La capa de transporte de datos

Xavier Vilajosana Guillén
René Serral i Gracià
Eduard Lara Ochoa
Miquel Font Rosselló

PID_00171190

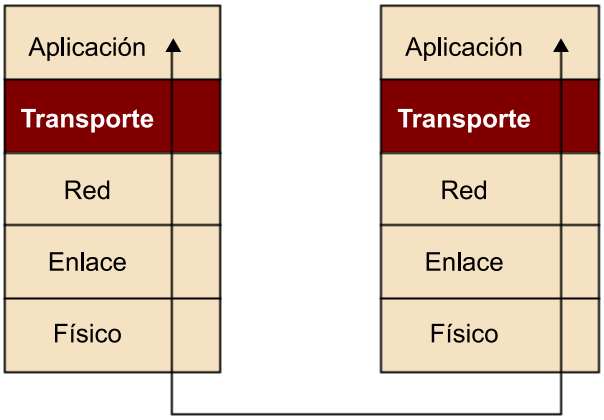
Universitat Oberta de Catalunya
www.uoc.edu

Índice

Introducción

La capa de transporte se encarga de proveer de la comunicación extremo a extremo entre procesos de aplicaciones ubicadas en diferentes equipos finales (*hosts*). Desde el punto de vista de la aplicación, es como si los equipos finales estuvieran directamente conectados, pero en realidad podrían estar en lugares opuestos del planeta, conectados mediante múltiples encaminadores (*routers*) y tipos de enlaces diferentes. La capa de transporte ofrece las funcionalidades básicas para alcanzar este nivel de comunicación lógica, sin que las aplicaciones deban preocuparse de la infraestructura de red subyacente.

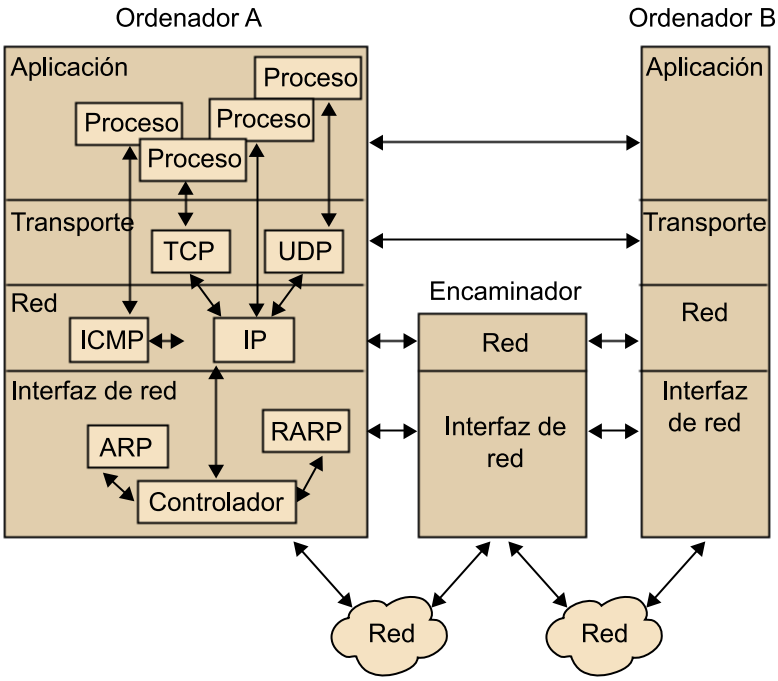
Figura 1. Mensajes extremo a extremo



Los protocolos de la capa de transporte se implementan en los dispositivos extremos de la red y no en los encaminadores ni en dispositivos intermedios. La información transmitida por las aplicaciones se convierte en paquetes de la capa de transporte, conocidos como **segmentos de la capa de transporte**. Estos segmentos se construyen dividiendo la información que quiere transmitir la aplicación en segmentos de un tamaño determinado y añadiéndoles una cabecera. Cada segmento es enviado a la capa de red en la que serán incluidos en los paquetes del nivel de red conocidos como **datagramas**. A partir de la capa de red, los datagramas son enviados a los destinatarios pasando por diferentes dispositivos que sólo examinarán la información correspondiente a la capa de red. Sólo los receptores, al recibir el datagrama en la capa de red, extraerán el segmento de transporte y lo pasarán a la capa de transporte del receptor. Esta capa procesará el segmento y lo pasará a la aplicación correspondiente.

En este módulo estudiaremos con detalle los protocolos y las funcionalidades de la capa de transporte de datos, haciendo énfasis en los protocolos de transporte de datos de Internet, el UDP y el TCP que ofrecen servicios diferentes a las aplicaciones que los invocan.

Figura 2. La jerarquía TCP/IP



1. Servicios ofrecidos por la capa de transporte

La capa de transporte ofrece las siguientes funcionalidades:

- Garantiza la transmisión sin errores, extremo a extremo, independientemente del tipo de red.
- Controla la transmisión extremo a extremo.
- Es responsable del control de flujo de los datos y control de congestión de la red.
- Se encarga de establecer, mantener y finalizar las conexiones entre dos equipos finales o un equipo final y un servidor en una red.
- Asegura que los datos lleguen sin pérdidas, sin errores y sin ser duplicados.
- Ordena los paquetes que llegan.
- Fragmenta los mensajes y los recompone en el destino cuando es necesario.
- Permite la multiplexación de diferentes conexiones de transporte sobre una misma conexión de red.

Servicios de la capa de transporte

No todos los protocolos de la capa de transporte ofrecen todos estos servicios. Por ejemplo, el UDP, como veremos más adelante, no garantiza la transmisión sin errores extremo a extremo, entre otros.

2. Relación entre la capa de transporte y la capa de red

La capa de transporte se ubica justo por encima de la capa de red en la pila de protocolos.

Mientras que la capa de transporte se encarga de proveer de comunicación lógica entre procesos y se ejecuta en equipos finales diferentes, la capa de red provee de comunicación lógica entre equipos finales.

Esta diferencia es sustancial, ya que la capa de transporte debe permitir que múltiples procesos se comuniquen de manera lógica, mediante una única conexión lógica provista por la capa de red. Este concepto se denomina **multiplexación** y **demultiplexación** de la capa de transporte.

Además, la capa de red no ofrece garantías de entrega de la información y no garantiza la entrega de los segmentos ni la integridad de la información contenida en el segmento. Por ello, la capa de red es **no confiable**. La misión de la capa de transporte es, por una parte, proveer de transmisión de datos fiables y, por otra, permitir la comunicación proceso a proceso en una red creada que comunique entre un equipo final y otro equipo final.

Para introducir los protocolos del nivel de transporte, nos centramos en la jerarquía de protocolos utilizados en Internet, que define dos protocolos de transporte: el UDP y el TCP.

El protocolo UDP es no orientado a la conexión, lo que significa que no implementa fases de establecimiento de la conexión, envío de datos y fin de la conexión, mientras que el TCP es orientado a la conexión.

En el caso de la jerarquía TCP/IP[1], se definen dos direcciones que relacionan el nivel de transporte con los niveles superior e inferior:

(1)TCP/IP es la sigla de *transmission control protocol/Internet protocol*.

1) La dirección IP es la dirección que identifica un subsistema en una red.
2) El puerto identifica la aplicación que requiere la comunicación.

Para identificar las diferentes aplicaciones, los protocolos TCP[2] y UDP[3] marcan cada paquete (o unidad de información) con un identificador de 16 bits denominado **puerto**.

(2)TCP es la sigla de *transmission control protocol*.

(3)UDP es la sigla de *user datagram protocol*.

- **Puertos conocidos**: están regulados por la IANA[4]. Ocupan el rango inferior a 1024 y son utilizados para acceder a servicios ofrecidos por servidores.

(4)IANA es la sigla de Internet Assigned Numbers Authority.

- **Puertos efímeros**: son asignados de manera dinámica por los clientes dentro de un rango específico por encima de 1023. Identifica el proceso del cliente mientras dura la conexión. Por otra parte, existen algunos puertos efímeros que son conocidos por su uso en aplicaciones específicas. Algunos ejemplos son el 8080, usado por algunos servidores de aplicaciones, y el 6667, usado por una red par a par[5]. Asimismo, existen aplicaciones que utilizan puertos de manera dinámica sin tener ninguna especificación.

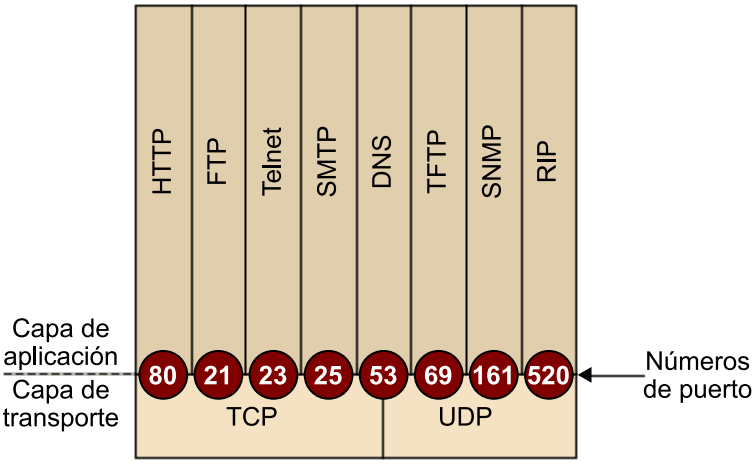
(5)En inglés, *peer to peer*.

0-255 (IANA)	Aplicaciones públicas
255-1023 (IANA)	Asignados a empresas con aplicaciones comerciales
>1023	No están registrados (efímeros)

Figura 3. Puertos de aplicaciones públicas establecidos por IANA

Decimal	Palabra clave	Descripción
0		Reservado
1-4		No asignado
5	RJE	Entrada remota de tareas
7	ECHO	Eco
9	DISCARD	Descartar
11	USERS	Usuarios activos
13	DAYTIME	De día
15	NETSTAT	Quien está conectado o NETSTAT
17	QUOTE	Cita del día
19	CHARGEN	Generador de caracteres
20	FTP-DATA	Protocolo de transferencia de archivos (datos)
21	FTP	Protocolo de transferencia de archivos
23	TELNET	Conexión de terminal
25	SMTP	Protocolo SMTP (simple mail transfer protocol)
37	TIME	Hora
39	RLP	Protocolo de ubicación de recursos
42	NAMESERVER	Servidor de nombre de host
43	NICNAME	Quién es
53	DOMAIN	Servidor de denominación
67	BOOTPS	Servidor de protocolo de *booststrap*
68	BOOTPC	Protocolo trivial de transferencia de archivos
69	TFTP	Cualquier servicio privado de conexión telefónica
75		Cualquier servicio privado de conexión telefónica
77		Cualquier servicio RJE privado
79	FINGER	Finger
80	HTTP	Protocolo de transferencia de hipertexto
95	SUPDUP	Protocolo SUPDUP
101	HOSTNAME	Servidor de nombre de host NIC
102	ISO-TSAP	ISO-TSAP
113	AUTH	Servicio de autenticación
117	UUCP-PATH	Servicio de ruta UUCP
123	NTP	Protocolo de tiempo de red
137	NetBIOS	Servicio de nombres
139	NetBIOS	Servicio de datagramas
143	IMAP	*Interim mail access protocol*
150	NetBIOS	Servicio de sesión
156	SQL	Servidor SQL
161	SNMP	*Simple network management protocol*
179	BGP	*Border gateway protocol*
190	GACP	*Gateway access control protocol*
194	IRC	*Internet relay chat*
197	DLS	Servicio de localización de directorios
224-241		No asignado
242-255		No asignado

Figura 4. Puertos usados por algunos de los protocolos de nivel de aplicación



Reflexión

¿Conocéis aplicaciones comerciales o de software libre que utilicen un puerto determinado? ¿Qué puertos utilizan Skype, Emule, BitTorrent y MSN?

3. Transporte no orientado a la conexión: UDP

El UDP es un protocolo no orientado a la conexión, por lo que no proporciona ningún tipo de control de errores ni de flujo, aunque utiliza mecanismos de detección de errores. En el caso de detectar un error, el UDP no entrega el datagrama a la aplicación, sino que lo descarta.

Cabe recordar que, por debajo, el protocolo UDP está utilizando IP, que también es un protocolo no orientado a la conexión. Lo que ofrece UDP por encima de IP es la posibilidad de multiplexar conexiones de diferentes procesos sobre una misma conexión de red, a través de los puertos.

La simplicidad del protocolo UDP provoca que sea ideal para aplicaciones que requieren poco retraso en el envío de la información (por ejemplo, aplicaciones en tiempo real o el DNS[6]). El UDP también es ideal para los sistemas que no pueden implementar un sistema tan complejo como el TCP.

[6]DNS es la sigla de *domain name server*.

El UDP es útil en aplicaciones en las que la pérdida de paquetes no sea un factor crítico, como telefonía o videoconferencia, monitoreo de señales, etc. Estas aplicaciones no toleran retrasos muy variables, es decir, si un datagrama llega más tarde del instante en el que debería ser recibido, entonces se descarta porque es inservible para la aplicación. Eso se traduce en disrupciones en el sonido o la imagen, que si bien no son deseables, no impiden la comunicación. La siguiente tabla muestra algunas de las aplicaciones más comunes que hacen uso de UDP.

Aplicaciones	Puertos
TFTP	69
SNMP	161
DHCP-BOOTP	67/68
DNS	53
RTP	Puertos pares dinámicos que empiezan en 1234

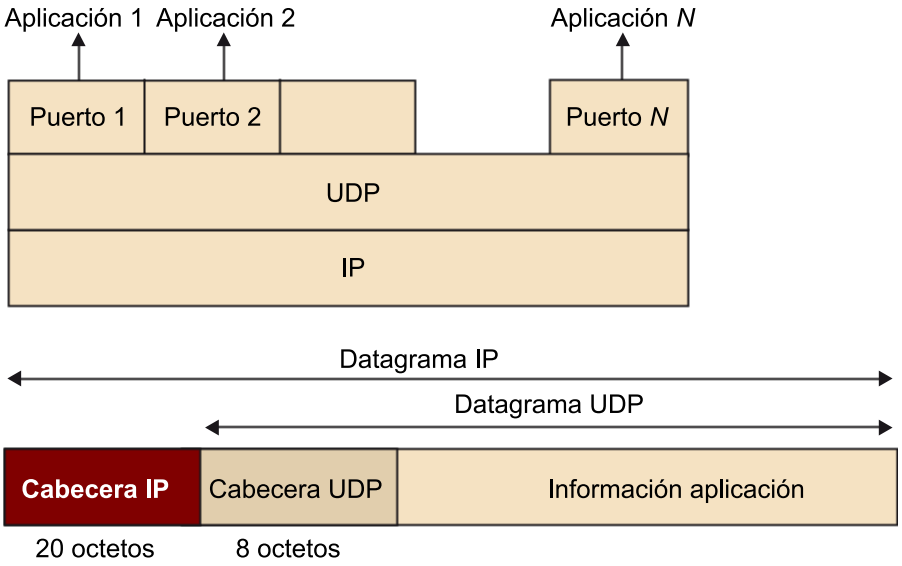
3.1. Encabezamiento UDP

La unidad de encabezamiento de UDP es el datagrama UDP.

- En la aplicación, cada escritura provoca la creación de un datagrama UDP. No se produce segmentación.

- Cada datagrama UDP creado es encapsulado dentro de un datagrama IP (nivel 3) para su transmisión.

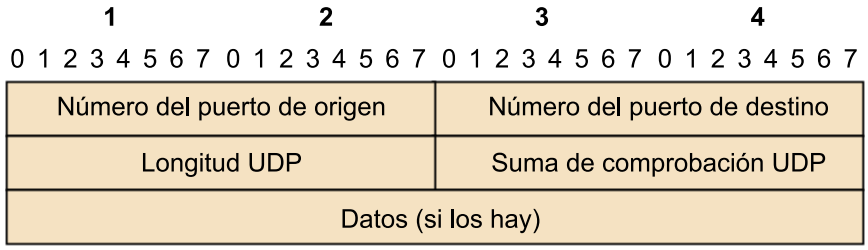
Figura 5



3.2. Cabecera UDP

La cabecera del datagrama UDP está formada por 8 bytes y tiene 4 campos:

Figura 6



- **Puertos** (**origen** y **destino**): permiten identificar a los procesos que se comunican.

- **Longitud total del datagrama UDP** (*payload* UDP + 8): es un campo redundante, ya que IP también contiene la longitud. Las dos cabeceras especifican un tamaño máximo para un datagrama de 65.335 bytes (16 bits de longitud de datagrama):
  - Longitud máxima de un datagrama UDP: 65.335 – 20 bytes = 65.315 bytes.
  - Longitud mínima de un datagrama UDP: 8 bytes.
  - Longitud de datos datagrama UDP: 65.315 – 8 bytes = 65.307 bytes.
  - El tamaño de un datagrama UDP depende del SO. Casi todas las API limitan la longitud de los datagramas UDP (lo mismo para TCP) a la

(7) En inglés, *buffer*.

máxima longitud que las memorias intermedias[7] de lectura y escritura definidas por el SO (llamadas en el sistema *read*() y *write*()). Se suele usar 8192 bytes como tamaño máximo del datagrama UDP (FreeBSD).

- Suma de comprobación UDP (*UDP checksum*): detector de errores cuyo objetivo es detectar que nadie ha modificado el datagrama UDP y que llega a su destino correctamente. Si la suma de comprobación[8] UDP es errónea, se descarta el datagrama UDP y no se genera ningún tipo de mensaje hacia el origen. La suma de comprobación se aplica junto a una pseudo-cabecera, una cabecera UDP y el campo de datos. Esta pseudo-cabecera tiene 3 campos de la cabecera del paquete IP que se envía. Debemos indicar que la suma de comprobación del datagrama IP sólo cubre la cabecera IP. La suma de comprobación se calcula haciendo el complemento a 1 de la suma de todas las palabras de 16 bits que configuran el datagrama UDP.

(8) En inglés, *checksum*.

Ejemplo

Supongamos que tenemos el datagrama (descompuesto en 3 palabras de 16 bits):

```
0110011001100000
0101010101010101
1000111100001100
```

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

```
0110011001100000
0101010101010101
_________________
1011101110110101
```

Si añadimos la tercera palabra, obtenemos:

```
1011101110110101
1000111100001100
_________________
0100101011000010
```

Ahora hacemos el complemento a 1 (cambiando 0 por 1), y obtenemos:

```
1011010100111101
```

Esta última palabra se añade también al datagrama UDP. En la recepción se suman todas las palabras de 16 bits, que debe dar:

```
1111111111111111
```

4. Principios de transferencia fiable de datos

Ved también

En el apartado 6 de este módulo didáctico, podéis ver con más detalle cómo el protocolo TCP explota los principios de transferencia de datos.

En esta sección se considerarán los principios de transferencia fiable de datos en un contexto general. Estos principios son genéricos y pueden ser aplicados a cualquier protocolo de nivel transporte que busque garantizar la transferencia fiable de los datos.

Un canal fiable de comunicación es aquel que asegura que la información transmitida llega de extremo a extremo sin errores, sin pérdidas y en el mismo orden en el que ha sido enviada. Como veremos, dirigir este problema no es trivial y requiere una serie de mecanismos que permitan, dado un canal no fiable, controlar el flujo de la información y controlar los errores que se producen en ese canal.

Con el fin de asegurar la transferencia fiable de datos, tal como se ha dicho, hay que controlar los errores que se producen en la transmisión.

Cuando se recibe un datagrama con errores, se pueden adoptar una de las siguientes soluciones:

- Descartar la trama: es útil en aplicaciones que toleran un cierto grado de error en la información recibida.

- Intentar corregir los errores con el código correcto correspondiente (se evita un retraso pero los códigos para corregir errores requieren mucha redundancia de la información).

- Solicitar la retransmisión: se produce un retraso.

La mayoría de los protocolos optan por solicitar la retransmisión del datagrama. Los protocolos de demanda de repetición automática ARQ[9] reúnen una serie de mecanismos que permiten asegurar la transmisión fiable de la información, que ésta llegue sin errores, sin duplicados y manteniendo el orden de envío.

(9)ARQ es la sigla de *automatic repeat request*.

4.1. Protocolos ARQ: control de errores

Los protocolos ARQ se encargan de retransmitir automáticamente la información que no llega o que lo hace con errores al destinatario. El funcionamiento se basa en el envío de mensajes de confirmación[10] solicitados por el destinatario de aquellos datagramas que han llegado. (En alguna versión sólo se avisa de los mensajes que no han llegado, como veremos más adelante.)

(10)En inglés, *acknowledgements*; abreviado, *ack*.

Existen dos técnicas ARQ:

Transmisiones orientadas a carácter			**Transmisiones orientadas a Bit**	
Idle RQ	Stop & Wait	Ret. implícita	**Retransmisión continua**[11]	Go-Back-N
		Ret. explícita		Retransmisión selectiva

(11)En inglés, *continuous RQ*.

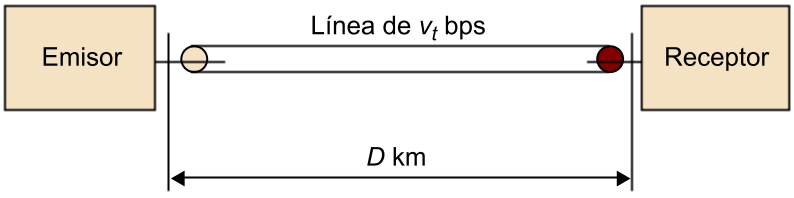
El funcionamiento básico de un protocolo ARQ es el siguiente:

- El emisor envía datagramas de información que se van guardando en una memoria intermedia de transmisión.
- Cuando la memoria intermedia de transmisión está llena, el emisor bloquea la escritura en el nivel superior hasta que reciba confirmaciones de datagramas recibidos por el receptor.
- A medida que las confirmaciones llegan al emisor, éste borra la información confirmada de la memoria intermedia para que el nivel de red superior pueda volver a escribirla.
- En caso de error, el emisor siempre puede volver a enviar la información no confirmada, ya que la mantiene en la memoria intermedia.

Terminología

En algunas referencias, el emisor también es conocido como *primario*. El receptor se denomina en muchos casos *secundario*.

Figura 7

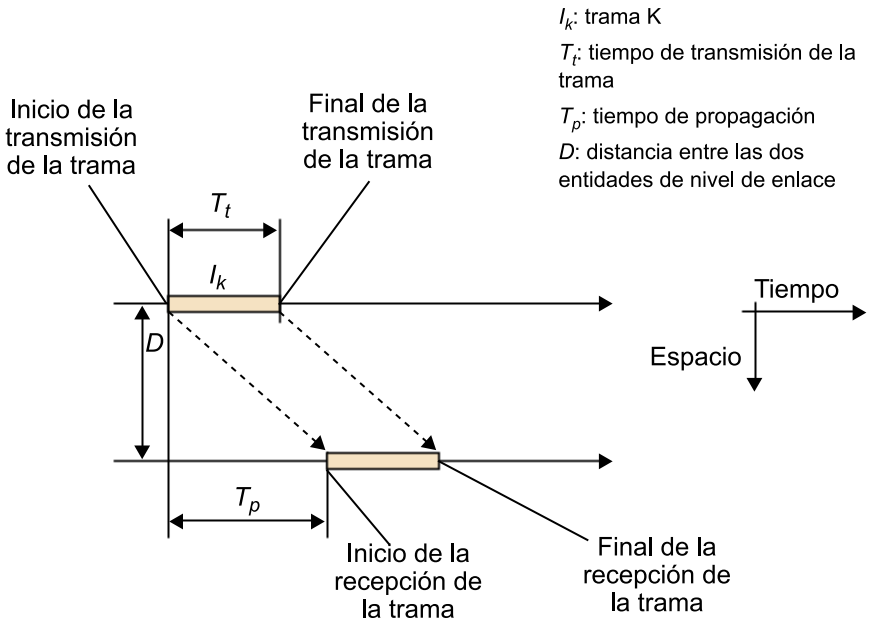


4.2. Diagramas de tiempo

Los diagramas de tiempo que utilizaremos en esta asignatura son una representación espacio-temporal de la transmisión de datagramas entre las dos entidades de nivel transporte. Este tipo de diagramas también se utiliza para el estudio de la comunicación entre entidades en muchos otros contextos como el nivel enlace. Como ejemplo de este tipo de diagramas, la figura 8 muestra

la transmisión de una trama denominada I_K. La interpretación del índice K es la siguiente: si enumeramos la secuencia de datagramas transmitidos con los números (1, 2, 3...), I_K representa el datagrama K de la secuencia transmitida.

Figura 8. Diagrama de tiempo de la transmisión de una trama

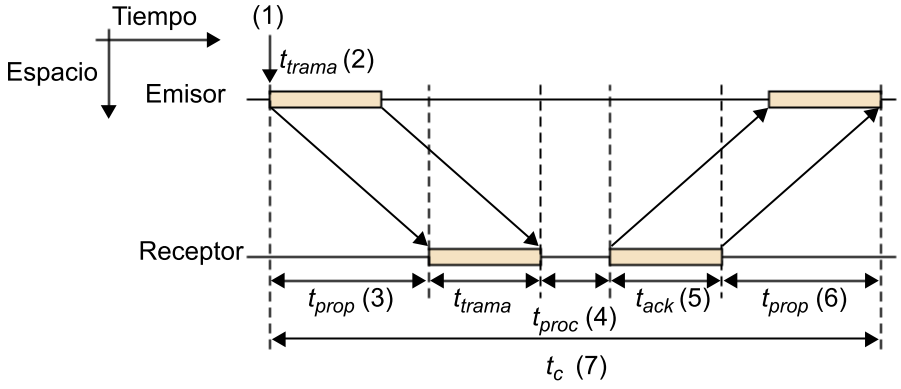


En la figura 8 aparecen dos ejes temporales. En ellos se representan los acontecimientos que tienen lugar en cada una de las estaciones (emisor y receptor): en el eje superior, los que se llevan a cabo en la estación que transmite el datagrama, y en el inferior, los que se llevan a cabo en la estación que lo recibe.

4.3. Cálculos sobre los diagramas de tiempo

La figura 9 nos ejemplifica la comunicación entre un emisor y un receptor. Los números entre paréntesis nos indican los puntos en los que tomamos medidas temporales.

Figura 9



1) El nivel superior escribe la información que se debe transmitir. El emisor acopla el datagrama de información I_K de L_t bits con esta información, la guarda en la memoria intermedia de transmisión y la pasa al nivel inferior para su transmisión. Generalmente, supondremos un tiempo de procesado de 0 segundos:

$$t_{procesamiento\ transmisión} = 0$$

2) t_{Trama} es el tiempo que se tarda en poner el datagrama de información en la línea de transmisión. Si v_t es la velocidad de transmisión del enlace:

$$t_{Trama} = \frac{L_{Trama}\ (\text{bits})}{v_t\ (\text{bps})}$$

3) t_{prop} es el tiempo de propagación de cada bit por el medio. Si v_{prop} es la velocidad de propagación del medio:

$$t_{prop} = \frac{D\ (\text{m})}{v_{prop}\ (\text{m/s})}, \text{ donde } v_{prop} = 3 \cdot 10^8 \text{ m/s en el vacío o } 2 \cdot 10^8 \text{ m/s en un conductor}$$

4) t_{proc} es el tiempo de procesamiento del datagrama recibido. Cuando llega el último bit de I_K al receptor, el nivel superior lee la información recibida en un tiempo que supondremos igual a 0.

$$t_{procesamiento\ recepción} = 0$$

5) t_{ack} es el tiempo que tarda el receptor en poner el datagrama de confirmación *ack* en la línea de transmisión. Normalmente, $L_{ack} < L_{Trama}$, por lo tanto, $t_{ack} < t_{Trama}$.

$$t_{ack} = \frac{L_{ack}\ (\text{bits})}{v_t\ (\text{bps})}$$

6) Los datagramas de confirmación tardan t_{prop} en llegar al emisor. Cuando el emisor lo recibe, borra I_K de la memoria intermedia de transmisión y repite el proceso para el siguiente datagrama ($I_K + 1$).

7) Tc está el tiempo invertido en la transmisión de un datagrama:

$$T_c = t_{Trama} + t_{ack} + t_{proc} + 2t_{prop} \approx t_{Trama} + t_{ack} + 2t_{prop}$$

Confirmación negativa

En los diagramas de tiempo indicaremos las confirmaciones negativas con el término *nak* (*no acknowledgement*).

4.4. Protocolo *idle RQ*: *stop & wait*

El protocolo *stop & wait*, que podemos traducir por "para y espera", es el protocolo ARQ más sencillo. Su principio de funcionamiento es no transmitir un datagrama nuevo hasta que no se tiene la certeza de la recepción correcta del datagrama anterior.

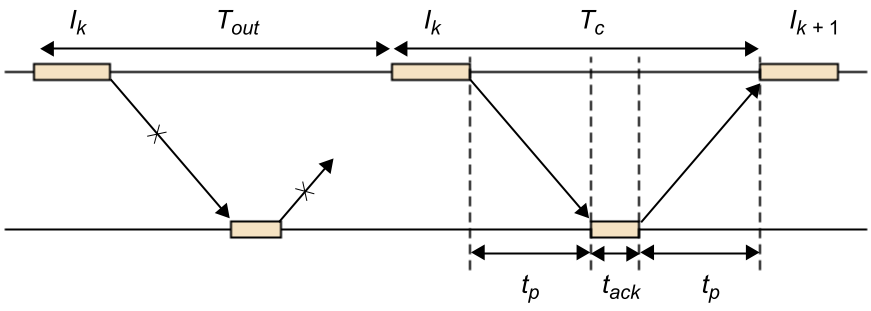
Para conseguirlo se definen dos tipos de datagramas:

1) Los **datagramas de información**, que llevan la información que se intercambian las entidades del nivel superior.

2) Los **datagramas de confirmación** (*ack*), que forman parte del protocolo y que no llevan información de los niveles superiores.

La figura 10 muestra el funcionamiento de este protocolo. En la parte derecha de esta figura el emisor envía datagramas de información al receptor, y el receptor las confirma:

Figura 10



Fijaos en que tanto el receptor como el emisor invierten un cierto tiempo en el procesamiento de los datagramas recibidos. Esto supone que, aunque el tiempo de propagación es constante, a causa del tiempo de procesamiento de los datagramas en el receptor, el tiempo de espera de las confirmaciones sea variable.

Para simplificarlo, supondremos generalmente que el tiempo de procesamiento de los datagramas es cero, excepto cuando sea necesario tenerlo en cuenta.

El tamaño de los datagramas de confirmación es normalmente mucho más pequeño que el de los datagramas de información, dado que no llevan datos del nivel superior. Además, se debe tener en cuenta que el código detector de errores también se ha de aplicar a las confirmaciones con el fin de asegurar la recepción correcta.

En el caso de que se produzca algún error, el receptor solicita la retransmisión del datagrama de información. Esto se puede llevar a cabo mediante las retransmisiones implícitas y las explícitas.

Trama perdida

Indicaremos que una trama se pierde (es decir, que los errores impiden reconocer la recepción) con una flecha que no llega al eje contrario y una cruz.

4.5. Protocolo *stop & wait* con retransmisiones implícitas

En este caso, las reglas que siguen el emisor y el receptor son las siguientes:

[12]En inglés, *time-out*.

1) El receptor envía confirmaciones positivas, A_K de los datagramas que recibe sin errores.

2) El emisor, después de enviar un datagrama I_K, espera un tiempo T_0 (activa un temporizador[12]) para recibir la confirmación positiva, I_K.

- Si una vez agotado el tiempo T_0 la confirmación no ha llegado, retransmite el datagrama I_K y vuelve a activar el temporizador.

- Si recibe la confirmación positiva, A_K, desactiva el temporizador, acepta nuevos datos del nivel superior, monta un nuevo datagrama I_{K+1} y repite este proceso.

El valor T_0 del *time-out* se debe fijar de manera que el emisor tenga tiempo de recibir la confirmación (es decir, $T_0 > T_c = t_{ack} + 2t_p + t_{Trama}$, donde t_{Trama} es el tiempo que se tarda en poner el datagrama de información en la línea de transmisión, t_{ack} el tiempo de cálculo del *ack* y t_p el de propagación). Si el valor del *time-out* fuera demasiado pequeño, se produciría la retransmisión innecesaria de las tramas.

4.6. Protocolo *stop & wait* con retransmisiones explícitas

En este caso, las reglas son las mismas que en la retransmisión implícita, con la diferencia de que el receptor también envía confirmaciones negativas si recibe una trama con errores.

En general, la confirmación negativa permite una retransmisión más rápida de la trama errónea porque no hay que esperar a que se agote el temporizador. Ahora bien, en la transmisión también es necesario utilizar un temporizador, ya que las confirmaciones pueden llegar con errores o se pueden perder.

Reflexión

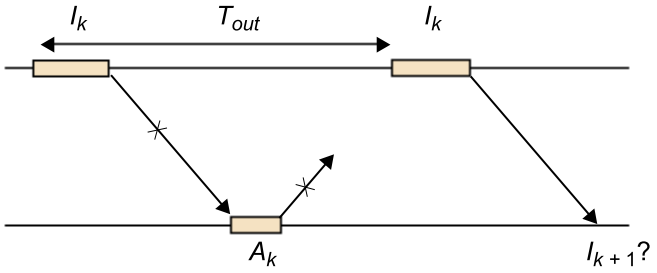
¿Qué ocurriría si se perdiera la confirmación A_K y el emisor volviera a recibir el datagrama I_K? ¿Cómo se daría cuenta de que la trama I_K está duplicada en el receptor?

4.7. Necesidad de los números de secuencia

Los protocolos ARQ necesitan un número de secuencia para poder relacionar los datagramas de información y sus correspondientes confirmaciones. En la práctica, este número de secuencia es uno de los campos de la cabecera que añade el protocolo.

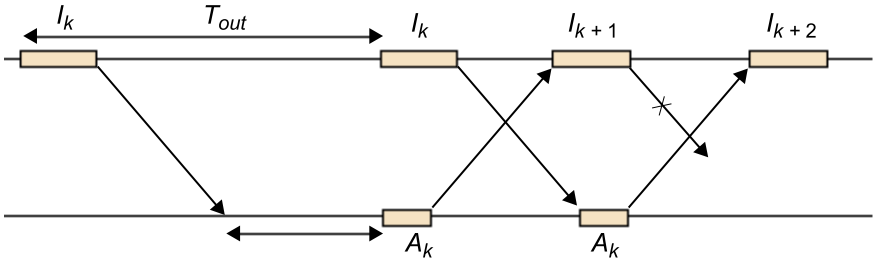
a) Necesidad de numerar los datagramas de información. Si se pierde la confirmación A_K y el emisor retransmite el datagrama I_K, el receptor no podría detectar la recepción duplicada del datagrama I_K, a no ser que el datagrama tenga un número de secuencia.

Figura 11



b) Necesidad de numerar los datagramas de confirmación. Supongamos que el tiempo de procesado del receptor es excesivo (es una estación multitarea muy cargada). Expiraría el temporizador del emisor y retransmitiría el datagrama I_K, antes de recibir su confirmación A_K.

Figura 12



La segunda confirmación de I_K se podría entender como una confirmación del paquete I_{K+1}, que se ha perdido y por lo tanto nunca se retransmitirá.

Si las confirmaciones no tuvieran número de secuencia, el emisor detectaría la recepción de las confirmaciones duplicadas. Podría suceder que se perdiera un datagrama de información.

4.8. Eficiencia del protocolo *stop & wait*

La eficiencia del protocolo *stop & wait* cuantifica la pérdida de capacidad de transmisión de información.

$$\text{Velocidad de transmisión} \rightarrow V_t = \frac{1}{\text{Tiempo de transmisión de un bit}}$$

$$\text{Velocidad efectiva } (throughput) \rightarrow V_{ef} = \frac{\text{Bits de información}}{\text{Tiempo de transmisión}}$$

La eficiencia es la relación entre la velocidad media a la que se transmiten los bits de información y la velocidad máxima a la que se puede transmitir.

$$Eficiencia = \frac{V_{ef}}{V_t} = \frac{Bits\ de\ información \times Tiempo\ de\ transmisión\ de\ un\ bit}{Tiempo\ de\ transmisión} = \frac{Duración\ de\ la\ transmisión\ de\ información}{Tiempo\ de\ transmisión} = \frac{Bits\ de\ información}{Bits\ que\ pasan\ en\ el\ tiempo\ de\ transmisión}$$

La eficiencia se puede ver también como el tiempo en el que se transmite un datagrama de información con respecto al tiempo total que se necesita como mínimo para transmitirla.

$$E = \frac{t_{Trama}}{T_c} = \frac{t_{Trama}}{t_{Trama} + t_{ack} + 2t_p} \overset{(1)}{\approx} \frac{t_{Trama}}{t_{Trama} + 2t_p} = \frac{1}{1 + 2t_p / t_t} = \frac{1}{1+2a} \quad \text{, en que } a = \frac{t_p}{t_t}$$

Supongamos que:

- $t_{Trama} > t_{ack}$
- Los datagramas de información son iguales de longitud L bits: $t_{Trama}=L_{Trama}/V_t$.
- Los datagramas de confirmación son de M bits y son todos iguales: $t_{ack}=M_{ack}/V_t$.

$$v_{ef} = v_t \cdot Eficiencia = \frac{v_t}{1+2a}$$

Concepto de $a = t_p / t_t$:

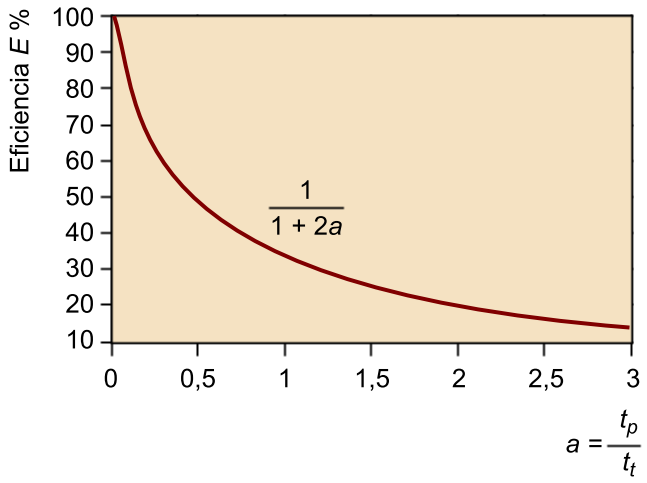
$$a = \frac{t_p}{t_t} = \frac{t_p}{L / v_t} = \frac{t_p v_t}{L} = \frac{Número\ de\ bits\ que\ caben\ en\ el\ enlace}{Número\ de\ bits\ de\ la\ trama}$$

- Si $a > 1$ cuando el primer bit del datagrama llega al receptor, el emisor ya ha acabado de transmitir la trama.
- Si $a < 1$ cuando el primer bit del datagrama llega al receptor, el emisor todavía no ha acabado de transmitir.

Figura 13

4.8.1. Eficiencia *stop & wait* en función de $a = t_{prop} / t_{Trama}$

Figura 14



Si a es muy pequeña ($t_p < t_t$), entonces la eficiencia se aproxima al 100%. Si no, la eficiencia decrece rápidamente ($t_p = t_t$ $E \approx 30\%$)

El protocolo *stop & wait* es bastante ineficiente. Imaginemos dos equipos finales ubicados a 3.000 km de distancia y conectados por una red. Supongamos que entre estos dos equipos finales los tiempos de propagación y retorno son 30 ms (*RTT*). Supongamos también que la red tiene una velocidad de transmisión de 1 Gbps (10^9 bits por segundo). El tamaño del datagrama es L = 1.000 bytes (8.000 bits) por datagrama, incluyendo cabecera y datos.

El tiempo de transmisión de un datagrama en un enlace de 1 Gbps es:

$$T_{Trama} = (800 \text{ bits/datagrama})/(10^9 \text{ bits} \times \text{s}^{-1}) = 8 \text{ microsegundos}$$

Si calculamos ahora el tiempo de transmisión del datagrama, suponiendo que el tiempo del *ack* es negligible (0 ms):

$$T_c = RTT + T_{ack} + T_{Trama} = 30 \text{ ms} + 8 \text{ microsegundos} + 0 \text{ s} = 30{,}008 \text{ ms}$$

De este modo, podemos ver la eficiencia del protocolo, es decir, podemos ver que en 30,008 ms sólo se está transmitiendo información durante 0,008 ms.

$$E_c = T_{Trama} / T_c = 0{,}008/30{,}008 \text{ ms} = 0{,}00027$$

Este resultado nos dice que sólo se está enviando información durante el 1% del tiempo.

La velocidad efectiva[13] es:

$$V_{ef} = 8.000 \text{ bits} / 30{,}008 \text{ ms} = 267 \text{ kbps}$$

(13) En inglés, *throughput*.

Si tenemos en cuenta que el ancho de banda accesible es de 1 Gbps, podemos comprobar que la eficiencia de *stop & wait* es muy baja. La solución a este problema es sencilla: el emisor, en lugar de enviar un único datagrama, podrá enviar múltiples datagramas, como veremos en los protocolos ARQ continuos que se describen en el siguiente subapartado.

Actividad 1

Calculad la eficiencia máxima que se puede conseguir en un enlace punto a punto que utiliza un protocolo *stop & wait* en los casos siguientes:

a) Enlace entre dos estaciones de una red de dimensiones reducidas con los siguientes datos:

- Distancia entre las estaciones = 1 km
- Velocidad de transmisión del enlace = 10 Mbps
- Velocidad de propagación en el enlace = 2 · 108 m/s
- Tamaño medio de las tramas de información = 5 kbit
- Tamaño de las confirmaciones negligible

b) Enlace vía satélite con los siguientes datos:

- Tiempo de propagación entre estaciones = 270 ms
- Velocidad de transmisión del enlace = 56 kbps
- Tamaño medio de las tramas de información = 4 kbit
- Tamaño de las confirmaciones negligible

Solución

a) $t_p = D/v_p = 103 / 2 \cdot 108 = 0{,}5 \cdot 10^{-5}$ s
$t_t = L/v_t = 5 \cdot 103 / 10 \cdot 106 = 5 \cdot 10^{-4}$ s
$E\ (\%) = t_t / (t_t + 2t_p) \cdot 100 = 98\%$

b) $t_t = 4 \cdot 103 / 56 \cdot 103 = 71{,}5$ ms, donde $E = 12\%$.

4.9. Protocolos continuos RQ

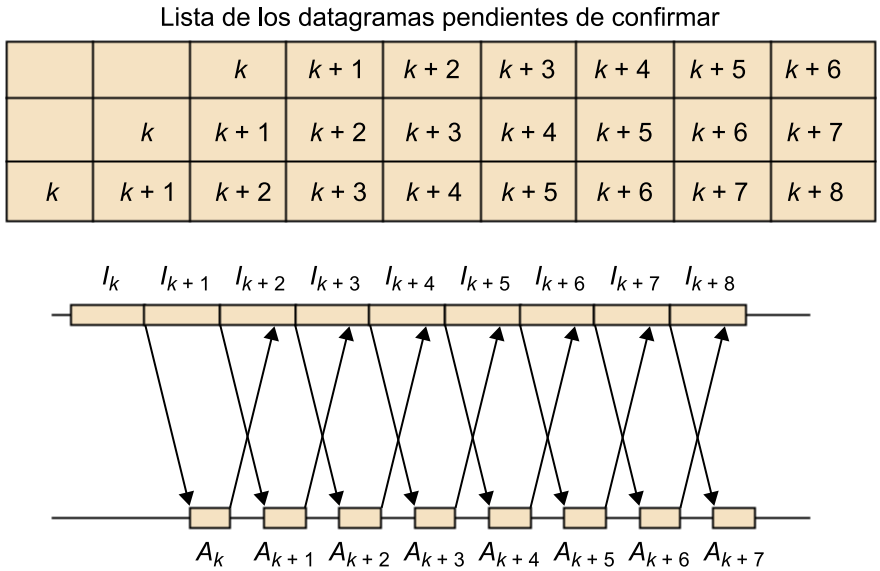
Acabamos de ver que el protocolo *stop & wait* puede llegar a ser muy ineficiente, debido a que el emisor no puede transmitir durante el tiempo de espera de las confirmaciones.

En los protocolos de transmisión continua se deja que el emisor transmita continuamente tramas de información mientras se esperan las confirmaciones.

Evidentemente, esto sólo es posible si el enlace es full dúplex, ya que el emisor y el receptor deben poder ocupar simultáneamente el canal. En el caso del protocolo *stop & wait*, esto no es necesario, pues el emisor y el receptor ocupan el canal alternativamente.

En caso de error, el emisor se puede ver obligado a retransmitir una trama anterior a la última transmitida. Así, en un protocolo de transmisión continua es necesario que el emisor guarde en una lista las tramas que se han enviado y que están pendientes de confirmar, con el fin de poder retransmitirlas en caso de error. Esta lista se conoce como **lista de transmisión**.

Figura 15



Cuando se producen errores, éstos se deben recuperar. Existen dos técnicas:

- *Go-back-N*
- Repetición selectiva

4.10. Protocolo *Go-back-N*

El protocolo *go-back-N* es un protocolo ARQ de transmisión continua en el que el emisor mantiene los datagramas en una memoria intermedia hasta que éstos han sido confirmados. La memoria intermedia de recepción del receptor sólo admite datagramas recibidos en secuencia, a punto para ser leídos por el nivel superior.

En caso de error, el receptor descarta todos los que llegan fuera de secuencia, y el emisor repite la transmisión a partir del último datagrama recibido correctamente.

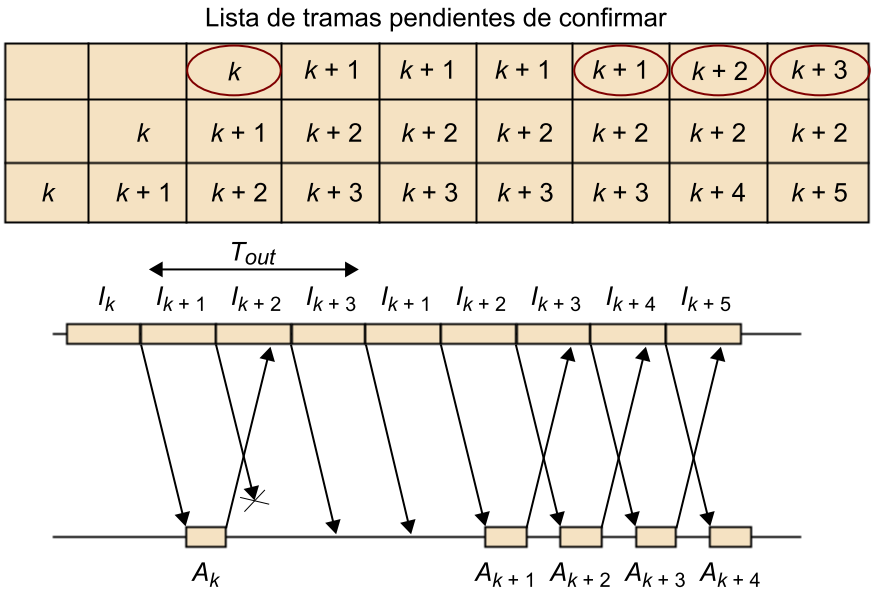
Con el fin de obtener este comportamiento, la memoria intermedia de transmisión debe tener una capacidad mayor de 1. En cambio, la memoria intermedia de recepción en el receptor sólo necesita tener un tamaño de 1. Con el fin de implementar este protocolo, podemos utilizar diferentes planteamientos, como temporizadores o confirmaciones negativas.

4.10.1. Implementación de *go-back-N* con temporizadores

Como se ha comentado, el emisor envía los datagramas y los mantiene en una memoria intermedia hasta que recibe la confirmación. Al recibir los datagramas, el receptor los confirma si no se producen errores. En caso de error o recepción de un datagrama fuera de secuencia, el receptor deja de enviar confirmaciones hasta que recibe correctamente el datagrama que falta (K). Se descartan todos los datagramas que recibe con un número de secuencia diferente al que se ha perdido o era erróneo (K).

Cuando salta el temporizador de retransmisión de un datagrama (I_K), el emisor retransmite el datagrama I_K y continúa con la transmisión de los datagramas siguientes.

Figura 16

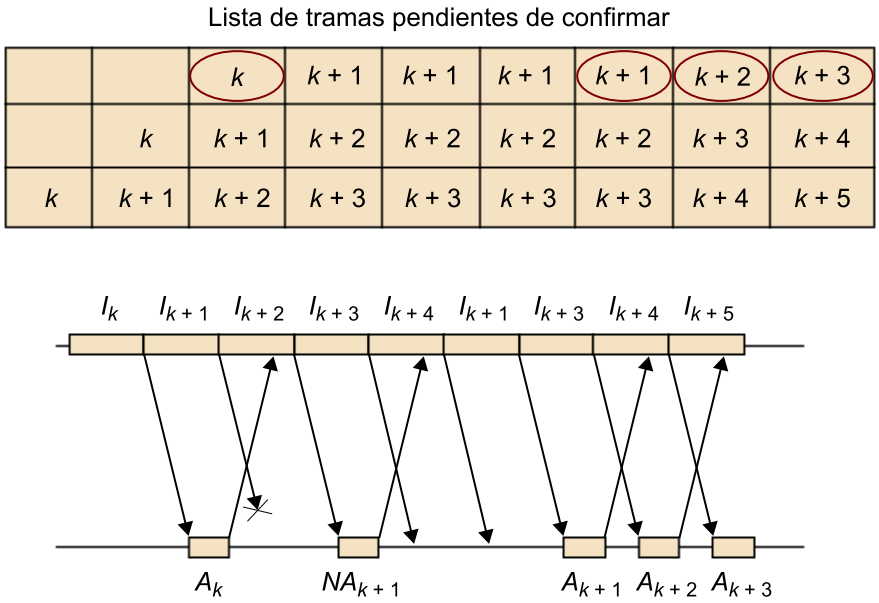


4.10.2. Implementación de *go-back-N* con confirmaciones negativas

En este caso, cuando el receptor recibe un datagrama con errores o fuera de secuencia, envía una confirmación negativa solicitando la retransmisión de la trama errónea o la que falta, y deja de enviar confirmaciones hasta que recibe el datagrama pendiente.

Cuando el emisor recibe una confirmación negativa, confirma todos aquellos datagramas con número de secuencia anterior al número de secuencia incluido en la confirmación negativa (este proceso se conoce como **confirmación acumulativa**). A continuación, el emisor envía de manera secuencial los datagramas a partir del que no ha sido confirmado.

Figura 17



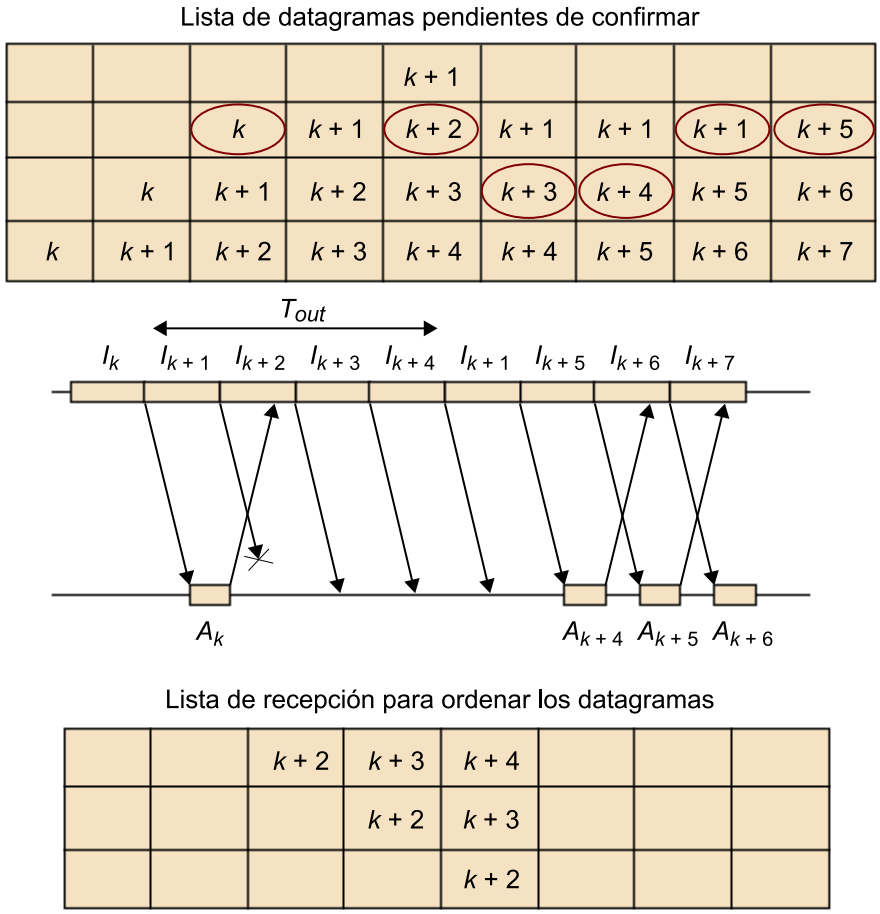
4.11. Protocolo de repetición selectiva

El protocolo de repetición selectiva es también un protocolo ARQ de transmisión continua. En este caso, a diferencia de *go-back-N*, el receptor almacena los datagramas correctos que han sido recibidos fuera de secuencia. La tarea del receptor consiste en almacenar y ordenar los datagramas que llegan fuera de secuencia antes de transmitirlos al nivel superior. En consecuencia, se necesita una memoria intermedia de recepción para mantener más de un datagrama. En el protocolo de repetición selectiva, el emisor sólo deberá retransmitir los datagramas erróneos. La implementación de este protocolo se detalla a continuación.

4.11.1. Implementación de la repetición selectiva

Al igual que en *go-back-N*, las confirmaciones son acumulativas, es decir, la confirmación A_K confirma todos los datagramas de información con número de secuencia menor que K. Cuando el receptor recibe un datagrama de información I_K con errores o fuera de secuencia, deja de enviar confirmaciones hasta que recibe correctamente el datagrama que falta. Mientras tanto, va guardando todos los datagramas que recibe con número de secuencia diferente de K. En el momento en el que en el emisor salta el temporizador de retransmisión de un datagrama I_K, éste retransmite el datagrama I_K pero no retransmite el resto de los datagramas que ya ha enviado.

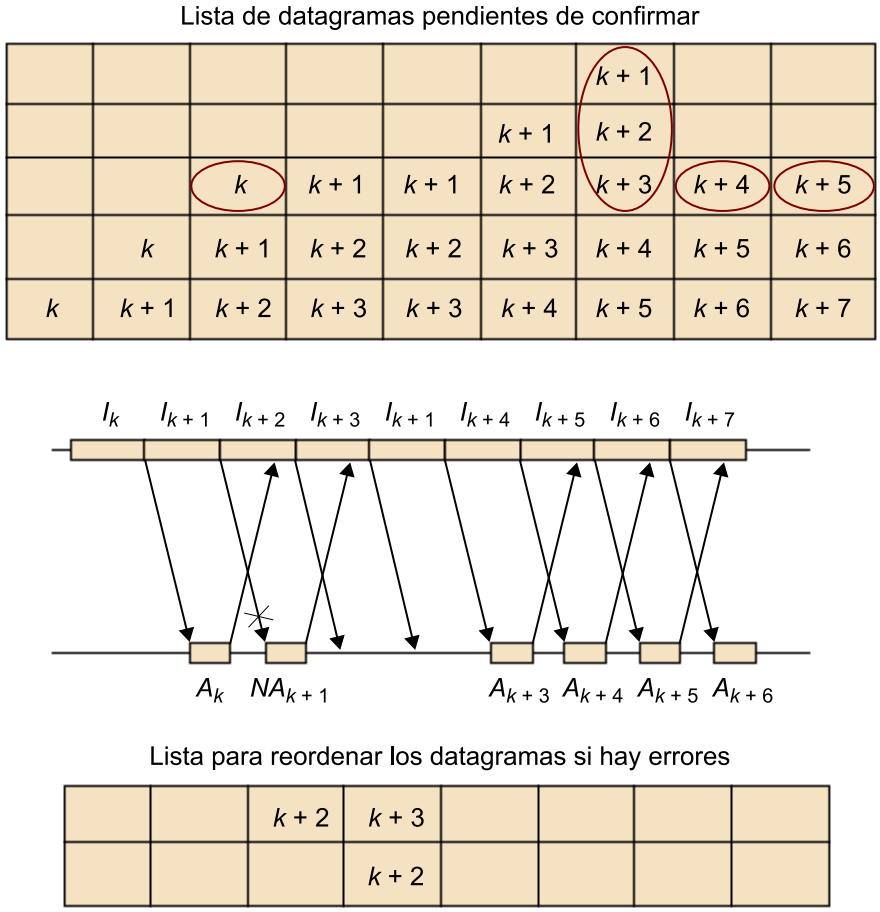
Figura 18



4.11.2. Implementación de la repetición selectiva explícita

En esta implementación el receptor envía confirmaciones positivas cuando el datagrama es correcto, y confirmaciones negativas cuando los recibe con errores o fuera de secuencia. Las confirmaciones son acumulativas, como en el caso anterior, de manera que una confirmación A_K confirma todos los datagramas de información con número de secuencia anterior a K. Mientras el receptor espera la retransmisión de una trama, se dejan de enviar confirmaciones positivas o negativas de los datagramas posteriores a K. El receptor activa un temporizador de retransmisiones negativas de los datagramas no recibidos con el fin de volver a enviar la demanda en caso de que no se reciba en un período de tiempo determinado. A medida que las confirmaciones negativas llegan al emisor, éste retransmite las tramas pedidas.

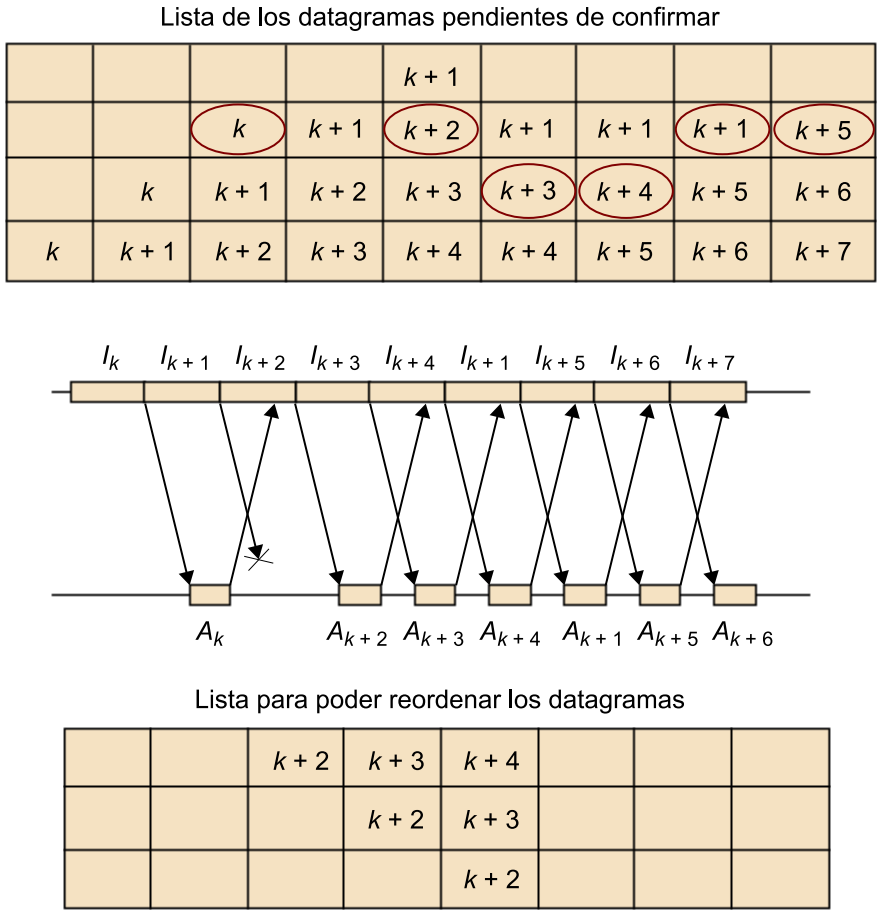
Figura 19



4.11.3. Implementación de la repetición selectiva implícita

En esta implementación las confirmaciones no son acumulativas. El receptor sólo envía confirmaciones positivas de aquellos datagramas recibidos correctamente. Si el emisor recibe un datagrama de confirmación no consecutivo, deduce que ha habido un error y retransmite el datagrama intermedio. Un temporizador T_{out} se utiliza para evitar que se pierdan las tramas pendientes, ya que el emisor sólo envía la primera perdida a partir de la última confirmada.

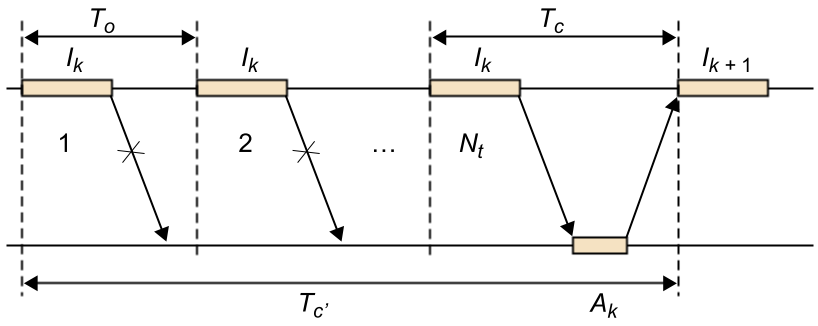
Figura 20



4.12. Cálculo de la eficiencia en presencia de errores

4.12.1. *Stop & wait*

Figura 21



En las $N_t - 1$ primeras retransmisiones, el datagrama I_k se retransmite después de un tiempo T_{out} (cuando salta el temporizador de retransmisión), mientras que en la última la transmisión del datagrama se realiza durante un tiempo T_c:

$$Eficiencia = \frac{t_{Trama}}{T'_c} = \frac{t_{Trama}}{(N_t - 1)T_{out} + T_c}$$

Si ajustamos el temporizador para no tener retransmisiones innecesarias, es decir, cuando $T_o > \approx T_c$, obtenemos la eficiencia máxima:

$$T_c' = N_t T_c = N_t(2t_p + t_{ack} + t_t) \approx N_t(1 + 2a)t_t$$

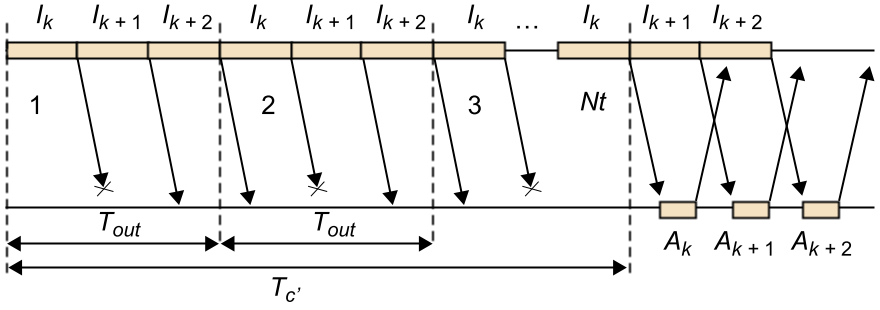
$$Eficiencia = \frac{t_{Trama}}{T'c} = \frac{t_{Trama}}{N't T_c} = \frac{t_{Trama}}{N_t(2t_p + t_{ack} + t_{Trama})} \overset{T_{Trama} << t_{ack}}{\approx} \frac{1}{N_t(1+2a)}$$

La eficiencia disminuye en un factor N_t.

4.12.2. *Go-back-N*

Cada vez que se produce un error se pierden los datagramas transmitidos y se aborta la transmisión:

Figura 22



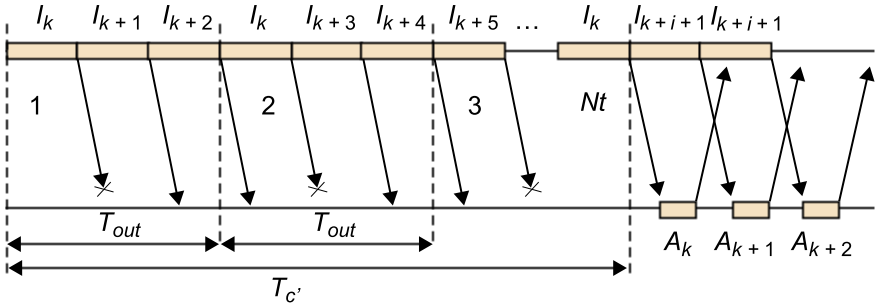
$T_{c'} = (N_t - 1)T_0 + t_{Trama} \approx (N_t - 1)T_c + t_{Trama}$, donde $T_c = t_{Trama} + 2t_{prop}$

$$Eficiencia = \frac{t_{Trama}}{(N_t - 1)T_c + t_t} = \frac{t_{Trama}}{(N_t - 1)(t_{Trama} + 2t_p) + t_{Trama}} \approx \frac{1}{2a(N_t - 1) + N_t} = \frac{1}{N_t(1+2a) - 2a}$$

4.12.3. Retransmisión selectiva

A diferencia de *stop & wait y go-back-N*, el tiempo que queda libre mientras se efectúan las N_t transmisiones de un mismo datagrama de información se aprovecha para transmitir otros datagramas. La eficiencia es independiente del temporizador de retransmisión.

Figura 23



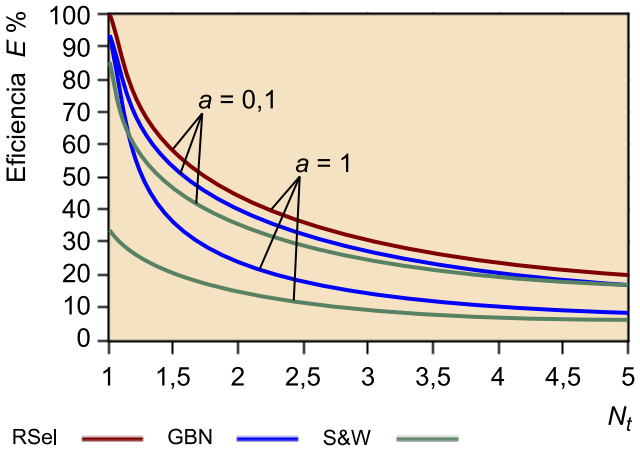
El tiempo invertido en la transmisión es $N_t \cdot t_{Trama}$:

$$Eficiencia = \frac{t_{Trama}}{N_t \cdot t_{Trama}} = \frac{1}{N_t}$$ La eficiencia es independiente de a.

4.12.4. Comparaciones y conclusión

- Para $a < 1$, el comportamiento de los 3 protocolos es similar.
- Si P_{error} es alta ($N > 1$), la eficiencia obtenida con los 3 protocolos es parecida.
- La eficiencia de *stop & wait* decrece al aumentar a, independientemente de la P_{error}.
- La retransmisión selectiva y el *go-back-N* tienen una $E \approx 100\%$ si la probabilidad de error es pequeña ($P_{error} \approx 0 = N \approx 1$).
- Si la probabilidad de error es pequeña ($N \approx 1$), la retransmisión selectiva y el *go-back-N* tienen una eficiencia parecida.
- La retransmisión selectiva tiene más eficiencia que *go-back-N* por una probabilidad de error no demasiado alta.

Figura 24. Comparación de la eficiencia de *stop & wait, go-back-N* y Retransmisión selectiva en función de *N* para dos valores de *a*



4.13. Cálculo de *N*

Calcularemos la relación que existe entre N (la media del número de transmisiones necesarias para la transmisión con éxito de un datagrama) y P_{bit} (probabilidad de error en el bit P_{bit}). Supondremos que cada bit del datagrama tiene una probabilidad de error independiente de los otros bits e igual a P_{bit}. Si el datagrama tiene L bits, la probabilidad de que el datagrama tenga algún error (P_p) es:

$$P_p = 1 - Prob\{\text{bit sin error}\}^L = 1 - (1 - P_{bit})^L$$

La probabilidad de transmitir un datagrama k veces es la de que las $k - 1$ primeras veces tenga error (P_p^{k-1}), multiplicado por la probabilidad de que la última transmisión no tenga error ($1 - P_p$):

$$N_t = \sum_{k=1}^{\infty} k \cdot Prob\left\{k \ transmisiones\right\} = \sum_{k=1}^{\infty} k \cdot P_p^{k-1}(1-P_p) =$$

$$= (1-P_p)\sum_{k=1}^{\infty} k \cdot P_p^{k-1} = \frac{(1-P_p)}{(1-P_p)^2} = \frac{1}{(1-P_p)}$$

5. Control de flujo

Los protocolos ARQ realizan recuperación de errores y control de flujo.

El control de flujo consiste en adaptar una velocidad de transmisión eficaz entre el emisor y el receptor, de manera que siempre haya recursos.

La velocidad de transmisión se controla por los siguientes motivos:

- Saturación de las memorias intermedias de transmisión, cuando hay tramas pendientes de confirmar.

- Saturación de las memorias intermedias de recepción. Un emisor que envía información a una velocidad eficaz mayor que la que puede consumir el receptor satura la memoria intermedia de recepción del receptor y se pierde la información.

Técnicas para controlar el flujo

Existen diferentes técnicas para controlar el flujo:

- X-OFF/X-ON: cuando el receptor no admite más caracteres, le envía X-OFF para bloquear el transmisor, y X-ON para reanudar la transmisión.
- En *stop* & *wait* no hay problema. Sólo mantiene un datagrama sin confirmar.
- En transmisión continua: **mecanismo de ventana deslizante**[14].
- *Echo checking*: realiza control de errores y control de flujo. Cuando las memorias intermedias se llenan, se detiene el envío de ECOS y el transmisor se bloquea hasta que recibe un ECO.
- Control de flujo fuera de banda: protocolo de nivel físico RS232 para comunicación entre puerto serie de un PC y un módem. Dos líneas sirven para el control de flujo: RTS[15] y CTS[16]. El funcionamiento es el siguiente:
  - RTS 1 = el emisor está listo para transmitir datos.
  - CTS 1 = el receptor puede recibir los datos. El emisor le envía datos, si tiene.
  - CTS 0 = el emisor no puede transmitir.

(14)En inglés, *sliding window*.

(15)RTS es la sigla de *request to send*.

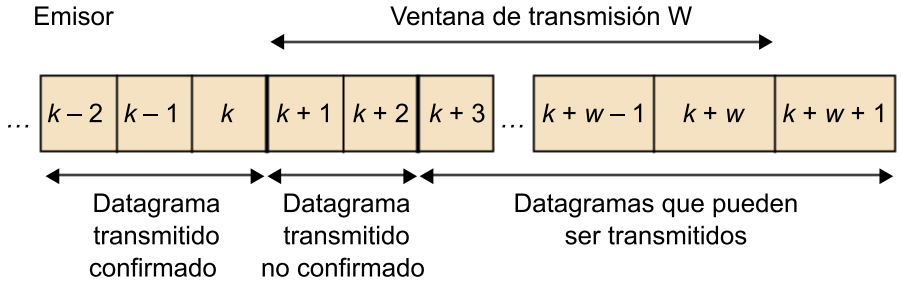
(16)CTS es la sigla de *clear to send*.

5.1. Protocolos de ventana. Concepto de ventana deslizante

Los protocolos ARQ de transmisión continua requieren el control del flujo de transmisión con el fin de evitar la saturación de las memorias intermedias de emisión o recepción, tal como hemos visto antes. Uno de los modos de dirigir este control de flujo es mediante un protocolo de ventana deslizante. La ventana deslizante define un número *W* de datagramas máximo que pueden

estar sin confirmar. En este sentido, el emisor puede enviar hasta W datagramas de información sin confirmar, que quedan almacenados en la memoria intermedia de transmisión.

Figura 25. Ventana de transmisión



Datagrama k: último datagrama confirmado
Datagrama $k + 2$: último datagrama transmitido no confirmado

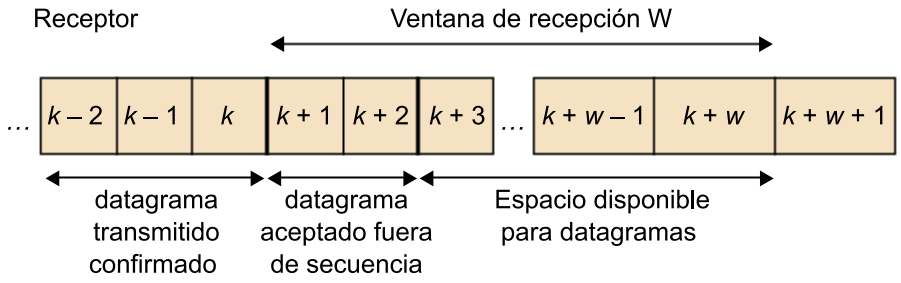
En el protocolo de ventana deslizante las estaciones deben estar conectadas de manera *full duplex*. Si I_k es el último datagrama confirmado (supongamos confirmaciones acumulativas) y W es el tamaño de la ventana de transmisión, el emisor puede transmitir hasta el datagrama I_{k+w}. Antes de cada transmisión, el emisor comprueba que el número de secuencia del datagrama que se ha de transmitir menos el número de secuencia del último datagrama confirmado es menor o igual que W. En caso contrario, el emisor se queda bloqueado esperando que lleguen confirmaciones. Según llegan las confirmaciones, el emisor puede volver a enviar datagramas.

La ventana de transmisión permite dimensionar el tamaño de la memoria intermedia de transmisión necesaria para almacenar los datagramas pendientes de confirmación, ya que los datagramas confirmados se pueden borrar de la lista.

Stop & wait es un caso particular, dado que $W = 1$. Después de la transmisión de un datagrama, el emisor se queda bloqueado hasta que llega la confirmación (sólo se puede enviar un datagrama sin confirmar).

En transmisión continua el tamaño de la memoria intermedia de transmisión es de W.

Figura 26. Ventana de recepción



datagrama *k*: último datagrama recibido y confirmado
datagrama *k* + 2: último datagrama recibido pero no confirmado

La ventana de recepción define el número máximo de datagramas recibidos sin errores, fuera de secuencia. En los protocolos en los que no se deben reordenar las tramas, como en *stop & wait y go-back-N*, el valor de la ventana de recepción es 1. En el protocolo de retransmisión selectiva en el que hay que reordenar los datagramas, se podrán recibir *W* datagramas fuera de secuencia (desde *k* + 1, *k* + 2..., hasta *k* + *w*), que es el número total de datagramas que puede transmitir el emisor. El valor máximo de la ventana de recepción será *W*.

Protocolo	Ventana TX	Ventana RX
Stop & wait	1	1
Go-back-N	*W*	1
Retransmisión selectiva	*W*	*W*

5.2. Ventana óptima

Establecer el valor de la ventana de transmisión es primordial para la eficiencia del protocolo ARQ. Si el tamaño de la ventana es demasiado pequeño, puede ocurrir que el protocolo se quede rápidamente bloqueado, esperando que lleguen las confirmaciones. Si la ventana del protocolo es demasiado grande, también puede ser un inconveniente, ya que las memorias intermedias de transmisión y recepción han de dimensionarse según el tamaño de la ventana.

Se define la ventana óptima como la mínima ventana que permite conseguir una eficiencia del 100% en el protocolo, es decir, la mínima ventana que permite conseguir la velocidad efectiva máxima.

- Si utilizamos una ventana $W < W_{optima}$, la velocidad efectiva será inferior a la que podríamos conseguir con una ventana mayor.

- Si utilizamos una ventana $W > W_{optima}$, no se aumentará la velocidad efectiva más allá de lo conseguido con la ventana óptima.

El tamaño óptimo de la ventana es:

$$W_{óptimo} = \frac{T_c}{t_{Trama}}$$

5.3. *Piggybacking*

Es una técnica utilizada tanto por el emisor como por el receptor cuando envían y reciben información. En este caso es necesario que exista una confirmación de tamaño mínimo, que se incorpora a la trama de información.

5.4. Números de secuencia

Los protocolos ARQ necesitan un número de secuencia para relacionar los datagramas de información y sus correspondientes confirmaciones. El número de secuencia lo lleva un campo de la cabecera del datagrama. Si este campo tiene *n* bits, entonces el número de secuencia podrá tener un valor en el intervalo $[0, 2^n - 1]$. Es decir, tenemos 2^n números de secuencia diferentes. Cuando se llega al número de secuencia $2^n - 1$, se vuelve a empezar con el valor 0, y se repite el ciclo de números de secuencia. La reutilización de los números de secuencia puede crear ambigüedades si no es lo bastante grande. Por lo tanto, cuando se confirma el datagrama *k*, ¿cómo puede saber el emisor si se confirma el datagrama del actual ciclo o del ciclo anterior?

Si los datagramas pueden llegar desordenados y con retrasos arbitrarios, esta ambigüedad no se puede resolver. Lo único que se puede hacer es elegir un número de bits suficientemente grande para que la probabilidad de que eso suceda sea prácticamente 0 (así actúa TCP).

Si los datagramas llegan en el mismo orden en el que se han enviado, se puede demostrar que necesitamos los siguientes números de secuencia para que los protocolos funcionen sin ambigüedades:

- Con *stop & wait* bastaría con un solo bit para el número de secuencia.
- *Go-back-N*: si deseamos una ventana de medida *W*, necesitaremos un número de bits *n* tal que $2^n \geq W + 1$.
- Retransmisión selectiva: $2^n \geq 2W$.

Protocolo	Número de identificadores necesarios	Tamaño de la ventana máxima si se utiliza *n* bits para los identificadores
Stop & wait	2	
Go-back-N	$W + 1$	2^{n-1}

Protocolo	Número de identificadores necesarios	Tamaño de la ventana máxima si se utiliza *n* bits para los identificadores
Retransmisión selectiva	$2 \cdot W$	2^{n-1}

6. Transporte orientado a la conexión: TCP

TCP es un protocolo de nivel de transporte que se usa en Internet para la transmisión fiable de información (quizá sea el protocolo más complejo e importante de los muchos protocolos de Internet).

Como hemos visto, UDP no garantiza la entrega de la información que le proporciona una aplicación. Tampoco reordena la información en el caso de que llegue en un orden diferente del orden en el que se ha transmitido. Existen aplicaciones que no pueden tolerar estas limitaciones. Para superarlas, el nivel de transporte proporciona TCP.

El TCP da fiabilidad a la aplicación, es decir, garantiza la entrega de toda la información en el mismo orden en el que ha sido transmitida por la aplicación de origen. Para conseguir esta fiabilidad, el TCP proporciona un servicio orientado a la conexión con un control de flujo y errores.

TCP es un protocolo ARQ, extremo a extremo, orientado a la conexión (fases de establecimiento de la conexión, envío de datos y cierre de la conexión) y bidireccional (dúplex). La unidad de datos TCP es el "segmento TCP". TCP, a diferencia de UDP, intenta generar segmentos de tamaño óptimo que se denominan MSS[17], generalmente lo mayor posible para minimizar el sobrecoste[18] de las cabeceras, pero que no produzca fragmentación a nivel IP. Al igual que UDP, TCP utiliza multiplexación mediante el uso de puertos, tal como hemos visto anteriormente.

[17] MSS es la sigla de *maximum segment size*.

[18] En inglés, *overhead*.

Proporciona fiabilidad mediante el:

- **Control de errores**. TCP es similar a *go-back-N*, pero no descarta segmentos posteriores cuando llegan fuera de secuencia. Permite las retransmisiones selectivas.

- **Control de flujo** (ventana advertida). Sirve para adaptar la velocidad entre el emisor y el receptor. Se encarga de que el emisor no envíe los segmentos a más velocidad de la que puede procesarlos el receptor y que se puedan perder paquetes por saturación de la memoria intermedia de recepción.

- **Control de la congestión** (ventana de congestión). Para adaptar la velocidad del emisor a los encaminadores intermedios de la red y evitar que se colapsen sus memorias intermedias y se puedan perder paquetes.

En transmisión, TCP fragmenta los datos del nivel aplicación y les asigna un número de secuencia antes de enviar los fragmentos al nivel IP.

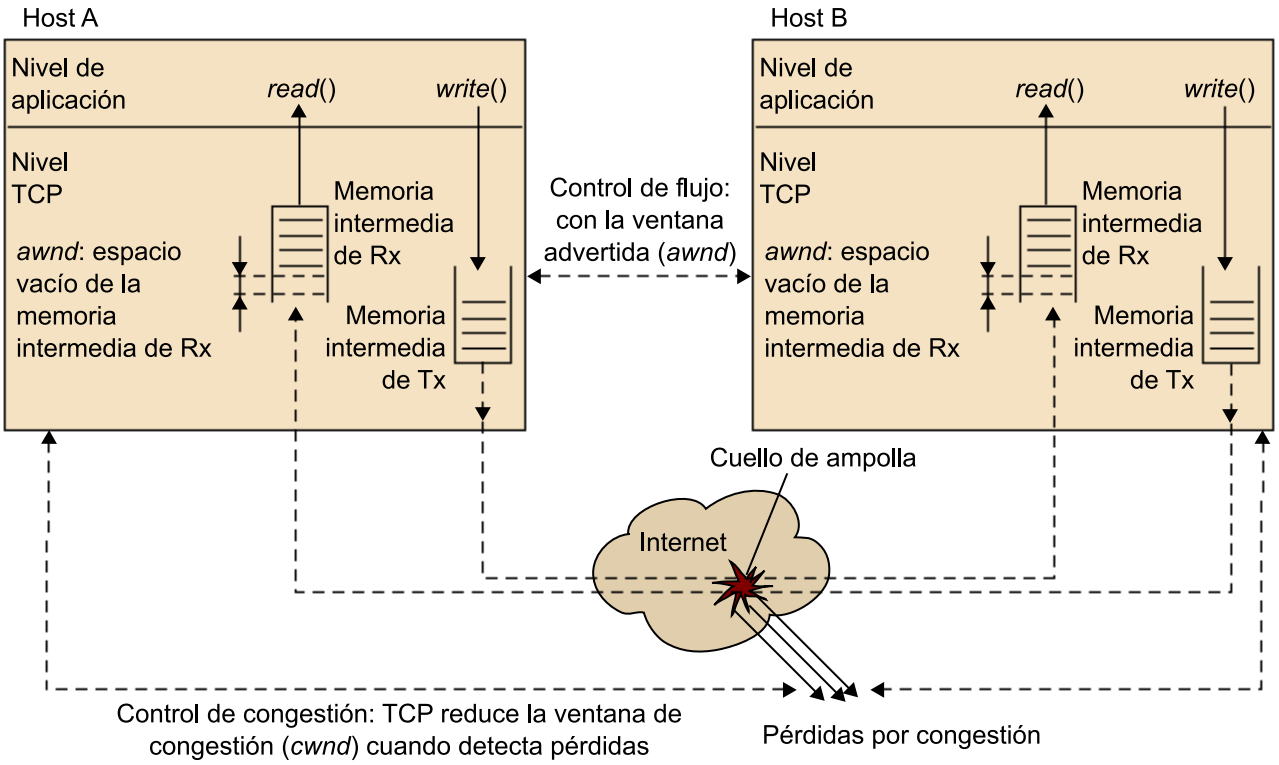
En recepción, como los segmentos pueden llegar fuera de orden, TCP los reordena antes de pasarlos a los niveles superiores.

6.1. Funcionamiento básico de TCP

En cada extremo, TCP mantiene una memoria intermedia de transmisión y otra de recepción. La aplicación utiliza las llamadas al sistema operativo *read*() y *write*() para leer y escribir en estas memorias intermedias. Cuando la aplicación escribe la información que se debe enviar al emisor, TCP la guarda en una memoria intermedia de transmisión. Cuando la memoria intermedia de Tx está llena, la llamada *write*() queda bloqueada hasta que vuelva a haber espacio.

Cada vez que llegan segmentos de datos al receptor, API *read*() los pasa al nivel superior y se envían sus correspondientes confirmaciones. Los datos, según van siendo confirmados por el receptor, se borran de la memoria intermedia de transmisión y queda espacio libre para que la aplicación siga escribiendo.

Figura 27



6.2. Cabecera TCP

Figura 28

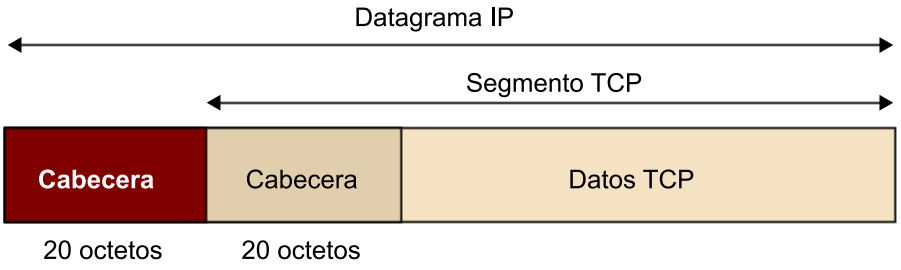
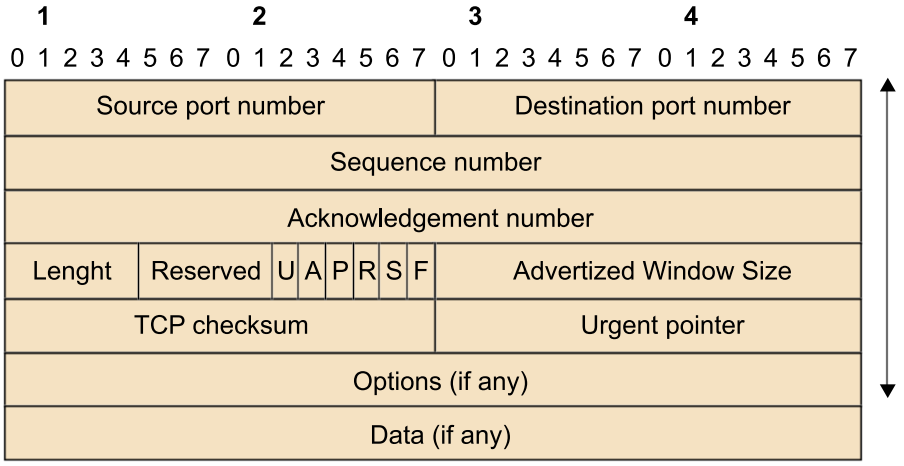


Figura 29



- **Puertos de origen y destino** (*Source port number* / *Destination port number*) que identifican las aplicaciones.

- **Número de secuencia del segmento** (*Sequence number*): identifica el primer byte dentro de este segmento de la secuencia de bytes enviados hasta aquel momento.

- **ISN** (*Initial sequence number*): primer número de secuencia elegido por el protocolo TCP. El número de secuencia será un número a partir de ISN. Por lo tanto, para saber cuántos bytes llevamos enviados hay que hacer:

  Número de secuencia – ISN

- **Número de reconocimiento** (*Acknowledgement number*): contiene el próximo número de secuencia que el transmisor del *ACK* espera recibir, es decir: número de secuencia + 1 del último byte recibido correctamente. TCP es dúplex: cada extremo mantiene un *seq. number* y un *ack number*.

- **Longitud total de la cabecera** (*Header length*) del segmento TCP en words de 32 bits (igual que el campo *header length* de la cabecera IP).
  - Tamaño mínimo de la cabecera TCP = 20 bytes (*header length* = 5).
  - Tamaño máximo de la cabecera TCP = 60 bytes (*header length* = 15).

- **Reservado** (*Reserved*): bits reservados para posibles ampliaciones del protocolo. Se ponen 0.

- **Indicadores** (*Flags*): existen 6 indicadores (bits) en la cabecera. Son válidos si están en 1.

- **URG** (*Urgent*): indica que se utiliza el campo Apuntador de urgencia[19] de la cabecera TCP.

- **ACK** (*Acknowledgement*): indica que se utiliza el campo Número de reconocimiento. Vale para todos menos para el SYN inicial.

- **PSH** (*Push*): indica que el receptor debe pasar los datos de la memoria intermedia de recepción a los niveles superiores tan rápido como pueda. Se trata de operar más rápido sin llenar la memoria intermedia. La activación de este indicador depende de la implementación. Las implementaciones derivadas de BSD lo activan cuando la memoria intermedia de Tx se queda vacía.

- **RST** (*Reset*): se activa cuando se quiere abortar la conexión. Un ejemplo de esto es cuando se recibe un segmento de un cliente dirigido a un puerto en el que no hay ningún servidor escuchando. En este caso, TCP contesta con un segmento con el indicador de *reset* activado.

- **SYN** (*Synchronize*): se utiliza en el establecimiento de la conexión, durante la sincronización de los números de secuencia al inicio de la conexión.

- **FINAL** (*Finalize*): se utiliza en la finalización de la conexión.

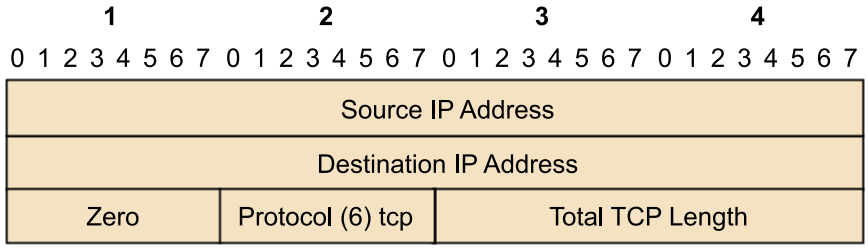
- **Tamaño de la ventana detectada** (*Advertised window size*): tamaño de la ventana detectada por el receptor en el transmisor (*sliding window*) para el control de flujo. El tamaño máximo de la ventana es 65.535 bytes.

- **Suma de comprobación TCP** (*TCP checksum*): se utiliza para detectar errores en el segmento TCP y para tener una mayor certeza de que el segmento no ha llegado a un destino equivocado.

  – Al igual que en UDP, la suma de comprobación TCP se aplica junto con una pseudocabecera + la cabecera TCP + el campo de datos TCP. Esta pseudocabecera tiene 3 campos de la cabecera IP y el tamaño del segmento TCP (cabecera + *payload*).

(19) En inglés, *urgent poiter*.

Figura 30



– La pseudocabecera sirve para calcular la suma de comprobación: no se transmite en el segmento TCP, ni se incluye en la longitud del paquete IP.

– El tamaño del segmento TCP no se pone en la cabecera TCP, sólo se tiene en cuenta en el cálculo de la suma de comprobación.

– A diferencia de UDP, en TCP el cálculo de la suma de comprobación es obligatorio.

- **Apuntador de urgencia** (*Urgent pointer*): campo válido si el indicador *URG* = 1. Implementa un mecanismo para indicar datos urgentes (es decir, que se deben atender lo mejor posible). Es un puntero al *seq. number* que indica la parte de datos urgentes dentro del campo de datos. Los datos urgentes irán desde el primer byte del segmento hasta el byte indicado por el *urgent pointer*. Este indicador se utiliza rara vez. Un ejemplo de ello es cuando se teclea un control-C (interrupción) desde la aplicación Telnet.

- **Opciones** (*Options*): TCP permite añadir opciones a la cabecera, pero a diferencia de IP, las opciones de TCP suelen utilizarse. Podemos destacar las siguientes:

  – **Tamaño máximo del segmento** (*Maximum segment size*): se utiliza durante el establecimiento de la conexión para sugerir el valor del MSS en el otro extremo MSS = MTU red directamente conectada – cabecera IP – cabecera TCP (sin opciones). Si la red es Ethernet (MTU 1.500), entonces MSS = 1.460.

  – **Factor de escala de la ventana** (*Window scale factor*): se utiliza durante el establecimiento de la conexión para indicar que el valor de la ventana advertida se debe multiplicar por este factor de escala. Esto permite advertir ventanas mayores de 216.

  – **Sello de tiempo** (*Timestamp*): se utiliza en el cálculo del *RTT*.

  – **SACK**: permite que TCP haga retransmisión selectiva (*selective ack*). TCP utiliza el campo *ack* para indicar hasta dónde se ha recibido correctamente. Con la opción *SACK* el receptor puede indicar bloques de segmentos

que se han recibido correctamente, más allá del segmento confirmado por el *ack*. De este modo, el emisor puede elegir mejor los segmentos que se deben retransmitir.

– ***Padding***: bytes de relleno añadidos para que la cabecera tenga un múltiplo de 32 bits.

Actividad 2

Asumimos que un extremo cliente TCP (emisor) ha elegido el 28.325 como número de secuencia inicial (ISN), mientras que el extremo receptor TCP (servidor) ha optado por el 12.555 como ISN. ¿Qué indica un segmento cliente (emisor) TCP con número de secuencia 29.201, número *ACK* 12.655 y ventana 1.024?

Solución actividad 2

El número de secuencia indica que el cliente ya ha transmitido desde el byte 28.325 hasta el byte 29.200 (875 bytes en total) y que en este segmento transmitirá a partir del byte 29.201. El número *ACK* indicará al receptor que el emisor ha recibido correctamente hasta el byte 12.654 y que espera recibir a partir del 12.655. La ventana indica al receptor que el cliente sólo puede aceptar 1.024 bytes antes de confirmarlos. Por lo tanto, el servidor TCP actualizará su ventana de transmisión a 1.024.

6.3. Control de flujo en TCP

El control de flujo es primordial para el funcionamiento de TCP. Con anterioridad hemos visto cómo los protocolos ARQ conseguían controlar el flujo de transmisión mediante el uso de memorias intermedias de emisión y de transmisión. TCP también utiliza esta técnica e intenta evitar la saturación de la memoria intermedia de recepción en el caso de que la velocidad de transmisión del emisor sea mayor que la velocidad de procesado del receptor. El control de flujo es impuesto por el receptor, que informa sobre el tamaño actual de la memoria intermedia de recepción mediante el campo ventana advertida[20] en la cabecera de las confirmaciones (*acks*). El receptor no puede enviar más bytes de los que indica la ventana advertida. De este modo, siempre habrá espacio suficiente para guardar los bytes enviados en la memoria intermedia de recepción del receptor. Cuando la memoria intermedia de recepción se llena, el receptor encontrará una ventana advertida de valor 0. Mientras la ventana de transmisión sea cero, el emisor no podrá enviar más información. Con el fin de regular la transmisión, TCP adapta la ventana a la condición más restrictiva, tanto de control de flujo como de control de congestión.

[20] En inglés, *advertised window*. Abreviado, *awnd*.

En todo momento TCP utiliza una ventana de transmisión que puede ser definida: ventana TX $wnd = mín(awnd, cwnd)$ donde *cwnd* es la ventana de congestión[21].

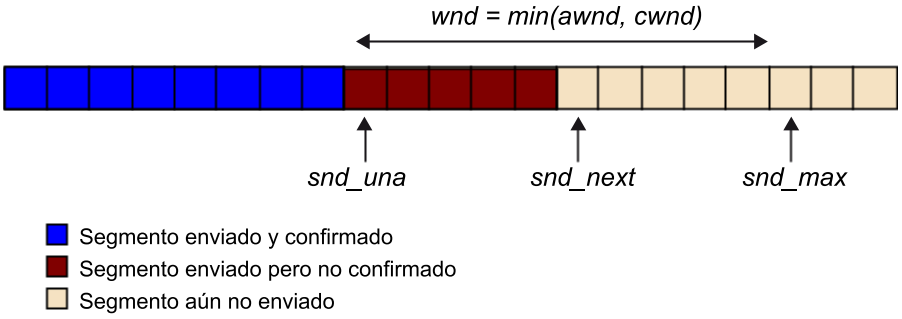
[21] En inglés, *congestion window*. Abreviado, *cwnd*.

6.3.1. Ventana deslizante en TCP

TCP implementa un protocolo de control de flujo mediante una ventana de tamaño deslizante.

El control de flujo extremo a extremo permite que el emisor envíe segmentos sin recibir su correspondiente *ack*, gracias al uso de una memoria intermedia. El receptor almacena en la memoria intermedia los segmentos que le han llegado pero que todavía no han sido utilizados. Si la aplicación no consume los datos recibidos, se dejan de reconocer segmentos.

Figura 31



awnd	Ventana advertida
cwnd	Ventana de congestión
snd_una	Primer segmento enviado no confirmado
snd_next	Próximo segmento para enviar
snd_max	Último segmento que puede ser enviado (si no, no se confirman nuevos datos)

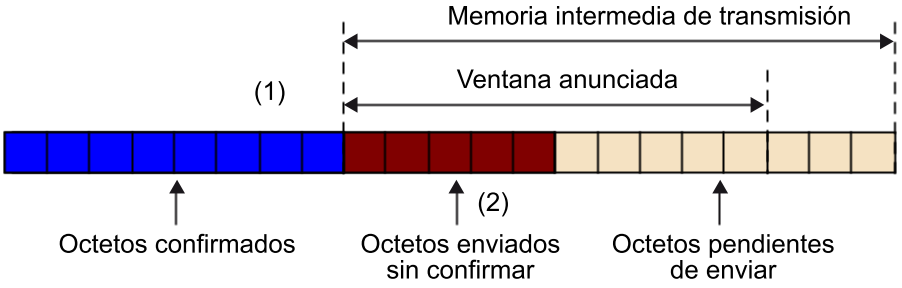
6.3.2. Ventana advertida de transmisión

El emisor guarda en la memoria intermedia:

- Los segmentos enviados y no reconocidos. Cuando las memorias intermedias se llenan, el emisor deja de enviar segmentos.

- Los bytes que todavía no han sido transmitidos.

Figura 32



En la figura 32 vemos que:

1) Último *ack* recibido ≤ último byte enviado y confirmado (consumido).
2) Último byte enviado ≤ último byte consumido (*write*).

También podemos ver que el tamaño de la memoria intermedia será:

Tamaño memoria intermedia *r* ≥ Último byte escrito – Último *ack* recibido

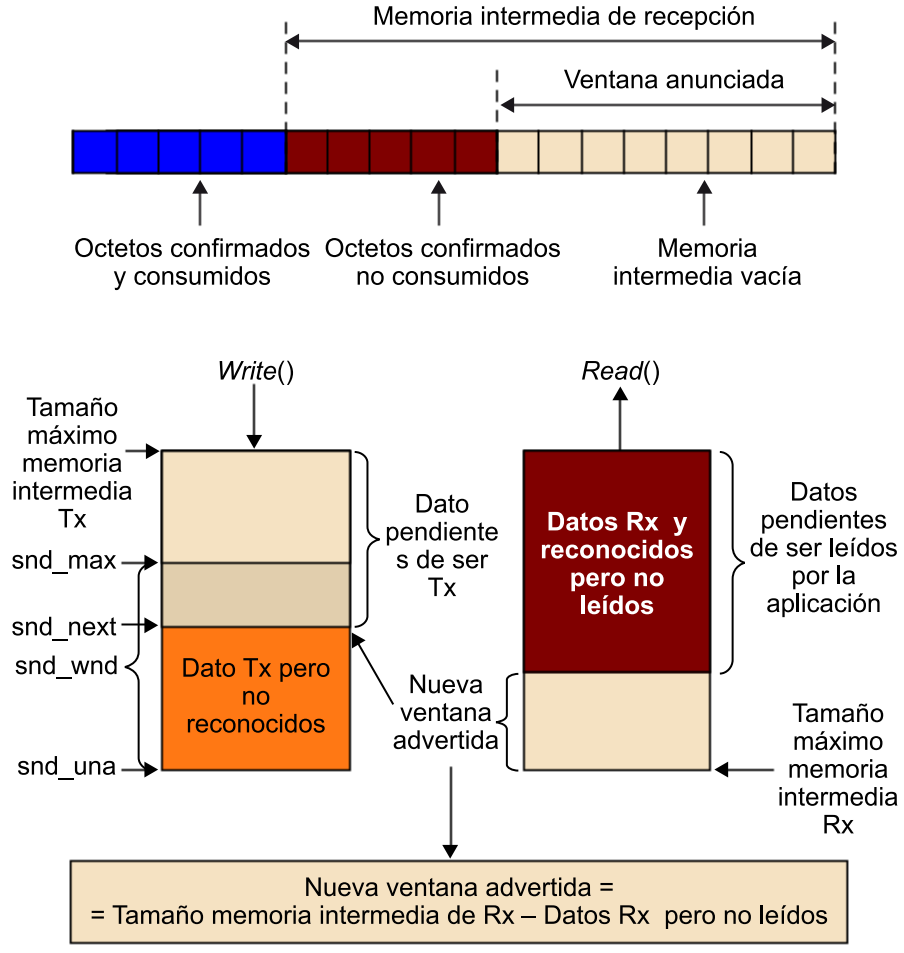
Y que la ventana advertida:

Ventana advertida ≥ Último byte consumido – Último *ack*

6.3.3. Ventana advertida de recepción

El receptor guarda en la memoria intermedia los bytes recibidos y los que están pendientes de consumir por la aplicación.

Figura 33



Casos que se pueden dar:

- Si el receptor es más rápido que el emisor, el primero estará esperando continuamente datos y la ventana podría llegar a ocupar toda la memoria intermedia.

- El receptor es igual de rápido que el emisor. En este caso, se van transmitiendo datos y la memoria intermedia del receptor alberga tanto los no consumidos como los que se han recibido del emisor.

- El receptor es más lento que el emisor. En este caso, la ventana advertida se irá reduciendo hasta que bloquee al emisor, ya que no podrá enviar más datos.

Tipos de aplicaciones que usan TCP

1) Transferencia masiva[22]:

- Aplicaciones que envían gran cantidad de datos.
- La memoria intermedia de transmisión siempre está llena y TCP envía segmentos de tamaño máximo.
- La ventana de transmisión limita la velocidad efectiva de la conexión.
- Aplicaciones que generan este tipo de tráfico: FTP, web, correo electrónico, etc.
- Los segmentos llevan más de 512 bytes de datos.
- Importan los algoritmos de control de la congestión (*slow start*, *congestion avoidance*, *fast retransmit* y *fast recovery*).

2) Aplicaciones interactivas:

- Aplicaciones en las cuales el usuario interactúa con la máquina remota: *rlogin*, *telnet*.
- La información se envía en segmentos de pocos bytes y de manera discontinua (el 90% de los segmentos tienen menos de 10 bytes de datos).
- No es importante el control de la congestión. No significa que no lo implementen, sólo que no se ven afectadas por este control de la congestión.

(22)En inglés, *bulk transfer*.

6.3.4. El problema de las aplicaciones interactivas

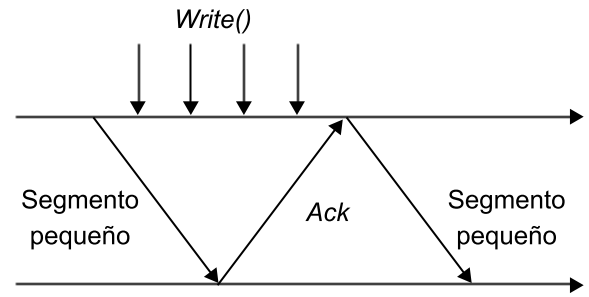
En conexiones TCP como *telnet* o *rlogin*, los datos se escriben, prácticamente, de byte en byte, lo que provoca que se transmitan muchos segmentos pequeños y sus correspondientes confirmaciones por cada tecla pulsada. El hecho de que cada segmento deba ser confirmado implica que la relación de bytes enviados de información con respecto a bytes transmitidos en confirmaciones sea baja y que la eficiencia del protocolo decaiga. Para solucionar este problema el algoritmo de Nagle propone:

- Evitar el envío de un segmento por cada tecla pulsada en las aplicaciones interactivas. Los bytes son almacenados en la memoria intermedia de transmisión, entre *ack* y *ack* recibidos, con el objetivo de enviarlos en un único segmento.

- Si la ventana permite enviar un nuevo segmento, éste se enviará si: hay suficientes bytes para enviar un segmento de tamaño máximo; no existen bytes pendientes de confirmación; hay bytes pendientes de confirmar (en

este caso, los bytes que van llegando a la memoria intermedia de transmisión se retienen hasta que llega la confirmación).

Figura 34



6.3.5. *Delayed acknowledgements*

Otro modo de dirigir el problema de las transmisión de datos de aplicaciones interactivas es mediante la técnica de los *delayed acknowledgements*. Esta técnica reduce el número de *acks* enviados por el receptor. El receptor no transmite el *ack* inmediatamente después de recibir nuevos datos, sino que espera un tiempo para ver si llegan los nuevos segmentos de información y después envía un *ack* (que los confirmará a todos).

Algunas implementaciones recomiendan:

- Retraso *delayed acks* 0,5 ≤ segundos.
- Enviar al menos 1 *ack* por cada 2 segmentos de tamaño máximo.

En las implementaciones de TCP/IP actuales el retraso de los *delayed acks* es menor o igual a 200 ms.

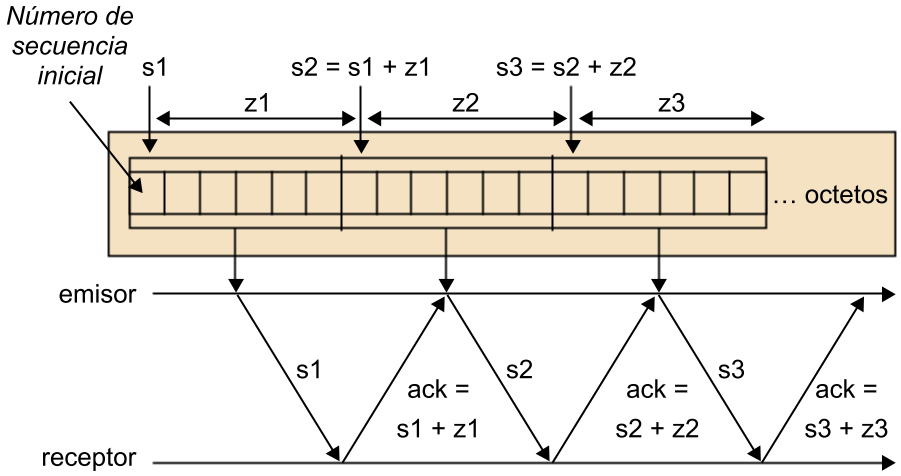
6.4. Números de secuencia en TCP

Como ya hemos visto en este módulo, el uso de los números de secuencia es primordial para el funcionamiento de los protocolos ARQ, ya que nos permiten establecer correspondencias entre las tramas enviadas y sus confirmaciones.

En TCP, los números de secuencia y las ventanas se miden en bytes, que identifican el primer byte de datos del segmento. Si un segmento tiene un número de secuencia S_i y lleva MSS bytes, los siguientes números de secuencia serán S_i + MSS, S_i + 2 MSS, S_i + 3 MSS, incrementos de valor MSS. La confirmación del segmento de datos con número de secuencia S_i, con MSS bytes, posee el valor: *ack* = S_i + MSS (*ack* es el valor del número de secuencia del próximo segmento de datos que espera recibir el receptor o el próximo byte que falta en el receptor). Las confirmaciones son acumulativas: la recepción de una confirmación

con *ack* implica que todos los bytes identificados con números de secuencia inferiores al del *ack* han llegado correctamente al receptor y que, por lo tanto, el emisor los puede borrar de la memoria intermedia de transmisión.

Figura 35



6.5. Establecimiento y finalización de una conexión en TCP

Tal como hemos comentado, TCP es un protocolo orientado a la conexión. Esto supone que en todo proceso de comunicación habrá una fase de establecimiento de la conexión en la que el emisor y el receptor se sincronizan con el fin de poder intercambiar datos. TCP utiliza el algoritmo three-way handshake[23], que consiste en el intercambio de tres segmentos que no llevan datos (sólo es la cabecera TCP).

[23] En castellano, *establecimiento a tres manos*.

Establecimiento de la conexión

1) El emisor envía un segmento SYN con:

- Puerto destino = puerto bien conocido[24] de un servidor (< 1.024).
- Puerto origen = puerto efímero (elegido por el núcleo[25], > 1.024).
- *Seq. number* = ISN cliente elegido al azar por el núcleo. Se utiliza para identificar los bytes de datos enviados por el cliente.
- Indicador SYN = 1 (activado).
- Indicador *ACK* = 0 (no se confirma nada).
- Es posible negociar (indicar) opciones como el MSS, *timestamp*, *window scale factor* o SACK, como indicación de que se quiere utilizar.

2) El receptor (servidor) devuelve un segmento SYN + ACK:

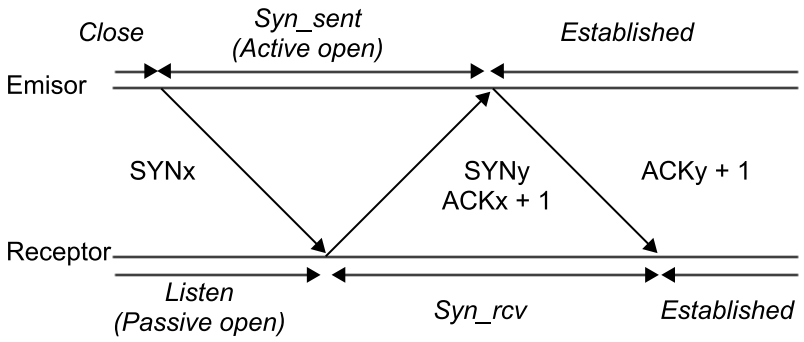
- *Seq. number* = ISN servidor para identificar los bytes de datos enviados por el servidor. Los segmentos de SYN, aunque no lleven ningún byte de datos, consumen un número de secuencia.
- ACK = ISN cliente + 1, reconociendo el segmento SYN e indicando su ISN.
- Indicador SYN = 1 (activado).
- Indicador ACK = 1 (activado).

[24] En inglés, *well known*.

[25] En inglés, *kernel*.

3) El cliente responde con un segmento ACK y reconoce el SYN + ACK. Por último, el cliente confirma la recepción del SYN + ACK mediante el envío de la confirmación con el valor ISN servidor + 1. Una vez establecida la conexión se pasa a la fase de envío de datos.

Figura 36




```
11:27:13.771041 147.83.35.18.3020 > 147.83.32.14.ftp: S
951111901:951111901(0) win 32120 <mss 1460,sackOK,timestamp
86683767 0,nop,wscale 0> (DF)
11:27:13.771491 147.83.32.14.ftp > 147.83.35.18.3020 : S
211543977:211543977(0) ack 951111902 win 10136 <nop,nop,timestamp
104199850 86683767,nop,wscale 0,nop,nop,sackOK,mss 1460> (DF)
11:27:13.771517 147.83.35.18.3020 > 147.83.32.14.ftp: . 1:1(0) ack 1
win 32120 <nop,nop,timestamp 86683767 104199850> (DF)
```

Cierre de la conexión

Al igual que en el establecimiento de la conexión, es necesario un proceso de finalización de la conexión. El cierre de la conexión puede deberse a varias causas:

- El cliente o el servidor cierran la conexión (LLS *close*()).
- Por alguna razón se envía un *reset* de la conexión (indicador activo RST).
- Cierre a causa de una interrupción, un control ^D o un control ^C, etc.

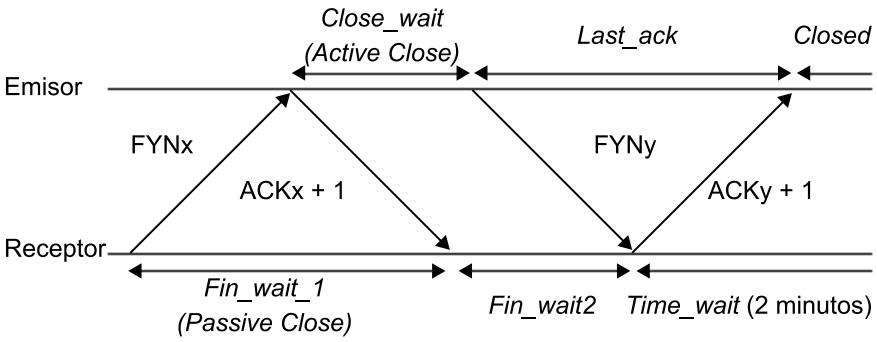
La finalización también se lleva a cabo mediante el envío de segmentos TCP. El primer segmento de FINAL puede ser enviado tanto por el cliente como por el servidor. El cierre normal se debe a un *close*() del cliente que provoca el intercambio de 3 o 4 segmentos TCP.

Como la conexión TCP es dúplex, cada participante en la conexión debe cerrar ésta. Se utilizan los mensajes FIN/*ack* en un sentido y en otro. Y al igual que SYN, el FINAL consume un número de secuencia. El segmento FINAL puede llevar datos, si todavía quedan bytes de datos que enviar en la memoria intermedia de transmisión de TCP cuando la aplicación realiza la llamada *close*().

Es posible que un extremo cierre su lado de la conexión pero no el otro. En este caso, el extremo que no ha cerrado puede enviar datos y el otro extremo los reconocerá, aunque haya cerrado su conexión. Este hecho se denomina conexión de cierre parcial[26].

[26]En inglés, *half closed*.

Figura 37

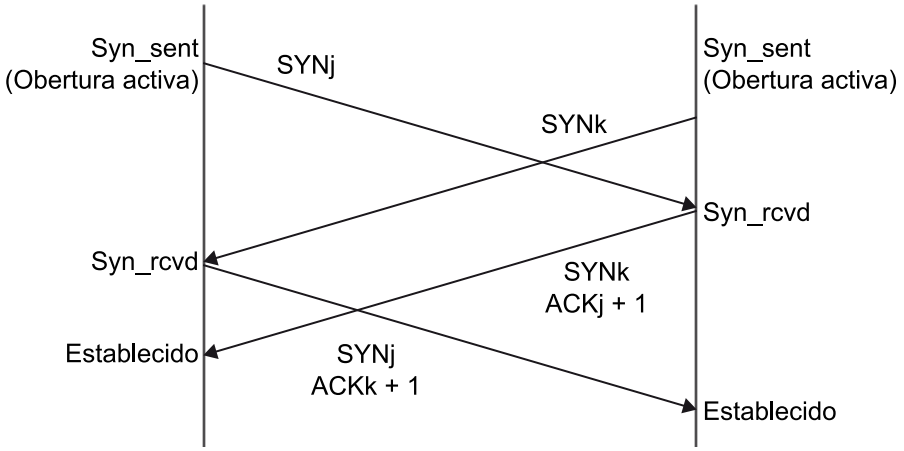



```
11:27:19.397349 147.83.32.14.3020 > 147.83.35.18.ftp: F
3658365:3658365(0) ack 1 win 10136 <nop,nop,timestamp 104200411
86684327> (DF)
11:27:19.397370 147.83.35.18.ftp > 147.83.32.14.3020: . 1:1(0) ack
3658366 win 31856 <nop,nop,timestamp 86684330 104200411> (DF)
```

```
11:27:19.397453 147.83.35.18.ftp > 147.83.32.14.3020: F 1:1(0) ack
3658366 win 31856 <nop,nop,timestamp 86684330 104200411> (DF)
11:27:19.398437 147.83.32.14.3020 > 147.83.35.18.ftp: .
3658366:3658366(0) ack 2 win 10136 <nop,nop,timestamp 104200412
86684330> (DF)
```

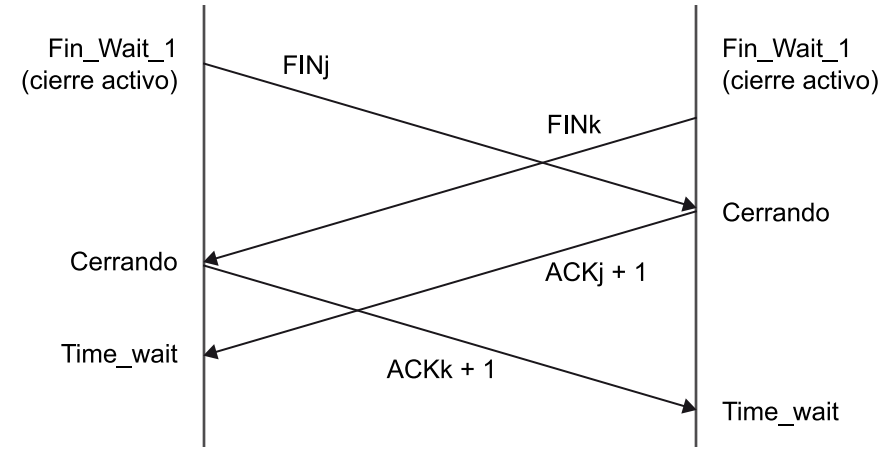
Los diagramas de tiempos siguientes muestran distintas situaciones que se pueden dar en el establecimiento o cierre de la conexión TCP.

Figura 38. *Simultaneous open*



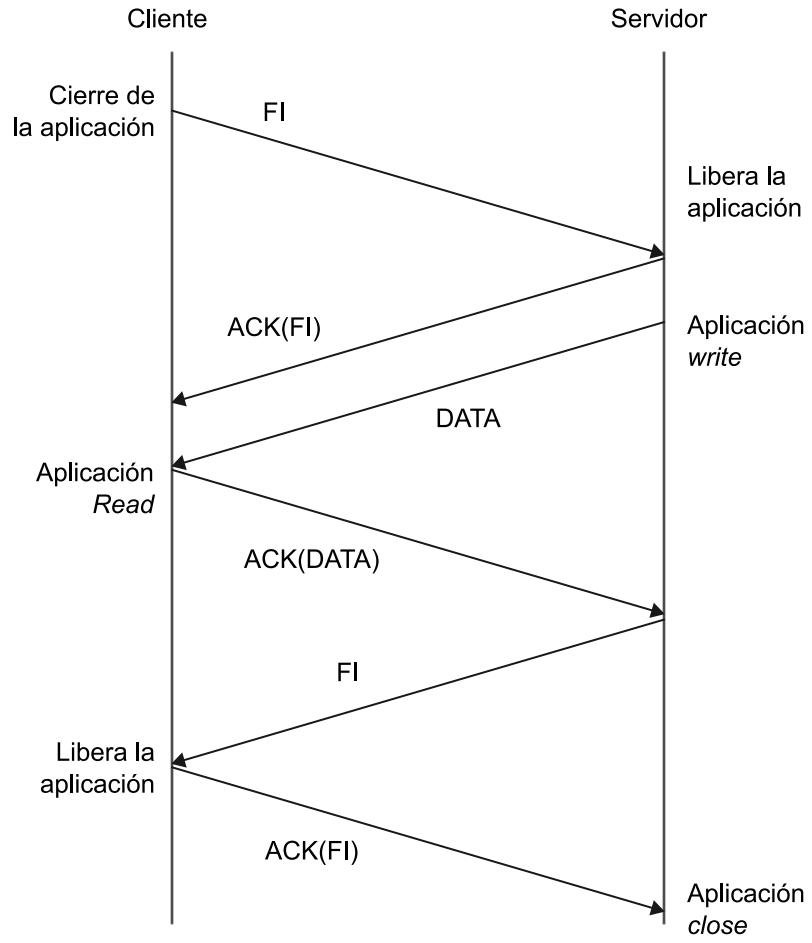
Simultaneous open es una situación en la que el emisor y el receptor intentan establecer la conexión de un modo concurrente. En este caso, el propio SYN-*ACK* confirma la conexión

Figura 39. *Simultaneous close*



Simultaneous close es similar a la situación anterior, en la que el emisor y el receptor deciden finalizar la comunicación de un modo concurrente, pero, en este caso, la finalización de la comunicación se lleva a cabo para cada extremo independiente

Figura 40. *Half closed*



Finalmente, el cierre parcial permite al extremo que ha finalizado la conexión seguir enviando confirmaciones al otro extremo

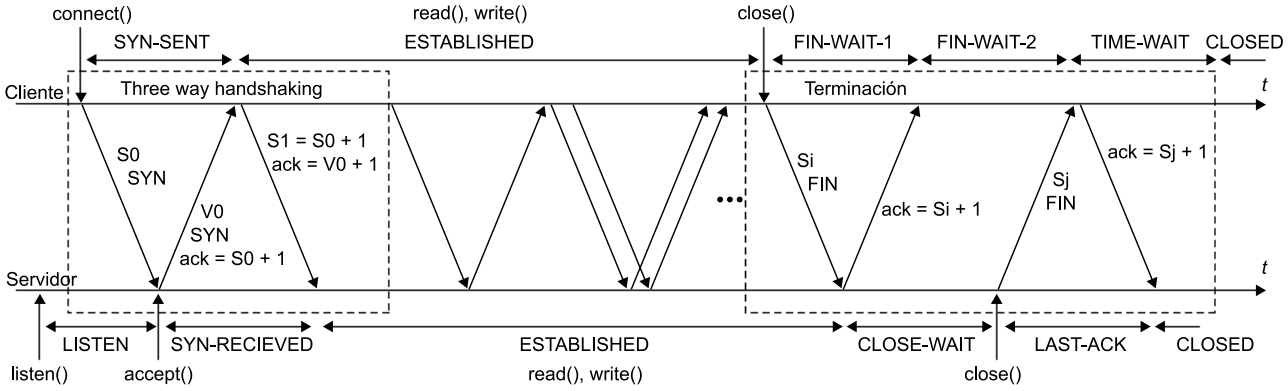
6.6. Diagrama de estados de TCP

El protocolo TCP es orientado a la conexión y, por lo tanto, tiene diferentes estados durante sus ciclos de vida. En concreto, pasa por 3 fases, divididas a su vez en un número determinado de estados.

1) Fase de establecimiento de la conexión.
2) Fase de transmisión.
3) Fase de finalización.

Entre estas fases podemos distinguir 11 estados diferentes:

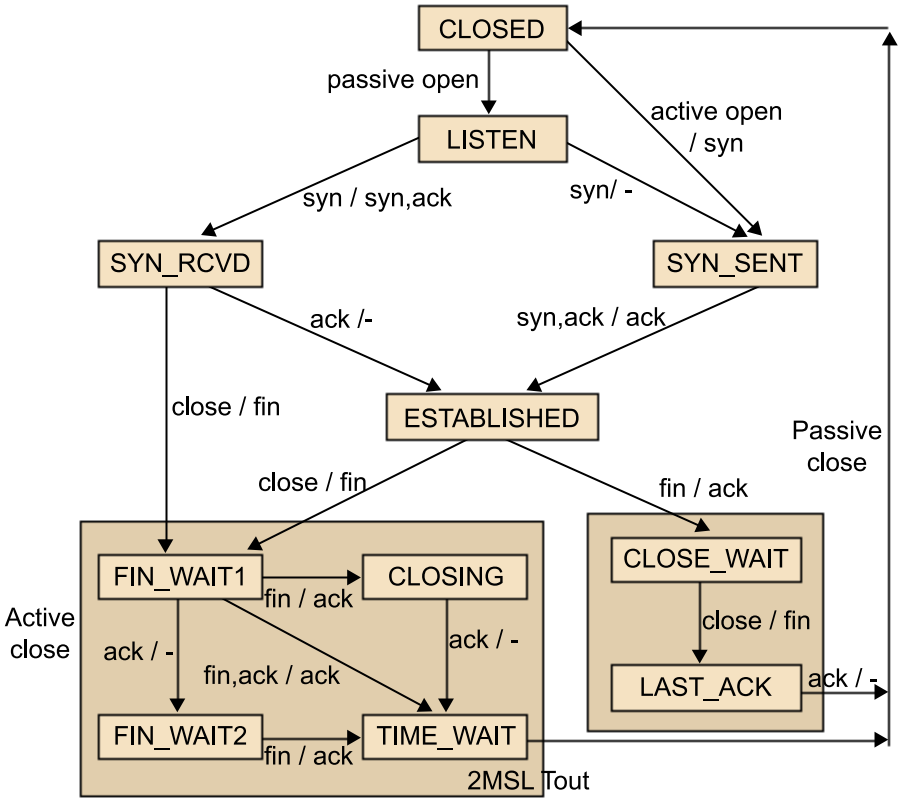
Figura 41



La siguiente tabla presenta los 11 estados de TCP:

Active open	***CLOSED***	La conexión no está siendo usada.
	SYN_SENT	El emisor envía un segmento SYN después de ejecutarse el API *connect*() y queda a la espera de recibir una confirmación.
Passive open	***LISTEN***	El receptor espera recibir peticiones de clientes (emisores).
	SYN_RECEIVED	El receptor recibe un segmento SYN y acepta la conexión (realiza la llamada al API *accept*(). Envía un segmento con los indicadores SYN y ACK activados.
ESTABLISHED		Conexión establecida. El cliente (emisor) pasa a éste cuando recibe el segmento SYN + ACK del (receptor) servidor y responde con un ACK. Cuando el servidor (receptor) recibe este ACK también pasa al estado de *ESTABLISHED*.
Active close	***FIN_WAIT_1***	El servidor o cliente realiza un *close*() de la conexión y envía el segmento FINAL. Se produce el cierre de la conexión origen.
	CLOSED	El otro extremo, cuando recibe el *ack* del FINAL que ha enviado, tiene la certeza de que los dos extremos han recibido el segmento *FINAL* y pasa directamente al estado *CLOSED*.
	FIN_WAIT_2	Se recibe confirmación de conexión de origen cerrado por parte del otro extremo y se queda a la espera de la desconexión remota.
	***TIME_WAIT* (2 minutos)**	El ordenador principal que ha iniciado la finalización, cuando recibe el segmento *FINAL* de desconexión de la conexión remota, envía el ACK y se queda en el estado de *ESTAFI-WAIT* durante un tiempo $T = 2 \times$ MSL (máximo tiempo de vida del segmento) = 2 minutos. En caso de pérdida de esta confirmación, el ordenador principal podría contestar a las posibles retransmisiones del último FINAL mientras se encuentre en este estado. Aproximadamente, un minuto es el tiempo que puede estar un datagrama como máximo en Internet.
Passive close	***CLOSE_WAIT***	Recepción del primer segmento FINAL, envío de confirmación y espera para la desconexión de la conexión remota.
	LAST_ACK	Inicio de desconexión remota. Queda a la espera de recibir reconocimiento por parte del otro extremo.

Figura 42



Ejemplo de funcionamiento del protocolo de ventana deslizante

Utilizaremos el programa Tcpdump para ver cómo funciona el protocolo de ventana deslizante. Asumimos que hemos efectuado un *rlogin* de argos a helios (argos % rlogin helios) y estamos conectados a helios. Una vez en helios, ejecutamos el mando ls. Éste devuelve por salida estándar el listado de directorios del directorio del usuario[27] en helios, que ocupan 811 caracteres (representa el envío de 811 bytes).helios % ls

[27]En inglés, *home directory*.

Las líneas que obtenemos con el programa Tcpdump (numeradas de la 1 a la 13) son las siguientes:

1) 15:56:59.506091 argos.1023 > helios.login: P 37:38 (1)*ack* 596 win 31744
2) 15:56:59.516091 helios.login > argos.1023: P 596:597 (1) *ack* 38 win 8760
3) 15:56:59.526091 argos.1023 > helios.login: .*ack* 597 win 31744
4) 15:56:59.846091 argos.1023 > helios.login: P 38:39 (1)*ack* 597 win 31744
5) 15:56:59.856091 helios.login > argos.1023: : P 597:600 (3) *ack* 39 win 8760
6) 15:56:59.866091 argos.1023 > helios.login: .*ack* 600 win 31744
7) 15:57:00.116091 argos.1023 > helios.login: P 39:40 (1)*ack* 600 win 31744
8) 15:57:00.126091 helios.login > argos.1023: P 600:603 (3) *ack* 40 win 8760
9) 15:57:00.136091 argos.1023 > helios.login: .*ack* 603 win 31744
10) 15:57:00.146091 helios.login > argos.1023: P 603:658 (55) *ack* 40 win 8760
11) 15:57:00.156091 argos.1023 > helios.login: .*ack* 658 win 31744
12) 15:57:00.166091 helios.login > argos.1023: P 658:1414 (756) *ack* 40 win 8760
13) 15:57:00.176091 argos.1023 > helios.login: .*ack* 1414 win 31744

La interpretación de estas líneas es la siguiente: argos ya ha enviado 36 bytes, mientras que helios ha mandado 595 (información que los dos han intercambiado desde el comienzo de la conexión, como pueden ser *logins*, *usernames*, etc.). Deducimos esta información de la primera línea del ejemplo.

1) argos envía el carácter 'l'. El indicador *P* señala PUSH. El número de secuencia avanza de 37 a 38.

2) helios devuelve un eco del carácter 'l'. Su número de secuencia avanza de 596 a 597 y reconoce el byte recibido (*ACK* = 37 + 1 = 38).

3) argos reconoce el eco: *ACK* = 597 + 1 = 598.

4) argos envía el carácter 's'. El número de secuencia avanza de 38 a 39. El *ACK* no reconoce nada porque vale igual que antes: *ACK* = 597.

5) helios hace un eco que ocupa 3 bytes (BS* = 1 + s). El número de secuencia avanza tres posiciones (de 597 a 600) y reconoce el carácter 's', dado que *ACK* = 38 + 1 = 39.

6) argos reconoce el eco: *ACK* = 597 + 1 = 598.

7) argos envía el retorno de carro (CR). El número de secuencia avanza una posición.

8) helios hace un eco del CR y, además, devuelve otro CR seguido de un LF (*line feed*). Esto significa el envío de 3 bytes. Reconoce el CR, dado que *ACK* = 40.

9) argos reconoce estos tres caracteres.

10) helios responde a 'ls' enviando 55 de los 811 bytes que debe enviar. El número de secuencia avanza de 603 a 658. El *ACK* se queda en 40.

11) argos reconoce estos 55 bytes enviando un *ACK* de 659.

12) helios transmite el resto de los 811 bytes, es decir, 756 bytes.

13) argos reconoce estos bytes avanzando el *ACK* a 1.414.

Como podemos ver en este ejemplo, el TCP divide la información al enviar dos segmentos: un segmento de 55 bytes y otro de 756 bytes. Debemos señalar que *rlogin* envía los mandos carácter a carácter y que la aplicación remota hace un eco de estos caracteres. Por ello, en las primeras líneas se envía primero la 'l', después la 's', después el retorno de carro, etc. Lo que nos interesa de este ejemplo es ver cómo avanzan las ventanas al emitir y reconocer bytes. Por lo tanto, no justificaremos por qué *rlogin* devuelve ecos ni por qué añade un carácter LF al retorno de carro.

6.7. Control de la congestión

Hasta ahora hemos visto cómo TCP implementaba el control de errores y control de flujo haciendo uso de variantes de los protocolos ARQ que hemos introducido al principio de este módulo. Uno de los problemas que pueden aparecer durante la comunicación entre un emisor y un receptor es la congestión: TCP intenta adaptar la velocidad de transmisión del emisor a la de los encaminadores intermedios de la red para evitar la saturación de sus memorias intermedias y las pérdidas de paquetes. Para ello, utiliza la ventana de congestión (*cwnd*). Este control de flujo es impuesto por el emisor, ya que es aquél el que regula la ventana de congestión. El protocolo intenta disminuir la congestión de la red en el momento en el que aparece; cuando se pierde un paquete o salta un temporizador, la ventana de congestión se divide por la mitad y provoca que la cantidad de información que se envía a la red disminuya de modo multiplicativo en presencia de múltiples pérdidas de paquetes. Asimismo, según detecta que ya no hay pérdida de paquetes, el tamaño de la ventana aumenta de modo aditivo para intentar maximizar la cantidad de información enviada.

En caso de congestión, la *cwnd* se reduce de modo multiplicativo (es decir, rápidamente) para evitar agravar la congestión. En ausencia de congestión, *cwnd* se incrementa lentamente, intentando buscar el punto en el que no haya congestión y en el que el aprovechamiento de las líneas de transmisión sea máximo.

Para conseguir este comportamiento de la red se han definido varios mecanismos que intentan regular la congestión. Los mecanismos básicos usados son:

- *Slow start/congestion avoidance*.
- *Fast retransmit/fast recovery*.

Actividad 3

Buscad cuál es la implementación actual de TCP. ¿Qué características posee?

6.7.1. Algoritmos *slow start* y *congestion avoidance*

Ambos mecanismos funcionan conjuntamente. Realizan el control básico de la ventana de TCP. Estos algoritmos emplean las siguientes variables:

- ***cnwd***: es la ventana de congestión.
- ***snd_una***[28]: es el primer segmento no confirmado. Es decir, es el segmento que hace más tiempo que espera en la memoria intermedia de transmisión para ser confirmado.
- ***ssthresh***[29]: límite entre la fase *slow start* y *congestion avoidance*.

(28) *snd_una* significa *unacknowledged*, sin confirmar.

(29) *ssthresh* significa *slow start threshold*, umbral de inicio lento.

6.7.2. *Slow start*

Slow start aumenta el valor de la ventana de congestión hasta llegar lo más rápidamente posible al umbral *ssthresh*, valor por encima del cual se pueden producir pérdidas de paquetes. *Slow start* inyecta segmentos a la red según va recibiendo *Acks* desde el otro extremo. Durante el *slow start*, el incremento de la ventana de congestión es muy rápido, crece de modo exponencial en función del tiempo.

En cada *RTT*[30] (el retraso que hay desde que TCP envía un segmento hasta que llega su confirmación), la *cwnd* se multiplica por 2. En un tiempo $n \times RTT$, la *cwnd* pasa a valer aproximadamente 2^n.

(30) *RTT* es la sigla de *round trip time*.

Se envían segmentos de manera exponencial, hasta que se satura el canal de comunicación y se llena la memoria intermedia de algún encaminador intermedio, lo que produce pérdidas de paquetes.

El nombre *slow start* es contradictorio porque *cwnd* aumenta rápidamente durante esta fase.

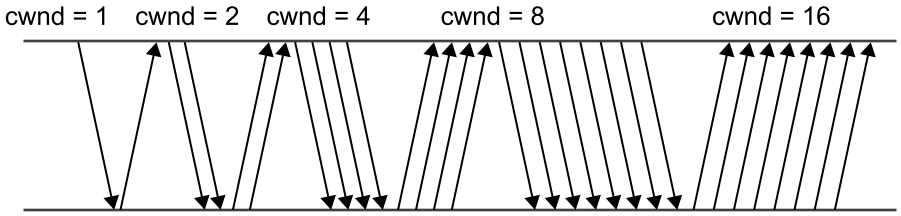
```
Inicialización
   cwnd = MSS;
   ssthresh = 216 = 65535 bytes;
```

```
Cuando se recibe un ack:
   if (cwnd < ssthresh) /* Estamos en Slow start (SS)*/
      cwnd = cwnd + MSS; //Crece exponencialmente
   else      /* Estamos en Congestion avoidance (CA)*/
      cwnd = cwnd + MSS/cwnd; //Crece linealmente

Cuando se produce una pérdida debida a time-out o a la recepción de 3 ACKS duplicados

Retransmite el segmento snd_una;
cwnd = MSS; //Sólo en caso de que la congestión se deba
      //al salto del temporizador.
ssthresh = max(2, min(snd_awnd, snd_cwnd) / 2)
      //threshold pasa a valer la mitad de la ventana de transmisión pero no
      //por debajo de 2 segmentos.
```

6.7.3. Funcionamiento de *slow start*

TCP inicia la ventana de congestión con el valor MSS. Inicialmente sólo puede enviar un segmento inicial sin confirmar y quedarse a la espera. Mientras no se produzcan pérdidas, TCP incrementa *cwnd* en MSS por cada nuevo *ack*. Cuando llega el *ACK* del primer segmento, *cwnd* aumenta a 2 MSS y se envían 2 segmentos. Cuando llegan los acuses de recibo, la ventana de congestión se incrementa en MSS por cada acuse recibido y acaba valiendo 4 MSS. Al llegar los acuses de recibo de los 4 segmentos enviados, la ventana se incrementa hasta 8 MSS, y así hasta que llega al límite de la ventana de recepción.

Figura 43



6.7.4. *Congestion avoidance*

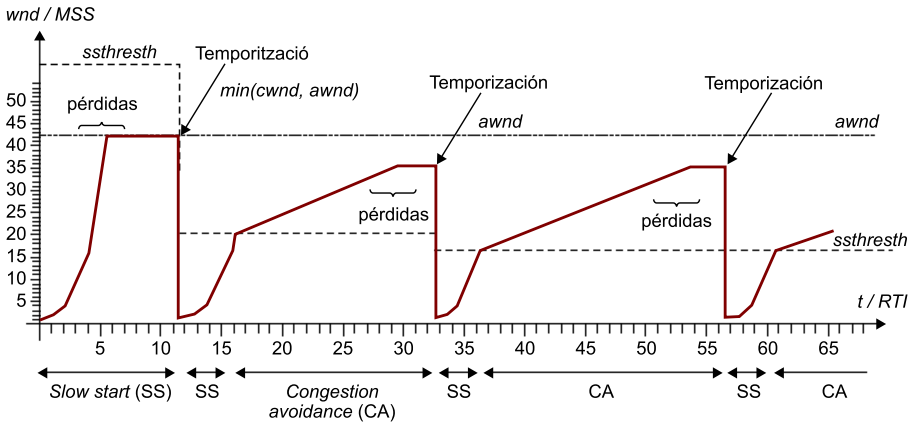
Congestion avoidance evita que se congestione la red, lo que provocaría el cese de la transmisión, que saltara el *timeout* y empezara de nuevo un proceso *slow start*. Por su parte, *congestion avoidance* intenta mantener la ventana de congestión *cwnd* en el límite de la transmisión de datos sin tener pérdidas. Cuando la ventana de congestión supera el umbral *ssthresh*, por una parte, *congestion avoidance* aumenta lentamente el valor de *cwnd* (transmisión de forma lineal) por si la ventana ha quedado por debajo de su valor óptimo y el enlace se está infrautilizando, y, por otra parte, intenta ajustar la variable *ssthresh* a un valor en el que TCP no tenga pérdidas.

6.7.5. Funcionamiento

En un principio, el valor de *ssthresh* es infinito (*slow start*). Cuando se produce alguna pérdida, salta el temporizador, se lleva a cabo la retransmisión y se pone *ssthresh* al valor de la ventana de transmisión que había en el momento de la pérdida dividido por 2: (*min*(*awnd, cwnd*) / *2*).

A continuación, *slow start* incrementa *cwnd* rápidamente, hasta el valor de *ssthresh*. Cuando *cwnd* supere *ssthresh*, TCP entra en la fase de *congestion avoidance*: la ventana aumenta poco a poco, aproximadamente 1 MSS cada vez que se reciben las confirmaciones de toda una ventana, y la ventana pasa a valer: *cwnd = cwnd + MSS/cwnd*.

Figura 44



Funcionamiento del TCP

Debemos señalar que si no hay un enlace congestionado (como sucede normalmente cuando el cliente y el servidor están dentro de una misma LAN), TCP está siempre en *slow start*. En este caso, la ventana aumenta hasta la ventana advertida (*awnd*) y, a partir de ese momento, actúa sólo el control de flujo: la ventana de TCP es siempre igual a la advertida.

6.8. Temporizadores en TCP. Cálculo del temporizador de retransmisión (*RTO*)

El protocolo TCP utiliza varios temporizadores, entre los que destaca el de retransmisión *RTO*[31]. El temporizador de retransmisión *RTO* determina cuándo se debe retransmitir un segmento ante la ausencia de reconocimiento de éste. Mientras hay datos pendientes de confirmación permanece activado. Si el temporizador expira antes de la llegada de una confirmación al emisor, éste retransmite el segmento sin confirmar.

Cada vez que llega una confirmación de nuevos datos:

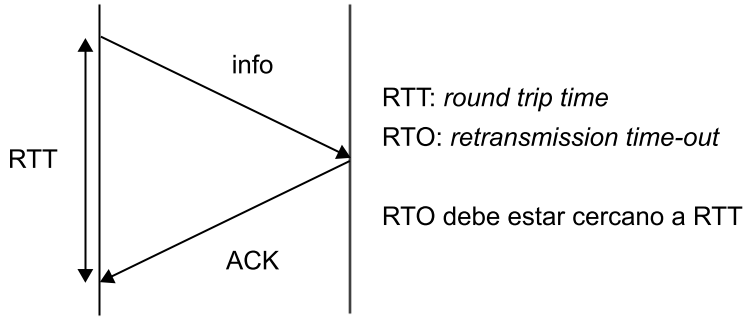
1) TCP calcula el valor del *RTO*.

2) Si hay más datos pendientes de confirmación, el temporizador se actualiza con el valor calculado. En caso contrario, el temporizador se desactiva.

El problema principal consiste en fijar el *RTO*.

(31) *RTO* es la sigla de *retransmission timeout*.

Figura 45



Si *RTO* < *RTT*, se envía un paquete duplicado inútilmente a la red, ya que el sistema cree que hay una pérdida, cuando en realidad sólo se ha producido un retraso.

Si *RTO* > *RTT*, puede llegar a esperarse más tiempo del que realmente es necesario. Como *RTT* varía en cada segmento, *RTO* no se puede calcular de una sola manera; para cada segmento se necesita realizar el cálculo de su temporizador.

6.8.1. Cálculo del temporizador *RTO* en las primeras implementaciones del protocolo TCP

Para calcular el valor del temporizador, en primer lugar se debe valorar el retraso medio de ida y vuelta del enlace, es decir, *MRTT*[32] (valor medio del *RTT*). Para lograr esta métrica se utilizaba una estimación ponderada (denominada estimación exponencial), que asigna un mayor peso al cómputo a las últimas estimaciones de *RTT*.

(32) *MRTT* es la valoración de la media del tiempo de ciclo.

Estimación exponencial

- Valor medio *RTT*:

$$MRTT(k) = \alpha \cdot MRTT(k-1) + (1-\alpha) \cdot RTT(k)$$

- Temporizador de retransmisión:

$$RTO = \beta \cdot MRTT$$

Donde $0 \leq \alpha \leq 1$:

- Si α está próximo a 1, el pronóstico se vuelve inalterable a los cambios.
- Si α está próximo a 0, se responde con rapidez a los cambios.

β = 2 es el valor que se toma como referencia (el doble del valor de la media). Si β = 1, se producían muchas retransmisiones por cualquier retraso.

6.8.2. Algoritmo de van Jacobsen para el temporizador TCP

En canales con alta fluctuación del valor instantáneo de *RTT*, el factor 2 puede ser insuficiente para protegerse de retransmisiones innecesarias, mientras que para canales muy estables este factor puede ser excesivo. Por ello, Jacobsen propuso un algoritmo basado en el cómputo de la desviación media del valor

(33) *DRTT* es la desviación media de la media del tiempo de ciclo.

instantáneo del *RTT* (*DRTT*[33]) con respecto a su media (*MRTT*). Como en el caso del cómputo de la media del *RTT*, para ponderar en mayor medida las estimaciones más recientes de la desviación se utiliza un algoritmo de tipo exponencial:

Algoritmo de van Jacobsen

- Valor medio de *RTT*:

$$MRTT(k) = \alpha \cdot MRTT(k-1) + (1-\alpha) \cdot RTT(k)$$

- Desviación media de *RTT*:

$$DRTT(k) = \gamma \cdot DRTT(k-1) + (1-\gamma) \cdot |RTT(k) - MRTT(k)|$$

- Temporizador de retransmisión:

$$RTO(k) = MRTT(k) + \beta \cdot DRTT(k) \; (\alpha = 7/8, \beta = 4 \text{ y } \gamma = 3/4)$$

Es decir, cuanto mayor sea la variabilidad de *RTT*, mayor será la variancia, y, por lo tanto, la diferencia entre el valor de *RTO* y la estimación de *RTT* (para evitar que *RTO* salte prematuramente y se hagan retransmisiones innecesarias).

6.8.3. Medidas de los temporizadores en TCP

RTO se calcula en términos de *slow-timer tics* (cada 500 o 200 ms).

RTT se mide de diferentes maneras:

- En función de los tics (aproximación dura).
- Usando sellos de tiempo[34]. Esta opción consiste en enviar un sello de tiempo a la cabecera de los segmentos (el valor del reloj del ordenador principal cuando TCP envía el segmento) y en añadir un *echo* del sello de tiempo del segmento que confirman. De este modo, cuando llega el *ack*, la diferencia entre el reloj del ordenador principal y el valor que lleva el *echo* del sello de tiempo es una medida precisa de *RTT*.

(34) En inglés, *timestamps*.

6.8.4. Algoritmo de Karn

En caso de fuerte congestión de la red, momento en el que se producen pérdidas de paquetes o valores de *RTT* extremadamente elevados, no es conveniente mantener el valor del temporizador de retransmisión. La temporización se debería incrementar para garantizar que los paquetes se retransmiten cuando la situación de congestión ha desaparecido.

Por esta razón, en situaciones de retransmisión se utiliza el denominado algoritmo de Karn, basado en el algoritmo de crecimiento exponencial de *RTO*[35]. En este caso de retransmisión de un segmento, su funcionamiento es el siguiente:

(35)En inglés, *exponential RTO backoff*.

1) No se debe utilizar el algoritmo de Jacobsen para estimar el nuevo valor del *RTO*. Tampoco se ha de actualizar el *RTT* en los segmentos retransmitidos. Sólo se mide *RTT* para los segmentos que se transmiten por primera vez.

2) Cada vez que se produce un proceso de retransmisión de un segmento, el valor de temporizador *RTO* se incrementa exponencialmente (TCP se espera el doble del tiempo esperado en la transmisión anterior para evitar que la red se pueda volver inestable):

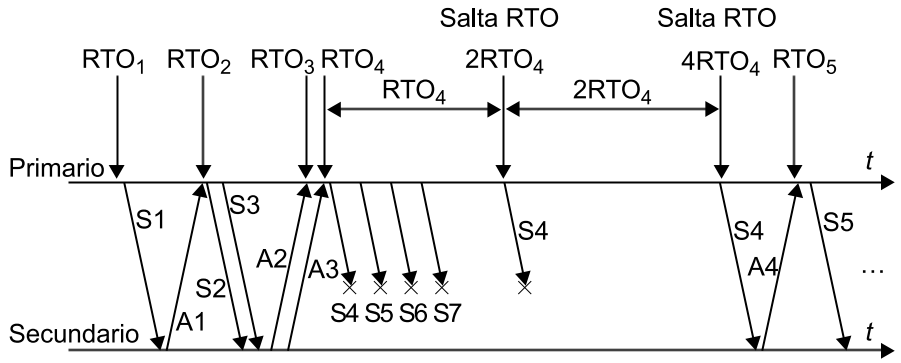
$$RTO(k+1) = q\ RTO(k), \text{ donde habitualmente } q = 2$$

3) Este cómputo de *RTO* se mantiene hasta que se recibe un segmento de reconocimiento correspondiente a un paquete no retransmitido (es decir, se vuelven a confirmar nuevos datos). A partir de ese momento, se emplea de nuevo el algoritmo de Jacobsen.

Utilización del algoritmo de Karn o Jacobsen

```
Si retransmisión:

RTO(k+1)=2·RTO(k) (Algoritmo de Karn-Crecimiento exponencial del RTO)

Si no:

      RTO(k) = MRTT(k) + 4 · DRTT(k) (Algoritmo de Jacobsen)
```

Figura 46



En la figura 46, el segmento S4 se retransmite dos veces. *RTO*1, *RTO*2... Son los instantes en los que se actualiza el temporizador, ya sea porque se confirman nuevos datos, ya sea porque se produce una retransmisión. En caso de retransmisión, *RTO* se fija a dos veces el valor que tenía anteriormente.

6.8.5. Problema de los temporizadores de retransmisión de TCP

Como hemos visto, la problemática de los temporizadores TCP no es trivial. El problema principal es que la determinación de *RTO* es imprecisa, ya que se basa en el valor *RTT* del último segmento de reconocimiento recibido. *RTT* depende del estado de la red, del tipo de reconocimiento efectuado por el protocolo TCP y de los retrasos en el terminal receptor. Si un segmento se pierde en transmisión, el emisor continuará enviando segmentos hasta que su ventana de transmisión se agote. El receptor recibirá estos segmentos, pero, como hay un segmento previo no recibido, no enviará ningún tipo de reconocimiento, esperando que llegue el segmento perdido. La retransmisión del segmento perdido se realizará cuando expire el temporizador *RTO*, lo que provoca que este proceso pueda ser extremadamente lento.

6.9. Algoritmos *fast retransmit/fast recovery*

El problema de los temporizadores ha sido dirigido con una serie de mecanismos que permiten la recuperación rápida de la congestión de la red.

6.9.1. *Fast retransmit*

El algoritmo *fast retransmit*[36] agiliza el procedimiento de retransmisión basado en el temporizador *RTO*. Cuando el receptor recibe un segmento de información fuera de secuencia, inmediatamente envía una confirmación del segmento que espera recibir. Estos *acks* se denominan *acks duplicados*. Cuando el emisor recibe confirmaciones duplicadas, desconoce si ello se debe a un retraso del segmento perdido en la red (el segmento llegará y por lo tanto no es necesario retransmitirlo), o si es que el segmento se ha perdido realmente (es necesario retransmitirlo lo más rápidamente posible).

(36)En castellano, retransmisión rápida.

Fast retransmit propone esperar 3 *acks* duplicados seguidos (es decir, 4 *acks* con el mismo número de secuencia) para estar razonablemente seguros de que el segmento se ha perdido y retransmitirlo enseguida, sin esperar a que salte el temporizador. A partir de ese momento:

- retransmite *snd_una*; //Retransmite el primer segmento no confirmado.
- *ssthresh* = *max*(*min*(*awnd*, *cwnd*) / 2,2 MSS); Calcula el *ssthesh* como la mitad de la ventana actual (igual que cuando salta el temporizador).
- *cwnd* = *ssthresh* + *3MSS*;

y se pasa al modo *fast recovery*.

6.9.2. *Fast recovery*

Una vez se ha recuperado un segmento perdido y se envía la confirmación con el número de secuencia del último segmento recibido correctamente, TCP intenta devolver *congestion avoidance*, sin necesidad de pasar otra vez por *slow start*. De ello se encarga *fast recovery*.

Hasta que no se reciba un nuevo *ack* no duplicado, el emisor aumenta *cwnd* en un MSS por cada *ack* duplicado recibido, para poder enviar un nuevo segmento (si le deja la ventana):

$$cwnd = cwnd + MSS$$

Cuando se confirma el segmento retransmitido, es decir, cuando se recibe un *ack* nuevo no duplicado, el emisor sale de *fast recovery* y pone:

$$cwnd = ssthresh$$

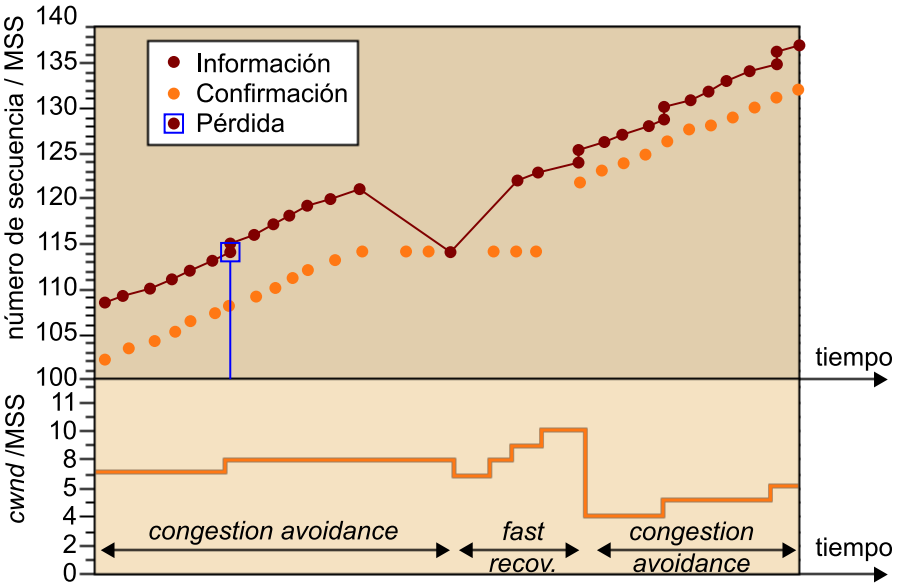
A partir de ese momento, el emisor entra en fase *congestion avoidance*.

Pseudocódigo explicativo de los algoritmos *fast retransmit/fast recovery*

```
Cada vez que se recibe un ack
   Si (se recibe un ack duplicado) {
      Si (es el 3.er ack duplicado) {//Fast retransmit
         retransmite snd_una; //Retransmite el primer segmento no confirmado
         ssthresh = max(min(awnd, cwnd) / 2, 2 MSS);
         // Calcula el ssthesh como la mitad de la ventana actual (igual que cuando
         // salta el timeout).
         cwnd = ssthresh + 3 MSS;
         // La ventana que se supone estaba antes de la pérdida más los 3 segmentos
         // que han generado los acks duplicados.
         fast_recovery = CIERTO;
      }
      sino {//Fast recovery
         si(fast_recovery == CIERTO) {
            cwnd += MSS;
            //Por cada ack duplicado se incrementa la ventana en un MSS, para
            //poder enviar un nuevo segmento (si lo permite la ventana).
         }
      }
} en caso contrario {//se recibe un ack nuevo (no duplicado)
   si(fast_recovery == CIERTO) {
      cwnd = ssthesh;
      fast_recovery = FALSO;
      //Cuando se confirman nuevos datos sale de fast recovery y se pone cwnd = ssthesh,
        para iniciar la fase de congestion avoidance.
```

```
    } en caso contrario {
       /* Slow start/Congestion avoidance */
    }
}
```

Figura 47



La parte superior de la figura muestra un rastro capturado en el emisor con la evolución de los números de secuencia y las confirmaciones en un intervalo de tiempo en el que se produce una pérdida. La parte inferior de la figura muestra la evolución de la *cwnd*

En la figura 47 observamos que cuando el emisor recibe 3 *acks* duplicados entra en *fast retransmit* (*cwnd* valía 8 *MSS*). En ese momento, el emisor retransmite el segmento perdido, calcula *ssthresh* = 4 *MSS*, la mitad de la ventana de congestión:

$$cwnd = ssthresh + 3 = 7 \text{ MSS}$$

A continuación, entra en fase de *fast recovery*. El emisor aumenta *cwnd* en MSS por cada *ack* duplicado. Como al entrar en *fast recovery* aparecen 8 segmentos sin confirmar, el emisor puede enviar nuevos segmentos después de recibir otros dos *ack* duplicados (*cwnd* aumenta incluso a 9 MSS).

Cuando llega un *ack* nuevo no duplicado, el emisor sale de *fast recovery* y pone:

$$cwnd = ssthresh = 4 \text{ MSS}$$

Como en este momento sólo hay 2 segmentos sin confirmar, el emisor puede enviar 2 más. A partir de ese momento, el emisor entra en fase *congestion avoidance*.

6.10. Implementaciones actuales de TCP

Todas las implementaciones actuales de TCP están obligadas a implementar *slow start* y *congestion avoidance*. Para identificar algunas de las diferentes versiones de TCP realizadas a lo largo del tiempo, se han utilizado nombres de ciudades: Tahoe, Hortes, Ren, etc.

- TCP Tahoe (1988): *slow start*, *congestion avoidance*, *fast retransmit*.
- TCP Reno (1990): Tahoe + *fast recovery* + *TCP header prediction*.

En la actualidad las implementaciones mejoran todavía más el comportamiento de TCP con algoritmos adicionales:

- TCP *Sack*: Reno + *selective ACK*.
- Otros: *multicasting*, *routing tables*.

7. Otros protocolos de transporte

Aunque el módulo se ha centrado en la presentación de TCP y UDP, que son los protocolos de transporte por excelencia, debemos señalar que existen otros diseñados para soportar mejor ciertos tipos de comunicaciones o que han mejorado algunos de los puntos débiles de TCP y UDP:

- SCTP[37] es un protocolo de transporte diseñado como alternativa a TCP y, además de ofrecer fiabilidad, control de flujo y secuenciación como TCP, permite el envío de mensajes fuera de orden. Y, en consecuencia, funciona como un protocolo no orientado a la conexión como UDP. Asimismo, permite el paralelismo de envío de mensajes.

(37)SCTP es la sigla de *stream control transmission protocol.*

- DCCP[38] es un protocolo de transporte no orientado a la conexión basado como SCTP en el envío de mensajes. DCCP es usado por aplicaciones que tienen necesidad de entrega rápida de datos. Esta categoría de aplicaciones incluye, por ejemplo, la telefonía en Internet y multimedia en tiempo real. DCCP está pensado para aplicaciones que requieren el control de flujo de TCP, pero no necesitan la entrega en orden ni la confiabilidad que ofrece TCP, ni demandan un control de congestión dinámico diferente del de TCP. El mismo modo, DCCP está definido para aplicaciones que no requieren las características especiales de SCTP, como la entrega secuencial de flujo múltiple.

(38)DCCP es la sigla de *datagrama congestion control protocol.*

Resumen

El objetivo principal de este módulo didáctico ha sido presentar en detalle los servicios, los protocolos y las funciones del nivel de transporte de una red. En el módulo se han introducido los protocolos que se utilizan en Internet. Estos protocolos son UDP y TCP.

Se ha comprobado que el objetivo principal del nivel de transporte es entregar la información a los niveles orientados a la aplicación en los extremos de la red. El módulo ha presentado los protocolos ARQ, protocolos usados por el control de errores y que aseguran la fiabilidad de la red. Estos protocolos han sido descritos de manera genérica, sin entrar en la especificidad de los protocolos en Internet. Sin embargo, más adelante se ha puesto el énfasis en los protocolos que implementa Internet. Los principales protocolos de transporte de esta red son:

1) El UDP, que es un protocolo no orientado a la conexión. Por lo tanto, no efectúa ningún control de errores ni de flujo. Si un datagrama UDP llega equivocado (el UDP utiliza un código detector de errores), el UDP lo descarta y no lo entrega a la aplicación. Ésta deberá ser capaz de responder a este tipo de servicio o habrá de asumir la pérdida de la información. Este tipo de servicio puede ser útil en aplicaciones en tiempo real, cuando es preferible que la información llegue en el momento correspondiente –esto es, con un retraso delimitado– a que se pierda una parte de ésta.

2) El TCP, que es un protocolo orientado a la conexión. Habrá una fase de establecimiento de la conexión (el denominado procedimiento *three-way handshake*), una fase de transmisión de la información y una fase de terminación de la conexión. El TCP entregará la información a la aplicación totalmente libre de errores. Para conseguirlo, necesita efectuar un control de errores y de flujo. El TCP utiliza un código detector de errores, junto con un protocolo de retransmisiones, que permite recuperar la información errónea. Como las memorias intermedias de recepción se pueden desbordar, el TCP utiliza un control de flujo por ventana deslizante. Se han presentado las implementaciones específicas de estos protocolos y se han relacionado con los protocolos ARQ en los que están basadas. El TCP debe dimensionar correctamente los temporizadores de retransmisión. Hay diferentes algoritmos, entre los que destaca el de Jacobson, basado en una estimación de RTT (tiempo de ida y vuelta de la información entre los extremos TCP) y el cálculo de la medida y la variancia de RTT. También hemos estudiado los algoritmos que TCP utiliza para aliviar la congestión en la red: *slow start* y *congestion avoidance*.

Actividades

1. Enviamos 8.192 bytes mediante cuatro encaminadores (asumimos que los encaminadores son del tipo *store and forward*, y que cada uno está conectado a una línea telefónica T1 a 1.544.000 bps). Y consideramos que el tiempo de propagación entre el primer y el último encaminador es una constante *K*.

a) ¿Cuál es el tiempo máximo para transmitir la información si la longitud del segmento de datos es de 4.096 bytes?

b) ¿Cuál es el tiempo máximo para transmitir la información si la longitud del segmento de datos es de 512 bytes?

2. Cuando se establece una conexión entre las estaciones argos y helios, tiene lugar el proceso siguiente:

```
15:56:54.796091 argos.1023 > helios.login S 3541904332: 3641904331 (0)win
31744 <mss 1460>
15:57:00.796591 argos.1023 > helios.login S 3541904332: 3641904331 (0)
win 31744 <mss 1460>
15:57:13.797035 argos.1023 > helios.login S 3541904332: 3641904331 (0)win
31744 <mss 1460>
15:57:13.797035 helios.login > argos.1023 S 548133143: 548133143 (0)ack
34519043333 win 8760 <mss 1460>
15:56:54.797035 argos.1023 > helios.login .ack 548133144 win 31744
```

El primer temporizador de argos se inicia a los 6 segundos. Determinad cuánto valen el segundo y el tercer temporizador de argos.

3. Un TCP transmisor ha advertido, durante el establecimiento de una conexión, un MSS de 512 bytes, y durante la transferencia de la información, una ventana de 512 bytes. Un TCP receptor advierte ventanas de 2.048 bytes. Dibujad un diagrama de tiempo en el que se perciba el algoritmo *slow start*.

Bibliografía

Kurose, James F.; Ross, Keith W. (2005). *Computer networking: a top-down approach featuring the Internet*. Addison-Wesley.

Tanenbaum, Andrew S. (2003). *Redes de computadores* (4.ª ed.). Pearson.