REDES: Tema 3. Capa de transporte

(Grado en Ingeniería Informática)

Francisco Argüello Pedreira y José Carlos Cabaleiro Domínguez
Departamento de Electrónica y Computación
Universidad de Santiago de Compostela

18 de octubre de 2011

Índice general

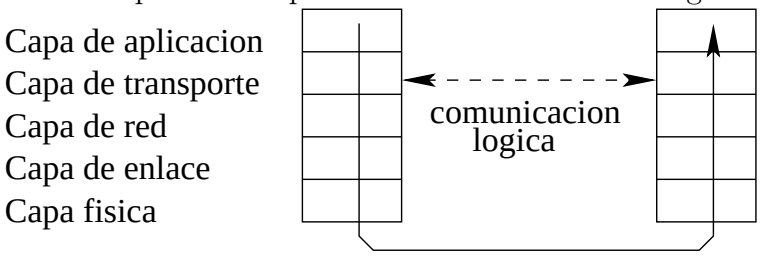
Capítulo 3

Capa de transporte

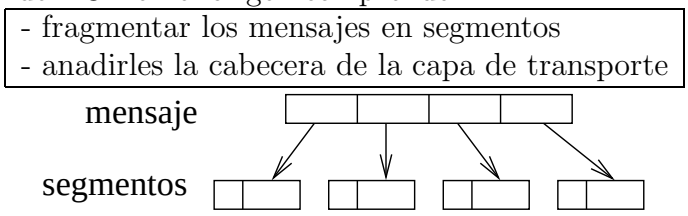
3.1. Introducción

La función de la capa de transporte se resume en los siguientes puntos:

- La capa de transporte en el origen coge los mensajes que originan las aplicaciones y los prepara para transmitir por un canal no fiable. En el destino recupera los mensajes y los entrega al proceso destino.
- Gracias al modelo de capas, la capa de transporte no se preocupa de cómo es el canal no fiable, sólo sabe que produce muchos errores. La comunicación que realiza la capa de transporte es una comunicación lógica.



- La capa de transporte sólo está implementada en origen y destino, no en los routers intermedios (en ocasiones el origen o destino puede ser un router).
- En TCP/IP, los protocolos de la capa de transporte son TCP y UDP. La operación de TCP en el origen comprende:



UDP, por otra parte, no segmenta los mensajes, sólo les añade una cabecera.

Nota sobre la nomenclatura: las unidades de datos transmitidas en cada una de las capas tienen la siguiente denominación:

Capa	Unidades de datos
aplicación	mensajes
transporte	segmentos
red	paquetes o datagramas
enlace	frames, tramas o marcos

3.1.1. Multiplexión y demultiplexión

Veamos como es la interacción entre las aplicaciones y la capa de transporte:

- Los mensajes de la capa de aplicación pasan a la capa de transporte a través de un socket.

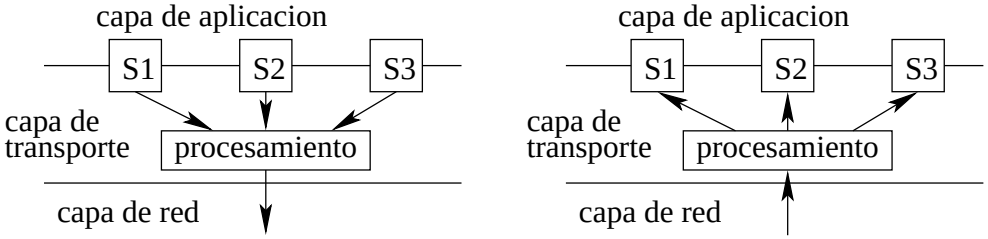
  El proceso escribe o lee los mensajes del socket como si fuese en un fichero y se despreocupa de todo lo demás.

  La capa de transporte recoge los mensajes del socket origen y los traslada al socket destino. Allí los procesos podrán recogerlos cuando quieran.

  Es decir, que la capa de transporte no se entiende directamente con los procesos sino que recoge y coloca los mensajes en los sockets.

- *Multiplexión.* Como en el origen puede haber mucho procesos transmitiendo, lo cual implica muchos sockets abiertos, la capa de transporte ha de recorrer uno a uno cada socket, dedicarles un cierto tiempo de proceso a cada uno y pasar los segmentos generados a la capa de red. Esto se denomina multiplexión.

- *Demultiplexión.* En el destino, la capa de transporte recoge los segmentos que le llegan de la capa de red, reconstruye los mensajes y los reparte entre los distintos sockets destino.

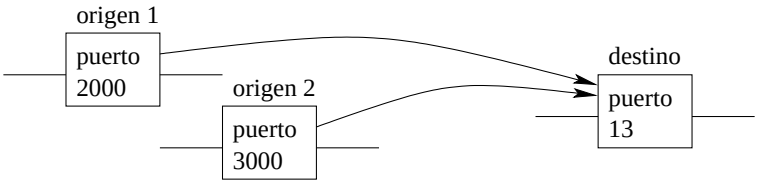


- La identificación del socket al que va el segmento se realiza a través de los denominados números de puerto que se colocan en la cabecera.

Los números de puerto son enteros de 16 bits (de 0 a $2^{16} - 1$). De 0-1023 son utilizados por el administrador para los servicios bien conocidos, y están estandarizados y son conocidos por todos. En el caso de Linux están almacenados en el fichero */etc/services*. Por encima de 1024 son libres para su utilización por parte de las aplicaciones de usuario.

Multiplexión con sockets sin conexión (en UDP)

Estos sockets se identifican solo por la pareja dirección IP de destino y puerto de destino. Así, dos segmentos procedentes de distintos hosts y de distintos puertos que se entreguen al mismo puerto destino los recoge el mismo proceso. Un ejemplo es el de un servidor de hora del día (*daytime*, puerto 13) para todos los ordenadores de una red. Cuando un host manda un datagrama a este puerto, el servidor le devuelve otro datagrama cuyo contenido es el día y la hora.

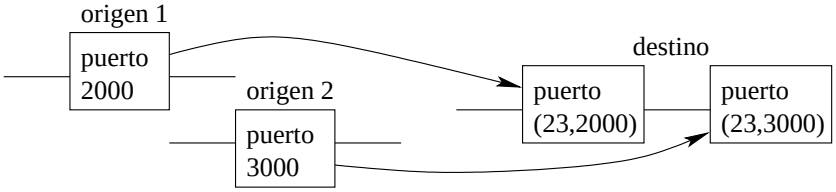


El puerto origen se usa exclusivamente para que el proceso sepa a quién tiene que responder. En el mismo socket se reciben transmisiones de puertos y hosts distintos y el proceso tiene que responder individualmente a cada uno.

Multiplexión con sockets orientados a conexión (en TCP)

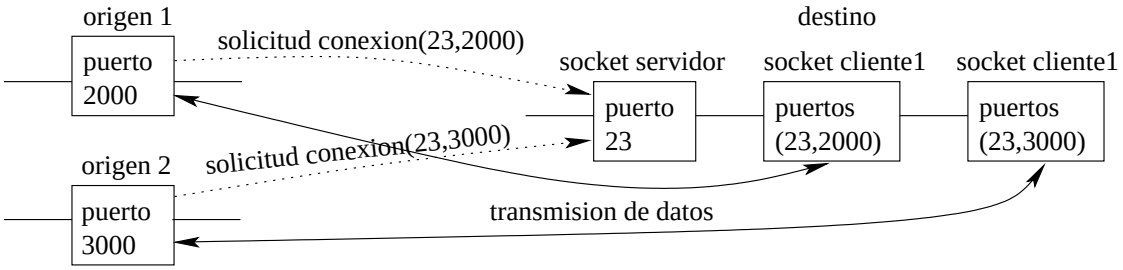
Estos sockets se identifican por una tupla de 4 elementos: direcciones IP de origen y destino y puertos de origen y destino. En este caso dos segmentos con direcciones IP origen distintas serán dirigidas a sockets distintos aunque vayan al mismo host y al mismo puerto. Y lo mismo pasa con segmentos de puertos origen distintos. Esto permite que varias conexiones con destino el mismo puerto puedan ser atendidas por procesos diferentes. Esto ocurre con las conexiones telnet, ftp, http, etc.

Cuando se invoca un comando telnet, la conexión se dirige al puerto destino 23, pero también se elige aleatoriamente un puerto origen, lo que permite distinguir varias conexiones telnet.

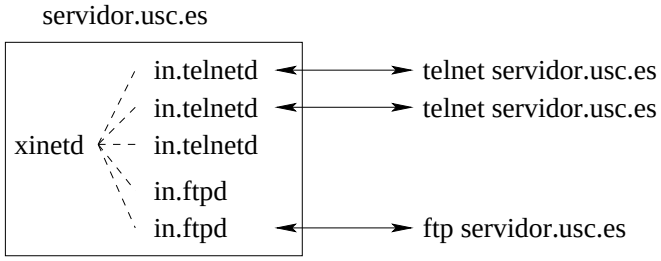


En las conexiones TCP se utilizan dos tipos de sockets: los *sockets de servidor* y los *sockets de conexión*. El socket de servidor lo único que hace es estar a la espera de conexiones de clientes. Una vez hecha la conexión, las transfiere a un

socket de conexión que se encarga realmente de la transmisión. Hay un solo socket de servidor y múltiples sockets de conexión, uno para cada conexión.



Por ejemplo, el proceso del sistema (daemon) encargado de gestionar muchas de las aplicaciones de red (telnet, ftp, echo, finger, etc), se llama `xinetd` (extended internet daemon). Es un proceso del sistema operativo que se encarga de controlar a otros procesos subordinados. Por ejemplo, para cada conexión telnet recibida por el ordenador, xinetd crea un proceso `in.telnetd` para atenderla, mientras que por cada conexión ftp recibida, xinetd crea un proceso `in.ftpd`. En concreto, si hay 3 usuarios conectados por telnet a un ordenador y otros dos por ftp, entonces en el ordenador habrá ejecutándose tres procesos `in.telnetd` y dos `in.ftpd`.

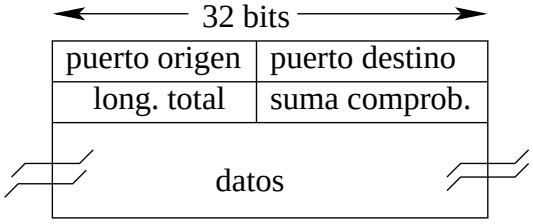


Algo similar ocurre en HTTP con conexiones persistentes, que suele generar un nuevo proceso o hilo para atender a cada conexión persistente.

3.1.2. Capa de transporte sin conexión: UDP

UDP significa *protocolo de datagrama de usuario*. Es un protocolo de transporte sencillo cuyo modo de operación es el siguiente:

- En el origen, UDP hace sólo una cosa: coger el mensaje y añadirle una cabecera para formar un segmento. La cabecera es muy sencilla pues contiene sólo 4 campos: puertos origen y destino, longitud total del segmento (cabecera + datos medida en bytes) y una suma de comprobación.



- En el destino, UDP hace sólo una cosa: comprobar si el paquete llegó sin errores. Si llegó bien se pasa al socket. Si llegó mal normalmente se descarta, aunque hay implementaciones UDP en las que un segmento dañado también se pasa al socket pero adjuntando un aviso de precaución para la aplicación.

- La suma de comprobación se utiliza para detectar errores. Es la suma de las palabras de 16 bits de todas las palabras del segmento, incluida la cabecera UDP (más algunos campos de la cabecera IP), seguida de un complemento a 1 y de la selección de los 16 bits menos significativos del resultado. Se calcula primero en origen y luego en destino. Si coinciden ambas es que el paquete llegó sin errores.

  Veamos un ejemplo pero por simplicidad usaremos palabras de 4 bits:

  origen:

0000	0001
0011	
1000	1001

  segmento:

0000	0001
0011	1001
1000	1001

  En el origen se hacen las siguientes operaciones y se inserta el resultado en el campo suma de comprobación del segmento:

  1) se suman todas las palabras: $0000 + 0001 + 0011 + 1000 + 1001 = 10101$,
  2) se toman 4 bits del resultado y se le suman los acarreos: $1 + 0101 = 0110$,
  3) se hace el complemento a 1 del resultado: $\overline{0110} = 1001$.

  En el destino se vuelve a realizar la misma operación y si el resultado coincide con el campo suma de comprobación es que el segmento se recibió bien.

Características:

- *Sin conexión.* No hay acuerdo previo entre emisor y receptor. Sólo se envían los datos. No hay retardo de establecimiento de conexión. Tampoco hay retransmisiones en caso de segmentos con errores.

- *Sin estado.* Cada segmento se envía con independencia de los demás. Por ejemplo no se numeran los segmentos, incluso si un emisor envía varios segmentos a los vez, estos pueden llegar en desorden al destino y la capa de transporte no tiene forma de ordenarlos. Los mensajes no se dividen en segmentos porque no hay información para reconstruirlos. Si el mensaje es demasiado grande para meterlo en un segmento se produce un error.

  El procesamiento de los segmentos es más rápido y requiere menos recursos, por lo que los servidores pueden atender a más clientes.

- *Cabeceras más pequeñas.* Lo que implica menor espacio de almacenamiento y lectura/escritura más rápida.

- *Más control de la aplicación.* La aplicación siempre puede añadir su propia cabecera con información de control (como vimos en el correspondiente

tema). Lo que no proporciona UDP, como por ejemplo numerar los datos, lo podría hacer sin problemas y a su gusto la aplicación. Y lo que se estima que no es necesario, como por ejemplo el control de congestión, simplemente no se incluye.

- *Usos*. Aquellas aplicaciones en las que es preferible velocidad frente fiabilidad. Por ejemplo, la transmisión de audio y vídeo digitalizado, los mensajes del DNS y algunas herramientas de gestión y administración de una red.

3.2. Fundamentos de la transmisión fiable

La transmisión fiable se basa en los protocolos de retransmisión, es decir, los que retransmiten los paquetes cuando hay errores. Estos protocolos se denominan *ARQ, Automatic Repeat reQuest* (solicitud automática de repetición). Vamos a ver dos protocolos ARQ:

- *Parar y esperar, stop-and-wait*. El emisor envía un paquete y mientras espera a recibir el reconocimiento del receptor permanece parado.
- *Ventana deslizante (entubamiento)*. Se pueden enviar varios paquetes antes de recibir los reconocimientos. Hay varias versiones de este protocolo entre las que destacan *Retroceder N* y *Repetición selectiva*.

Consideramos enlaces bidirecionales (o *full dúplex*), es decir, que permiten la transmisión en los dos sentidos y a la vez ($A \rightarrow B$ y $A \leftarrow B$).

Numeraremos los paquetes, por ejemplo, 0-1-2-3-4, etc. Si se usa un número limitado de bits para numerar lo paquetes, al llegar al máximo se recomienza con el 0. Por ejemplo, si usamos 8 bits para los números, recorremos de 0 a 255 y volvemos a 0. Podemos incluso usar sólo 1 bit para numerar los paquetes: 0-1-0-1-0, etc (numeración del bit alternante).

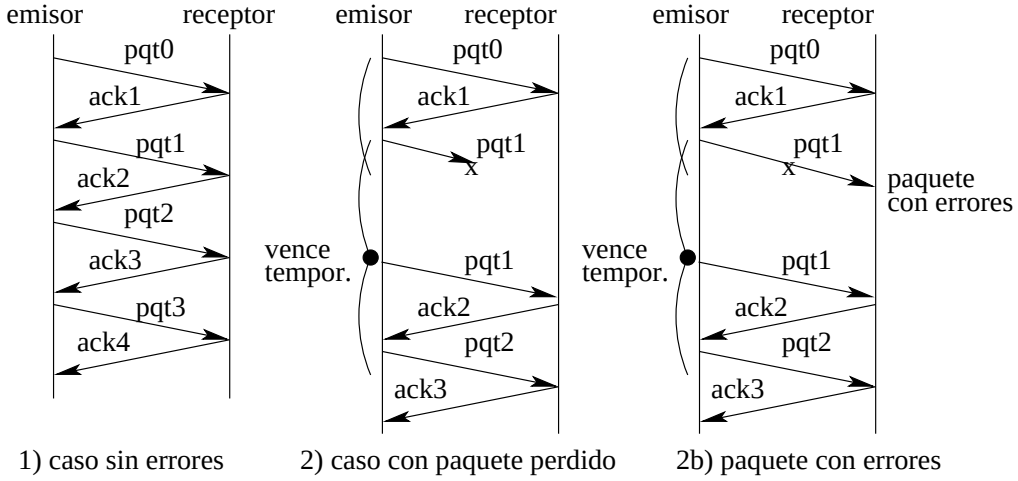
3.2.1. Protocolo ARQ parar y esperar

Los distintos casos que pueden darse en un protocolo *ARQ parar y esperar*:

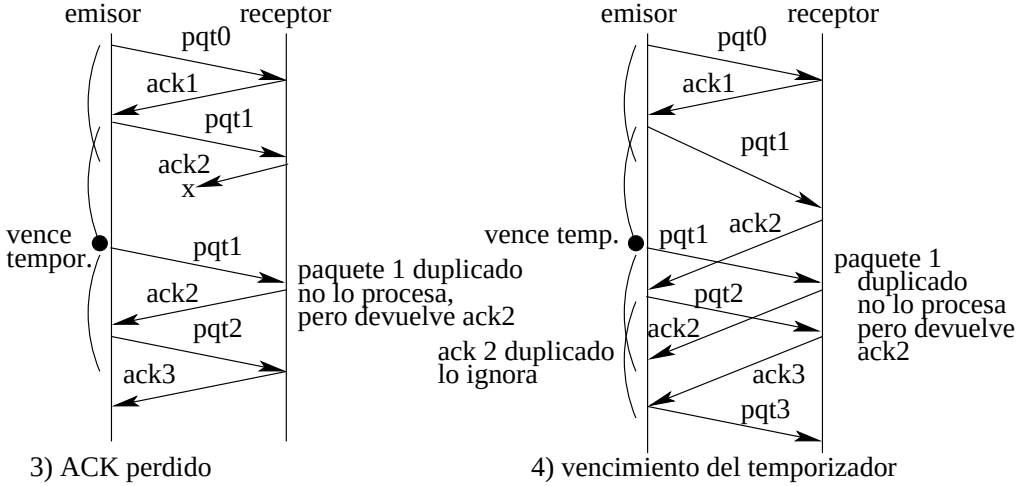
1. *Caso sin errores*. Cuando el receptor recibe correctamente un paquete devuelve un ACK (acknowledge) al emisor. ACK se puede traducir por reconocimiento, asentimiento o confirmación. Por convenio, el número del ACK será el del siguiente paquete a enviar. En el momento en que el emisor obtiene un ACK pasa a enviar el siguiente paquete.
2. *Caso de pérdida de paquetes*. Si se pierde un paquete completamente, el receptor no se enterará de nada y por lo tanto no devolverá un reconocimiento. Para desbloquear esta situación, el emisor, a la vez que envía un

paquete pone un temporizador a contar. Cuando vence el tiempo sin que el emisor obtenga un reconocimiento, retransmite el paquete.

El caso de un paquete recibido con errores se puede considerar igual: el receptor descarta el paquete con errores y no devuelve ningún reconocimiento.



3. *ACK perdido*. El emisor retransmite el paquete en cuanto le venza el temporizador. El receptor recibe un duplicado del paquete pero sabe que está repetido por el número de secuencia del paquete, así que no lo procesa pero devuelve el ACK.

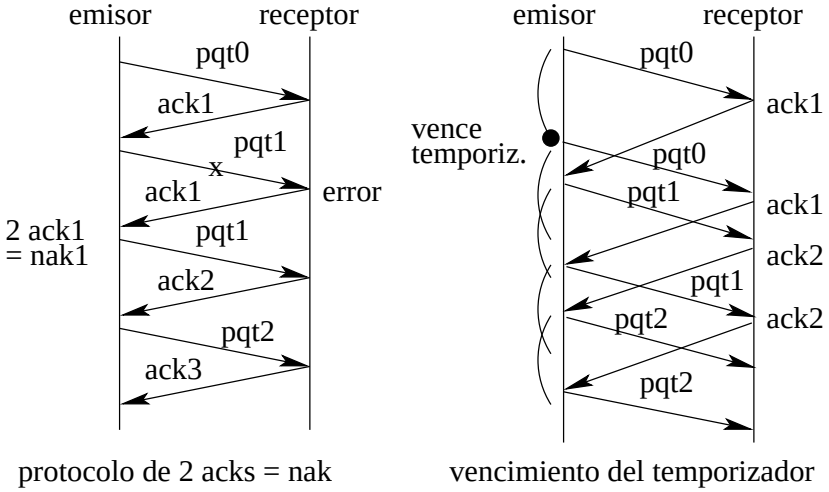


4. *Vencimiento del temporizador*. Puede ocurrir cuando el tiempo del temporizador se ha elegido demasiado corto o cuando un paquete o un ACK llegan con retraso por problemas de congestión de la red. En este habrá tanto paquetes como ACKs duplicados. El paquete duplicado se trata como en el caso anterior.

   En cuanto al ACK duplicado, el emisor sabe que está duplicado por el número de secuencia y simplemente lo ignora.

Nota: hay muchas variantes de protocolos *ARQ parar y esperar*. Nosotros hemos supuesto que existen ACK pero no NAK (reconocimentos negativos). Los reconocimientos negativos se emplean para que el receptor indique al emisor que se ha recibido un paquete con errores. Esto ahorra tiempo pues en este caso el emisor no tiene que esperar a que venza el temporizador para retransmitir el paquete.

Otra posibilidad es considerar que dos ACKs con el mismo número equivalen a un NAK. Para ello hay que considerar que cuando el receptor reciba un paquete con error debe devolver un reconocimento al último paquete recibido correctamente. Además, el emisor deberá considerar que un ACK duplicado equivale a un NAK. Este protocolo tiene un problema y es el vencimiento del temporizador: en este caso el emisor enviará continuamente duplicados de paquetes.

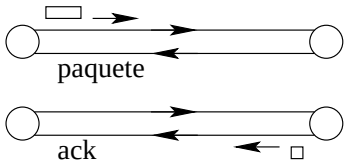


protocolo de 2 acks = nak vencimiento del temporizador

El considerar que 3 ACKs repetidos equivalen a un NAK resuelve algunos problemas, pero no todos. No obstante con alguna variación se emplea en TCP.

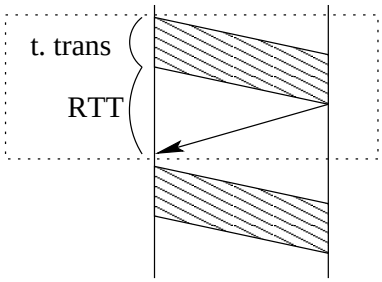
3.2.2. Tiempo desperdiciado en ARQ parar-y-esperar

Los protocolos *ARQ parar y esperar* son simples pero tienen como desventaja el que desperdician muchísimo tiempo, pues mientras se espera la respuesta el enlace puede estar desocupado.



Como detalle adicional vamos a tener en cuenta que el paquete de datos tiene un determinado tamaño, pero despreciaremos el tamaño del ACK. Consideraremos los dos tiempos que hemos venido usando hasta ahora: *tiempo de ida y vuelta* (RTT) y *tiempo de transmisión*. Viendo el dibujo es fácil obtener el tiempo que se requiere para transmitir cada paquete. Basta considerar una de las zonas que

se repiten.



Vamos a calcular la utilización del enlace por parte del emisor. El tiempo útil es aquel en el cual el emisor transmite, y el inútil cuando espera. Se define la utilización como el cociente entre el tiempo útil y el total.

$$U = \frac{t_{trans}}{RTT + t_{trans}} \tag{3.1}$$

El tiempo de transmisión puede calcularse de la siguiente forma. La fórmula de la velocidad, $v = l/t$, también se aplica para a la transmisión de bits, $R = L/t_{trans}$, donde R es la velocidad de transmisión en bps y L la longitud del paquete en bits. De aquí,

$$t_{trans} = \frac{L}{R} \tag{3.2}$$

Ejemplo. Pongamos un ejemplo para ver de que magnitud es el porcentaje de utilización. Supongamos que con *ARQ parar y esperar* transmitimos paquetes de longitud 4000 bits sobre un enlace de velocidad 1 Gbps y de longitud 1000 km y sin routers intermedios. La velocidad de propagación de una señal en el cable es 200000 km/s. Calculemos el porcentaje de utilización del enlace.

l=1000 km
v=200 000 km/s
L=4000 bits
R=1 Gbps

RTT. En este caso depende sólo de la propagación de la señal en el cable (no hay routers intermedios). Tenemos:

$$v = \frac{l}{t};\quad t = \frac{l}{v} = \frac{1000\ km}{200000\ km/sg} = 0,005,\quad RTT = 2t = 0,01\ sg \tag{3.3}$$

Tiempo de transmisión. Depende sólo del tamaño del paquete.

$$R = \frac{L}{t_{trans}};\quad t_{trans} = \frac{L}{R} = \frac{4000\ bits}{10^9\ bits/sg} = 4 \cdot 10^{-6}\ sg \tag{3.4}$$

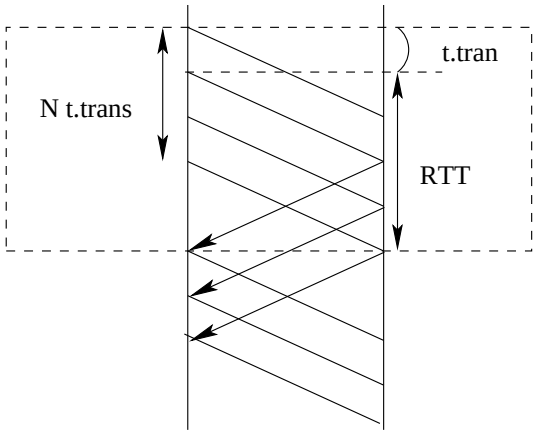
Porcentaje de utilización.

$$U = \frac{t_{trans}}{RTT + t_{trans}} = \frac{4 \cdot 10^{-6}}{0,01 + 4 \cdot 10^{-6}} = \frac{0,004}{10,004} = 0,0004 \tag{3.5}$$

Esta cantidad es en tantos por uno, es decir que el máximo sería 1. Sin embargo, sólo conseguimos una utilización 0,0004. Puesto que la capacidad del enlace es 1 Gbps, resulta una capacidad utilizada de 400 kbps, por lo que el enlace resulta enormemente desaprovechado.

3.2.3. ARQ con ventana deslizante (entubamiento)

La solución al tiempo perdido es *ARQ con ventana deslizante* también denominado *Entubamiento*. Consiste en que el emisor envíe un determinado número de paquetes (sean estos N) antes de recibir los reconocimientos.



Porcentaje de utilización. De la figura se obtiene fácilmente:

tiempo útil en el emisor	$N \cdot t_{trans}$
tiempo total	$t_{trans} + RTT$

No habrá tiempo perdido ($U = 1$) cuando $N \cdot t_{trans} \geq t_{trans} + RTT$. De aquí,

$$N \geq 1 + \frac{RTT}{t_{trans}} \tag{3.6}$$

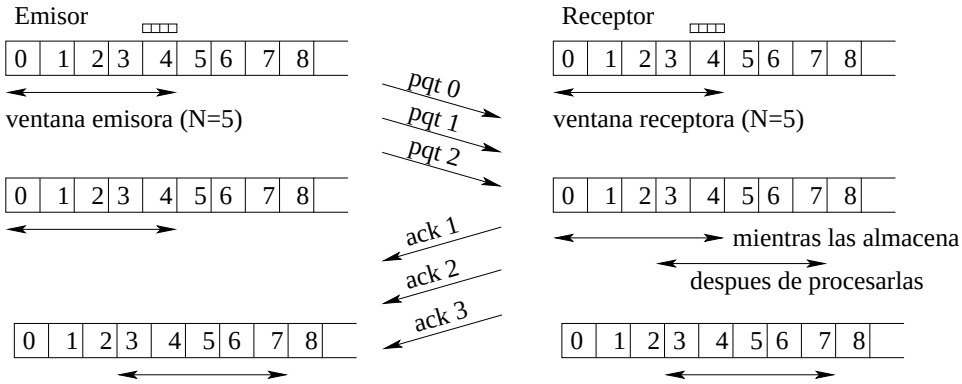
Requerimientos. Para poder usar entubamiento, necesitamos dos cosas:

- El rango de números de secuencia tiene que abarcar por lo menos el doble de paquetes que pueden enviarse de golpe.
- Emisor y receptor tendrán que guardar más de un paquete. Por ejemplo, el emisor ha de mantener en memoria los paquetes que ha enviado y todavía no le han reconocido, porque si al final se pierden tendrá que reenviarlos.

Ventana. Por estas razones se limita a N el número de paquetes que se puede enviar de golpe y a este número se le denomina *tamaño de la ventana*.

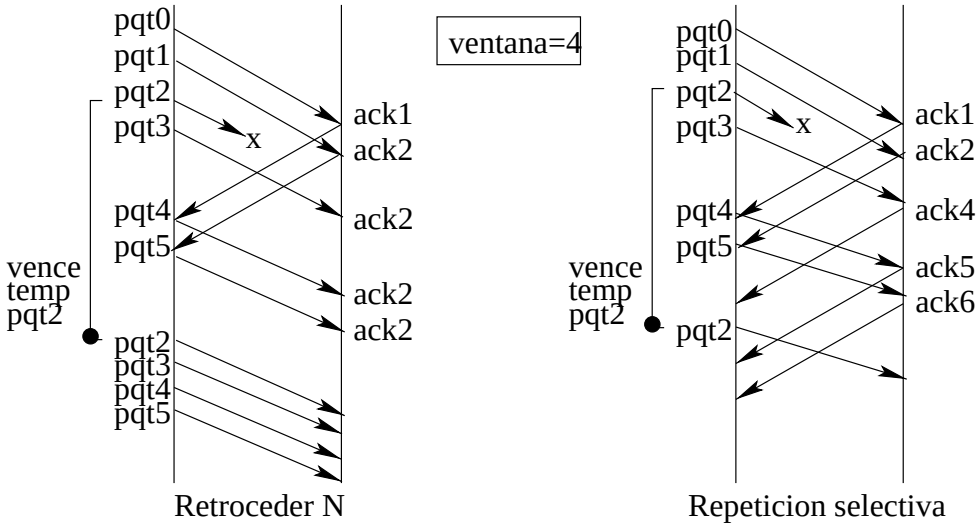
1. La ventana emisora es el conjunto de N paquetes que el emisor puede enviar o que ya se ha enviado pero que todavía el receptor no ha reconocido. El emisor ha de contener copia de estos N paquetes. La ventana se desplaza una posición en cuanto recibe la confirmación de un paquete.

2. La ventana receptora es el conjunto de N paquetes que el receptor puede aceptar o que ya ha aceptado pero que está todavía procesando. El receptor ha de tener memoria para almacenar estos N paquetes. La ventana se desplaza una posición cuando el receptor devuelve el reconocimiento para ese paquete[1].



Temporizadores. Al igual que en los protocolos Parar y Esperar, el emisor pone en marcha un temporizador cada vez que manda un paquete.

Tipos. Hay dos tipos de protocolos de ventana deslizante:



- *Retroceder N* (GBN, *Go Back N*). El receptor sólo acepta paquetes en orden.

  (1) Si un paquete llega al receptor con errores o no llega, el receptor descarta todos los siguientes.

[1]Nota: hay otras formas de definir las ventanas emisora y receptora.

(2) El emisor inicia un temporizador cada vez que manda un paquete. Si al emisor le vence el temporizador que inició con un paquete, entonces reenviará también a partir de aquí los siguientes paquetes.

(3) Aquí un ACK a un paquete implica un ACK a todos los paquetes previos, por lo que se denominan ACKs acumulativos.

- *Repetición selectiva*. El receptor acepta paquetes en desorden.

  (1) Solo se retransmiten los paquetes que llegan con error o no llegan.

  (2) Hay que enviar los ACKs para cada uno de los paquetes recibidos.

3.3. TCP

3.3.1. Transmisión

Una vez que conocemos los principios de la transmisión fiable, veamos como se aplican estos a TCP.

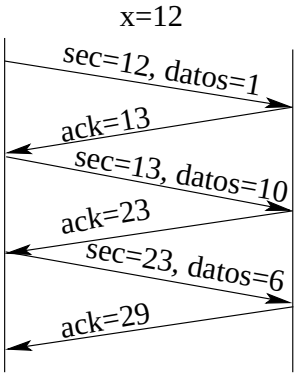
1) Números de secuencia

En TCP, se usan números de 32 bits para numerar los segmentos, pero con dos particularidades:

1. En vez de usar la secuencia 0,1,2,3,4, etc, se empieza por un número aleatorio x, y en vez de incrementar en una unidad, cada segmento se incrementa en el número de bytes enviados ($x + bytes$).

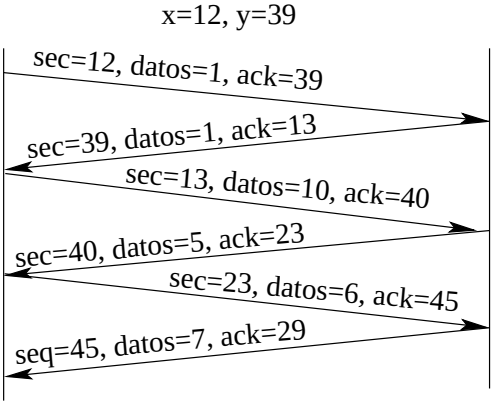
2. Los reconocimientos indican el siguiente byte que se desea recibir (se suman los bytes recibidos al número de secuencia del paquete).

Por ejemplo:



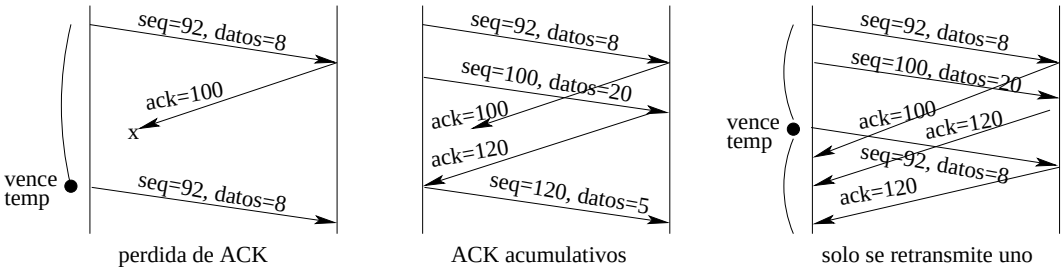
2) Superposición

En los ejemplos anteriores los segmentos de datos se trasmitían en una sola dirección. En la mayoría de las situaciones prácticas se transmiten datos en ambas direcciones. La forma obvia es intercalar segmentos de datos y reconocimientos, sin embargo, es posible una mejora. Cuando llega un segmento de datos, en lugar de enviar inmediatamente un reconocimiento, en algunas implementaciones de TCP, el receptor espera hasta que tenga que enviar datos y los envía juntos.



Esta técnica se denomina *superposición* (piggybacking). Hay que seleccionar un tiempo de espera máximo en el receptor hasta que haya datos. Si se sobrepasa este tiempo el receptor debe envíar el reconocimiento solo.

3) Transferencia fiable de datos



- El emisor utiliza temporizadores para retransmitir los segmentos a los que no se han obtenido ACKs.
- Se usa entubamiento y los ACK son acumulativos, pues implican el reconocimiento de los segmentos previos.
- Como la gestión de los temporizadores consume bastante recursos, se recomienda usar un único temporizador a la vez, aunque se haya enviado un conjunto de segmentos. Cuando llega un ACK, el emisor rearranca el temporizador si existieran segmentos todavía por reconocer.
- En caso de que un ACK no llegue en el tiempo del temporizador, TCP retransmitirá sólo el segmento que no se reconoció, se reinicia el tempori-

zador y se espera a recibir un ACK antes de seguir. Este ACK puede ser el correspondiente al segmento enviado o al último si se han completado un hueco.

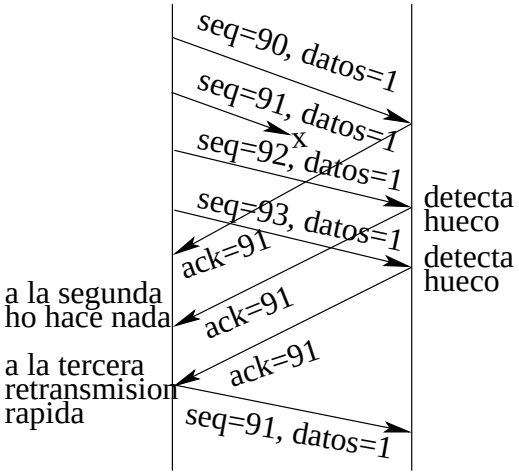
4) Duplicación del tiempo de espera

- Cuando se produce un vencimiento del temporizador, el emisor además de retransmitir el segmento, duplica el tiempo de temporización. Si vuelve a ocurrir otro vencimiento, se vuelve a duplicar el tiempo de temporización.
- En otros casos se estima y actualiza el temporizador según la fórmula:

$$\begin{aligned} \text{Temporizador} &= \text{EstimacionRTT} + 4\text{DevRTT} \\ \text{EstimacionRTT} &= (1-\alpha)\text{EstimacionRTT} + \alpha\text{MuestraRTT} \\ \text{DevRTT} &= (1-\beta)\text{DevRTT} + \beta|\text{MuestraRTT} - \text{EstimacionRTT}| \end{aligned}$$

En donde la estimación del RTT se hace con segmentos transmitidos y reconocidos, pero nunca con segmentos que hayan sido retransmitidos. La operación realizada en la fórmula anterior se denomina *media ponderada de desplazamiento exponencial*. DevRTT es una medida de la variación del RTT. Usualmente $\alpha = 0,125$ y $\beta = 0,25$.

4) Retransmisión rápida



TCP no tiene reconocimientos negativos (NAK) específicos, pero se interpretan los ACK triplicados como un NAK. Veamos cómo se usan:

- Cuando el receptor recibe un segmento con mayor número al esperado, es decir, cuando se forma un agujero, el receptor envía un ACK duplicado. El emisor lo intrepreta sólo como un aviso.

- Si esto vuelve a ocurrir otra vez, se envía otro ACK repetido. El emisor recibe entonces tres ACKs repetidos y en este caso realiza una *retransmisión rápida*, en la cual retransmite el segmento ausente antes de que venza el temporizador.

TCP distingue dos tipos de eventos de pérdida:

- *Vencimiento del temporizador*. Si vence el temporizador, se considera que los problemas son graves, pues se han perdido tanto los segmentos como los reconocimientos.
- *ACKs triplicados*. Este caso se considera un problema leve, puesto que al menos han llegado los reconocimientos.

5) TCP es un ARQ intermedio entre retroceder N y repetición selectiva

- Como *retroceder a N*, en TCP los reconocimientos son acumulativos, puesto que un reconocimiento a un segmento implica reconocimientos a los segmentos previos.
- Como *repetición selectiva*, la mayoría de las implementaciones TCP pueden aceptar segmentos en desorden y esperar a que lleguen los intermedios. Por ejemplo, cuando al emisor le vence el temporizador sólo retransmite el segmento que causó el vencimiento (y no todos los siguientes como haría retroceder a N).

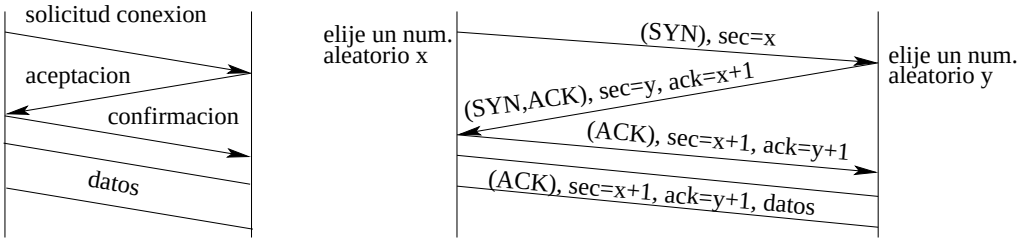
6) Control de flujo

El control de flujo es el mecanismo por el cual el receptor puede indicar al emisor el ritmo en el cual puede recibir datos. En el caso de TCP se usa de la siguiente forma:

- En el momento de la conexión el receptor le indica al emisor el tamaño de su ventana receptora (medido en bytes). El emisor fija su ventana emisora a este valor.
- Para ello se utiliza un campo en la cabecera de los segmentos TCP. El tama no de la ventana deslizante se puede modificar en cada transmisión, con objeto de optimizar el uso de memoria.

7) Conexión

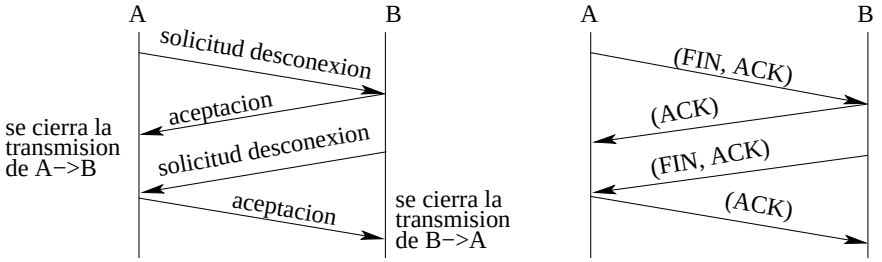
La conexión utiliza un acuerdo en 3 fases. Hay que solicitar, después aceptar y por último confirmar. La confirmación no cuesta nada y da mayor seguridad. Más en detalle:



- El emisor elige un número aleatorio x y el receptor uno propio y. El receptor tiene que reconocer al número $x + 1$ y el emisor reconocer al $y + 1$.
- Los segmentos de conexión, aceptación y confirmación son segmentos normales que tienen activados dos bits de la cabecera, SYN y ACK. Se usa SYN=1 cuando se envía por primera vez x o y y ACK=1 cuando hay que reconocer algún dato.
- Completada la etapa de confirmación pueden enviarse ya datos.

8) Desconexión

La desconexión se realiza en dos fases, en la primera se desconecta la transmisión en un sentido y en la segunda en el otro. Después de que se cierra la conexión en un sentido es posible seguir enviando datos en el otro.



- Se usa FIN en la solicitud de desconexión y ACK para aceptarla.

3.3.2. Estructura del segmento TCP

Antes que nada veremos algunas cosas.

- *Envío.* El proceso va escribiendo sus datos al socket, y las especificaciones de TCP indican que debe coger los datos del socket para transmitir según convenga. Normalmente espera que se vayan acumulando datos durante un cierto tiempo y después los envía.
- *Segmentación.* TCP segmenta los datos y el tamaño máximo del segmento (sin tener en cuenta la cabecera) se denomina *MSS, Maximun Segment Size.* El MSS depende de la implementación concreta en el sistema operativo y valores comunes son 1460 o 536 bytes.

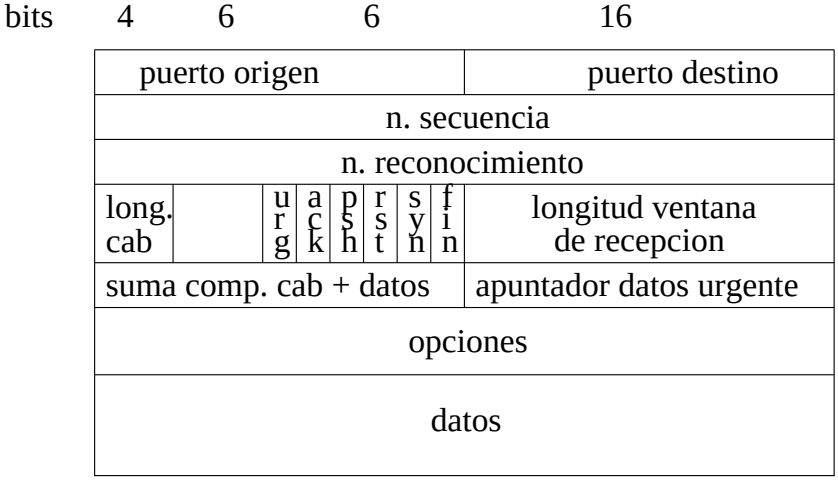
Otras cuestiones:

- *Datos empujados*. El proceso origen va escribiendo sus datos en el socket, la capa de transporte espera que se vayan acumulando durante un cierto tiempo y después los envía. La aplicación (sobre todo si es interactiva) o el S.O. pueden requerir que se realice el envío inmediato, incluso aunque hayan muy pocos datos. Si esto ha ocurrido se activa un bit en la cabecera del segmento TCP. En el destino, la capa de transporte entregará los datos al socket inmediatamente.

- *Datos urgentes*. El proceso origen va escribiendo sus datos en el socket y puede ocurrir que un cierto instante se generen datos urgentes, como por ejemplo cuando el usuario ha tecleado CNT-C o se haya generado una excepción. Hay un bit en el segmento para indicar que existen datos urgentes, y también hay un puntero para indicar la posición en la que empiezan. El proceso destino ha de decidir que hacer con estos datos urgentes.

- *Reset*. Si uno de los extremos no se aclara con los datos recibidos o tiene algún problema puede pedir que se reincie la conexión enviando un *reset*.

La estructura de la cabecera TCP se muestra en la siguiente figura:



1. *Puertos origen y destino*. El puerto destino es estándar y depende de la aplicación, mientras que el puerto origen se elige aleatoriamente.

2. *Número de secuencia*. Numera los segmentos y no tiene que empezar en 0, sino que durante la conexión se establecen números de comienzo (x e y) para ambos lados de la transmisión. A partir de estos números se incrementa en el número de bytes transmitidos, es decir, $x = x + bytes$.

3. *Número de reconocimiento*. Reconoce el número de bytes recibidos correctamente siempre que el bit ACK (que veremos después) está activado. Si el bit ACK no está activado este campo no se usa para nada.

4. *Longitud de la cabecera.* La cabecera tiene longitud variable porque existe el campo de opciones. La cabecera se mide en palabras de 32 bits.

5. *Tamaño de la ventana de recepción.* El receptor le indica al emisor cuantos bytes de datos puede recibir. Se usa para el control de flujo.

6. *Suma de comprobación.* Se usa como código detector de errores y es la suma de palabras de 16 bits en complemento a 1 de la cabecera TCP y cuerpo (más algunos campos de la cabecera IP). Se calcula tanto en el emisor como en el receptor y, si no hay errores de transmisión, ambos coinciden.

7. *Apuntador de datos urgentes.* Indica la posición de los datos urgentes dentro del segmento. Si los hay, también se activa el bit URG.

8. *Campo de opciones.* Entre otras cosas, se usa para negociar algunos parámetros de la conexión, como por ejemplo el tamaño máximo de los segmentos. También se usa para otras cosas.

9. *Campo de indicadores.* Son 6 bits que se usan para indicar lo siguiente:

URG	el segmento tiene datos urgentes
ACK	el segmento tiene un reconocimiento
PSH	el segmento fue empujado
RST	se solicita reinicio de la conexión
SYN	indica solicitud de conexión
FIN	indica solicitud de desconexión

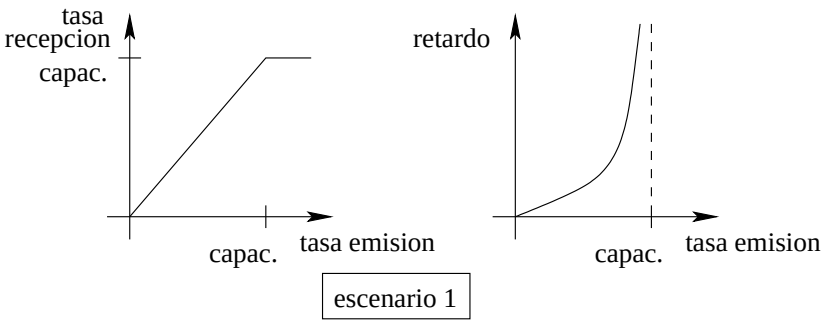
3.4. Congestión

3.4.1. Introducción

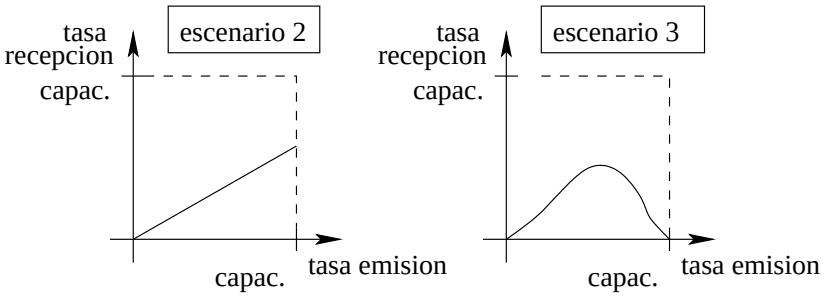
Congestión significa que hay demasiados paquetes en la red y sus síntomas son grandes retardos en las transmisiones y muchos paquetes perdidos. La congestión se suele producir por el desbordamiento de las memorias de los routers. Las memorias de los routers se llenan porque el router no es capaz de reenviar los paquetes al mismo ritmo que los recibe. Y el router no puede aceptar más envios.

Veamos en tres pasos como se produce la congestión:

- Escenario 1. En primer lugar, supongamos el caso ideal de que los routers tengan memoria infinita. Conforme la tasa de emisión va igualando la capacidad de los enlaces, los paquetes se van acumulando en las colas originando retardos cada vez más grandes. Hay que tener en cuenta que un retardo producido en el envío de paquete origina retardos acumulativos en los siguientes paquetes.



- Escenario 2. A continuación, consideremos que la memoria es finita y que hay sólo un router. En este caso el router tiene que descartar paquetes cuando la memoria se llena y el emisor responde realizando retransmisiones para compensar los paquetes desechados, lo cual agrava el problema de la congestión.



- Escenario 3. Por último consideremos un camino de varios routers. En este caso, cuando se descarta un paquete a mitad de camino, el trabajo realizado por los routers anteriores se desperdicia. En consecuencia el trabajo útil realizado es cada vez menor y el rendimiento de la red tiende a cero.

De aquí se desprende que es conveniente añadir mecanismos de control de la congestión. Estos mecanismos de control pueden ser proporcionados por la capa de transporte o por la de red. En el caso de TCP/IP el control de la congestión recae principalmente sobre TCP.

3.4.2. Congestión en TCP

El mecanismo utilizado por TCP para el control de la congestión es que el emisor limite su tasa de envío en función de la congestión de la red que percibe. TCP considera que hay congestión cuando se produce el vencimiento de un temporizador o la recepción de ACKs triplicados. El emisor disminuye la tasa de emisión cuando nota que hay congestión, pero en contrapartida, cuando nota que no hay congestión aumenta esta tasa. El procedimiento es el siguiente:

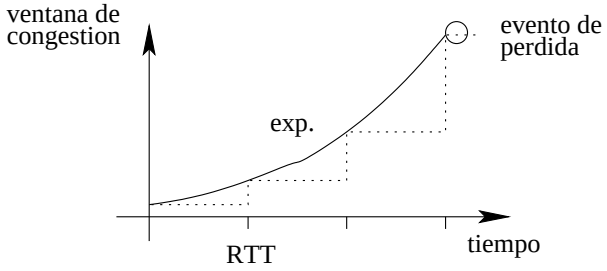
- TCP define en el emisor dos variables numéricas que va actualizando periódicamente: la *ventana de congestión* y el RTT (tiempo de ida y vuelta).

- El RTT es estimado periódicamente por el emisor midiendo el tiempo que pasa entre que envía un paquete y obtiene su reconocimiento.
- La ventana de congestión es actualizada periódicamente por el emisor según el algoritmo que veremos a continuación.
- Dadas estas cantidades, la tasa a la que envía el emisor es

$$\text{tasa de envío} = \frac{\text{ventana de congestion}}{RTT} \text{ bytes/segundo} \qquad (3.7)$$

Vemos el algoritmo de actualización de la *ventana de congestion*. A lo largo de la transmisión, TCP considera 3 fases:

1. *Arranque lento*. Cuando comienza una conexión TCP, se hace la variable ventana de congestión igual al tamaño máximo del segmento, *MSS* y se mide RTT en la etapa de conexión. Por ejemplo, si $MSS = 500$ bytes y $RTT = 0{,}2$ s, resulta una tasa de transmisión inicial $500 \times 8/0{,}2 = 20$ kbps, la cual es una velocidad bajísima, de ahí el nombre de arranque lento. Como el ancho de banda de conexión suele ser mucho mayor que MSS/RTT, se procede a continuación a aumentar la tasa de transmisión exponencialmente.

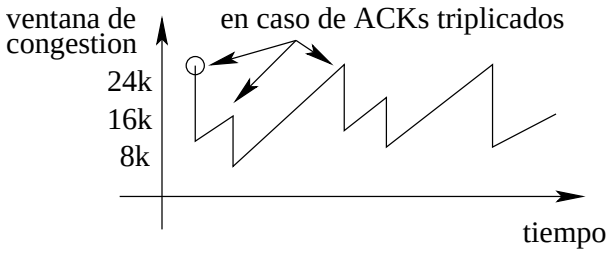


   En concreto, se duplica la ventana de congestión cada intervalo RTT. Se sigue duplicando hasta que se produce un evento de pérdida de segmento, en cuyo caso se pasa a considerar uno de los dos siguientes casos:

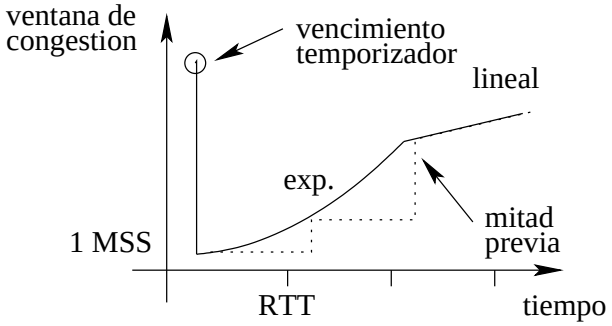
ACKs triplicados	incremento aditivo, decremento multiplicativo
Vencimiento del temporizador	similar al arranque lento

2. *Incremento aditivo, decremento multiplicativo*. Cuando se pierde un segmento y el emisor recibe ACKs triplicados, el emisor considera que hay algo de congestión, pero que el caso no es demasiado grave. El emisor responde reduciendo a la mitad su ventana de congestión (decremento multiplicativo). Sin embargo, la ventana de congestión no puede descender del valor de 1 MSS.

   Si se ha resuelto el problema, a continuación la ventana de congestión se incrementa linealmente, es decir sumando cantidades pequeñas, probando cautelosamente el ancho de banda disponible. Esta es la etapa de incremento aditivo.



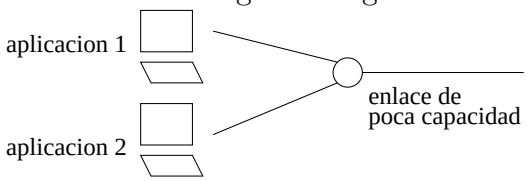
3. *Vencimiento del temporizador*. En cambio, si se produce un vencimiento de temporizador, se considera que la congestión es muy grave puesto que no ha llegado ni siguiera un ACK. TCP entra entonces en una fase de arranque lento, tal como hemos visto. Se pasa a considerar la ventana de congestión igual a 1 MSS y se incrementa exponencialmente. Sin embargo, ahora la ventana de congestión se para a la mitad del valor que tenía cuando se produjo el vencimiento de temporizador y a continuación pasa a incrementarse linealmente. Hay más detalles pero con esto es suficiente.



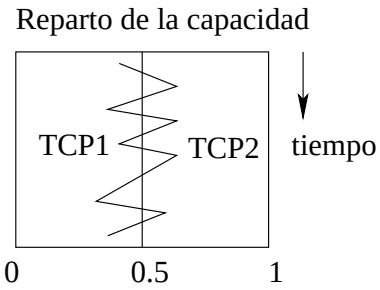
La razón de distinguir entre recibir ACKs triplicados y los eventos de vencimiento del temporizador, es que el primer caso se considera menos grave que el segundo, puesto que con ACKs triplicados se perdió un segmento, mientras que con el vencimiento del temporizador se ha perdido por lo menos un segmento y 3 reconocimientos.

3.4.3. Imparcialidad

Vamos a considerar dos aplicaciones que comparten un enlace de capacidad limitada, tal como se muestra en la siguiente figura:



- Dos conexiones TCP se reparten a medias la capacidad del enlace. En la siguiente figura se muestra como las conexiones TCP realizan el ajuste de su tasa de emisión según el algoritmo que hemos visto antes.



- Con una conexión TCP y otra UDP, la conexión UDP al no tener control de congestión acabará acaparando la mayor parte de la capacidad del enlace.
- Si la primera aplicación usa una conexion TCP y la segunda genera 9 conexiones TCP paralelas, la capacidad del enlace se repartirá, respectivamente, en los porcentajes 1/10 y 9/10.