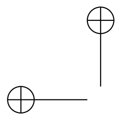
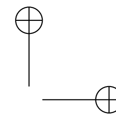



Capítulo 12

Confiabilidad y control de flujo en TCP

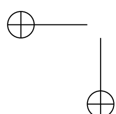
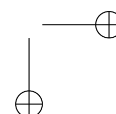
Al terminar este capítulo, entenderás:

- Cómo se establece y termina una conexión TCP.
- Qué implicaciones tiene el control de flujo de TCP.
- Cómo detecta y resuelve TCP los errores de transmisión.

Ya hemos hablado del protocolo de transporte TCP. Aquí veremos el soporte que ofrece para confiabilidad y control de flujo [23]. TCP utiliza un mecanismo de retransmisión de mensajes y confirmaciones de tipo repetición selectiva con confirmación acumulativa y ventana deslizante. Es *full duplex* de modo que aplica lo dicho en § 11.4. También puede utilizar un sistema de confirmación selectiva para mejorar la eficiencia, aunque es algo opcional que deben negociar explícitamente los extremos durante el establecimiento de la conexión.

Para cada conexión establecida, el SO crea un TCB (Transmission Control Block). Se trata de una estructura de datos que almacena toda la información relacionada con la conexión: dirección del otro extremo, números de secuencia, temporizadores, puertos, estado, etc. y también el buffer dónde coloca los datos que el proceso emisor envía —`socket.send()`—, llamado *buffer de envío* (*sending buffer*). En el otro extremo, el receptor tiene un *buffer de recepción* (*receiving buffer*) dónde almacena los datos hasta que el proceso receptor los solicite, con `socket.recv()`.

A partir de los datos almacenados en el buffer de envío, el procedimiento de **segmentación**, determina cuántos bytes se incluirán en el siguiente mensaje TCP (llamado «segmento») que se envíe. El tamaño del segmento depende de varias variables que veremos en las siguientes secciones y también cuando hablemos de congestión (§ 13.5).

Una diferencia interesante respecto a los protocolos de estilo ARQ es que TCP no identifica los segmentos. En lugar de ello, numera cada uno de los bytes del flujo completo, para lo que usa un entero de 32 bits. En particular, el número de secuencia que corresponde al primer byte de la carga útil de cada segmento aparece en el campo *sequence number* de su cabecera y es por eso que se le llama «número de secuencia del segmento». Si por ejemplo se envía un segmento con número de secuencia 1000 y su carga útil tiene 500 bytes, el número de secuencia del siguiente byte (que será el primero del siguiente segmento) será 1500. Del mismo modo, la confirmación que envía el receptor corresponde al número de secuencia del siguiente byte que espera recibir, y que aparecerá en el campo *acknowledge number* de la cabecera.

12.1. Conexión

El procedimiento de conexión de TCP permite a los dos extremos intercambiar varios parámetros de configuración. El más importante es el *número de secuencia inicial* de cada extremo (o ISN). Cada uno de los extremos genera su ISN mediante una función pseudo-aleatoria, que aparecerá únicamente en el primer segmento que envíen. A partir de ese valor, se numera cada byte del flujo de datos. La conexión responde a un patrón muy concreto de tres mensajes, llamado *three-way handshake*, que se suele traducir literalmente como *triple apretón de manos*.

```
1. --> seq:1200, SYN
2  <-- seq:4800, SYN/ACK, ack:1201
3. --> seq:1201, ACK, ack:4801
```

Listado 12.1: TCP: Secuencia de una conexión

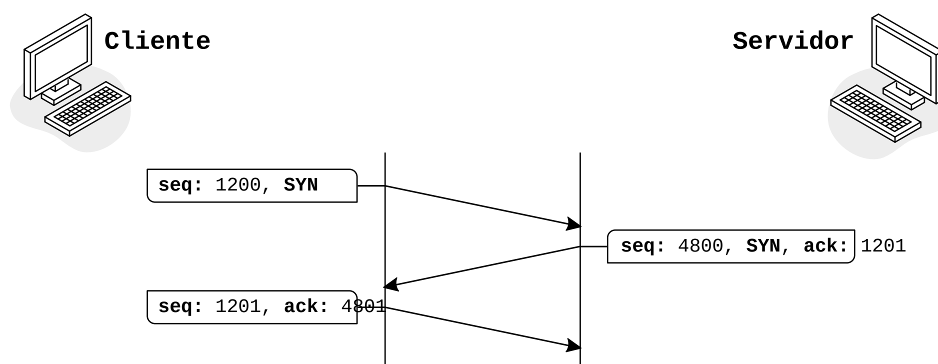


Figura 12.1: TCP: Patrón *triple handshake* durante la conexión

La conexión la inicia siempre el cliente, por lo que el primer segmento es suyo. Este segmento incluye el ISN del cliente (es el 1200 del ejemplo de la Figura 12.1 y del esquema equivalente del Listado 12.1) y lleva activo el flag SYN[1] únicamente. El servidor responde con un segmento que incluye su ISN (4800 en la figura) y lleva activos los flags SYN y ACK, y por tanto el contenido del campo *acknowledge number* es relevante y sirve como confirmación del primer mensaje del cliente (1201). En realidad el flag ACK ya siempre estará activo hasta el fin de la conexión. El segundo segmento del cliente lleva el ACK 4801, reconociendo así el segmento del servidor. El número de secuencia de este tercer segmento es 1201 a pesar de que el cliente no ha enviado aún ningún byte de datos. Esto ocurre tanto en la conexión como en la desconexión y es una excepción al modo en que se incrementan los números de secuencia. A partir de ese instante, la conexión queda establecida y ambos extremos conocen los números de secuencia que va a utilizar el otro, es decir, los números de secuencia están sincronizados (*SYNchronized*).

Puedes ver la captura de la conexión real en la Figura 12.2. El comando ejecuta `tshark` en segundo plano, espera 1 segundo y luego conecta al puerto 80 del servidor `example.net` con `ncat`. Si lo pruebas en tu computador, recuerda parar `tshark` al terminar, por ejemplo con `kill%1`[2].

```
$ sudo tshark -f "tcp port 80" & sleep 1; ncat example.net 80
1 192.168.0.37 → 93.184.215.14 TCP 74 50192 → 80 [SYN] Seq=0 Win=32120 Len=0
                                                 MSS=1460 SACK_PERM WS=128
2 93.184.215.14 → 192.168.0.37 TCP 74 80 → 50192 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0
                                                 MSS=1452 SACK_PERM WS=512
3 192.168.0.37 → 93.184.215.14 TCP 66 50192 → 80 [ACK] Seq=1 Ack=1 Win=32128 Len=0
```

Listado 12.2: TCP: Captura de una conexión

La captura indica tanto para el cliente como para el servidor que los ISN son 0 (Seq=0), pero cuidado, porque esto no es real. Para poder hacer un seguimiento más sencillo de los números de secuencia, `tshark`[3] no muestra el número de secuencia que aparece en el segmento, si no un número relativo al ISN; por eso siempre es 0 en el primer segmento. Hay una opción para forzar a `tshark` a mostrar los números de secuencia reales. Puedes ver el

[1] por *synchronize*

[2] El comando `kill` envía una señal (por defecto SIGTERM) al proceso indicado, en este caso con la intención de terminarlo. En el capítulo 2 vimos el uso de este y otro muchos comandos y recuerda también que el anexo A es un pequeño catálogo de comandos habituales. Y por supuesto, siempre puedes usar el comando `man` para obtener información de cualquier comando.

[3] Y la mayoría de los *sniffers*

mismo ejemplo con números de secuencia reales en la captura 12.3. Se han omitido algunos campos que no son relevantes para este ejemplo.

```
$ sudo tshark -f "tcp port 80" -o tcp.relative_sequence_numbers:FALSE &
$ sleep 1; ncat example.net 80
1 192.168.0.37 → 93.184.215.14 TCP 74 48200 → 80 [SYN] Seq=7967281 Win=32120 Len=0
2 93.184.215.14 → 192.168.0.37 TCP 74 80 → 48200 [SYN, ACK] Seq=6696166 Ack=7967282 Len=0
3 192.168.0.37 → 93.184.215.14 TCP 66 48200 → 80 [ACK] Seq=7967282 Ack=6696167 Len=0
```

Listado 12.3: TCP: Captura de una conexión (con números de secuencia reales)

12.2. Tamaño máximo de segmento

La especificación de TCP dice que todo computador debe ser capaz de recibir segmentos de como mínimo 536 bytes en IPv4 y 1220 en IPv6 [23]. Este requisito está relacionado con la capacidad de almacenamiento en el *buffer de recepción*. Sin embargo, si el emisor tiene la certeza de que el receptor puede manejar segmentos mayores, es preferible que lo haga ya que eso aumenta el rendimiento del canal.

En general es más eficiente enviar pocos mensajes grandes que muchos pequeños. Enviando menos mensajes se reducen las interacciones entre capas a través de las interfaces, se reduce el peso relativo de las cabeceras en el total de datos transmitidos —la *sobrecarga de cabeceras*—, se minimiza el impacto de las retransmisiones y el trabajo que conlleva para routers y conmutadores.

Sin embargo, hay un límite superior relacionado con el tamaño máximo de datagrama IP, que a su vez está limitado por la MTU de la red. Si el tamaño del segmento elegido provoca que se supere el MTU, tendremos un paquete que no cabe en una única trama y se tendrá que fragmentar. La fragmentación debería evitarse cuando sea posible porque aumenta el impacto de los errores de transmisión. Si alguno de los fragmentos se pierde o corrompe, habrá que retransmitir el segmento completo, que de nuevo será fragmentado.

MSS determina el tamaño máximo que tendrá la carga útil de los segmentos durante la conexión. Si por ejemplo, cliente y servidor son vecinos de un mismo enlace Ethernet (que tiene una MTU de 1500 bytes) podrían utilizar un MSS de $1500 - 20 - 20 = 1460$ bytes. Esos 40 bytes que descontamos corresponden a las cabeceras estándar[4] de TCP e IP.

[4] estándar = sin opciones

Para utilizar un MSS distinto al valor por defecto, un receptor puede incluir la opción TCP MSS (tipo 2) en el primer segmento (el que lleva el flag SYN). Esta opción anuncia cuál es el tamaño máximo de segmento que puede manejar ese extremo. Hay que destacar que esto no es una negociación, no están acordando un valor común para el MSS; cada extremo decide por su cuenta y puede ser un valor diferente en cada uno de los los dos sentidos de la comunicación. La opción MSS proporciona un campo de 2 bytes para especificar el tamaño de segmentos, es decir, el valor máximo que se puede anunciar es 64 KiB, aunque el valor 65 535 es especial: significa «infinito» y está pensado para utilizar con los *jumbogramas* de IPv6.

En la captura 12.2 puedes ver que los dos mensajes iniciales llevan la opción MSS. El cliente indica 1460 bytes mientras que el servidor indica 1452 bytes.

Como los extremos de una conexión TCP pueden no conocer el MTU de la ruta, podrían elegir valores que obligan a realizar fragmentación. Un router o firewall intermedio podría evitar este problema aplicando una técnica conocida como MSS *clamping*, que consiste en *sobrescribir* el valor de MSS en las cabeceras de la conexión fijando un valor más bajo, adecuado para la ruta que seguirán los paquetes de esa conexión.

El programador también tiene la posibilidad de consultar y modificar el MSS de una conexión establecida. El Listado 12.4 muestra un cliente Python que conecta con un servidor escuchando en el mismo nodo (localhost). Consulta el valor de MSS antes de establecer la conexión y obtiene 536 bytes, que es el valor mínimo. Después, lo vuelve a consular una vez establecida la conexión y obtiene 32 741 bytes. Este valor tan alto se debe precisamente a que ambos cliente y servidor se ejecutan el propio nodo y por eso el SO sabe de antemano que no habrá fragmentación.

```
$ python
>>> import socket
>>> sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> sock.getsockopt(socket.IPPROTO_TCP, socket.TCP_MAXSEG)
536
>>> sock.connect(('127.0.0.1', 2000))
>>> sock.getsockopt(socket.IPPROTO_TCP, socket.TCP_MAXSEG)
32741
```

Listado 12.4: TCP: Consulta del MSS de una conexión

Si se fija el valor de MSS antes de establecer la conexión, el socket lo aplicará en la opción MSS de la cabecera al establecerse la conexión. Sin embargo, cambiarlo después normalmente no tiene ningún efecto.

12.3. Desconexión

La desconexión se parece a la conexión en que ambos extremos necesitan asegurarse que el otro ha terminado su tarea y está de acuerdo en dar por terminada la conexión (al menos en la modalidad más *civilizada*). La desconexión sin embargo puede implicar 3 o 4 mensajes, como vamos a ver.

A diferencia de la conexión, que solo la puede iniciar el cliente, la desconexión la puede comenzar cualquiera de los dos extremos en cualquier momento. La desconexión se basa en el envío del flag FIN[5] que indica que su emisor ha terminado de enviar sus datos, pero a pesar de eso todavía está dispuesto a aceptar nuevos datos. El receptor confirma la recepción de ese mensaje aumentando el número de secuencia en su siguiente confirmación de manera análoga a lo que ocurre con el flag SYN durante la conexión. Si el receptor también ha terminado de enviar datos, ese mismo segmento llevará también el flag FIN. El Listado 12.5 y la Figura 12.2 representan la desconexión con 3 mensajes que acabamos de describir. El Listado 12.6 muestra una captura de una desconexión real.

```
1. --> seq:4000, FIN
2. <-- seq:5000, FIN, ack:4001
3. --> seq:4001, ack:5001
```

LISTADO 12.5: TCP: Secuencia de una desconexión en 3 segmentos

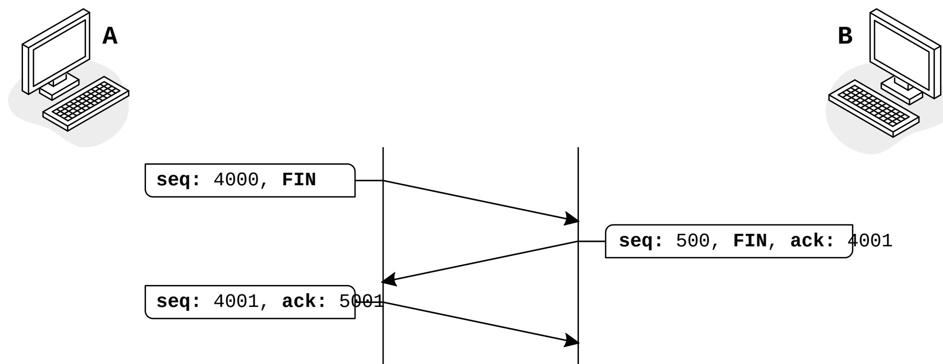


FIGURA 12.2: TCP: Patrón de mensajes de la desconexión en 3 segmentos

[5] por *finish*

```
$ tshark -f "tcp port 80"
[...]
4 192.168.0.37 → 93.184.215.14 TCP 66 50192 → 80 [FIN, ACK] Seq=1 Ack=1 Win=32128 Len=0
5 93.184.215.14 → 192.168.0.37 TCP 66 80 → 50192 [FIN, ACK] Seq=1 Ack=2 Win=65536 Len=0
6 192.168.0.37 → 93.184.215.14 TCP 66 50192 → 80 [ACK] Seq=2 Ack=2 Win=32128 Len=0
```

Listado 12.6: TCP: Captura de una desconexión

También puede ocurrir que el receptor del mensaje FIN tengo datos por enviar. En ese caso enviará una confirmación, y continuará enviando sus datos pendientes. Cuando termine emitirá por su cuenta un segmento con el flag FIN. En esta situación la desconexión requiere 4 segmentos $(2 + 2)$, en lugar de 3. El Listado 12.7 y la Figura 12.3 representa una situación en que el nodo que no inicia la desconexión todavía tenía 100 bytes por enviar. Fíjate que el mensaje que lleva datos y su correspondiente confirmación no forman parte de la desconexión; Por eso no se contabilizan en el Listado 12.7 y aparecen en gris en la Figura 12.3.

```
1. --> seq:4000, FIN
2. <-- seq:5000, ack:4001
   <-- seq:5000, len:100
   --> seq:4001, ack:5100
3. <-- seq:5100, FIN
4. --> seq:4001, ack:5101
```

Listado 12.7: TCP: Secuencia de una desconexión en 4 segmentos

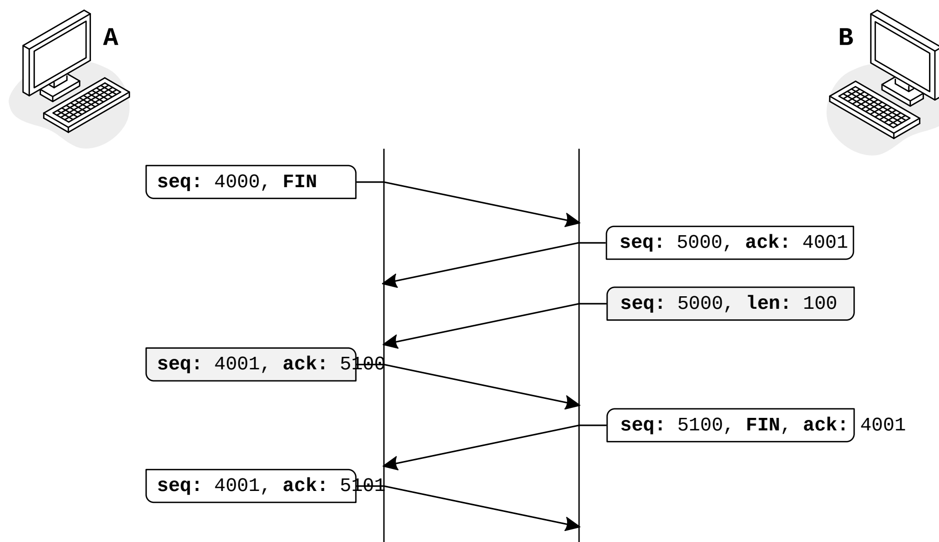


Figura 12.3: TCP: Patrón de mensajes de la desconexión en 4 segmentos

Una vez terminada la conexión, los dos extremos pueden liberar sus respectivos TCB, que incluyen los buffers de envío y recepción, y otros recursos asociados.

12.3.1. Tiempo de silencio

Cuando una conexión termina y vuelve a establecerse inmediatamente después, existe una pequeña posibilidad de que los números de secuencia empleados en la nueva conexión se repitan o solapen con los de la conexión anterior. Si esto ocurriera, el receptor podría confundir los segmentos de la nueva conexión con los de la anterior, con resultados impredecibles.

Para evitar este problema, cuando se termina un servidor, el SO impone un *tiempo de silencio* (*quiet time*). Durante ese tiempo el socket se mantiene en un estado de *limbo* (llamado `TIME_WAIT`) que impide vincular un nuevo socket (`bind`) al mismo puerto. Si se intenta levantar de nuevo un servidor en el mismo puerto se obtendrá el mismo error que si realmente estuviera ocupado: *Dirección en uso* (*Address already in use*).

El tiempo de silencio es igual a MSL (Maximum Segment Lifetime), que como su nombre indica, es el tiempo máximo de vida de un segmento, es decir, el tiempo máximo que un segmento puede estar moviéndose por la red antes de llegar a su destino o ser descartado. Se asume que este tiempo es aproximadamente 2 minutos, aunque depende del implementador.

Aunque es una restricción de seguridad importante para un servidor en producción, el programador puede desactivar esta salvaguarda para un socket concreto. Esto es útil por ejemplo para un servidor en su fase de desarrollo o pruebas, y para el que sabe que una confusión entre segmentos no tendrá consecuencias. El Listado 12.8 muestra cómo desactivar el tiempo de silencio.

```
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

LISTADO 12.8: TCP: Desactivación del tiempo de silencio

Puedes profundizar más en esta cuestión consultado la sección «The TCP Quiet Time Concept» en la RFC 793 [24].

12.4. Reset

La utilidad principal del flag RST es rechazar una conexión entrante. Ante un intento de conexión (segmento con el flag SYN), el SO del nodo receptor responderá con un segmento con el flag RST si la conexión no puede ser establecida. Esto puede ocurrir por varios motivos:

- El puerto destino está cerrado, es decir, no está vinculado a ningún proceso servidor. Puedes ver un ejemplo de este caso en el Listado 12.9.
- Hay un servidor, pero no tiene posibilidad de aceptar nuevas conexiones en ese momento, quizá por una limitación de recursos.
- El segmento de inicio de conexión no es válido.

```
$ tshark -f "tcp por 2000"
1 192.168.1.235 → 192.168.1.1 TCP 74 48318 → 2000 [SYN] Seq=0 Win=32120 Len=0 MSS=1460
2 192.168.1.1 → 192.168.1.235 TCP 60 2000 → 48318 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
```

Listado 12.9: TCP: Rechazo de un intento de conexión a un puerto cerrado

El flag RST también se utiliza para cerrar una conexión activa de forma inmediata sin esperar a que el otro extremo esté de acuerdo, de hecho ni siquiera espera una confirmación. Esto se puede hacer en situaciones anómalas, cuando el otro extremo está infringiendo las reglas del protocolo, como por ejemplo usando números de secuencia incorrectos, flags ausentes o inesperados, etc. Todas estas situaciones las maneja automáticamente el SO y no es necesario que el programador intervenga.

Pero el programador también puede usar RST para finalizar la conexión de manera abrupta aunque esté funcionando correctamente. Obviamente puede conllevar la pérdida de datos si el otro extremo no ha terminado de enviar, pero es un riesgo que el emisor asume. El Listado 12.10 muestra cómo desactivar la «permanencia» (*linger*) de la conexión hasta recibir todos los datos. De este modo, el método `close()` cierra la conexión con un segmento RST en lugar del método convencional que hemos visto en § 12.3.

```
import socket, struct

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_LINGER, struct.pack('ii', 1, 0))
sock.connect((host, port))
[... envío de datos ...]
sock.close()  # envía RST, no habrá handshake de desconexión
```

Listado 12.10: TCP: Cerrar una conexión de forma abrupta con RST

El SO del otro extremo (el que recibe el RSTT) enviará al proceso un error de tipo `errno.ECONNRESET` para indicarle esta situación.

12.5. Control de flujo en TCP

El control de flujo en TCP se basa en el mecanismo de *ventana deslizante* similar al que hemos visto antes en este mismo capítulo. El receptor se preocupa de informar en cada mensaje del espacio disponible (en bytes) es su buffer de recepción —a esto se le llama «actualizar la ventana». Esa cantidad de espacio libre es lo que llamamos *ventana de recepción* y es el valor que aparece en el campo *window* de la cabecera TCP. Determina también los números de secuencia de los bytes que puede recibir. Por ejemplo, si el campo *window* tiene un valor 1000 y el campo *acknowledge number* tiene un valor de 3000, significa que el receptor puede aceptar los bytes desde el 3000 hasta el 3999, y no otros. Al recibir este mensaje, el emisor sabe que como mucho podrá enviar segmentos por un total de esos 1000 bytes a partir del último byte confirmado. De este modo, el receptor TCP influye continuamente en la tasa de salida del emisor. A este mecanismo concreto es a lo que llamamos *control de flujo de receptor*, aunque en este capítulo lo llamaremos simplemente *control de flujo*.

La Figura 12.4 representa el flujo de datos completo desde que salen del proceso emisor hasta que son consumidos por el proceso receptor. Recuerda que estamos representando solo un sentido de la comunicación de una única conexión. Cada conexión establecida conlleva una estructura como esta en cada sentido de la comunicación.

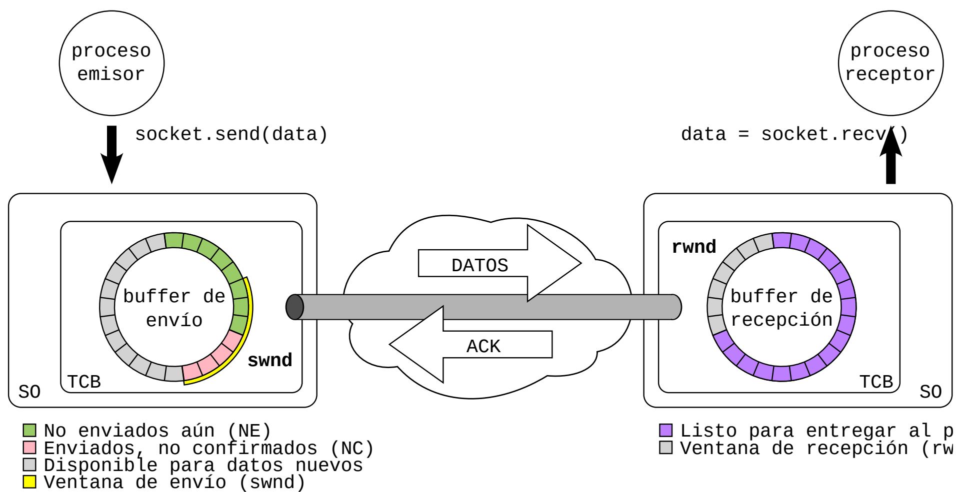


FIGURA 12.4: TCP: Flujo de datos
(se representa un solo sentido de la comunicación)

El emisor mantiene el buffer de envío (*sending buffer*) implementado como una cola circular. Los datos entran en este buffer vía el método `socket.send()` y salen cuando el SO decide construir un nuevo segmento y enviarlo a la red, pero no hay una relación directa entre la invocación de `send()` y el envío de segmentos, y de hecho varios `send()` podrían corresponder a uno, varios o ningún segmento en ese instante.

El buffer de envío está formado por:

- Espacio disponible para almacenar datos nuevos procedentes del proceso emisor.
- Bytes enviados que aún no han sido confirmados. Nos referiremos a esta parte del buffer como NC (no confirmados).
- Bytes que aún no han sido enviados. Nos referiremos a esta parte del buffer como NE (no enviados).

El buffer de recepción solo tiene datos en dos estados: recibidos confirmados y espacio libre. Cuando el computador recibe mensajes a través de una interfaz de red, el SO procesa el segmento, identifica el proceso al que corresponde el puerto destino que aparece en la cabecera de transporte y finalmente inserta su carga útil en este buffer. Los datos salen del buffer cuando el proceso receptor los solicita mediante el método `socket.recv()`.

Eso significa que si un proceso receptor, que mantiene una conexión abierta, deja de recoger datos —con `recv()`— mientras el emisor sigue enviando, ambos buffers (envío y recepción) se llenarán y como consecuencia el emisor quedará bloqueado en una invocación a `send()`.

Es importante aclarar que el SO envía el mensaje de confirmación cuando los datos son almacenados en el buffer de recepción, es decir, antes de que hayan sido procesados por el proceso receptor. Si el proceso receptor no llega a pedirlos y termina, el buffer de recepción se destruye y los datos se pierden. Dicho de otro modo, el mecanismo garantiza que los datos llegan al SO destino, no al proceso receptor.

El tamaño de estos buffers depende de cada SO y de su configuración específica. Puedes consultar estos valores para un socket concreto por medio de los identificadores `socket.SO_SNDBUF` y `socket.SO_RCVBUF`. El Listado 12.11 muestra una *shell* Python con el código que permite conseguirlo. Los valores mostrados: 16 384 y 131 072 bytes, son los valores reales (por defecto) que se han obtenido en un sistema Debian con Linux 6.12.9.

El programador puede modificar el tamaño de estos buffers, aunque si lo pruebas (como en el listado) verás algo curioso. El SO reserva el doble de

lo que pides. Esto se debe a que debe contar con el espacio necesario para almacenar cabeceras y metadatos, que estima que puede llegar a ser el doble de la carga útil. Estos tamaños tienen un valor mínimo y máximo que depende del SO. En Linux el mínimo ronda los 2-4 kB y un máximo 400 kB, y estas primitivas truncarán valores fuera de ese rango.

También es posible consultar el tamaño de los buffers, y muchos otros datos, para un socket activo con el comando `ss` como veremos un poco más adelante.

```
$ python
>>> import socket
>>> sock = socket.socket()
>>> sock.getsockopt(socket.SOL_SOCKET, socket.SO_SNDBUF)
16384
>>> sock.getsockopt(socket.SOL_SOCKET, socket.SO_RCVBUF)
131072
>>> sock.setsockopt(socket.SOL_SOCKET, socket.SO_SNDBUF, 4096)
>>> sock.setsockopt(socket.SOL_SOCKET, socket.SO_RCVBUF, 4096)
>>> sock.getsockopt(socket.SOL_SOCKET, socket.SO_SNDBUF)
8192
>>> sock.getsockopt(socket.SOL_SOCKET, socket.SO_RCVBUF)
8192
```

LISTADO 12.11: TCP: Consultar y modificar el tamaño de los buffers

12.6. Ventanas de envío y recepción

Acabamos de ver que la *ventana de recepción* (abreviado como *rwnd*) es el espacio libre (no ocupado) en el buffer de recepción. De la otra parte, la *ventana de envío* (abreviado como *swnd*) define la **parte concreta** del buffer de envío (los bytes específicos) que el emisor tiene permitido enviar en ese momento. El tamaño de la ventana de envío está limitado en todo momento por la ventana de recepción.

$$\text{swnd} \leq \text{rwnd} \tag{12.1}$$

Como la velocidad a la que el proceso receptor consume datos del buffer de recepción puede variar drásticamente a lo largo del tiempo, la ventana de recepción puede cambiar del mismo modo, y a su vez estos cambios, notificados al emisor, condicionan la ventana de envío. Dicho de otro modo, la tasa a la que el proceso receptor consume datos condiciona indirectamente la tasa a la que el proceso emisor puede enviar. La ventana de envío por tanto puede crecer, decrecer o incluso cerrarse en cualquier momento.

Control de flujo en una conexión real

Para ver el control de flujo en acción vamos a usar el ejemplo /flow-control. Consta de un servidor que únicamente recibe, pero que nos da la opción de limitar la tasa a la que consume. El siguiente comando arranca el servidor en el puerto 2000 y limita la tasa a 200 kB/s.

```
code/flow-control$ ./server.py --limit 200 2000
```

Y un cliente que se conecta al servidor indicado y sólo envía. Lo hace tan rápido como pueda. Para ejecutarlo, abre otra terminal y hazlo así:

```
code/flow-control$ ./client.py 127.0.0.1 2000
```

El cliente informa del tamaño del buffer de envío, la cantidad de datos que ha enviado (pero que pueden estar aún en el buffer) y la tasa media de envío. El servidor informa del tamaño del buffer de recepción, la cantidad de datos recibidos y la tasa media de recepción. Lo puedes ver en acción en la Figura 12.5. Más adelante en el capítulo 16 veremos cómo miden y limitan la tasa, de momento solo los vamos a usar para generar tráfico.

Cliente

```
~$ ./client.py 127.0.0.1 2000
Sending buffer size: 2,626 kB
 (-) sent:10,630 kB, rate:268.2 kB/s
```

Servidor

```
~$ ./server.py 2000 200
Receiving buffer size: 131 kB
 received:9,067 kB, rate:200.1 kB/s
```

Figura 12.5: Flujo continuo con tasa limitada por el receptor (servidor) — /flow-control

Si los ejecutas ambos en tu computador, tanto el emisor como «la red» pueden enviar más rápido de esa tasa de 200 kB/s. Por eso, tanto el buffer de recepción del servidor como el de envío del cliente eventualmente se llenan, lo que produce un bloqueo intermitente del emisor durante unos segundos. Cada bloqueo (cuando se detiene la cuenta de bytes enviados) corresponde al momento en que el receptor cierra la ventana. Esto es una prueba clara de cómo el receptor condiciona absolutamente la capacidad del emisor para enviar datos, a punto de detenerlo por completo.

Una vez tienes en marcha los dos programas en sendas consolas, puedes usar el comando `ss` que consulta el TCB del socket para obtener muchos datos interesantes. El comando concreto que se muestra filtra la conexión establecida que incluye el socket cliente y el socket conectado en el lado del cliente, es decir, los flujos con destino y origen en el puerto 2000. Si no se aplica un filtro, listaría todos los sockets activos.

```
$ ss -tin 'sport = :2000 or dport = :2000'
State         Recv-Q   Send-Q     Local Address:Port        Peer Address:Port
ESTAB         0        2260096    127.0.0.1:33188           127.0.0.1:2000
    wscale:7,7 rto:220 backoff:1 rtt:17.291/0.018 mss:54976 bytes_sent:58781120
    bytes_retrans:164928 bytes_acked:58616193 segs_out:1608 segs_in:1602
    data_segs_out:1073 send 254356602bps lastsnd:512 lastrcv:293028 lastack:272
    pacing_rate 508687456bps delivery_rate 50765624bps delivered:1074 busy:293024ms
    rwnd_limited:293016ms(100.0%) retrans:0/3 dsack_dups:3 rcv_space:65495
    rcv_ssthresh:65495 notsent:2260096 minrtt:0.01 rcv_wnd:65536
ESTAB         7552      0         127.0.0.1:2000            127.0.0.1:33188
    wscale:7,7 rto:200 rtt:0.017/0.008 ato:40 mss:32768 bytes_received:58616192
    segs_out:1601 segs_in:1608 data_segs_in:1073 send 154202352941bps lastsnd:293028
    lastrcv:512 lastack:512 pacing_rate 308404705880bps delivered:1 app_limited rcv_rtt:1
    rcv_space:65483 rcv_ssthresh:109848 minrtt:0.017 snd_wnd:65536
```

Listado 12.12: TCP: Información de la conexión con ss

En la primera línea aparece la cabecera con unos campos que se aplican a todos los sockets listados. Si te fijas solo en esas filas puedes observar lo siguiente:

```
State         Recv-Q   Send-Q     Local Address:Port        Peer Address:Port
ESTAB         0        2260096    127.0.0.1:33188           127.0.0.1:2000
ESTAB         7552     0          127.0.0.1:2000            127.0.0.1:33188
```

Estas columnas indican:

- State: es el estado de la conexión. En ambos es `ESTAB`, es decir, la conexión está establecida.
- Recv-Q: es la ocupación (en bytes) de la cola de recepción. La primera fila corresponde con el cliente, que no recibe nada, y la segunda con el servidor.
- Send-Q es la ocupación de cola de envío.
- Local Address:Port es la dirección IP y el puerto local del socket.
- Peer Address:Port es la dirección IP y el puerto del socket remoto.

La lista de variables que hay debajo de esas filas corresponden a cada socket. Incluimos solo la que resultan relevantes en este momento. La Tabla 12.1 muestra los datos del cliente mientras que la Tabla 12.2 muestra los del servidor.

En próximas secciones y capítulos veremos otros datos interesantes que aparecen es la información que ofrece `ss`.

12.7. RTO: el temporizador de retransmisión

En un protocolo confiable de la capa de enlace, la duración del temporizador de retransmisión es fácil de calcular. Como el envío de datos se realiza directamente a un vecino, resulta sencillo determinar el tiempo necesario para que un mensaje de datos llegue al destino y, de vuelta, la confirmación

Variable	Valor	Descripción
mss	54 976	Tamaño máximo de segmento (MSS)
bytes_sent	58 781 120	Bytes enviados
bytes_retrans	164 928	Bytes retransmitidos
bytes_acked	58 616 193	Bytes confirmados (ACK)
segs_out	1 608	Segmentos totales
segs_in	1 602	Segmentos totales
data_segs_out	1 073	Segmentos de datos
send	25 435	Tasa de envío (Mbps)
delivery_rate	5 076	Tasa de entrega (Mbps)
delivered	1 074	Segmentos entregados
rwnd_limited	293 016	La tasa está limitada por rwnd (ms) – 100 %
rcv_space	65 495	Espacio en el búfer de recepción
rcv_wnd	65 536	Tamaño de la ventana de recepción

CUADRO 12.1: Métricas del estado del socket cliente relacionadas con buffers y ventanas — /`flow-control`

Variable	Valor	Descripción
mss	32 768	Tamaño máximo de segmento (MSS)
bytes_received	58 616 192	Total de bytes recibidos
segs_out	1 601	Segmentos totales enviados
segs_in	1 608	Segmentos totales recibidos
data_segs_in	1 073	Segmentos de datos recibidos
send	154.20	Tasa de envío (Gbps)
delivered	1	Segmentos entregados
app_limited	-	Indica limitación por la aplicación
snd_wnd	65 536	Tamaño de la ventana de envío

CUADRO 12.2: Métricas de estado del socket servidor relacionadas con buffers y ventanas — /`flow-control`

correspondiente. Considerando el tamaño del mensaje y el ancho de banda del enlace se puede calcular este tiempo con precisión. Se debe añadir obviamente el tiempo de procesamiento de los mensajes en los extremos, su almacenamiento, etc., pero en general todo esto es bastante predecible, determinista y constante.

Con TCP la situación es muy distinta [25]. El envío de un segmento a un destino arbitrario depende de muchas variables ya que el segmento debe atravesar probablemente muchas redes y dispositivos intermedios:

- La distancia física entre emisor y receptor.
- La topología de la red y la ruta que van a tomar los datagramas.
- La carga de la red.
- El ancho de banda de los enlaces.
- La posible congestión en la red.
- La carga de trabajo de los routers intermedios, y de los nodos finales.

Y la situación es más compleja aún. Todas estas variables toman valores diferentes en función del destino y ruta, y pueden cambiar rápida y significativamente a lo largo de una misma conexión por cambios en las condiciones en distintos lugares de la interred.

Para resolverlo, TCP mide constantemente el RTT, es decir, el tiempo que transcurre desde que se envía un segmento hasta que llega su correspondiente confirmación. A partir de esta medida, calcula el valor del temporizador de retransmisión (RTO) que va a aplicar. Así pues, es posible que se el período de retransmisión sea distinto incluso para segmentos consecutivos. Para la obtención del valor final de RTO, TCP calcula una versión 'suavizada' (SRTT) del RTT con la fórmula 12.2. El suavizado proporciona estabilidad al cálculo del RTO, evitando que variaciones puntuales en la medida provoquen cambios bruscos.

$$\mathrm{SRTT} = (1-\alpha)\cdot\mathrm{SRTT} + \alpha\cdot\mathrm{RTT} \tag{12.2}$$

El coeficiente α es el *factor de suavizado*. Un valor entre 0 y 1 que determina si se le da más importancia al valor RTT que se acaba de medir (α alto) o si por el contrario es más importante el valor SRTT previo (α bajo). El valor de α suele ser $1/8$, es decir, conservador. Para el cálculo inicial $SRTT = RTT$.

TCP también calcula RTTVAR, que representa la desviación media del RTT respecto a SRTT. Permite estimar la variabilidad de RTT según la fórmula 12.3 (algoritmo de Jacobson [26]).

$$\mathrm{RTTVAR} = (1-\beta)\cdot\mathrm{RTTVAR} + \beta\cdot|\mathrm{SRTT}-\mathrm{RTT}| \tag{12.3}$$

La fórmula es similar a la de SRTT, β también un factor de suavizado cuyo valor habitual es $1/4$. Para el cálculo inicial se toma $RTTVAR = RTT/2$. Esta variable se actualiza antes que SRTT al llegar un ACK adecuado.

A partir de SRTT y RTTVAR se calcula el RTO según la fórmula 12.4.

$$\mathrm{RTO} = \mathrm{SRTT} + max(G, K\cdot\mathrm{RTTVAR}) \tag{12.4}$$

donde:

G es la precisión del reloj (*p. ej.* 1ms).
K es un factor de ponderación, normalmente 4.

Lo que dice esta formula es que el RTO es el SRTT más un margen de seguridad que será mayor cuanto más variable sea la latencia de la red (mayor sea RTTVAR). Adicionalmente se aplica un valor mínimo de 1 segundo y un valor máximo de 60 segundos.

TCP utiliza el algoritmo de Karn [27], que estipula que no se deben realizar medidas de RTT con segmentos retransmitidos, ya que la ambigüedad entre posibles ACK correspondientes a segmentos duplicados podría falsear el cálculo. En todo caso, no se recalcula el RTO para cada segmento, se realiza solo una vez por cada ronda. La única excepción a esta norma es la opción TCP Timestamp.

Cada emisor TCP mantiene un único RTO, que está asociado al primer segmento enviado no confirmado (SND.UNA). Eso implica que si todos los datos están confirmados, el RTO se desactiva. Cuando se envía un nuevo segmento, se reactiva el temporizador con el último valor RTO calculado. Cuando llega un ACK, se reinicia el temporizador con el valor del RTO actualizado.

Cuando el RTO expira, el emisor retransmite el segmento no confirmado más antiguo y duplica el valor del RTO. Si vuelve a expirar se duplicará de nuevo hasta un máximo según la fórmula 12.5. Esto se denomina *backoff exponencial* y pretende adaptar las retransmisiones a las condiciones de la red.

SND.UNA

Es el acrónimo de Send Unacknowledged Number, y es el número de secuencia relacionado con el segmento más antiguo enviado y no confirmado. Más concretamente, indica el limite inferior de la ventana de envío.

$$\mathrm{RTO} = min(2 \cdot \mathrm{RTO}, \mathrm{RTO}_{max}) \tag{12.5}$$

donde:

RTO_{max} es el valor máximo que puede tomar el RTO, normalmente 60s.

Cuando el emisor consiga realizar una nueva medida de RTT, es decir, cuando pueda enviar un nuevo segmento (no una retransmisión) y éste sea confirmado, el valor de RTO volverá a sus cotas habituales.

Ejemplo de cálculo de RTO

Veamos un pequeño ejemplo que ilustra cómo evoluciona el valor del RTO. Supongamos un valor calculado de $SRTT = 100ms$, $RTTVAR = 25ms$ que resulta en $\mathrm{RTO} = \mathrm{SRTT} + 4 \cdot \mathrm{RTTVAR} = 200ms$. Cuando correspond, se recalculan SRTT, RTTVAR y RTO según las fórmulas 12.2, 12.3 y 12.4.

- Se envían 3 segmentos S1 en $t = 0ms$, S2 en $t = 2ms$ y S3 en $t = 2ms$, asumiendo que la ventana así lo permite. El RTO queda asociado a S1.
- En $t = 90$ llega el ACK de S1. El RTT medido es 90ms. Se recalcula RTO para S2 en 183,75ms, expira en el instante $t = 273,75$.
- En $t = 120$ lleva un ACK duplicado para S1. No tiene ningún efecto.
- En $t = 273,75$ expira el temporizador de S2. Se aplica *backoff* y RTO se duplica hasta 367,5ms. Expirará en el instante $t = 641,25$.
- En $t = 330$ llega un ACK para S3. No se recalcula el RTO porque según Karn, el RTT medido debe descartarse. Como los tres segmentos enviados están confirmados, el RTO se desactiva.
- En $t = 400$ se envía S4. El temporizador se reinicia con el valor del RTO actualizado (367,5mstt). Expirará en $t = 767,5$.
- En $t = 500$ llega el ACK de S4. Se recalcula el valor RTO con una nueva muestra válida, resultando en 178ms, y se desactiva porque no hay confirmaciones pendientes.

La Tabla 12.3 resume el ejemplo y la evolución de las variables implicadas. Todos los valores están expresados en ms.

t	evento	RTT	RTTVAR	SRTT	RTO	expira en
0,00	envío S1	–	25,00	100,00	200,00	200,00
1,00	envío S2	–	25,00	100,00	–	–
2,00	envío S3	–	25,00	100,00	–	–
90,00	ACK S1	90	21,25	98,75	183,75	273,75
120,00	ACK (dup) S1	–	21,25	98,75	183,75	273,75
273,75	timeout S2	–	21,25	98,75	367,50	641,25
330,00	ACK (acum) S3	–	21,25	98,75	367,50	–
400,00	envío S4	–	21,25	98,75	367,50	767,50
500,00	ACK S4	100	16,25	98,91	163,91	–

Cuadro 12.3: TCP: Evolución de las variables en el ejemplo de RTO

RTO en una conexión real

Con `ss` se puede consultar información importante sobre el cálculo de RTO. Retomemos el ejemplo de la sección 12.6 y veamos las variables del cliente relacionadas en la Tabla 12.4.

Variable	**Valor**	**Descripción**
rto	220	Valor actual del RTO (ms)
backoff	1	Multiplicador de tiempo de espera (por timeout)
rtt	17.291 / 0.018	RTT medido (promedio / desviación) ms
minrtt	0.01	RTT mínimo observado

Cuadro 12.4: Métricas del estado del socket cliente relacionadas con el RTO— /flow-control

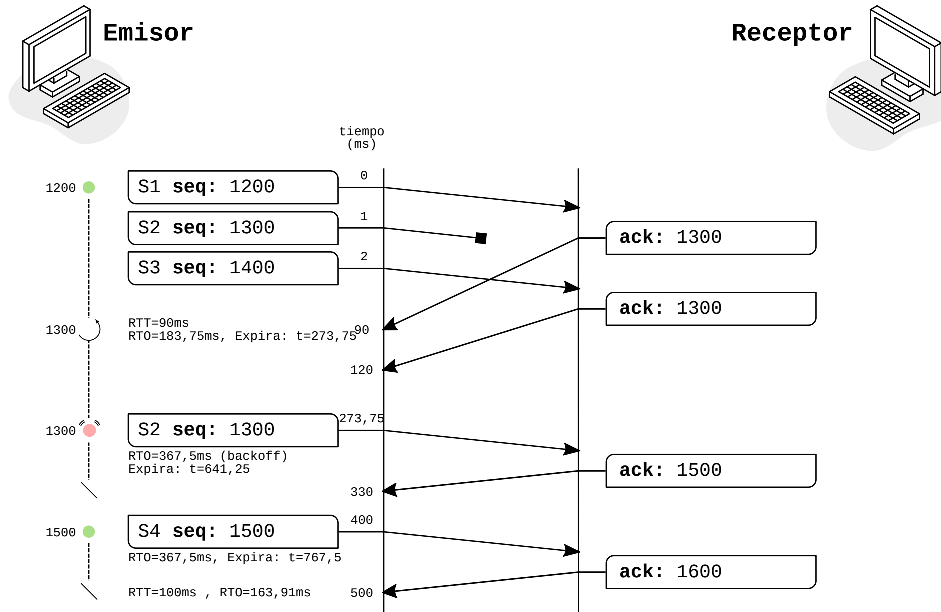


Figura 12.6: TCP: Ejemplo de cálculo del RTO

12.8. El síndrome de la *ventana tonta*

El *síndrome de la ventana tonta* (o SWS) es un efecto que conlleva una importante ineficiencia en la transferencia de datos propiciada por el envío de segmentos cada vez más pequeños. Ambos, receptor y emisor, son responsables de este efecto: el receptor porque anuncia tamaños de ventana pequeños tan pronto como dispone de algo de espacio en el buffer de recepción, y el emisor porque envía segmentos aunque tenga pocos datos disponibles. Por eso, resolver el problema requiere la colaboración de ambos extremos.

El receptor puede ayudar a paliar el problema esperando a disponer de una cantidad de espacio libre significativa antes de anunciarlo (normalmente entre MSS y la mitad del buffer de recepción) y reservando espacio libre adicional que no se anuncia. Además el receptor puede hacer uso del ACK retardado (con un máximo de medio segundo) y así favorecer la liberación de espacio adicional.

Por su parte, el emisor puede esperar a disponer de una cantidad significativa de datos antes de enviar un nuevo segmento. El algoritmo de Nagle [28] se ocupa de esto. Lo que dice Nagle básicamente es:

«Si el buffer de envío ya contiene datos sin confirmar, los nuevos datos que lleguen desde el proceso emisor se añaden al buffer hasta que se confirmen los datos pendientes o bien se acumulen MSS bytes nuevos».

12.9. Cierre de la ventana

El cierre de la ventana de recepción ocurre cuando el receptor anuncia un valor de 0 bytes en el campo `window` de la cabecera TCP, y provoca una situación peculiar, Esta acción suspende el tráfico (en ese sentido de los datos) a pesar de que el emisor tenga aún datos que enviar. Termina cuando el receptor, en algún momento, envía una confirmación indicando un valor distinto de 0. Pero hay un problema: ¿qué ocurre si se pierde esa confirmación? El emisor quedará esperando indefinidamente el mensaje de apertura, que no llegará; y el receptor queda esperando que el emisor envié datos nuevos, que tampoco van a llegar, es decir, tenemos un bloqueo.

Para evitarlo, TCP dispone de un mecanismo conocido como *prueba de ventana cero*. Consiste en el uso del llamado *temporizador de persistencia*, que al expirar, le dice al emisor que envíe una *prueba de ventana cero* (Zero Window Probe, ZWP), que puede ser un segmento sin datos o una retransmisión sobre datos ya confirmados. Eso fuerza al receptor a enviar una confirmación que permite al emisor saber si la ventana sigue cerrada. Emisor y receptor deberían permitir que la ventana quede cerrada indefinidamente. La figura 12.7 muestra un ejemplo de este mecanismo.

El temporizador de persistencia comienza siendo igual a RTO, pero como el RTO, se duplica cada vez que expira, hasta un valor máximo de 60 segundos.

12.10. Confirmación selectiva

Con la confirmación convencional que hemos visto, cuando se pierde un segmento, pero los siguientes llegan correctamente, el emisor recibe repetidamente confirmaciones para el último segmento que llegó correcto y en orden (confirmaciones duplicadas), y conforme vaya expirando el temporizador para los segmentos afectados, los irá retransmitiendo, a pesar de que hubieran llegado correctamente (el emisor no puede saberlo), y seguirá haciéndolo hasta recibir una confirmación sobre datos nuevos. Eso por supuesto conlleva retransmisiones innecesarias y un desperdicio significativo de tiempo y recursos.

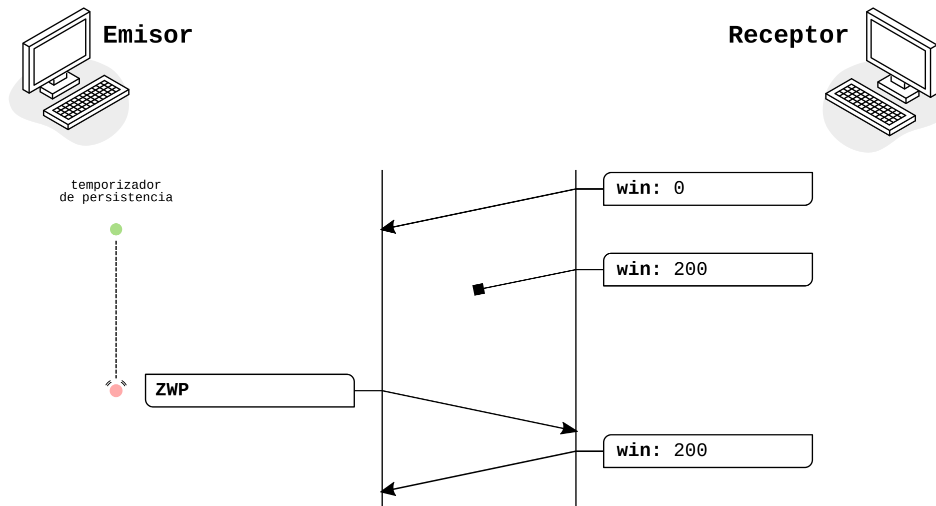


Figura 12.7: TCP: «Prueba de ventana cero» tras expirar el temporizador de persistencia

Para mejorar esta situación, el receptor puede enviar *confirmaciones selectivas*[6] o SACK. En ese caso, el receptor puede informar de los segmentos que han llegado correctamente incluso fuera de orden. Entonces el emisor puede enviar inmediatamente los segmentos que faltan sin tener que esperar a que expire el temporizador.

Para proporcionar esta funcionalidad, TCP dispone de dos opciones. La primera se llama SACK-permitted (tipo 4) y se incluye en el segmento SYN para indicar que quién la emite es capaz de procesar confirmaciones selectivas. La segunda opción es la confirmación SACK en sí misma (tipo 5), que lógicamente solo se debería enviar a un emisor que haya indicado que puede manejarlas.

La opción SACK contiene una lista de bloques que han llegado correctamente. Cada bloque (que puede incluir varios segmentos) se especifica con los números de secuencia del primer byte del bloque y del siguiente al último. Como cada número de secuencia requiere 4 bytes, la opción ACK puede contener un máximo de 4 bloques. Cuando el emisor retransmite los segmentos que faltan, envían una confirmación convencional actualizada.

La Figura 12.8 muestra un ejemplo de confirmación SACK en la que el receptor informa que ha recibido correctamente los datos fuera de orden (con números de secuencia 1601 a 2000). En emisor procede a retransmitir

[6] No confundir con *repetición selectiva*

inmediatamente el segmento que falta (con número de secuencia 1401) sin esperar a que expire el temporizador.

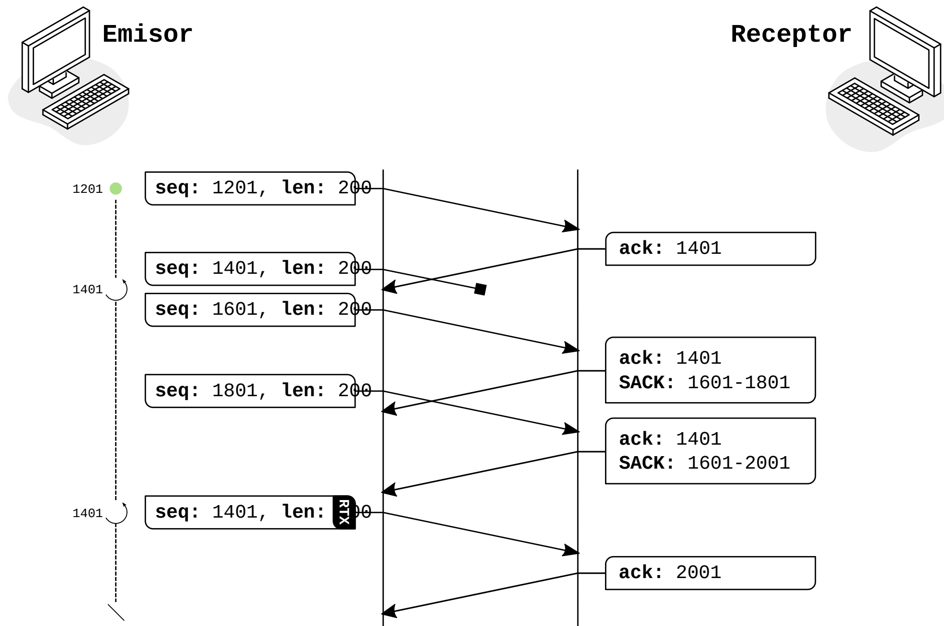


Figura 12.8: TCP: Confirmación selectiva

12.11. Aplicaciones interactivas

TCP transmite un flujo continuo de bytes entre emisor y receptor. El emisor (gracias al algoritmo de Nagle entre otros) trata de realizar una gestión óptima que maximice el uso del canal respetando, como hemos visto, las limitaciones del receptor y evitando efectos negativos como el envío de segmentos demasiado pequeños. El SO decide cuando enviar cada segmento y qué cantidad de bytes colocar en cada uno.

Sin embargo, esta búsqueda de la eficiencia es muy perjudicial para las aplicaciones interactivas tales como shell remota, escritorio remoto o videojuegos. Si por ejemplo movemos el ratón o pulsamos una tecla y esperamos ver el resultado de forma inmediata, el segmento que lleva esos datos al servidor debe salir inmediatamente, no puede esperar a que se acumulen MSS bytes. Este tipo de aplicaciones serían inviables.

Para lograrlo, hay varias cosas que podemos hacer. Una de ellas es desactivar el algoritmo de Nagle para la conexión particular que lo requiera. Puedes ver cómo hacerlo en Python en el Listado 12.13.

```
sock.setsockopt(socket.IPPROTO_TCP, socket.TCP_NODELAY, 1)
```

Listado 12.13: TCP: Desactivación del algoritmo de Nagle

Además de desactivar Nagle, TCP proporciona en su cabecera 2 flags que modifican el comportamiento habitual y que pueden ayudar a llevar los datos hasta el receptor lo antes posible.

Cuando el emisor incluye el flag PSH (*push*) en el segmento, el receptor lo interpreta como una señal de que debe entregar los datos a la capa superior inmediatamente, sin esperar a nuevos mensajes, pero no tiene ningún efecto en el emisor, es decir, no le obliga a enviar el segmento antes de los habitual. El flag PSH puede ser añadido automáticamente por el SO en ciertas circunstancias, por ejemplo, si los datos llevan determinado tiempo en el buffer de envío.

El flag URG (*urgent*) indica que la primera parte de la carga útil del segmento deben tratarse de forma prioritaria. El flag habilita el campo *puntero urgente* que corresponde con el offset (sumado al número de secuencia) que indica dónde acaban los datos urgentes (es el puntero al último byte urgente). En todo caso, es un mecanismo que históricamente ha tenido interpretaciones diferentes en las distintas implementaciones, ha tenido poco uso y no se recomienda.

12.12. Control de errores

En esta sección veremos cómo TCP resuelve los errores que pueden surgir durante una conexión.

12.12.1. Segmento perdido/corrupto

A efectos prácticos no hay diferencias entre un segmento que llega a su destino con un checksum incorrecto (corrupto) y uno que nunca llega. El receptor no puede procesar la información que contiene un segmento corrupto y por tanto lo descarta, de modo que las consecuencias son las mismas. En ambos casos, el receptor no envía confirmación, el RTO expirará y el emisor lo retransmitirá. La Figura 12.9 ilustra esta situación.

12.12.2. Confirmación perdida/corrupta

Una confirmación corrupta es descartada igual que cualquier otro segmento corrupto. La consecuencia, como en el caso anterior, es que el RTO expirará y habrá una retransmisión. Sin embargo, como las confirmaciones son

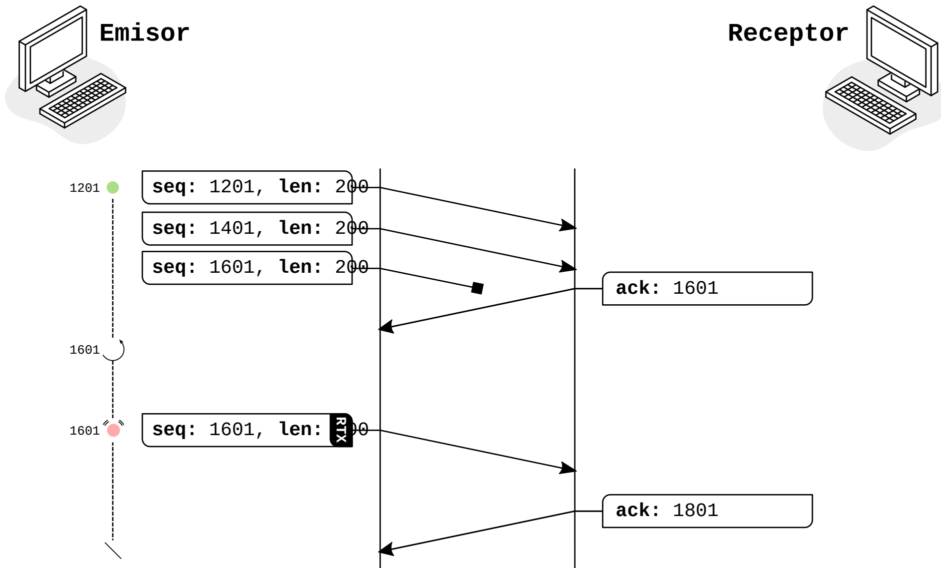


Figura 12.9: TCP: Segmento perdido

acumulativas, si llegara una confirmación sobre un número de secuencia posterior, el segmento quedaría confirmado a pesar de todo y la pérdida/corrupción del segmento de confirmación no tendría ninguna consecuencia. Puedes ver un ejemplo de esto en la Figura 12.10

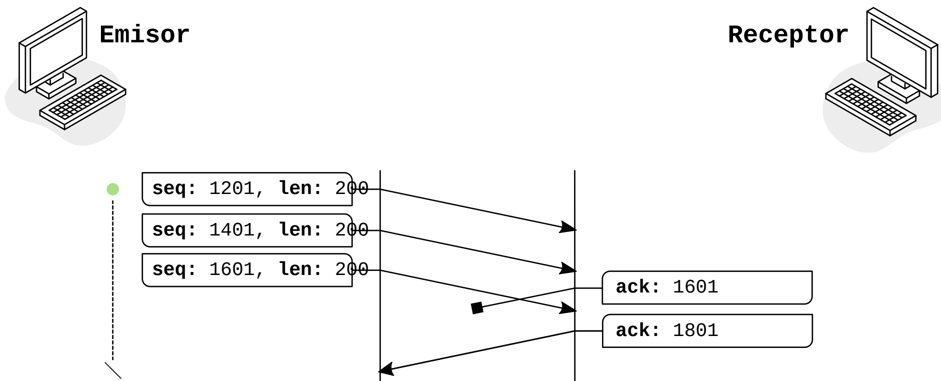


Figura 12.10: TCP: Confirmación perdida

12.12.3. Segmento duplicado

Cuando una confirmación se pierde y no existe otra que la sustituya de forma acumulativa, el emisor retransmite el segmento correspondiente y

el receptor recibirá un duplicado. En esta situación el receptor ignora el duplicado, pero está obligado a enviar una confirmación inmediatamente (ver Figura 12.11).

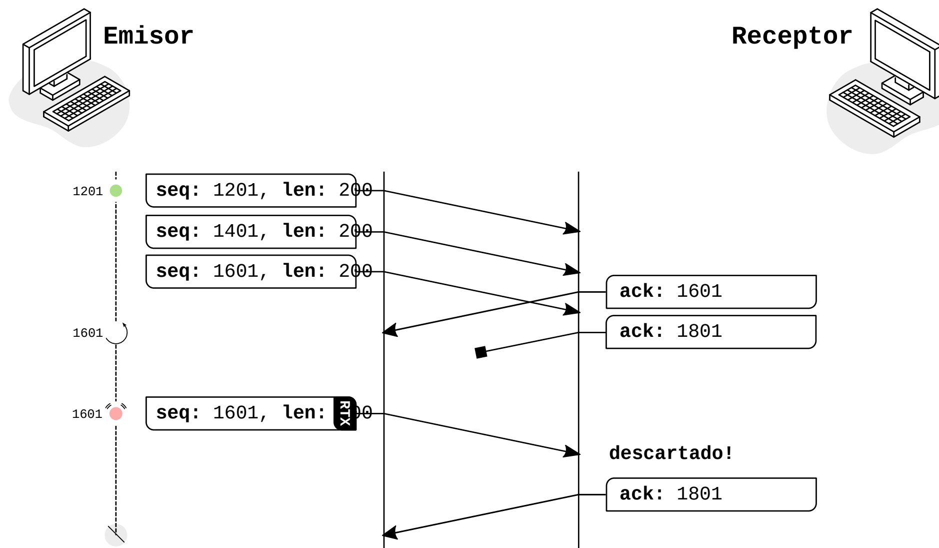


Figura 12.11: TCP: Segmento duplicado

12.12.4. Segmento fuera de orden

En una red de conmutación de paquetes como Internet es posible que segmentos de un mismo flujo o conexión lleguen al receptor en un orden diferente al que fueron enviados (Figura 12.12).

Esto puede ocurrir por varias razones, por ejemplo, porque el segmento que llega tarde se perdió y fue retransmitido, o porque el segmento que llegó antes se retrasó por alguna razón (congestión, colas en los routers, etc.). En cualquier caso, el receptor no puede procesar un segmento fuera de orden hasta que lleguen los segmentos anteriores.

Como hemos visto anteriormente, el receptor almacenará los segmentos fuera de secuencia en el buffer de recepción y enviará confirmaciones para el último segmento que llegó respetando la secuencia. Cuando finalmente llegue el segmento que falta, el receptor enviará una confirmación por todo el conjunto.

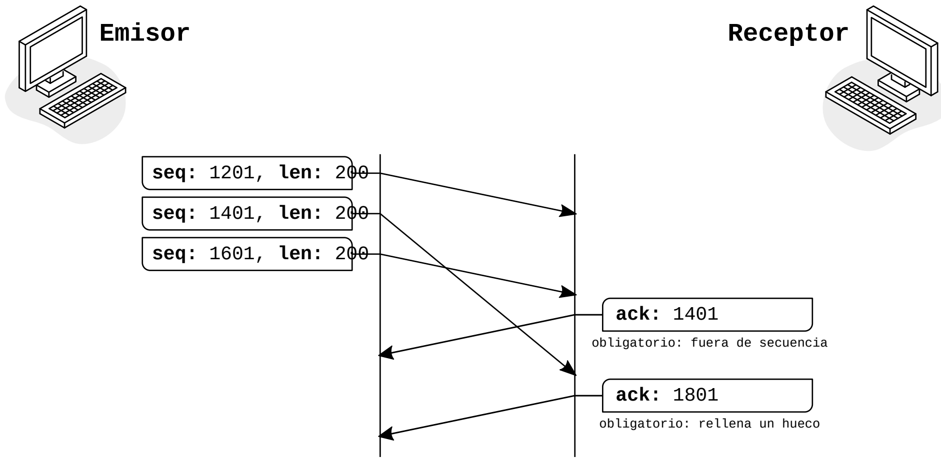


FIGURA 12.12: TCP: segmento fuera de orden

12.12.5. Confirmación duplicada

Tanto si un segmento de datos se pierde como si llega fuera de orden, el receptor enviará varias veces la misma confirmación para los siguientes segmentos que lleguen correctamente a continuación. Al emisor no le afectan las confirmaciones duplicadas, pero como veremos más adelante, le dan información valiosa sobre el estado de la red. Puedes ver un escenario en el que aparecen confirmaciones duplicadas en la Figura 12.13

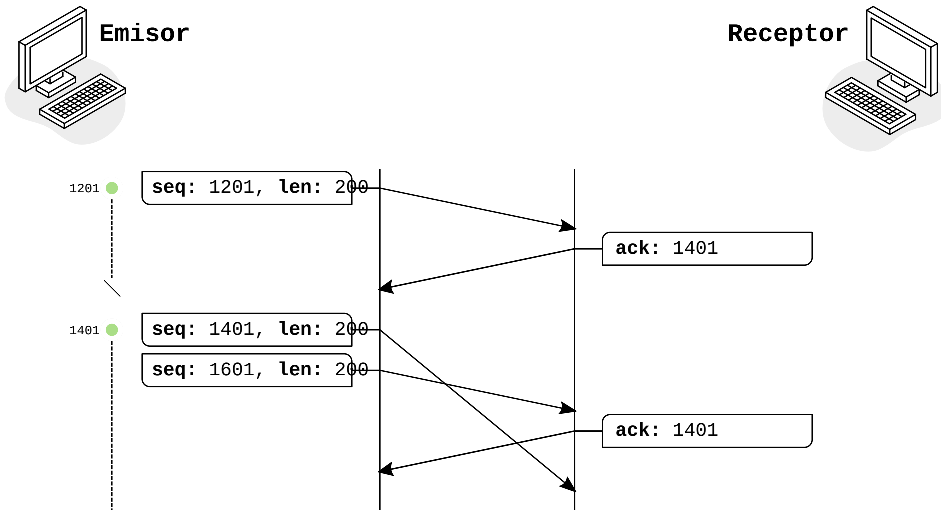


FIGURA 12.13: TCP: Confirmación duplicada

12.12.6. Demasiadas retransmisiones

Si durante el transcurso de una conexión, se suceden las retransmisiones sin lograr confirmar nuevos datos es una señal clara de que algo va realmente mal. Se definen dos umbrales que implican medidas distintas (ver [29]). Si el número de retransmisiones sobrepasa R1, que tiene normalmente un valor de 3, se entiende como un indicio de que podría haber un problema en el rutado de paquetes y se informa al proceso. Si sobrepasa R2, que suele tener un valor de 15, se cierra la conexión. En el caso de Python, eleva una excepción `OSError` con el valor `errno.ETIMEDOUT`.

El Linux puedes consultar estos valores como parámetros del núcleo ($2.10): R1 se corresponde con `tcp_retries1` y R2 con `tcp_retries2`. En el Listado 12.14 muestra cómo hacerlo y puedes ver los valores por defecto.

```
$ sysctl net.ipv4.tcp_retries1
net.ipv4.tcp_retries1 = 3
$ sysctl net.ipv4.tcp_retries2
net.ipv4.tcp_retries2 = 15
```

LISTADO 12.14: TCP: Configuración de los umbrales de retransmisión

Estos valores raramente se modifican. Solo ocurre en situaciones muy concretas, con enlaces de muy alta latencia o mucha inestabilidad, como enlaces vía satélite.

12.13. *Keep alive*

La conexión TCP puede quedar abierta durante mucho tiempo sin que ninguno de los dos extremos envíe segmentos (datos ni confirmaciones). Es estas condiciones se dice que la conexión está «inactiva» (*idle*). Para asegurarse de que el otro extremo sigue ahí, TCP proporciona un mecanismo llamado *keep alive* (mantener vivo) que consiste en enviar un segmento «sonda» para provocar una respuesta (una confirmación) del otro extremo y así verificar que sigue «vivo».

El segmento sonda se envía cuando expira el temporizador *keep-alive* (2 horas). El temporizador se reinicia cada vez que se recibe un segmento. Si el otro extremo no responde a la sonda, el emisor repetirá la operación cada 75 segundos hasta un máximo de 10 intentos. Si a pesar de eso no recibe respuesta, el emisor cierra la conexión. Todos estos valores son configurables por el programador para cada conexión.

- `SO_KEEPALIVE`: Activa o desactiva el mecanismo *keep alive*.
- `TCP_KEEPIDLE`: Tiempo de inactividad antes de la primera sonda.
- `TCP_KEEPINTVL`: Intervalo entre sondas.

- `TCP_KEEPCNT`: Número máximo de intentos antes de cerrar la conexión.

El Listado 12.15 muestra cómo fijar estos valores en un programa Python.

```
ç#ç Activar keep-alive
sock.setsockopt(socket.SOL_SOCKET, socket.SO_KEEPALIVE, 1)

# Configurar el temporizador keep-alive (en segundos)
sock.setsockopt(socket.IPPROTO_TCP, socket.TCP_KEEPIDLE, 3600)
```

Listado 12.15: TCP: Configuración del mecanismo *keep alive*

Y ¿qué más?

Hemos visto el control de flujo que, junto al control de errores, son los mecanismos fundamentales que proporcionan confiabilidad a TCP y por tanto a la mayoría de las comunicaciones en Internet en la actualidad.

Sin embargo, queda al menos otra gran cuestión que abordar: la congestión. Se trata de un problema más complejo que el control de errores, porque no afecta solo a una conexión concreta, sino a una parte o a toda la interred. En el próximo capítulo veremos cómo TCP aporta soluciones para paliar este problema.