Capítulo 11

Confiabilidad y control de flujo

Al terminar este capítulo, entenderás:

- Qué es la confiabilidad en las comunicaciones.
- Qué es el control de flujo y por qué es necesario para lograr confiabilidad.
- Qué relación existe entre confiabilidad y control de flujo.
- Qué son y cómo funcionan los protocolos ARQ.
- En qué cosiste y cómo funciona la ventana deslizante.

Que los datos lleguen de forma fiable a su destino se requiere al menos dos mecanismos básicos: confiabilidad y control de flujo. Ambos se pueden implementar por protocolos de capas diferentes, y se pueden lograr con enfoques dispares, sin embargo lo más habitual es encontrarlos en la capa de transporte (como TCP) o en la de enlace (como HDLC o PPP).

Decimos que una comunicación es fiable o **confiable** si los datos llegan a su destino exactamente tal cual partieron del origen, sin ningún cambio ni omisión, pero también sin partes duplicadas o desordenadas. Hay muchas razones por las que podrían darse todos esos problemas: interferencias, atenuación, retrasos, disparidad de rutas, limitación en el almacenamiento intermedio, etc.

Por otro lado, el **control de flujo** es un mecanismo que permite a los participantes limitar o interrumpir temporalmente el flujo de datos entre ellos. Aunque el concepto se puede aplicar de forma genérica para varias situaciones, aquí emplearemos la expresión «control de flujo» para un caso específico: el ajuste que el receptor solicita al emisor para evitar que le sature.

La razón por la que estudiamos la confiabilidad y el control de flujo a la vez se debe a que, habitualmente, ambos se diseñan e implementan mediante un mismo mecanismo. En todo caso, el control de flujo es imprescindible

para lograr confiabilidad. Sin control de flujo, el receptor perdería datos por saturación.

Todo mensaje que viaja entre dos dispositivos lo hace a través de algún medio físico, sea el aire, un cable de cobre o una fibra óptica, y por eso está restringido por las limitaciones físicas. Incluso en el vacío, la señal sufre cambios causados por interferencias o ruido electromagnético, y la intensidad de la señal se degrada por la distancia o la resistencia del medio. Por eso, cualquier mensaje que se envíe a través de una red podría llegar modificado o incompleto, o incluso no llegar. Llamamos a todo esto «errores de transmisión» y para solucionarlos existen distintas técnicas, que se clasifican en dos categorías: detección y corrección de errores.

La detección de errores más simple consiste en añadir información redundante al mensaje original, normalmente solo unos pocos bytes. Puede ser algo tan sencillo como un bit de paridad, pero típicamente se utiliza un polinomio de redundancia cíclica (CRC). El emisor aplica ese algoritmo a los datos de partida y adjunta el resultado al mensaje. Al llegar, el receptor realiza el mismo cálculo y compara el resultado con el que ha recibido. Si estos valores coinciden, el mensaje es correcto; si difieren, el mensaje ha sufrido algún cambio.

En la mayoría de los casos, estos algoritmos solo permiten detectar la alteración, pero no pueden determinar qué parte del mensaje es la que ha sido alterada y menos aún cuál fue la alteración. Este es el caso del FCS de Ethernet, o del checksum utilizado en las cabeceras de IP, UDP, TCP, etc., que es un valor de tan solo 16 bits. Puedes ver una explicación detallada de este algoritmo en /`inet-checksum/checksum.ipynb`.

Si la probabilidad de error es baja, lo más sencillo y eficiente es detectar los errores y retransmitir los mensajes afectados. Si por el contrario la probabilidad es alta, o no es posible o conveniente realizar la retransmisión, entonces se utilizan algoritmos de corrección de errores. Estos algoritmos añaden una cantidad significativa de información redundante a los mensajes y también requieren recursos de cómputo adicionales. El más conocido de estos algoritmos de corrección es Reed-Solomon, que se utiliza en comunicaciones vía satélite.

Para la retransmisión de mensajes erróneos se utilizan variantes de los protocolos ARQ (Automatic Repeat reQuest) en los que el receptor debe enviar mensajes de **confirmación** (también llamados «reconocimientos» o ACK). Estos protocolos permiten al emisor obtener información actualizada sobre el estado del receptor. Cada vez que el emisor recibe una confirmación tiene constancia de que el receptor está listo para continuar, y envía un nuevo

mensaje, es decir, al mismo tiempo proporciona confirmación y controla el flujo.

Aunque se denominan «protocolos», los ARQ no definen un formato concreto de mensaje o una sintaxis precisa. Únicamente definen una serie de reglas y pautas que deben cumplir los mensajes de datos y confirmación. Podríamos entenderlos como una especie de patrones que se pueden utilizar para crear protocolos confiables concretos.

Veamos como funciona cada uno de ellos.

11.1. Parada y espera

En el protocolo *parada y espera (stop and wait)* el emisor envía un mensaje y espera su confirmación. Si no la recibe en un tiempo determinado, reenvía el mensaje. Si recibe la confirmación, envía el siguiente mensaje.

En detalle:

- El emisor envía un mensaje de datos que incluye un campo checksum.
- El emisor inicia un temporizador de retransmisión.
- El receptor recibe el mensaje y calcula el checksum.
  - Si el checksum coincide, envía una confirmación.
  - Si el checksum no coincide, descarta el mensaje.
- Si el emisor recibe la confirmación, descarta el temporizador y envía el siguiente mensaje.
- Si el temporizador expira antes de recibir la confirmación, reenvía el mensaje.

Si el tiempo de propagación del mensaje es variable e impredecible o el temporizador no está bien ajustado, puede ocurrir que expire antes de que la confirmación llegue al emisor. Si eso ocurre, el emisor envía una retransmisión prematura[1] y el receptor recibirá un duplicado del mensaje. Para que el receptor pueda detectar esta situación se utiliza un número de secuencia de un único bit: alterna entre 0 y 1. Si el receptor recibe dos mensajes consecutivos con el mismo número de secuencia, sabrá que es un duplicado, y lo descartará.

Respecto a las confirmaciones, hay un detalle que puede causar confusión y merece la pena aclarar. Es muy frecuente que muchas implementaciones tomen el convenio de enviar en la confirmación el número de secuencia del **siguiente** mensaje esperado en lugar del que corresponde al mensaje que acaba de llegar. Es decir, para confirmar el mensaje de datos con número de

[1] llamada a menudo «retransmisión espuria»

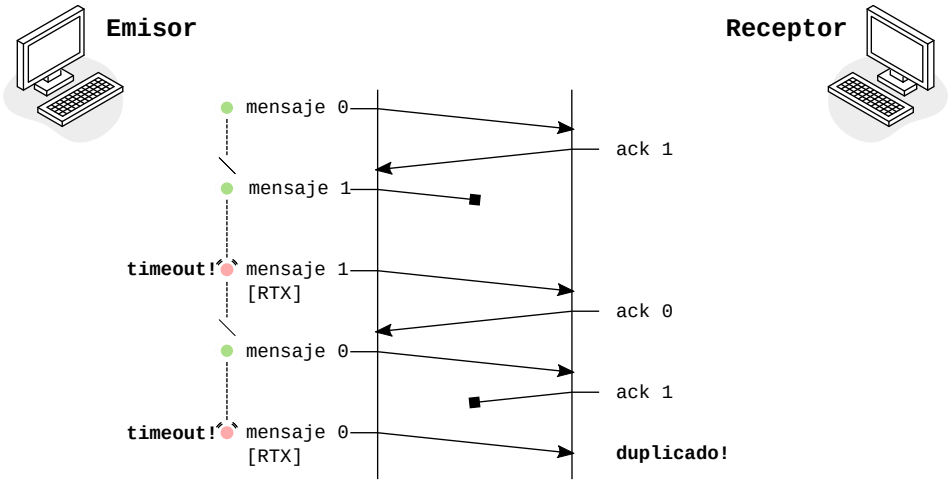

FIGURA 11.1: Diagrama de secuencia del protocolo *parada y espera*

secuencia 0, el receptor envía una confirmación con el número de secuencia 1. En este texto también tomaremos este convenio.

La implementación de *parada y espera* es simple, pero ineficiente. Como el emisor debe esperar la confirmación antes de poder enviar el siguiente mensaje, el canal de comunicación se infrautiliza en gran medida, afectando gravemente a la velocidad de transmisión neta.

11.2. Repetición continua

Repetición continua (go back N) intenta maximizar la ocupación del canal. Para ello, el emisor envía varios mensajes consecutivos, sin esperar una confirmación para enviar el siguiente. Esa técnica se llama *canalización (pipelining)* y consigue una mejora sustancial del ancho de banda efectivo.

Para que funcione, el emisor tiene que guardar temporalmente los mensajes no confirmados por si fuera necesario retransmitirlos. El espacio donde se almacenan estos mensajes suele ser una cola circular que se conoce como *buffer de envío*.

Cuando el receptor recibe un mensaje, comprueba el *checksum*, envía una confirmación al emisor y entrega el mensaje a la capa superior. Si algún mensaje se pierde, el receptor recibe mensajes fuera de secuencia. Si eso pasa, los descarta aunque sean correctos y, para cada uno de ellos, envía obligatoriamente una confirmación para el último mensaje recibido correctamente (y que respetaba la secuencia sin huecos), es decir, el emisor puede

recibir múltiples confirmaciones para un mismo mensaje de datos. Se las llama «confirmaciones duplicadas».

Los mensajes incluyen también un número de secuencia que el receptor utiliza para confirmarlos. El espacio de número de secuencia suele ser potencia de 2 porque se representa en un *campo de n bits* en la cabecera del mensaje. De este modo, con n bits se obtiene un espacio se números de secuencia de 2^n mensajes, si bien no es posible enviar n mensajes sin confirmación, porque provoca una situación ambigua que hay que evitar. Imagina una implementación hipotética con un número de secuencia de 2 bits (2^2 mensajes) en la que se utilizaran todos los números disponibles:

- El emisor envía los mensajes con números de secuencia 0 a 3 incluido.
- Todos los mensajes llegan correctamente a su destino
- Todas las confirmaciones se pierden, pero el receptor lo desconoce.
- El temporizador del mensaje 0 expira.
- El emisor retransmite el mensaje 0.

En esa situación el receptor tomaría esa retransmisión por un nuevo mensaje que está reutilizando el número de secuencia 0, provocando el fallo del protocolo. Para evitar esta situación, se define una *ventana de emisión* cuyo tamaño debe ser menor que 2^n, normalmente:

$$\text{tamaño de ventana} \leq 2^n - 1 \tag{11.1}$$

La ventana determina qué mensajes puede enviar el emisor en un momento dado, sin esperar confirmaciones y sin provocar ambigüedad. Cuando un mensaje es confirmado, su número de secuencia queda libre para ser reutilizado con un nuevo mensaje y el buffer se reutiliza una y otra vez durante la transmisión. Esa reutilización cíclica de los números de secuencia se denomina *ventana deslizante (sliding window).*

En la Figura 11.2 puedes cómo se evita la ambigüedad. El mensaje con número de secuencia 3 no forma parte de la ventana (marcada con el recuadro gris) hasta que el al menos mensaje 0 sea reconocido. El ACK 1, confirmando el mensaje 0, es el que provoca el deslizamiento de la ventana, y abre la posibilidad de enviar el mensaje 3.

Normalmente estos protocolos utilizan confirmación *acumulativa*, es decir, que incluye también a los mensajes anteriores. Por ejemplo, una confirmación acumulativa con número de secuencia 4 significa que los mensajes hasta el 3 incluido han llegado correctamente, y no hay necesidad de enviar previamente una confirmación individual para cada uno de los números de secuencia previos. Esto permite que algún mensaje de confirmación se pue-

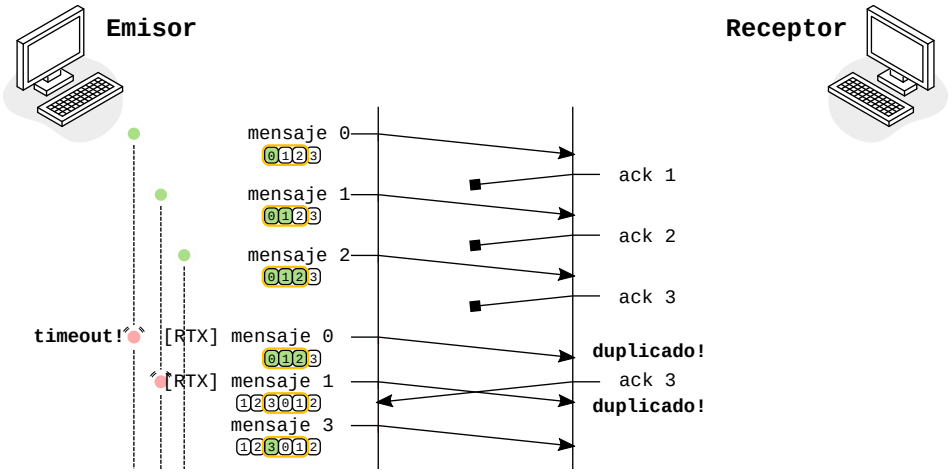

FIGURA 11.2: Diagrama de secuencia del protocolo *repetición continua* con confirmación acumulativa

da perder (o no enviarse) sin consecuencias. En realidad esto aumenta la eficiencia y muchos protocolos retrasan intencionadamente la confirmación (se llama «ACK retardado» o *delayed ACK*) para poder enviar una única confirmación para varios mensajes de datos.

Nótese en todo caso que el receptor no establece temporizadores ni retransmisiones cuando envía confirmaciones, y por supuesto el emisor no confirma las confirmaciones que recibe.

El nombre *go back N* hace referencia al hecho de que en caso de perder un mensaje quizá haya que «volver N mensajes atrás» y reenviarlos.

11.3. Repetición selectiva

Con *repetición selectiva* (*selective repeat*), el emisor también envía varios mensajes de forma consecutiva y espera las confirmaciones, pero a diferencia de *repetición continua*, si un mensaje se pierde o corrompe, el emisor reenvía solo el mensaje afectado, pero no los siguientes, y eso da nombre del protocolo.

Cuando se pierde un mensaje y el receptor empieza a recibir mensajes fuera de secuencia, aparte de confirmar el último mensaje correcto y en orden, ahora en lugar de descartarlos, los almacena en un *buffer de recepción*. Cuando los temporizadores de los mensajes no confirmados van expirando, el emisor realiza las correspondientes retransmisiones, pero tan pronto como el receptor «rellene los huecos» enviará una confirmación para todo

el conjunto y los temporizadores pendientes se cancelarán evitando más retransmisiones innecesarias.

En la Figura 11.3 puedes ver un ejemplo: Supongamos una ventada de 3 bits, en la que un emisor ha enviados los mensajes [0-4] perdiéndose el 2. El receptor enviará una confirmaciones para los mensajes con secuencia 1 y 2, reconociendo respectivamente los mensajes 0 y 1. Fíjate que al disponer de confirmación acumulativa, podría no enviar la primera de esas confirmaciones. Cuando lleguen los mensajes 3 y 4 los almacenará en su buffer de recepción y enviará sendas confirmaciones con secuencia 2. Más tarde, cuando expire el temporizador del mensaje 2, será retransmitido. El receptor al recibirlo dispondrá de la secuencia completa [0-4] y enviará un ACK con secuencia 5 para confirmar todos los mensajes recibidos. Mientras tanto es posible que más temporizadores expiren (el mensaje 3 en la figura) con lo que habrá retransmisiones innecesarias. Al llegar el ACK 5 se confirman todos los mensajes pendientes y se cancelan sus temporizadores (la figura solo muestra los de los mensajes [2-4] que son los relevantes para el ejemplo).

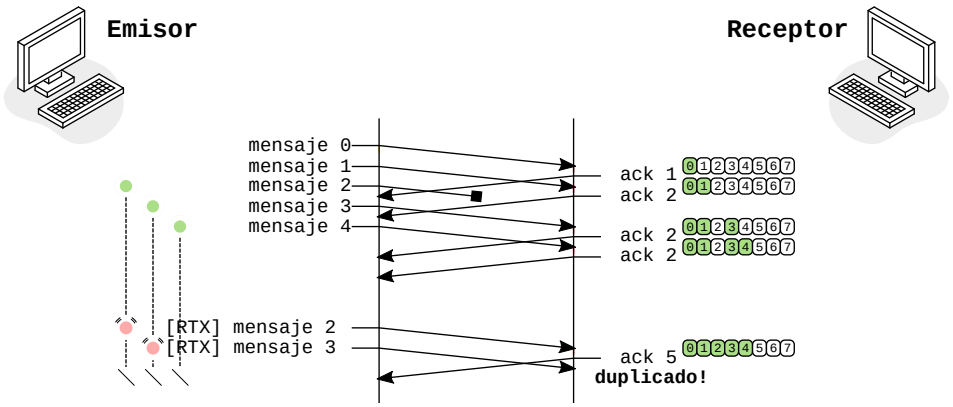

FIGURA 11.3: Diagrama de secuencia del protocolo *repetición selectiva*

Es posible mejorar también esta situación, aunque como siempre, a cambio de complejidad adicional. El receptor puede utilizar un nuevo tipo de confirmación —la confirmación negativa o NACK— para notificar explícitamente cuál es el mensaje corrupto o perdido. De ese modo, el emisor puede retransmitir inmediatamente ese mensaje en lugar de esperar a que expire su temporizador, y de ese modo aprovechando mejor el canal.

11.4. Confiabilidad en comunicaciones duplex

Los protocolos de confiabilidad que hemos visto tratan la comunicación solo en un sentido de la comunicación. Hemos hablado todo el tiempo de «emisor» y «receptor». En realidad se explica de este modo por simplicidad y para ayudar a entender su funcionamiento. Nada impide que en un canal duplex ambos extremos actúen como emisores y receptores a la vez. En ese caso, los protocolos que hemos visto trabajan de forma independiente para cada uno de los sentidos de la comunicación empleando ventanas y números de secuencia independientes.

En realidad, utilizar protocolos de confiabilidad en ambos sentidos permite optimizar algunas situaciones. Por ejemplo, resulta ineficiente enviar varios mensajes pequeños consecutivos. Por eso, cuando uno de los extremos necesita enviar una confirmación, que solo ocupa una simple cabecera, puede esperar «un poco» y aprovechar un mensaje de datos saliente para colocar la confirmación en su cabecera. A este tipo de optimizaciones se las llama *piggybacking* y es tan habitual que los formatos de las cabeceras se diseñan específicamente para hacerlo posible.

Y ¿qué más?

Los protocolos ARQ son la base de la mayoría de los protocolos que proporcionan confiabilidad en distintas capas. Algunos de estos protocolos son IEEE 802.11 (WiFi), HDLC, Bluetooth o LTE, que son protocolos de la capa de enlace[2], TCP o SCTP que son protocolos de transporte, o TFTP que es un protocolo de aplicación. Sin duda alguna, el más importante de todos estos protocolos es TCP. Por eso trataremos en detalle su funcionamiento y características en el capítulo 12.

[2]En realiad, algunos de estos protocolos abarcan varias capas, pero los mecanismos de confiabilidad los ofrecen en la capa de enlace.