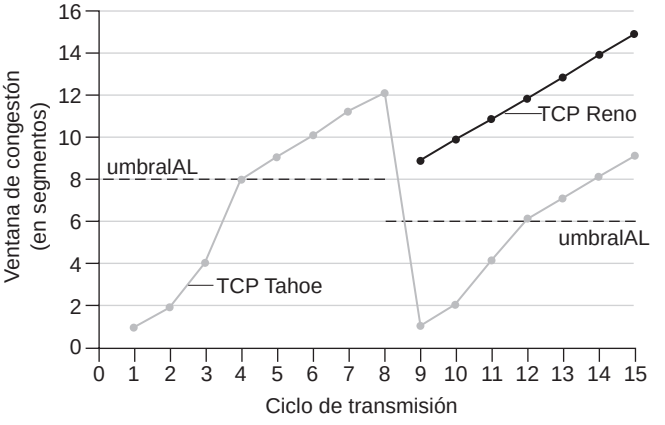


Figura 3.52 ♦ Evolución de la ventana de congestión de TCP (Tahoe y Reno).

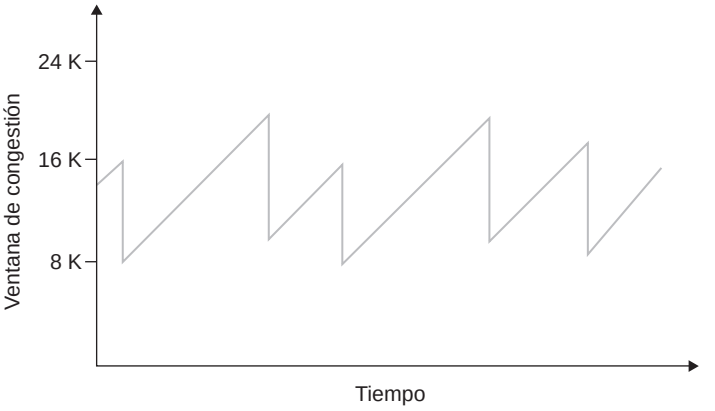


Figura 3.53 ♦ Control de congestión con crecimiento aditivo y decrecimiento multiplicativo.

(**AIMD, *Additive-Increase, Multiplicative-Decrease*)** de control de congestión. El control de congestión AIMD presenta un comportamiento en forma de "diente de sierra", como se muestra en la Figura 3.53, lo que también ilustra nuestra anterior intuición de que TCP "va tanteando" el ancho de banda. (TCP aumenta linealmente el tamaño de su ventana de congestión, y por tanto su velocidad de transmisión, hasta que tiene lugar la recepción de tres ACK duplicados. A continuación, divide entre dos su ventana de congestión, pero vuelve después a crecer linealmente, tanteando para ver si hay disponible ancho de banda adicional.)

Como hemos mencionado anteriormente, la mayor parte de las implementaciones TCP actuales emplean el algoritmo Reno [Padhye 2001]. Se han propuesto muchas variantes del algoritmo Reno [RFC 3782; RFC 2018]. El algoritmo TCP Vegas [Brakmo 1995; Ahn 1995] intenta evitar la congestión manteniendo una buena tasa de transferencia. La idea básica del algoritmo Vegas es (1) detectar la congestión en los routers existentes entre el origen y el destino *antes* de que se pierda un paquete y (2) reducir la velocidad linealmente cuando se detecta una pérdida inminente de paquetes. La pérdida inminente de un paquete se predice observando el RTT. Cuanto mayor es el RTT de los paquetes, mayor es la congestión en los routers. A finales de 2015, la implementación Linux Ubuntu de TCP proporcionaba arranque lento, evitación de congestión, recuperación rápida, retransmisión rápida y SACK, de manera predeterminada; también se proporcionan algoritmos alternativos de control de congestión, como TCP Vegas y BIC [Xu 2004]. En [Afanasyev 2010] se proporciona un resumen de las muchas variantes de TCP.

El algoritmo AIMD de TCP fue desarrollado basándose en un enorme trabajo de ingeniería y de experimentación con los mecanismos de control de congestión en redes reales. Diez años después del desarrollo de TCP, los análisis teóricos mostraron que el algoritmo de control de congestión de TCP sirve como un algoritmo de optimización asíncrona distribuido que da como resultado la optimización simultánea de diversos aspectos importantes, tanto de las prestaciones proporcionadas al usuario como del rendimiento de la red [Kelly 1998]. Desde entonces se han desarrollado muchos aspectos teóricos del control de congestión [Srikant 2004].

Descripción macroscópica de la tasa de transferencia de TCP

Visto el comportamiento en diente de sierra de TCP, resulta natural preguntarse cuál es la tasa de transferencia media (es decir, la velocidad media) de una conexión TCP de larga duración. En este análisis vamos a ignorar las fases de arranque lento que tienen lugar después de producirse un fin de temporización (estas fases normalmente son muy cortas, ya que el emisor sale de ellas rápidamente de forma exponencial). Durante un intervalo concreto de ida y vuelta, la velocidad a la que TCP envía datos es función del tamaño de la ventana de congestión y del RTT actual.

Cuando el tamaño de la ventana es *w* bytes y el tiempo actual de ida y vuelta es RTT segundos, entonces la velocidad de transmisión de TCP es aproximadamente igual a *w*/*RTT*. TCP comprueba entonces si hay ancho de banda adicional incrementando *w* en 1 MSS cada *RTT* hasta que se produce una pérdida. Sea *W* el valor de *w* cuando se produce una pérdida. Suponiendo que *RTT* y *W* son aproximadamente constantes mientras dura la conexión, la velocidad de transmisión de TCP varía entre $W/(2 \cdot RTT)$ y W/RTT.

Estas suposiciones nos llevan a un modelo macroscópico extremadamente simplificado del comportamiento en régimen permanente de TCP. La red descarta un paquete de la conexión cuando la velocidad aumenta hasta *W/RTT*; la velocidad entonces se reduce a la mitad y luego aumenta MSS/*RTT* cada *RTT* hasta que de nuevo alcanza el valor *W/RTT*. Este proceso se repite una y otra vez. Puesto que la tasa de transferencia (es decir, la velocidad) de TCP aumenta linealmente entre los dos valores extremos, tenemos

$$\text{tasa de transferencia media de una conexión} = \frac{0,75 \cdot W}{RTT}$$

Utilizando este modelo altamente idealizado para la dinámica del régimen permanente de TCP, también podemos deducir una expresión interesante que relaciona la tasa de pérdidas de una conexión con su ancho de banda disponible [Mahdavi 1997]. Dejamos esta demostración para los problemas de repaso. Un modelo más sofisticado que, según se ha comprobado, concuerda empíricamente con los datos medidos es el que se describe en [Padhye 2000].

TCP en rutas con un alto ancho de banda

Es importante darse cuenta de que el control de congestión de TCP ha ido evolucionando a lo largo de los años y todavía sigue evolucionando. Puede leer un resumen sobre las variantes de TCP actuales y una exposición acerca de la evolución de TCP en [Floyd 2001, RFC 5681, Afanasyev 2010]. Lo que era bueno para Internet cuando la mayoría de las conexiones TCP transportaban tráfico SMTP, FTP y Telnet no es necesariamente bueno para la Internet actual, en la que domina HTTP, o para una futura Internet que proporcione servicios que hoy ni siquiera podemos imaginar.

La necesidad de una evolución continua de TCP puede ilustrarse considerando las conexiones TCP de alta velocidad necesarias para las aplicaciones de computación reticular (*grid-computing*) y de computación en la nube. Por ejemplo, considere una conexión TCP con segmentos de 1.500 bytes y un *RTT* de 100 milisegundos y suponga que deseamos enviar datos a través de esta conexión a 10 Gbps. Siguiendo el documento [RFC 3649], observamos que utilizar la fórmula anterior para la tasa de transferencia de TCP, con el fin de alcanzar una tasa de transferencia de 10 Gbps, nos daría un tamaño medio de la ventana de congestión de 83.333 segmentos, que son *muchos* segmentos, lo que despierta el temor a que uno de esos 83.333 segmentos en tránsito pueda perderse. ¿Qué ocurriría si se produjeran pérdidas? O, dicho de otra manera, ¿qué fracción de los segmentos transmitidos podría perderse que permitiera al algoritmo de control de congestión de TCP de la Figura 3.51 alcanzar la velocidad deseada de 10 Gbps? En las cuestiones de repaso de este capítulo, mostraremos cómo derivar una fórmula que expresa la tasa de transferencia de una conexión TCP en función de la tasa de pérdidas (*L*), el tiempo de ida y vuelta (*RTT*) y el tamaño máximo de segmento (*MSS*):

$$\text{tasa de transferencia media de una conexión} = \frac{1,22 \cdot MSS}{RTT\sqrt{L}}$$

Con esta fórmula, podemos ver que para alcanzar una tasa de transferencia de 10 Gbps, el actual algoritmo de control de congestión de TCP solo puede tolerar una probabilidad de pérdida de segmentos de $2 \cdot 10^{-10}$ (o, lo que es equivalente, un suceso de pérdida por cada 5.000.000.000 segmentos), lo cual es una tasa muy baja. Esta observación ha llevado a una serie de investigadores a buscar nuevas versiones de TCP que estén diseñadas específicamente para estos entornos de alta

velocidad; consulte [Jin 2004; Kelly 2003; Ha 2008; RFC 7323] para obtener información sobre estos trabajos.

3.7.1 Equidad

Considere ahora *K* conexiones TCP, cada una de ellas con una ruta terminal a terminal diferente, pero atravesando todas ellas un enlace de cuello de botella con una velocidad de transmisión de *R* bps (con *enlace de cuello de botella* queremos decir que, para cada conexión, todos los restantes enlaces existentes a lo largo de la ruta de la conexión no están congestionados y tienen una capacidad de transmisión grande comparada con la capacidad de transmisión del enlace de cuello de botella). Suponga que cada conexión está transfiriendo un archivo de gran tamaño y que no existe tráfico UDP atravesando el enlace de cuello de botella. Se dice que un mecanismo de control de congestión es *equitativo* si la velocidad media de transmisión de cada conexión es aproximadamente igual a *R/K*; es decir, cada conexión obtiene la misma cuota del ancho de banda del enlace.

¿Es el algoritmo AIMD de TCP equitativo, teniendo en cuenta que diferentes conexiones TCP pueden iniciarse en instantes distintos y, por tanto, pueden tener distintos tamaños de ventana en un instante determinado? [Chiu 1989] proporciona una explicación elegante e intuitiva de por qué el control de congestión de TCP converge para proporcionar la misma cuota de ancho de banda de un enlace de cuello de botella a las conexiones TCP que compiten por el ancho de banda.

Consideremos el caso simple de dos conexiones TCP que comparten un mismo enlace, cuya velocidad de transmisión es *R*, como se muestra en la Figura 3.54. Suponemos que las dos conexiones tienen el mismo MSS y el mismo RTT (por lo que si tienen el mismo tamaño de ventana de congestión, entonces tienen la misma tasa de transferencia). Además, tienen que enviar una gran cantidad de datos y ninguna otra conexión TCP ni datagrama UDP atraviesan este enlace compartido. Asimismo, vamos a ignorar la fase de arranque lento de TCP y vamos a suponer que las conexiones TCP están operando en el modo de evitación de la congestión (AIMD) durante todo el tiempo.

La gráfica de la Figura 3.55 muestra la tasa de transferencia de las dos conexiones TCP. Si TCP está diseñado para que las dos conexiones compartan equitativamente el ancho de banda del enlace, entonces la tasa de transferencia alcanzada debería caer a lo largo de la flecha que sale del origen con un ángulo de 45 grados (cuota equitativa de ancho de banda). Idealmente, la suma de las dos tasas de transferencia tiene que ser igual a *R* (evidentemente, que cada conexión reciba una cuota equitativa de la capacidad del enlace, pero igual a cero, no es una situación deseable). Por tanto, el objetivo debería ser que las tasas de transferencia alcanzadas se encuentren en algún punto próximo a la intersección de la línea de cuota equitativa de ancho de banda con la línea de utilización del ancho de banda completo mostradas en la Figura 3.55.

Suponga que los tamaños de ventana de TCP son tales que, en un instante determinado, las conexiones 1 y 2 alcanzan las tasas de transferencia indicadas por el punto *A* de la Figura 3.55. Puesto que la cantidad de ancho de banda del enlace conjunto consumido por las dos conexiones es menor que *R*, no se producirá ninguna pérdida y ambas conexiones incrementarán sus tamaños de ventana en

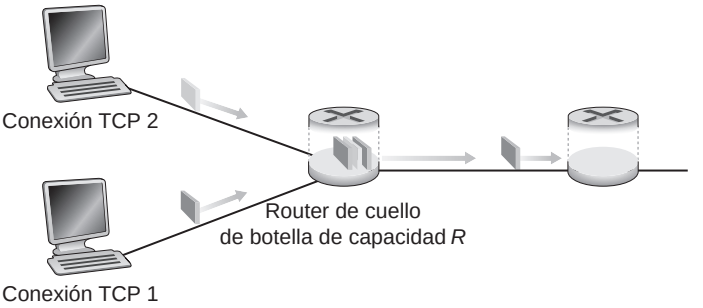


Figura 3.54 ♦ Dos conexiones TCP que comparten un mismo enlace cuello de botella.

1 MSS por RTT como resultado del algoritmo de evitación de la congestión de TCP. Por tanto, la tasa de transferencia conjunta de las dos conexiones sigue la línea de 45 grados (incremento equitativo para ambas conexiones) partiendo del punto *A*. Finalmente, el ancho de banda del enlace consumido conjuntamente por las dos conexiones será mayor que *R*, por lo que terminará produciéndose una pérdida de paquetes. Suponga que las conexiones 1 y 2 experimentan una pérdida de paquetes cuando alcanzan las tasas de transferencia indicadas por el punto *B*. Entonces las conexiones 1 y 2 reducen el tamaño de sus ventanas en un factor de dos. Las tasas de transferencia resultantes se encuentran por tanto en el punto *C*, a medio camino de un vector que comienza en *B* y termina en el origen. Puesto que el ancho de banda conjunto utilizado es menor que *R* en el punto *C*, de nuevo las dos conexiones incrementan sus tasas de transferencia a lo largo de la línea de 45 grados partiendo de *C*. Finalmente, terminará por producirse de nuevo una pérdida de paquetes, por ejemplo, en el punto *D*, y las dos conexiones otra vez reducirán el tamaño de sus ventanas en un factor de dos, y así sucesivamente. Compruebe que el ancho de banda alcanzado por ambas conexiones fluctúa a lo largo de la línea que indica una cuota equitativa del ancho de banda. Compruebe también que las dos conexiones terminarán convergiendo a este comportamiento, independientemente de su posición inicial dentro de ese espacio bidimensional. Aunque en este escenario se han hecho una serie de suposiciones ideales, permite proporcionar una idea intuitiva de por qué TCP hace que el ancho de banda se reparta de forma equitativa entre las conexiones.

En este escenario ideal hemos supuesto que solo las conexiones TCP atraviesan el enlace de cuello de botella, que las conexiones tienen el mismo valor de RTT y que solo hay una conexión TCP asociada con cada pareja origen-destino. En la práctica, estas condiciones normalmente no se dan y las aplicaciones cliente-servidor pueden por tanto obtener cuotas desiguales del ancho de banda del enlace. En particular, se ha demostrado que cuando varias conexiones comparten un cuello de botella común, aquellas sesiones con un valor de RTT menor son capaces de apropiarse más rápidamente del ancho de banda disponible en el enlace, a medida que este va liberándose (es decir, abren más rápidamente sus ventanas de congestión) y por tanto disfrutan de una tasa de transferencia más alta que aquellas conexiones cuyo valor de RTT es más grande [Lakshman 1997].

Equidad y UDP

Acabamos de ver cómo el control de congestión de TCP regula la velocidad de transmisión de una aplicación mediante el mecanismo de la ventana de congestión. Muchas aplicaciones multimedia,

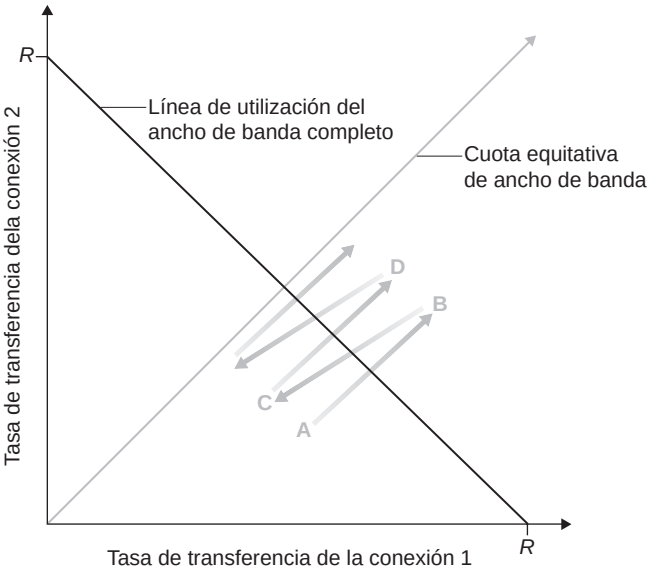


Figura 3.55 ♦ Tasa de transferencia alcanzada por las conexiones TCP 1 y 2.

como las videoconferencias y la telefonía por Internet, a menudo no se ejecutan sobre TCP precisamente por esta razón (no desean que su velocidad de transmisión se regule, incluso aunque la red esté muy congestionada). En lugar de ello, estas aplicaciones prefieren ejecutarse sobre UDP, que no incorpora un mecanismo de control de la congestión. Al ejecutarse sobre UDP, las aplicaciones pueden entregar a la red sus datos de audio y de vídeo a una velocidad constante y, ocasionalmente, perder paquetes, en lugar de reducir sus velocidades a niveles "equitativos" y no perder ningún paquete. Desde la perspectiva de TCP, las aplicaciones multimedia que se ejecutan sobre UDP no son equitativas (no cooperan con las demás conexiones ni ajustan sus velocidades de transmisión apropiadamente). Dado que el control de congestión de TCP disminuye la velocidad de transmisión para hacer frente a un aumento de la congestión (y de las pérdidas) y los orígenes de datos UDP no lo hacen, puede darse el caso de que esos orígenes UDP terminen por expulsar al tráfico TCP. Un área actual de investigación es el desarrollo de mecanismos de control de congestión para Internet que impidan que el tráfico UDP termine por reducir a cero la tasa de transferencia de Internet [Floyd 1999; Floyd 2000; Kohler 2006; RFC 4340].

Equidad y conexiones TCP en paralelo

Pero aunque se pudiera forzar al tráfico UDP a comportarse equitativamente, el problema de la equidad todavía no estaría completamente resuelto. Esto es porque no hay nada que impida a una aplicación basada en TCP utilizar varias conexiones en paralelo. Por ejemplo, los navegadores web a menudo utilizan varias conexiones TCP en paralelo para transferir los distintos objetos contenidos en una página web (en la mayoría de los navegadores puede configurarse el número exacto de conexiones múltiples). Cuando una aplicación emplea varias conexiones en paralelo obtiene una fracción grande del ancho de banda de un enlace congestionado. Por ejemplo, considere un enlace cuya velocidad es R que soporta nueve aplicaciones entrantes cliente-servidor, utilizando cada una de las aplicaciones una conexión TCP. Si llega una nueva aplicación y también emplea una conexión TCP, entonces cada conexión tendrá aproximadamente la misma velocidad de transmisión de $R/10$. Pero si esa nueva aplicación utiliza 11 conexiones TCP en paralelo, entonces la nueva aplicación obtendrá una cuota no equitativa de más de $R/2$. Dado que el tráfico web es el dominante en Internet, las conexiones múltiples en paralelo resultan bastante comunes.

3.7.2 Notificación explícita de congestión (ECN): control de congestión asistido por la red

Desde la estandarización inicial del arranque lento y de la evitación de la congestión a finales de la década de 1980 [RFC 1122], TCP ha implementado el tipo de control de congestión de extremo a extremo que hemos estudiado en la Sección 3.7.1: un emisor TCP no recibe ninguna indicación explícita de congestión por parte de la capa de red, deduciendo en su lugar la existencia de congestión a partir de la pérdida de paquetes observada. Más recientemente, se han propuesto, implementado e implantado extensiones tanto a IP como a TCP [RFC 3168] que permiten a la red señalizar explícitamente la congestión a un emisor y un receptor TCP. Este tipo de control de congestión asistido por la red se conoce con el nombre de **notificación explícita de congestión (ECN, *Explicit Congestion Notification*)**. Como se muestra en la Figura 3.56, están implicados los protocolos TCP e IP.

En la capa de red, se utilizan para ECN dos bits (lo que da un total de cuatro posibles valores) del campo Tipo de Servicio de la cabecera del datagrama IP (de la que hablaremos en la Sección 4.3). Una de las posibles configuraciones de los bits ECN es usada por los routers para indicar que están experimentando congestión. Esta indicación de congestión es transportada entonces en dicho datagrama IP hasta el host de destino, que a su vez informa al host emisor, como se muestra en la Figura 3.56. RFC 3168 no proporciona una definición de cuándo un router está congestionado; esa decisión es una opción de configuración proporcionada por el fabricante del router y sobre la cual decide el operador de la red. Sin embargo, RFC 3168 sí que recomienda que la indicación de

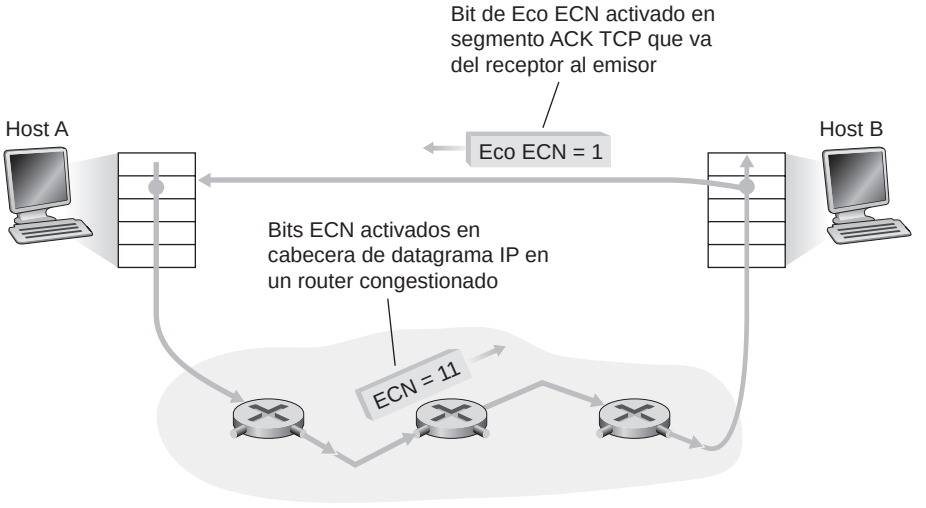


Figura 3.56 ♦ Notificación explícita de congestión: control de congestión asistido por la red.

congestión ECN solo se active cuando exista una congestión persistente. Una segunda configuración de los bits ECN es utilizada por el host emisor para informar a los routers de que el emisor y el receptor son compatibles con ECN, y por tanto capaces de llevar a cabo acciones es respuesta a las indicaciones de congestión de la red proporcionadas por ECN.

Como se muestra en la Figura 3.56, cuando la capa TCP en el host receptor detecta una indicación ECN de congestión en un datagrama recibido, informa de esa situación de congestión a la capa TCP del host emisor, activando el bit ECE (*Explicit Congestion Notification Echo*, bit de Eco ECN; véase la Figura 3.29) en un segmento ACK TCP enviado por el receptor al emisor. Al recibir un ACK con una indicación ECE de congestión, el emisor TCP reacciona a su vez dividiendo por dos la ventana de congestión, igual que lo haría al perderse un segmento mientras se usa retransmisión rápida, y activa el bit CWR (*Congestion Windows Reduced*, ventana de congestión reducida) en la cabecera del siguiente segmento TCP transmitido por el emisor hacia el receptor.

Además de TCP, también otros protocolos de la capa de transporte pueden hacer uso de la señalización ECN de la capa de red. El protocolo DCCP (*Datagram Congestion Control Protocol*, protocolo de control de congestión de datagramas) [RFC 4340] proporciona un servicio no fiable tipo UDP con control de congestión y baja sobrecarga administrativa, que emplea ECN. DCTCP (Data Center TCP, TCP para centros de datos) [Alizadeh 2010], que es una versión de TCP diseñada específicamente para redes de centros de datos, también utiliza ECN.

3.8 Resumen

Hemos comenzado este capítulo estudiando los servicios que un protocolo de la capa de transporte puede proporcionar a las aplicaciones de red. En uno de los extremos, el protocolo de capa de transporte puede ser muy simple y ofrecer un servicio poco sofisticado a las aplicaciones, poniendo a su disposición únicamente una función de multiplexación/demultiplexación para los procesos que se están comunicando. El protocolo UDP de Internet es un ejemplo de ese tipo de protocolo de la capa de transporte poco sofisticado. En el otro extremo, un protocolo de la capa de transporte puede proporcionar a las aplicaciones diversos tipos de garantías, como por ejemplo la de entrega fiable de los datos, garantías sobre los retardos y garantías concernientes al ancho de banda. De todos modos,

los servicios que un protocolo de transporte puede proporcionar están a menudo restringidos por el modelo de servicio del protocolo subyacente de la capa de red. Si el protocolo de la capa de red no puede proporcionar garantías sobre los retardos o de ancho de banda a los segmentos de la capa de transporte, entonces el protocolo de la capa de transporte no puede proporcionar garantías de retardo ni de ancho de banda a los mensajes intercambiados por los procesos.

En la Sección 3.4 hemos visto que un protocolo de la capa de transporte puede proporcionar una transferencia fiable de los datos incluso aunque el protocolo de red subyacente sea no fiable. Allí vimos que son muchas las sutilezas implicadas en la provisión de una transferencia fiable de los datos, pero que dicha tarea puede llevarse a cabo combinando cuidadosamente los paquetes de reconocimiento, los temporizadores, las retransmisiones y los números de secuencia.

Aunque hemos hablado de la transferencia fiable de los datos en este capítulo, debemos tener presente que esa transferencia fiable puede ser proporcionada por los protocolos de las capas de enlace, de red, de transporte o de aplicación. Cualquiera de las cuatro capas superiores de la pila de protocolos puede implementar los reconocimientos, los temporizadores, las retransmisiones y los números de secuencia, proporcionando así un servicio de transferencia de datos fiable a la capa que tiene por encima. De hecho, a lo largo de los años, los ingenieros e informáticos han diseñado e implementado de manera independiente protocolos de las capas de enlace, de red, de transporte y de aplicación que proporcionan una transferencia de datos fiable (aunque muchos de estos protocolos han desaparecido silenciosamente de la escena).

En la Sección 3.5 hemos examinado en detalle TCP, el protocolo fiable y orientado a la conexión de la capa de transporte de Internet. Hemos visto que TCP es bastante complejo, incluyendo técnicas de gestión de la conexión, de control de flujo y de estimación del tiempo de ida y vuelta, además de una transferencia fiable de los datos. De hecho, TCP es bastante más complejo de lo que nuestra descripción deja entrever; hemos dejado fuera de nuestra exposición, intencionadamente, diversos parches, correcciones y mejoras de TCP que están ampliamente implementadas en distintas versiones de dicho protocolo. De todos modos, todas estas complejidades están ocultas a ojos de la aplicación de red. Si el cliente en un host quiere enviar datos de forma fiable a un servidor implementado en otro host, se limita a abrir un socket TCP con el servidor y a bombear datos a través de dicho socket. Afortunadamente, la aplicación cliente-servidor es completamente inconsciente de toda la complejidad de TCP.

En la Sección 3.6 hemos examinado el control de congestión desde una perspectiva amplia, mientras que en la Sección 3.7 hemos mostrado cómo se implementa ese mecanismo de control de congestión en TCP. Allí vimos que el control de congestión es obligatorio para la buena salud de la red. Sin él, una red se puede colapsar fácilmente, sin que al final puedan transportarse datos de terminal a terminal. En la Sección 3.7 vimos que TCP implementa un mecanismo de control de congestión terminal a terminal que incrementa de forma aditiva su tasa de transmisión cuando se evalúa que la ruta seguida por la conexión TCP está libre de congestión, mientras que esa tasa de transmisión se reduce multiplicativamente cuando se producen pérdidas de datos. Este mecanismo también trata de proporcionar a cada conexión TCP que pasa a través de un enlace congestionado una parte equitativa del ancho de banda del enlace. También hemos examinado con cierta profundidad el impacto que el establecimiento de la conexión TCP y el lento arranque de la misma tienen sobre la latencia. Hemos observado que, en muchos escenarios importantes, el establecimiento de la conexión y el arranque lento contribuyen significativamente al retardo terminal a terminal. Conviene recalcar una vez más que, aunque el control de congestión de TCP ha ido evolucionando a lo largo de los años, continúa siendo un área intensiva de investigación y es probable que continúe evolucionando en los próximos años.

El análisis realizado en este capítulo acerca de los protocolos específicos de transporte en Internet se ha centrado en UDP y TCP, que son los dos "caballos de batalla" de la capa de transporte de Internet. Sin embargo, dos décadas de experiencia con estos dos protocolos han permitido identificar una serie de casos en los que ninguno de los dos resulta ideal. Los investigadores han dedicado, por tanto, grandes esfuerzos al desarrollo de protocolos adicionales de la capa de transporte, varios de los cuales son actualmente estándares propuestos por IETF.

El Protocolo de control de congestión para datagramas (DCCP, *Datagram Congestion Control Protocol*) [RFC 4340] proporciona un servicio no fiable con baja carga administrativa y orientado a mensajes, similar a UDP, pero que cuenta con un tipo de control de congestión seleccionado por la aplicación y que es compatible con TCP. Si una aplicación necesita una transferencia de datos fiable o semifiable, entonces el control de congestión sería implementado dentro de la propia aplicación, quizá utilizando los mecanismos que hemos estudiado en la Sección 3.4. DCCP está previsto para utilizarlo en aplicaciones tales como los flujos multimedia (véase el Capítulo 9) que pueden jugar con los compromisos existentes entre los requisitos de temporización y de fiabilidad en la entrega de datos, pero que quieran a la vez poder responder a situaciones de congestión en la red.

El protocolo QUIC (*Quick UDP Internet Connections*) de Google [Iyengar 2016], implementado en el navegador Chromium de Google, proporciona fiabilidad por medio de las retransmisiones, así como corrección de errores, configuración rápida de la conexión y un algoritmo de control de congestión basado en la velocidad que trata de ser compatible con TCP, todo ello implementado como un protocolo de la capa de aplicación por encima de UDP. A principios de 2015, Google informó de que aproximadamente la mitad de todas las solicitudes enviadas desde Chrome a los servidores Google se sirven a través de QUIC.

DCTCP (*Data Center TCP*) [Alizadeh 2010] es una versión de TCP diseñada específicamente para las redes de centros de datos y utiliza ECN para proporcionar un mejor soporte a la mezcla de flujos con tiempos de vida cortos y largos que caracterizan a las cargas de trabajo de los centros de datos.

El Protocolo de transmisión para control de flujos (SCTP, *Stream Control Transmission Protocol*) [RFC 4960, RFC 3286] es un protocolo fiable orientado a mensajes, que permite multiplexar diferentes "flujos" de nivel de aplicación a través de una única conexión SCTP (una técnica conocida con el nombre de "*multi-streaming*"). Desde el punto de vista de la fiabilidad, los diferentes flujos que comparten la conexión se gestionan de forma separada, de modo que la pérdida de paquetes en uno de los flujos no afecte a la entrega de los datos en los otros. SCTP también permite transferir datos a través de dos rutas de salida cuando un host está conectado a dos o más redes; también existe la posibilidad de la entrega opcional de datos fuera de orden, así como otra serie de características interesantes. Los algoritmos de control de flujo y de control de congestión de SCTP son prácticamente los mismos que en TCP.

El protocolo de Control de tasa compatible con TCP (TFRC, *TCP-Friendly Rate Control*) [RFC 5348] es un protocolo de control de congestión más que un protocolo completo de la capa de transporte. TFRC especifica un mecanismo de control de congestión que podría ser utilizado en algún otro protocolo de transporte, como DCCP (de hecho, uno de los dos protocolos seleccionables por la aplicación existentes en DCCP es TFRC). El objetivo de TFRC es suavizar el comportamiento típico en "diente de sierra" (véase la Figura 3.53) que se experimenta en el control de congestión de TCP, al mismo tiempo que se mantiene una tasa de transmisión a largo plazo "razonablemente" próxima a la TCP. Con una tasa de transmisión de perfil más suave que TCP, TFRC está bien adaptado a aplicaciones multimedia tales como la telefonía IP o los flujos multimedia, en donde es importante mantener ese perfil suave de la tasa de transmisión. TFRC es un protocolo "basado en ecuaciones" que utiliza la tasa medida de pérdida de paquetes como entrada para una ecuación [Padhye 2000] que permite estimar cuál sería la tasa de transferencia TCP si una sesión TCP experimentara dicha tasa de pérdidas. Entonces, dicha tasa de transferencia se adopta como objetivo de tasa de transmisión para TFRC.

Solo el futuro nos dirá si DCCP, SCTP, QUIC o TFRC serán adoptados ampliamente o no. Aunque estos protocolos proporcionan claramente una serie de capacidades mejoradas respecto a TCP y UDP, estos dos protocolos han demostrado ser a lo largo de los años "lo suficientemente buenos". El que un "mejor" protocolo termine venciendo a otro que es "suficientemente bueno" dependerá de una compleja mezcla de aspectos técnicos, sociales y empresariales.

En el Capítulo 1 hemos visto que una red de computadoras puede dividirse entre lo que se denomina la "frontera de la red" y el "núcleo de la red". La frontera de la red cubre todo lo que sucede en los sistemas terminales. Habiendo ya cubierto la capa de aplicación y la capa de transporte,

nuestro análisis de la frontera de la red está completo, así que ha llegado el momento de explorar el núcleo de la red. Comenzaremos nuestro viaje en los dos capítulos siguientes, donde estudiaremos la capa red, y seguiremos en el Capítulo 6 dedicado a la capa de enlace.

Problemas y cuestiones de repaso

Capítulo 3 Cuestiones de repaso

SECCIONES 3.1–3.3

R1. Suponga que la capa de red proporciona el siguiente servicio: la capa de red del host de origen acepta un segmento con un tamaño máximo de 1.200 bytes y una dirección de host de destino de la capa de transporte. La capa de red garantiza la entrega del segmento a la capa de transporte en el host de destino. Suponga que en el host de destino pueden ejecutarse muchos procesos de aplicaciones de red.

   a. Diseñe el protocolo de la capa de transporte más simple posible que entregue los datos de la aplicación al proceso deseado en el host de destino. Suponga que el sistema operativo del host de destino ha asignado un número de puerto de 4 bytes a cada proceso de aplicación en ejecución.

   b. Modifique este protocolo de manera que proporcione una "dirección de retorno" al proceso de destino.

   c. En sus protocolos, ¿la capa de transporte "tiene que hacer algo" en el núcleo de la red de computadoras?

R2. Imagine una sociedad en la que todo el mundo perteneciera a una familia de seis miembros, todas las familias vivieran en su propia casa, cada casa tuviera una dirección única y cada persona de cada casa tuviera un nombre único. Imagine que esa sociedad dispone de un servicio de correos que transporta las cartas desde una vivienda de origen hasta una vivienda de destino. El servicio de correos requiere que (i) la carta se introduzca en un sobre y que (ii) la dirección de la casa de destino (y nada más) esté claramente escrita en el sobre. Suponga también que en cada familia hay un delegado que tiene asignada la tarea de recoger y distribuir las cartas a los restantes miembros de la familia. Las cartas no necesariamente proporcionan una indicación acerca de los destinatarios.

   a. Partiendo de la solución del Problema R1, describa un protocolo que el delegado de la familia pueda utilizar para entregar las cartas de un miembro de la familia emisora a un miembro de la familia receptora.

   b. En su protocolo, ¿el servicio de correos tiene que abrir el sobre y examinar la carta para proporcionar este servicio?

R3. Considere una conexión TCP entre el host A y el host B. Suponga que los segmentos TCP que viajan del host A al host B tienen un número de puerto de origen *x* y un número de puerto de destino *y*. ¿Cuáles son los números de puerto de origen y de destino para los segmentos que viajan del host B al host A?

R4. Describa por qué un desarrollador de aplicaciones puede decidir ejecutar una aplicación sobre UDP en lugar de sobre TCP.

R5. ¿Por qué razón el tráfico de voz y de vídeo suele enviarse sobre TCP en lugar de sobre UDP en la Internet de hoy día? (*Sugerencia*: la respuesta que estamos buscando no tiene nada que ver con el mecanismo de control de congestión de TCP.)

R6. ¿Es posible que una aplicación disfrute de una transferencia de datos fiable incluso si se ejecuta sobre UDP? En caso afirmativo, explique cómo.

R7. Sea un proceso del host C que tiene un socket UDP con el número de puerto 6789. Suponga también que los hosts A y B envían cada uno de ellos un segmento UDP al host C con el número de puerto de destino 6789. ¿Serán dirigidos ambos segmentos al mismo socket del host C? En caso afirmativo, ¿cómo sabrá el proceso del host C que estos dos segmentos proceden de dos hosts distintos?

R8. Suponga que un servidor web se ejecuta en el puerto 80 del host C. Suponga también que este servidor web utiliza conexiones persistentes y que actualmente está recibiendo solicitudes de dos hosts diferentes, A y B. ¿Están siendo enviadas todas las solicitudes al mismo socket del host C? Si están siendo pasadas a través de sockets diferentes, ¿utilizan ambos sockets el puerto 80? Explique y justifique su respuesta.

SECCIÓN 3.4

R9. En los protocolos `rdt` estudiados, ¿por qué necesitábamos introducir números de secuencia?

R10. En los protocolos `rdt` estudiados, ¿por qué necesitábamos introducir temporizadores?

R11. Suponga que el retardo de ida y vuelta entre el emisor y el receptor es constante y conocido por el emisor. ¿Se necesitaría en este caso un temporizador en el protocolo `rdt 3.0`, suponiendo que los paquetes pueden perderse? Explique su respuesta.

R12. Visite el applet de Java *Go-Back-N* en el sitio web del libro.

a. Haga que el emisor envíe cinco paquetes y luego detenga la animación antes de que cualquiera de los cinco paquetes alcance su destino. A continuación, elimine el primer paquete y reanude la animación. Describa lo que ocurre.

b. Repita el experimento, pero ahora deje que el primer paquete alcance su destino y elimine el primer paquete de reconocimiento. Describa lo que ocurre.

c. Para terminar, pruebe a enviar seis paquetes. ¿Qué ocurre?

R13. Repita el problema R12, pero ahora utilizando el applet de Java con repetición selectiva (SR). ¿En qué se diferencian los protocolos SR y GBN?

SECCIÓN 3.5

R14. ¿Verdadero o falso?

a. El host A está enviando al host B un archivo de gran tamaño a través de una conexión TCP. Suponga que el host B no tiene datos que enviar al host A. El host B no enviará paquetes de reconocimiento al host A porque el host B no puede superponer esos reconocimientos sobre los datos.

b. El tamaño de la ventana de recepción de TCP `VentCongestion` nunca varía mientras dura la conexión.

c. Suponga que el host A está enviando al host B un archivo de gran tamaño a través de una conexión TCP. El número de bytes no reconocidos que A envía no puede exceder el tamaño del buffer del receptor.

d. Suponga que el host A está enviando al host B un archivo de gran tamaño a través de una conexión TCP. Si el número de secuencia de un segmento en esta conexión es m, entonces el número de secuencia del siguiente segmento necesariamente tiene que ser $m + 1$.

e. El segmento TCP contiene un campo en su cabecera para `VentRecepcion`.

f. Suponga que el último `RTTMuestra` en una conexión TCP es igual a 1 segundo. El valor actual del `IntervaloFinTemporización` para la conexión será necesariamente ≥ 1 segundo.

g. Suponga que el host A envía al host B un segmento con el número de secuencia 38 y 4 bytes de datos a través de una conexión TCP. En este mismo segmento el número de reconocimiento necesariamente tiene que ser 42.

R15. Suponga que el host A envía dos segmentos TCP seguidos al host B a través de una conexión TCP. El primer segmento tiene el número de secuencia 90 y el segundo tiene el número de secuencia 110.

a. ¿Cuántos datos hay en el primer segmento?

b. Suponga que el primer segmento se pierde pero el segundo llega a B. En el paquete de reconocimiento que el host B envía al host A, ¿cuál será el número de reconocimiento?

R16. Considere el ejemplo de la conexión Telnet de la Sección 3.5. Unos pocos segundos después de que el usuario escriba la letra 'C', escribe la letra 'R'. Después de escribir la letra 'R', ¿cuántos segmentos se envían y qué valores se almacenan en los campos número de secuencia y número de reconocimiento de los segmentos?

SECCIÓN 3.7

R17. Suponga que existen dos conexiones TCP en un cierto enlace de cuello de botella con una velocidad de *R* bps. Ambas conexiones tienen que enviar un archivo de gran tamaño (en la misma dirección a través del enlace de cuello de botella). Las transmisiones de los archivos se inician en el mismo instante. ¿Qué velocidad de transmisión podría proporcionar TCP a cada una de las conexiones?

R18. ¿Verdadero o falso? En el control de congestión de TCP, si el temporizador del emisor caduca, el valor de `umbralAL` se hace igual a la mitad de su valor anterior.

R19. En la exposición acerca de la división TCP del recuadro de la Sección 3.7, se establecía que el tiempo de respuesta con la división TCP es aproximadamente igual a $4 \cdot \text{RTT}_{\text{FE}} + \text{RTT}_{\text{BE}} +$ el tiempo de procesamiento. Justifique esta afirmación.

Problemas

P1. Suponga que el cliente A inicia una sesión Telnet con el servidor S. Aproximadamente en el mismo instante, el cliente B también inicia una sesión Telnet con el servidor S. Proporcione los posibles números de puerto de origen y de destino para:

a. Los segmentos enviados de A a S.

b. Los segmentos enviados de B a S.

c. Los segmentos enviados de S a A.

d. Los segmentos enviados de S a B.

e. Si A y B son hosts diferentes, ¿es posible que el número de puerto de origen en los segmentos que van de A a S sea el mismo que en los segmentos que van de B a S?

f. ¿Qué ocurre si A y B son el mismo host?

P2. Considere la Figura 3.5. ¿Cuáles son los valores de los puertos de origen y de destino en los segmentos que fluyen desde el servidor de vuelta a los procesos cliente? ¿Cuáles son las direcciones IP de los datagramas de la capa de red que transportan los segmentos de la capa de transporte?

P3. UDP y TCP utilizan el complemento a 1 para calcular sus sumas de comprobación. Suponga que tiene los tres bytes de 8 bits siguientes: 01010011, 01100110, 01110100. ¿Cuál es el complemento a 1 de la suma de estos bytes? (Observe que aunque UDP y TCP emplean palabras de 16 bits para calcular la suma de comprobación, en este problema le pedimos que considere sumas de 8 bits). Explique cómo funciona. ¿Por qué UDP utiliza el complemento a 1 de la suma; es decir, por qué no simplemente emplea la suma? Con el esquema del complemento a 1, ¿cómo detecta el receptor los errores? ¿Es posible que un error de un solo bit no sea detectado? ¿Qué ocurre si hay 2 bits erróneos?

P4. a. Suponga que tiene los 2 bytes siguientes: 01011100 y 01100101. ¿Cuál es el complemento a 1 de la suma de estos 2 bytes?

b. Suponga que tiene los 2 bytes siguientes: 11011010 y 01100101. ¿Cuál es el complemento a 1 de la suma de estos 2 bytes?

c. Para los bytes del apartado (a), proporcione un ejemplo en el que un bit cambie de valor en cada uno de los 2 bytes y aún así el complemento a 1 no varíe.

P5. Suponga que el receptor UDP calcula la suma de comprobación de Internet para el segmento UDP recibido y comprueba que se corresponde con el valor almacenado en el campo de suma de comprobación. ¿Puede el receptor estar completamente seguro de que no hay ningún bit erróneo? Explique su respuesta.

P6. Recuerde el motivo de corregir el protocolo `rdt2.1`. Demuestre que el receptor mostrado en la Figura 3.57 y el emisor mostrado en la Figura 3.11 pueden llegar a entrar en un estado de bloqueo tal que cada uno de ellos esté esperando a que se produzca un suceso que no ocurrirá nunca.

P7. En el protocolo `rdt3.0`, los paquetes ACK que fluyen del receptor al emisor no tienen números de secuencia (aunque tienen un campo ACK que contiene el número de secuencia del paquete que están reconociendo). ¿Por qué estos paquetes ACK no requieren números de secuencia?

P8. Dibuje la máquina de estados finitos correspondiente al lado receptor del protocolo `rdt3.0`.

P9. Dibuje un esquema que muestre la operación del protocolo `rdt3.0` cuando los paquetes de datos y los paquetes de reconocimiento están corrompidos. Utilice un esquema similar al mostrado en la Figura 3.16.

P10. Sea un canal que puede perder paquetes pero del que se conoce su retardo máximo. Modifique el protocolo `rdt2.1` para incluir los fines de temporización y las retransmisiones del emisor. Argumente de manera informal por qué su protocolo puede comunicarse correctamente a través de este canal.

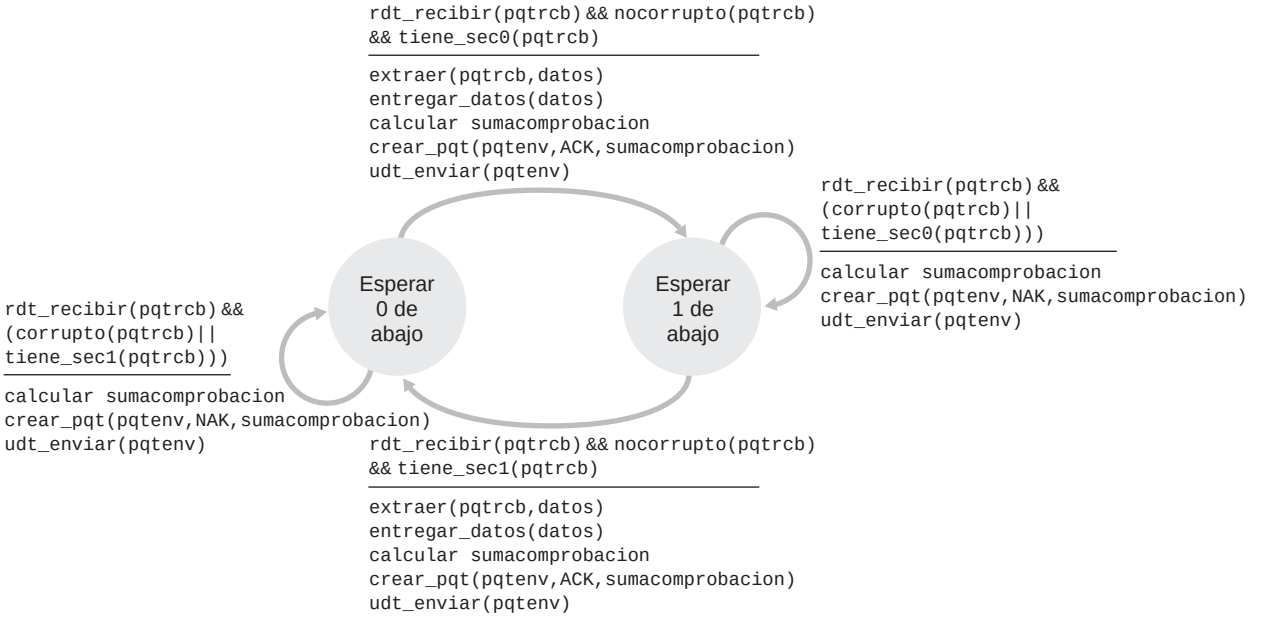


Figura 3.57 ♦ Un receptor incorrecto para el protocolo `rdt 2.1`.

P11. Considere el receptor `rdt2.2` mostrado en la Figura 3.14 y la creación de un paquete nuevo en la transición a sí mismo (esto es, la transición del estado de vuelta hacia sí mismo) en los estados Esperar 0 de abajo y Esperar 1 de abajo: `pqtenv=crear_pqt(ACK,1,sumacomprobacion)` y `pqtenv=crear_pqt(ACK,0,sumacomprobacion)`. ¿Funcionaría correctamente el protocolo si se eliminara esta acción de la transición al mismo estado en el estado Esperar 1 de abajo? Justifique su respuesta. ¿Qué ocurrirá si se elimina este suceso de la transición del estado Esperar 0 de abajo a sí mismo? [*Sugerencia*: en este último caso, considere lo que ocurriría si el primer paquete que va del emisor al receptor estuviera corrompido.]

P12. El lado del emisor de `rdt3.0` simplemente ignora (es decir, no realiza ninguna acción) todos los paquetes recibidos que contienen un error o que presentan un valor erróneo en el campo número de reconocimiento (`acknum`) de un paquete de reconocimiento. Suponga que, en tales circunstancias, `rdt3.0` simplemente retransmite el paquete de datos actual. ¿Funcionaría en estas condiciones el protocolo? (*Sugerencia*: piense en lo que ocurriría si solo hubiera errores de bit; no se producen pérdidas de paquetes pero sí pueden ocurrir sucesos de fin prematuro de la temporización. Considere cuántas veces se envía el paquete *n*, cuando *n* tiende a infinito.)

P13. Considere el protocolo `rdt 3.0`. Dibuje un diagrama que muestre que si la conexión de red entre el emisor y el receptor puede reordenar los mensajes (es decir, que dos mensajes que se propagan por el medio físico existente entre el emisor y el receptor pueden ser reordenados), entonces el protocolo de bit alternante no funcionará correctamente (asegúrese de identificar claramente el sentido en el que no funcionará correctamente). En el diagrama debe colocar el emisor a la izquierda y el receptor a la derecha, con el eje de tiempos en la parte inferior de la página y deberá mostrar el intercambio de los mensajes de datos (D) y de reconocimiento (A). No olvide indicar el número de secuencia asociado con cualquier segmento de datos o de reconocimiento.

P14. Considere un protocolo de transferencia de datos fiable que solo utiliza paquetes de reconocimiento negativo. Imagine que el emisor envía datos con muy poca frecuencia. ¿Sería preferible un protocolo con solo emplea paquetes NAK a uno que utilice paquetes ACK? ¿Por qué? Suponga ahora que el emisor tiene muchos datos que transmitir y que la conexión terminal a terminal experimenta muy pocas pérdidas. En este segundo caso, ¿sería preferible un protocolo que solo emplee paquetes NAK a otro que utilice paquetes ACK? ¿Por qué?

P15. Considere el ejemplo mostrado en la Figura 3.17. ¿Cuál tiene que ser el tamaño de la ventana para que la tasa de utilización del canal sea mayor del 98 por ciento? Suponga que el tamaño de un paquete es de 1.500 bytes, incluyendo tanto los campos de cabecera como los datos.

P16. Suponga que una aplicación utiliza el protocolo `rdt 3.0` como su protocolo de la capa de transporte. Como el protocolo de parada y espera tiene una tasa de utilización del canal muy baja (como se ha demostrado en el ejemplo de conexión que atraviesa el país de costa a costa), los diseñadores de esta aplicación permiten al receptor devolver una serie (más de dos) de ACK 0 y ACK 1 alternantes incluso si los correspondientes datos no han llegado al receptor. ¿Debería este diseño de aplicación aumentar la tasa de utilización del canal? ¿Por qué? ¿Existe algún problema potencial con esta técnica? Explique su respuesta.

P17. Suponga que tenemos dos entidades de red, A y B, que están conectadas mediante un canal bidireccional perfecto (es decir, cualquier mensaje enviado será recibido correctamente; el canal no corromperá, no perderá y no reordenará los paquetes). A y B se entregan mensajes de datos entre sí de manera alternante: en primer lugar, A debe entregar un mensaje a B, después B entrega un mensaje a A; a continuación, A entregará un mensaje a B, y así sucesivamente. Si una entidad se encuentra en un estado en el que no debería enviar un mensaje al otro lado y se produce un suceso como `rdt_enviar(datos)` procedente de arriba puede simplemente ignorarlo con una llamada a `rdt_inhabilitar_enviar(datos)`, que informa a la capa superior de que actualmente no es posible enviar datos. [Nota: esta simplificación se hace para no tener que preocuparse por el almacenamiento de datos en buffer].

Diseñe una especificación de la máquina de estados finitos (FSM) para este protocolo (una máquina FSM para A y una máquina FSM para B). Fíjese en que en este caso no tiene que preocuparse por el mecanismo de fiabilidad; el punto clave de esta cuestión es crear la especificación de una máquina FSM que refleje el comportamiento sincronizado de las dos entidades. Deberá emplear los siguientes sucesos y acciones, los cuales tienen el mismo significado que en el protocolo `rdt1.0` de la Figura 3.9: `rdt_enviar(datos)`, `pqt=crear_pqt(datos)`, `udt_enviar(pqt)`, `rdt_recibir(pqt)`, `extraer (pqt,datos)`, `entregar_datos(datos)`. Asegúrese de que el protocolo refleja la alternancia estricta de envío entre A y B. Asegúrese también de indicar los estados iniciales para A y B en sus descripciones de la FSM.

P18. En el protocolo SR genérico que hemos estudiado en la Sección 3.4.4, el emisor transmite un mensaje tan pronto como está disponible (si se encuentra dentro de la ventana) sin esperar a recibir un paquete de reconocimiento. Suponga ahora que deseamos disponer de un protocolo SR que envíe mensajes de dos en dos. Es decir, el emisor enviará una pareja de mensajes y enviará la siguiente pareja de mensajes solo cuando sepa que los dos mensajes de la primera pareja se han recibido correctamente.

Suponga que el canal puede perder mensajes pero no corromperlos ni tampoco reordenarlos. Diseñe un protocolo de control de errores para un servicio de transferencia de mensajes fiable y unidireccional. Proporcione una descripción de las máquinas de estados finitos del emisor y del receptor. Describa el formato de los paquetes intercambiados por el emisor y el receptor, y viceversa. Si utiliza alguna llamada a procedimiento distinta de las empleadas en la Sección 3.4 (por ejemplo, `udt_enviar()`, `iniciar_temporizador()`, `rdt_recibir()`, etc.), defina claramente las acciones que realizan. Proporcione un ejemplo (una gráfica temporal del emisor y del receptor) que muestre cómo este protocolo se recupera de la pérdida de paquetes.

P19. Considere un escenario en el que el host A desea enviar simultáneamente paquetes a los hosts B y C. El host A está conectado a B y C a través de un canal de multidifusión (*broadcast*) (un paquete enviado por A es transportado por el canal tanto a B como a C). Suponga que el canal de multidifusión que conecta A, B y C puede perder y corromper de manera independiente los paquetes (es decir, puede ocurrir, por ejemplo, que un paquete enviado desde A llegue correctamente a B, pero no a C). Diseñe un protocolo de control de errores similar a un protocolo de parada y espera que permita transferir paquetes de forma fiable de A a B y C, de manera que A no obtendrá nuevos datos de la capa superior hasta que sepa que tanto B como C han recibido correctamente el paquete actual. Proporcione las descripciones de las máquinas de estados finitos de A y C. (*Sugerencia*: la FSM de B será prácticamente la misma que la de C.) Proporcione también una descripción del formato o formatos de paquete utilizados.

P20. Considere un escenario en el que el host A y el host B desean enviar mensajes al host C. Los hosts A y C están conectados mediante un canal que puede perder y corromper (pero no reordenar) los mensajes. Los hosts B y C están conectados a través de otro canal (independiente del canal que conecta a A y C) que tiene las mismas propiedades. La capa de transporte del host C tiene que alternar la entrega de los mensajes que A y B tienen que pasar a la capa superior (es decir, primero entrega los datos de un paquete de A y luego los datos de un paquete de B, y así sucesivamente). Diseñe un protocolo de control de errores de tipo parada y espera para transferir de forma fiable los paquetes de A y B a C, con una entrega alternante en el host C, como hemos descrito anteriormente. Proporcione las descripciones de las FSM de A y C. (*Sugerencia*: la FSM de B será prácticamente la misma que la de A.) Proporcione también una descripción del formato o formatos de paquete utilizados.

P21. Suponga que tenemos dos entidades de red, A y B. B tiene que enviar a A un conjunto de mensajes de datos, cumpliendo los siguientes convenios. Cuando A recibe una solicitud de la capa superior para obtener el siguiente mensaje de datos (D) de B, A tiene que enviar un mensaje de solicitud (R) a B a través del canal que va de A a B. Solo cuando B recibe

un mensaje R puede devolver un mensaje de datos (D) a A a través del canal de B a A. A tiene que entregar exactamente una copia de cada mensaje D a la capa superior. Los mensajes R se pueden perder (pero no corromper) en el canal de A a B; los mensajes D, una vez enviados, siempre son correctamente entregados. El retardo a lo largo de ambos canales es desconocido y variable.

Diseñe (proporcione una descripción de la FSM de) un protocolo que incorpore los mecanismos apropiados para compensar las pérdidas del canal de A a B e implemente el paso de los mensajes a la capa superior de la entidad A, como se ha explicado anteriormente. Utilice solo aquellos mecanismos que sean absolutamente necesarios.

P22. Sea un protocolo GBN con un tamaño de ventana de emisor de 4 y un rango de números de secuencia de 1.024. Suponga que en el instante *t* el siguiente paquete en orden que el receptor está esperando tiene el número de secuencia *k*. Suponga que el medio de transmisión no reordena los mensajes. Responda a las siguientes cuestiones:

a. ¿Cuáles son los posibles conjuntos de números de secuencia que pueden estar dentro de la ventana del emisor en el instante *t*? Justifique su respuesta.

b. ¿Cuáles son todos los valores posibles del campo ACK en todos los posibles mensajes que están actualmente propagándose de vuelta al emisor en el instante *t*? Justifique su respuesta.

P23. Considere los protocolos GBN y SR. Suponga que el tamaño del espacio de números de secuencia es *k*. ¿Cuál es la máxima ventana de emisor permitida que evitará la ocurrencia de problemas como los indicados en la Figura 3.27 para cada uno de estos protocolos?

P24. Responda verdadero o falso a las siguientes preguntas y justifique brevemente sus respuestas:

a. Con el protocolo SR, el emisor puede recibir un ACK para un paquete que se encuentra fuera de su ventana actual.

b. Con GBN, el emisor puede recibir un ACK para un paquete que se encuentra fuera de su ventana actual.

c. El protocolo de bit alternante es igual que el protocolo SR pero con un tamaño de ventana en el emisor y en el receptor igual a 1.

d. El protocolo de bit alternante es igual que el protocolo GBN pero con un tamaño de ventana en el emisor y en el receptor igual a 1.

P25. Hemos dicho que una aplicación puede elegir UDP como protocolo de transporte porque UDP ofrece a la aplicación un mayor grado de control (que TCP) en lo relativo a qué datos se envían en un segmento y cuándo.

a. ¿Por qué una aplicación tiene más control sobre qué datos se envían en un segmento?

b. ¿Por qué una aplicación tiene más control sobre cuándo se envía el segmento?

P26. Se desea transferir un archivo de gran tamaño de *L* bytes del host A al host B. Suponga un MSS de 536 bytes.

a. ¿Cuál es el valor máximo de *L* tal que los números de secuencia de TCP no se agoten? Recuerde que el campo número de secuencia de TCP tiene 4 bytes.

b. Para el valor de *L* que haya obtenido en el apartado (a), calcule el tiempo que tarda en transmitirse el archivo. Suponga que a cada segmento se añade un total de 66 bytes para la cabecera de la capa de transporte, de red y de enlace de datos antes de enviar el paquete resultante a través de un enlace a 155 Mbps. Ignore el control de flujo y el control de congestión de modo que A pueda bombear los segmentos seguidos y de forma continuada.

P27. Los hosts A y B están comunicándose a través de una conexión TCP y el host B ya ha recibido de A todos los bytes hasta el byte 126. Suponga que a continuación el host A envía dos segmentos seguidos al host B. El primer y el segundo segmentos contienen, respectivamente, 80 y 40 bytes de datos. En el primer segmento, el número de secuencia es 127, el número del

puerto de origen es 302 y el número de puerto de destino es 80. El host B envía un paquete de reconocimiento cuando recibe un segmento del host A.

a. En el segundo segmento enviado del host A al B, ¿cuáles son el número de secuencia, el número del puerto de origen y el número del puerto de destino?

b. Si el primer segmento llega antes que el segundo segmento, ¿cuál es el número de reconocimiento, el número del puerto de origen y el número del puerto de destino en el ACK correspondiente al primer segmento?

c. Si el segundo segmento llega antes que el primero, ¿cuál es el número de reconocimiento en el ACK correspondiente a este segmento?

d. Suponga que los dos segmentos enviados por A llegan en orden a B. El primer paquete de reconocimiento se pierde y el segundo llega después de transcurrido el primer intervalo de fin de temporización. Dibuje un diagrama de temporización que muestre estos segmentos y todos los restantes segmentos y paquetes de reconocimiento enviados. (Suponga que no se producen pérdidas de paquetes adicionales.) Para cada uno de los segmentos que incluya en su diagrama, especifique el número de secuencia y el número de bytes de datos; para cada uno de los paquetes de reconocimiento que añada, proporcione el número de reconocimiento.

P28. Los hosts Ay B están directamente conectados mediante un enlace a 100 Mbps. Existe una conexión TCP entre los dos hosts y el host A está enviando al host B un archivo de gran tamaño a través de esta conexión. El host A puede enviar sus datos de la capa de aplicación a su socket TCP a una velocidad tan alta como 120 Mbps pero el host B solo puede leer los datos almacenados en su buffer de recepción TCP a una velocidad máxima de 50 Mbps. Describa el efecto del control de flujo de TCP.

P29. En la Sección 3.5.6 se han estudiado las cookies SYN.

a. ¿Por qué es necesario que el servidor utilice un número de secuencia inicial especial en SYNACK?

b. Suponga que un atacante sabe que un host objetivo utiliza cookies SYN. ¿Puede el atacante crear conexiones semi-abiertas o completamente abiertas enviando simplemente un paquete ACK al host objetivo? ¿Por qué?

c. Suponga que un atacante recopila una gran cantidad de números de secuencia iniciales enviados por el servidor. ¿Puede el atacante hacer que el servidor cree muchas conexiones completamente abiertas enviando paquetes ACK con esos números de secuencia iniciales? ¿Por qué?

P30. Considere la red mostrada en el escenario 2 de la Sección 3.6.1. Suponga que ambos hosts emisores A y B tienen definidos valores de fin de temporización fijos.

a. Demuestre que aumentar el tamaño del buffer finito del router puede llegar a hacer que se reduzca la tasa de transferencia (λ_{out}).

b. Suponga ahora que ambos hosts ajustan dinámicamente su valores de fin de temporización (como lo hace TCP) basándose en el retardo del buffer del router. ¿Incrementar el tamaño del buffer ayudaría a incrementar la tasa de transferencia? ¿Por qué?

P31. Suponga que los cinco valores de `RTTMuestra` medidos (véase la Sección 3.5.3) son 106 ms, 120 ms, 140 ms, 90 ms y 115 ms. Calcule el valor de `RTTEstimado` después de obtener cada uno de estos valores de `RTTMuestra`, utilizando un valor de $\alpha = 0{,}125$ y suponiendo que el valor de `RTTEstimado` era 100 ms justo antes de obtener el primero de estas cinco muestras. Calcule también el valor de `RTTDesv` después de obtener cada muestra, suponiendo un valor de $\beta = 0{,}25$ y que el valor de `RTTDesv` era de 5 ms justo antes de obtener la primera de estas cinco muestras. Por último, calcule el `IntervaloFinTemporización` de TCP después de obtener cada una de estas muestras.

P32. Considere el procedimiento de TCP para estimar RTT. Suponga que $\alpha = 0{,}1$. Sea `RTTMuestra`$_1$ la muestra de RTT más reciente, `RTTMuestra`$_2$ la siguiente muestra de RTT más reciente, y así sucesivamente.

a. Para una conexión TCP determinada, suponga que han sido devueltos cuatro paquetes de reconocimiento con las correspondientes muestras de RTT, `RTTMuestra`$_4$, `RTTMuestra`$_3$, `RTTMuestra`$_2$ y `RTTMuestra`$_1$. Exprese `RTTEstimado` en función de las cuatro muestras de RTT.

b. Generalize la fórmula para *n* muestras de RTT.

c. En la fórmula del apartado (b), considere que *n* tiende a infinito. Explique por qué este procedimiento de cálculo del promedio se conoce como media móvil exponencial.

P33. En la Sección 3.5.3, se ha estudiado la estimación de RTT en TCP. ¿Por qué cree que TCP evita medir `RTTMuestra` para los segmentos retransmitidos?

P34. ¿Cuál es la relación entre la variable `EnviarBase` de la Sección 3.5.4 y la variable `UltimoByteRecibido` de la Sección 3.5.5?

P35. ¿Cuál es la relación entre la variable `UltimoByteRecibido` de la Sección 3.5.5 y la variable y de la Sección 3.5.4?

P36. En la Sección 3.5.4 hemos visto que TCP espera hasta que ha recibido tres ACK duplicados antes de realizar una retransmisión rápida. ¿Por qué cree que los diseñadores de TCP han decidido no realizar una retransmisión rápida después de recibir el primer ACK duplicado correspondiente a un segmento?

P37. Compare GBN, SR y TCP (sin paquetes ACK retardados). Suponga que los valores de fin de temporización de los tres protocolos son los suficientemente grandes como para que 5 segmentos de datos consecutivos y sus correspondientes ACK puedan ser recibidos (si no se producen pérdidas en el canal) por el host receptor (host B) y el host emisor (host A), respectivamente. Suponga que el host A envía 5 segmentos de datos al host B y que el segundo segmento (enviado desde A) se pierde. Al final, los 5 segmentos de datos han sido recibidos correctamente por el host B.

a. ¿Cuántos segmentos ha enviado en total el host A y cuántos ACK ha enviado en total el host B? ¿Cuáles son sus números de secuencia? Responda a esta pregunta para los tres protocolos.

b. Si los valores de fin de temporización para los tres protocolos son mucho mayores que 5 RTT, ¿qué protocolo entregará correctamente los cinco segmentos de datos en el menor intervalo de tiempo?

P38. En la descripción de TCP de la Figura 3.53, el valor del umbral, `umbralAL`, se define como `umbral=VentCongestion/2` en varios sitios y el valor de `umbralAL` se hace igual a la mitad del tamaño de la ventana cuando se produce un suceso de pérdida. ¿Tiene que ser la velocidad a la que el emisor está transmitiendo cuando se produce un suceso de pérdida aproximadamente igual a `VentCongestion` segmentos por RTT? Explique su respuesta. Si su respuesta es no, ¿puede sugerir una forma diferente en la que se podría fijar el valor de `umbralAL`?

P39. Considere la Figura 3.46(b). Si λ'_{in} aumenta por encima de $R/2$, ¿puede λ_{out} incrementarse por encima de $R/3$? Explique su respuesta. Considere ahora la Figura 3.46(c). Si λ'_{in} aumenta por encima de $R/2$, ¿puede λ_{out} aumentar por encima de $R/4$ suponiendo que un paquete será reenviado dos veces como media desde el router al receptor? Explique su respuesta.

Nota de vídeo

Examen del comportamiento de TCP

P40. Considere la Figura 3.58. Suponiendo que TCP Reno es el protocolo que presenta el comportamiento mostrado en la figura, responda a las siguientes preguntas. En todos los casos, deberá proporcionar una breve explicación que justifique su respuesta.

a. Identifique los intervalos de tiempo cuando TCP está operando en modo de arranque lento.

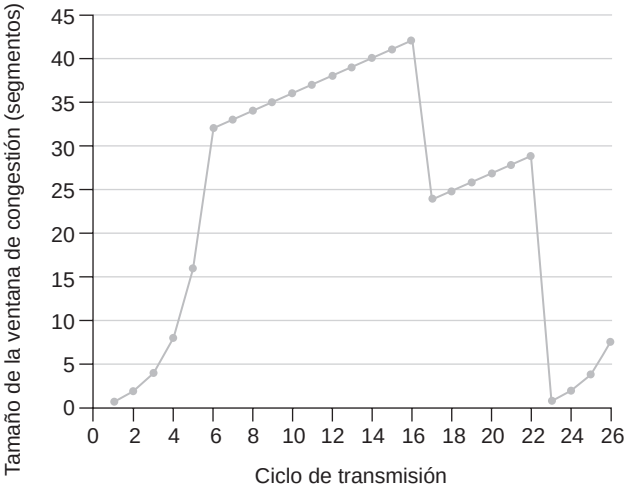


Figura 3.58 ♦ Tamaño de ventana de TCP en función del tiempo.

b. Identifique los intervalos de tiempo cuando TCP está operando en el modo de evitación de la congestión.

c. Después del ciclo de transmisión 16, ¿se detecta la pérdida de segmento mediante tres ACK duplicados o mediante un fin de temporización?

d. Después del ciclo de transmisión 22, ¿se detecta la pérdida de segmento mediante tres ACK duplicados o mediante un fin de temporización?

e. ¿Cuál es el valor inicial de `umbralAL` en el primer ciclo de transmisión?

f. ¿Cuál es el valor de `umbralAL` transcurridos 18 ciclos de transmisión?

g. ¿Cuál es el valor de `umbralAL` transcurridos 24 ciclos de transmisión?

h. ¿Durante cuál ciclo de transmisión se envía el segmento 70?

i. Suponiendo que se detecta una pérdida de paquete después del ciclo de transmisión 26 a causa de la recepción de un triple ACK duplicado, ¿cuáles serán los valores del tamaño de la ventana de congestión y de `umbralAL`?

j. Suponga que se utiliza TCP Tahoe (en lugar de TCP Reno) y que se han recibido triples ACK duplicados en el ciclo de transmisión 16. ¿Cuáles serán los valores del tamaño de la ventana de congestión y de `umbralAL` en el ciclo de transmisión 19?

k. Suponga otra vez que se utiliza TCP Tahoe y que se produce un suceso de fin de temporización en el ciclo de transmisión 22. ¿Cuántos paquetes han sido enviados entre los ciclos de transmisión 17 a 22, ambos inclusive?

P41. Utilice la Figura 3.55, que ilustra la convergencia del algoritmo AIMD de TCP. Suponga que en lugar de un decrecimiento multiplicativo, TCP disminuye el tamaño de la ventana en una cantidad constante. ¿Convergería el algoritmo AIAD resultante hacia un algoritmo de cuota equitativa? Justifique su respuesta utilizando un diagrama similar al de la Figura 3.55.

P42. En la Sección 3.5.4, hemos explicado que el intervalo de fin de temporización se duplica después de un suceso de fin de temporización. Este mecanismo es una forma de control de congestión. ¿Por qué TCP necesita un mecanismo de control de congestión basado en ventana (como hemos estudiado en la Sección 3.7) además de un mecanismo de duplicación del intervalo de fin de temporización?

P43. El host A está enviando un archivo de gran tamaño al host B a través de una conexión TCP. En esta conexión nunca se pierden paquetes y los temporizadores nunca caducan. La velocidad

de transmisión del enlace que conecta el host A con Internet es *R* bps. Suponga que el proceso del host A es capaz de enviar datos a su socket TCP a una velocidad de *S* bps, donde $S = 10 \cdot R$. Suponga también que el buffer de recepción de TCP es lo suficientemente grande como para almacenar el archivo completo y que el buffer emisor solo puede almacenar un porcentaje del archivo. ¿Qué impide al proceso del host A pasar datos de forma continua a su socket TCP a una velocidad de *S* bps? ¿El mecanismo de control de flujo de TCP, el mecanismo de control de congestión de TCP o alguna otra cosa? Razone su respuesta.

P44. Se envía un archivo de gran tamaño de un host a otro a través de una conexión TCP sin pérdidas.

a. Suponga que TCP utiliza el algoritmo AIMD para su control de congestión sin fase de arranque lento. Suponiendo que `VentCongestion` aumenta 1 MSS cada vez que se recibe un lote de paquetes ACK y suponiendo que los intervalos RTT son aproximadamente constantes, ¿Cuánto tiempo tarda `VentCongestion` en aumentar de 6 MSS a 12 MSS (si no se producen sucesos de pérdida de paquetes)?

b. ¿Cuál es la tasa de transferencia media (en función de MSS y RTT) para esta conexión hasta llegar al periodo RTT número 6?

P45. Recuerde la descripción macroscópica de la tasa de transferencia de TCP. En el periodo de tiempo que va desde que la velocidad de la conexión varía entre $W/(2 \cdot RTT)$ y W/RTT, solo se pierde un paquete (justo al final del periodo).

a. Demuestre que la tasa de pérdidas (fracción de paquetes perdidos) es igual a:

$$L = \text{tasa de pérdidas} = \frac{1}{\frac{3}{8}W^2 + \frac{3}{4}W}$$

b. Utilice el resultado anterior para demostrar que si una conexión tiene una tasa de pérdidas igual a *L*, entonces su tasa promedio es aproximadamente igual a

$$\approx \frac{1{,}22 \cdot MSS}{RTT\sqrt{L}}$$

P46. Considere una única conexión TCP (Reno) que emplea un enlace a 10Mbps que no almacena en buffer ningún dato. Suponga que este enlace es el único enlace congestionado entre los hosts emisor y receptor. Suponga también que el emisor TCP tiene que enviar al receptor un archivo de gran tamaño y que el buffer de recepción del receptor es mucho más grande que la ventana de congestión. Haremos además las siguientes suposiciones: el tamaño de segmento TCP es de 1.500 bytes, el retardo de propagación de ida y vuelta de esta conexión es igual a 150 milisegundos y esta conexión TCP siempre se encuentra en la fase de evitación de la congestión, es decir, ignoramos la fase de arranque lento.

a. ¿Cuál es el tamaño máximo de ventana (en segmentos) que esta conexión TCP puede alcanzar?

b. ¿Cuáles son el tamaño medio de ventana (en segmentos) y la tasa de transferencia media (en bps) de esta conexión TCP?

c. ¿Cuánto tiempo tarda esta conexión TCP en alcanzar de nuevo su tamaño de ventana máximo después de recuperarse de una pérdida de paquete?

P47. Continuando con el escenario descrito en el problema anterior, suponga que el enlace a 10Mbps puede almacenar en buffer un número finito de segmentos. Razone por qué para que el enlace esté siempre ocupado enviando datos, deberíamos seleccionar un tamaño de buffer que sea al menos igual al producto de la velocidad del enlace *C* y el retardo de propagación de ida y vuelta entre el emisor y el receptor.

P48. Repita el Problema 46, pero sustituyendo el enlace a 10 Mbps por un enlace a 10 Gbps. Observe que en la respuesta al apartado (c) habrá demostrado que se tarda mucho tiempo en que el tamaño de la ventana de congestión alcance su máximo después de recuperarse de una pérdida de paquete. Diseñe una solución que resuelva este problema.

P49. Sea *T* (medido en RTT) el intervalo de tiempo que una conexión TCP tarda en aumentar el tamaño de su ventana de congestión de *W*/2 a *W*, donde *W* es el tamaño máximo de la ventana de congestión. Demuestre que *T* es una función de la tasa de transferencia media de TCP.

P50. Considere un algoritmo AIMD de TCP simplificado en el que el tamaño de la ventana de congestión se mide en número de segmentos, no en bytes. En la fase de incremento aditivo, el tamaño de la ventana de congestión se incrementa en un segmento cada RTT. En la fase de decrecimiento multiplicativo, el tamaño de la ventana de congestión se reduce a la mitad (si el resultado no es un entero, redondee al entero más próximo). Suponga que dos conexiones TCP, C_1 y C_2, comparten un enlace congestionado cuya velocidad es de 30 segmentos por segundo. Suponemos que tanto C_1 como C_2 están en la fase de evitación de la congestión. El intervalo RTT de la conexión C_1 es igual a 50 milisegundos y el de la conexión C_2 es igual a 100 milisegundos. Suponemos que cuando la velocidad de los datos en el enlace excede la velocidad del enlace, todas las conexiones TCP experimentan pérdidas de segmentos de datos.

a. Si en el instante t_0 el tamaño de la ventana de congestión de ambas conexiones, C_1 y C_2, es de 10 segmentos, ¿cuáles serán los tamaños de dichas ventanas de congestión después de transcurridos 1.000 milisegundos?

b. ¿Obtendrán estas dos conexiones, a largo plazo, la misma cuota de ancho de banda del enlace congestionado? Explique su respuesta.

P51. Continúe con la red descrita en el problema anterior, pero ahora suponga que las dos conexiones TCP, C_1 y C_2, tienen el mismo intervalo RTT de 100 milisegundos. Suponga que en el instante t_0, el tamaño de la ventana de congestión de C_1 es de 15 segmentos pero el tamaño de la ventana de congestión de C_2 es igual a 10 segmentos.

a. ¿Cuáles serán los tamaños de las ventanas de congestión después de transcurridos 2.200 ms?

b. ¿Obtendrán estas dos conexiones, a largo plazo, la misma cuota de ancho de banda del enlace congestionado?

c. Decimos que dos conexiones están sincronizadas si ambas conexiones alcanzan su tamaño de ventana máximo al mismo tiempo y alcanzan su tamaño mínimo de ventana también al mismo tiempo. ¿Terminarán con el tiempo sincronizándose estas dos conexiones? En caso afirmativo, ¿cuáles son sus tamaños máximos de ventana?

d. ¿Ayudará esta sincronización a mejorar la tasa de utilización del enlace compartido? ¿Por qué? Esboce alguna idea para evitar esta sincronización.

P52. Veamos una modificación del algoritmo de control de congestión de TCP. En lugar de utilizar un incremento aditivo podemos emplear un incremento multiplicativo. Un emisor TCP incrementa su tamaño de ventana según una constante pequeña positiva a ($0 < a < 1$) cuando recibe un ACK válido. Halle la relación funcional existente entre la tasa de pérdidas *L* y el tamaño máximo de la ventana de congestión *W*. Demuestre que para esta conexión TCP modificada, independientemente de la tasa media de transferencia de TCP, una conexión TCP siempre invierte la misma cantidad de tiempo en incrementar el tamaño de su ventana de congestión de *W*/2 a *W*.

P53. En nuestra exposición sobre el futuro de TCP de la Sección 3.7 hemos destacado que para alcanzar una tasa de transferencia de 10 Gbps, TCP solo podría tolerar una probabilidad de pérdida de segmentos de $2 \cdot 10^{-10}$ (o lo que es equivalente, un suceso de pérdida por cada 5.000.000.000 segmentos). Indique de dónde se obtienen los valores $2 \cdot 10^{-10}$ (1 por cada 5.000.000) para los valores de RTT y MSS dados en la Sección 3.7. Si TCP tuviera que dar soporte a una conexión a 100 Gbps, ¿qué tasa de pérdidas sería tolerable?

P54. En nuestra exposición sobre el control de congestión de TCP de la Sección 3.7, implícitamente hemos supuesto que el emisor TCP siempre tiene datos que enviar. Consideremos ahora el caso en que el emisor TCP envía una gran cantidad de datos y luego en el instante t_1 se queda inactivo (puesto que no tiene más datos que enviar). TCP permanece inactivo durante un periodo de tiempo relativamente largo y en el instante t_2 quiere enviar más datos. ¿Cuáles son las ventajas y las desventajas de que TCP tenga que utilizar los valores de `VentCongestion` y `umbralAL` de t_1 cuando comienza a enviar datos en el instante t_2? ¿Qué alternativa recomendaría? ¿Por qué?

P55. En este problema vamos a investigar si UDP o TCP proporcionan un cierto grado de autenticación del punto terminal.

a. Considere un servidor que recibe una solicitud dentro de un paquete UDP y responde a la misma dentro de un paquete UDP (por ejemplo, como en el caso de un servidor DNS). Si un cliente con la dirección IP X suplanta su dirección con la dirección Y, ¿a dónde enviará el servidor su respuesta?

b. Suponga que un servidor recibe un SYN con la dirección IP de origen Y, y después de responder con un SYNACK, recibe un ACK con la dirección IP de origen Y y con el número de reconocimiento correcto. Suponiendo que el servidor elige un número de secuencia inicial aleatorio y que no existe ningún atacante interpuesto (*man-in-the-middle*), ¿puede el servidor estar seguro de que el cliente está en la dirección Y (y no en alguna otra dirección X que esté intentando suplantar a Y)?

P56. En este problema, vamos a considerar el retardo introducido por la fase de arranque lento de TCP. Se tiene un cliente y un servidor web directamente conectados mediante un enlace a velocidad *R*. Suponga que el cliente desea extraer un objeto cuyo tamaño es exactamente igual a 15 *S*, donde *S* es el tamaño máximo de segmento (MSS). Sea RTT el tiempo de transmisión de ida y vuelta entre el cliente y el servidor (suponemos que es constante). Ignorando las cabeceras del protocolo, determine el tiempo necesario para recuperar el objeto (incluyendo el tiempo de establecimiento de la conexión TCP) si

a. $4\ S/R > S/R + RTT > 2\ S/R$

b. $S/R + RTT > 4\ S/R$

c. $S/R > RTT$.

Tareas de programación

Implementación de un protocolo de transporte fiable

En esta tarea de programación tendrá que escribir el código para la capa de transporte del emisor y el receptor con el fin de implementar un protocolo de transferencia de datos fiable simple. Hay disponibles dos versiones de esta práctica de laboratorio: la versión del protocolo de bit alternante y la versión del protocolo GBN. Esta práctica de laboratorio le resultará entretenida y su implementación diferirá muy poco de lo que se necesita en una situación real.

Puesto que probablemente no dispone de máquinas autónomas (con un sistema operativo que pueda modificar), su código tendrá que ejecutarse en un entorno simulado hardware/software. Sin embargo, la interfaz de programación proporcionada a sus rutinas (el código que efectuará las llamadas a sus entidades desde las capas superior e inferior) es muy similar a la que se utiliza en un entorno UNIX real. (De hecho, las interfaces software descritas en esta tarea de programación son mucho más realistas que los emisores y receptores con bucles infinitos que se describen en muchos textos.) También se simula el arranque y la detención de temporizadores, y las interrupciones de los temporizadores harán que se active su rutina de tratamiento de temporizadores.

La tarea completa de laboratorio, así como el código que tendrá que compilar con su propio código está disponible en el sitio web del libro en www.pearsonhighered.com/cs-resources.

Práctica de laboratorio con Wireshark: exploración de TCP

En esta práctica de laboratorio tendrá que utilizar su navegador web para acceder a un archivo almacenado en un servidor web. Como en las prácticas de laboratorio con Wireshark anteriores, tendrá que utilizar Wireshark para capturar los paquetes que lleguen a su computadora. A diferencia de las prácticas anteriores, *también* podrá descargar una traza de paquetes que Wireshark puede leer del servidor web del que haya descargado el archivo. En esta traza del servidor encontrará los paquetes que fueron generados a causa de su propio acceso al servidor web. Analizará las trazas del lado del cliente y de lado del servidor para explorar los aspectos de TCP. En particular, tendrá que evaluar el rendimiento de la conexión TCP entre su computadora y el servidor web. Tendrá que trazar el comportamiento de la ventana de TCP e inferirá la pérdida de paquetes, las retransmisiones, el comportamiento del control de flujo y del control de congestión y el tiempo de ida y vuelta estimado.

Como con el resto de las prácticas de laboratorio con Wireshark, la descripción completa de esta práctica está disponible en el sitio web del libro en www.pearsonhighered.com/cs-resources.

Práctica de laboratorio con Wireshark: exploración de UDP

En esta corta práctica de laboratorio, realizará una captura y un análisis de paquetes de su aplicación favorita que utilice UDP (por ejemplo, DNS o una aplicación multimedia como Skype). Como hemos visto en la Sección 3.3, UDP es un protocolo de transporte simple. En esta práctica de laboratorio tendrá que investigar los campos de cabecera del segmento UDP, así como el cálculo de la suma de comprobación.

Como con el resto de las prácticas de laboratorio con Wireshark, la descripción completa de esta práctica está disponible en el sitio web del libro en www.pearsonhighered.com/cs-resources.

Van Jacobson

Van Jacobson trabaja actualmente en Google y fue previamente investigador en PARC. Antes de eso, fue co-fundador y Director Científico de Packet Design. Y antes aun, fue Director Científico en Cisco. Antes de unirse a Cisco, dirigió el Grupo de Investigación de Redes en el Lawrence Berkeley National Laboratory y dio clases en las universidades de Berkeley y Stanford en California. Van recibió el premio SIGCOMM de la ACM en 2001 por su sobresaliente contribución profesional al campo de las redes de comunicaciones, así como el premio Kobayashi del IEEE en 2002 por "contribuir a la comprensión de la congestión de red y a desarrollar mecanismos de control de congestión que han permitido la exitosa expansión de Internet". Fue elegido como miembro de la Academia Nacional de Ingeniería de los Estados Unidos en 2004.

Describa uno o dos de los proyectos más interesantes en los que haya trabajado durante su carrera. ¿Cuáles fueron los principales desafíos?

La escuela nos enseña muchas formas de encontrar respuestas. En todos los problemas interesantes en los que he trabajado, el desafío consistía en encontrar la pregunta correcta. Cuando Mike Karels y yo empezamos a analizar el tema de la congestión TCP, nos pasamos meses observando trazas de protocolos y paquetes, preguntándonos "¿Por qué está fallando?". Un día, en la oficina de Mike, uno de nosotros dijo "La razón por la que no consigo ver por qué falla, es porque ni siquiera comprendo por qué llegó a funcionar". Resultó que esa era la pregunta correcta, y nos obligó a inventar el "reloj ack" que hace que TCP funcione. Después de eso, el resto fue sencillo.

Hablando en términos más generales, ¿cuál cree que es el futuro de las redes y de Internet?

Para la mayoría de la gente, Internet es la Web. Los expertos en redes sonríen educadamente, porque sabemos que la Web no es otra cosa que una aplicación que se ejecuta sobre Internet, pero...¿y si la gente tuviera razón? El objetivo de Internet es permitir la conversación entre parejas de hosts. El de la Web es la producción y el consumo distribuidos de información. La "propagación de información" es una visión muy general de la comunicación, de la cual las "conversaciones de dos" constituyen un diminuto subconjunto. Necesitamos pasar al ámbito más general. Las redes, hoy en día, tienen que ver con los medios de difusión (radios, redes PON, etc.), simulando ser un cable punto a punto. Eso es espantosamente ineficiente. Por todo el mundo se intercambian terabytes por segundo de datos mediante memorias USB o teléfonos inteligentes, pero no sabemos cómo tratar eso como "redes". Los ISP están ocupados instalando cachés y redes CDN para distribuir vídeo y audio de forma escalable. Las cachés son una parte necesaria de la solución, pero no hay ninguna parte de las redes actuales - desde la Teoría de la Información, la de Colas o la de Tráfico, hasta las especificaciones de los protocolos Internet - que nos diga cómo proyectarlas e implantarlas. Creo y espero que, en los próximos años, las redes evolucionarán para abarcar esa visión de la comunicación mucho más amplia que subyace a la Web.

¿Qué personas le han inspirado profesionalmente?

Cuando estaba en la Universidad, Richard Feynman vino de visita y nos dio una charla. Nos habló acerca de una parte de la Teoría Cuántica con la que yo había estado luchando todo el semestre, y su explicación fue tan simple y tan lúcida, que lo que había sido un galimatías incomprensible para mi, se convirtió en algo obvio e inevitable. Esa capacidad para ver y transmitir la simplicidad que subyace a nuestro complejo mundo, me parece un don raro y maravilloso.

¿Qué le recomendaría a los estudiantes que quieran orientar su carrera profesional hacia la computación y las redes?

Es un campo maravilloso: las computadoras y las redes probablemente han tenido más impacto sobre la sociedad que cualquier otro invento, desde la imprenta. Las redes se ocupan, fundamentalmente, de conectar cosas, y su estudio nos ayuda a hacer conexiones mentales: el forrajeo de las hormigas y las danzas de las abejas ilustran el diseño de protocolos mejor que los documentos RFC; los atascos de tráfico o la masa de gente que sale de un estadio abarrotado son la esencia de la congestión; y los estudiantes que tratan de encontrar vuelo de vuelta hacia la universidad tras una escapada por Navidad son la mejor representación del enrutamiento dinámico. Si sientes interés por multitud de cosas distintas y quieres dejar huella, es difícil imaginar un campo mejor que el de las redes.