TEMA 3. CAPA DE TRANSPORTE EN INTERNET.

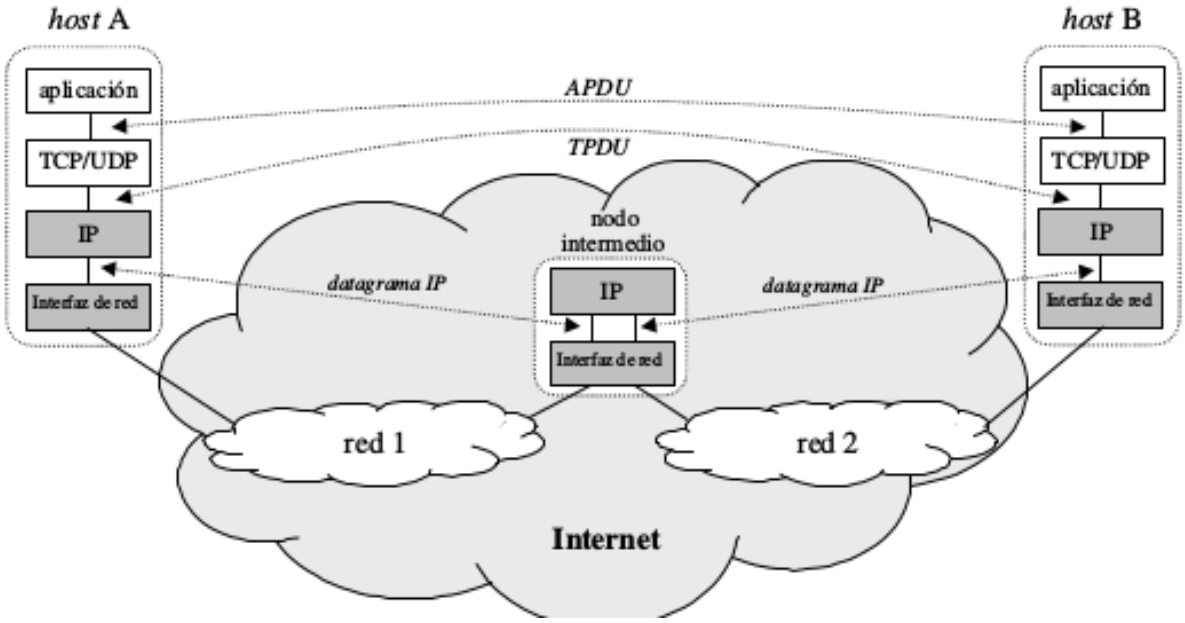
1. Introducción.

Los principales servicios a los que vamos a acceder en capa de transporte son la comunicación extremo/extremo controlando dicha comunicación y además, la multiplexación/demultiplexación de aplicaciones utilizando puertos.

Hasta ahora siempre hemos hablado de dispositivos (Servidor y Cliente), dentro de cada dispositivo tenemos miles de procesos y muchos de ellos se quieren comunicar con internet. Así, cuando llega un nuevo paquete por la tarjeta de red, debemos indicar al sistema operativo a qué aplicación va destinado dicho paquete y así el sistema operativo pueda mantener varias comunicaciones en paralelo.

Ésto se hace mediante el uso de los distintos tipos de puertos que hay.

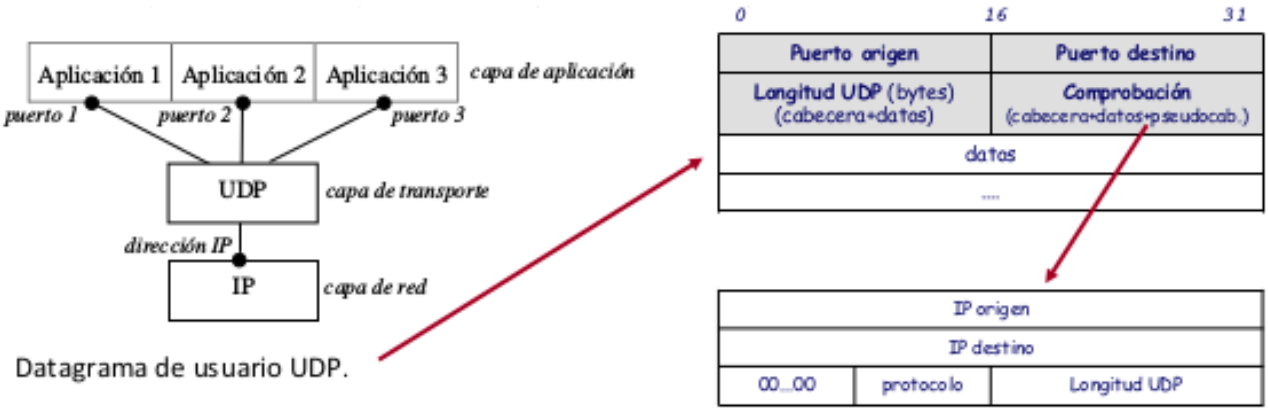
Comunicación extremo a extremo:



El modelo de esta comunicación supone que se dispone de dos computadoras que se comunican (y son los extremos de la comunicación) que están conectados por algún tipo de medio de transmisión y/o red de comunicación.
El protocolo de transporte debe conseguir que sea cual sea el medio de comunicación entre ambos extremos, en particular su fiabilidad (que puede no ser muy buena), se pueda asegurar una comunicación fiable entre los extremos.

2. Protocolo de datagrama de usuario (UDP).

UDP, podemos decir que prácticamente lo único que hace es utilizar los puertos, ya que es un servicio no orientado a conexión, no fiable, etc... es decir, no implementa casi funcionalidad alguna. Por tanto, UDP es un protocolo best-effort, ya que intentará que el paquete llegue a su destino, pero si no llega no se preocupa en realizar el reenvío del paquete.



Vemos la estructura general de la cabecera de un paquete UDP. Tenemos 8 bytes distribuidos en 4 campos de dos bytes cada uno donde tenemos en primer el puerto origen, el puerto destino, la longitud UDP del datagrama.
En UDP, al no haber conexión, nos solemos referir al paquete de datos como Datagrama, que es el nombre usado en internet para paquetes que van individualmente a su destino que nos da el tamaño de la cabecera y el payload que guarda toda la información de los protocolos de la capa de aplicación y por último, tenemos la comprobación que nos permite hacer un pequeño control de errores sobre el paquete.

Ejercicio:
5. Calcule la suma de comprobación en UDP y TCP de las siguientes palabras de 8 bits (observe que aunque UDP y TCP utilicen palabras de 16 bits, en este ejercicio se pide el mismo cálculo sobre palabras de 8 bits): 01010011, 01010100, 01110100.

01010011	10100111	00011011	00011100 → 11100011
+	+	+	
01010100	01110100	1	
---	---	---	
10100111	100011011	00011100	

En este ejemplo, podemos imaginar que el primer número es la IP origen, el segundo, la IP destino y el tercero, la longitud del datagrama. Como se ve, vamos sumando todos los números y si en esa suma hay un acarreo al final (número resaltado en rojo), lo sumamos al principio invirtiendo los primeros bits. Por último hacemos el complemento a uno del resultado.

a)¿Por qué UDP/TCP utilizan el complemento a uno de la suma complemento a uno, en lugar de directamente la suma en complemento a uno?
Si en vez de realizar el complemento a uno, nos quedamos sólo en hacer la suma, el receptor tendría que volver a calcular el checksum y hacer una comparación bit a bit. Si hacemos el complemento a uno, le ahorramos al receptor volver a calcularlo (que es una operación algo costosa) y éste sólo tendría que hacer la suma complemento a uno de toda la cabecera, sumarle el checksum con su complemento a uno y obtener un resultado compuesto únicamente por unos. La comprobación de que una palabra está únicamente compuesta por unos es menos costoso que hacer una comparación bit a bit. En resumen, se hace el complemento a uno de la suma complemento a uno por eficiencia.

b) ¿cómo detecta el receptor los errores?
Si en el resultado hay algún cero, quiere decir que ha habido algún error.

c) ¿se detectan todos los errores de 1 bit?
Sí, porque si cambia un sólo bit obtendríamos un cero.

d) ¿se detectan todos los errores que afectan simultáneamente a 2 bits?
No, porque puede darse la casualidad de que los bits afectados formen una combinación con la que obtengamos el mismo resultado en la suma.

Ejemplos de puertos UDP preasginados:

Ejemplos de puertos UDP preasignados

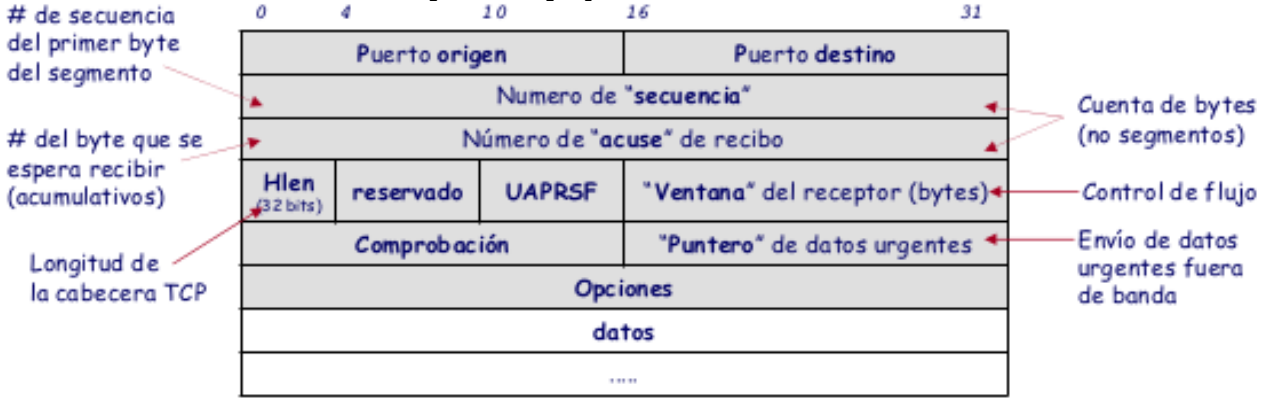
Puerto	Aplicación/Servicio	Descripción
53	DNS	Servicio de nombres de dominio
69	TFTP	Transferencia simple de ficheros
123	NTP	Protocolo de tiempo de red
161	SNMP	Protocolo simple de administración de red
520	RIP	Protocolo de información de encaminamiento

-DNS que normalmente usa UDP aunque también puede usar TCP.
-TFTP es una versión de bajo consumo de recursos TFP.
-NTP es un protocolo de sincronización de tiempo por la red.
-SNMP aparte de para hacer gestión de la red, que sirve para pequeños enlaces, controlar el tráfico…
-RIP es un protocolo de enrutamiento, para escribir la ruta que siguen los paquetes hasta un determinado destino.

3. Protocolo de control de transmisión (TCP).

Introducción a TCP:

- En TCP a cada paquete del flujo de datos se le llama segmento pues hace referencia a un segmento dentro de una linea recta, una parte del todo.
- TCP es un servicio orientado a conexión por lo que hace una operación de conexión y otra de desconexión antes de empezar la transmisión de los datos, ésto se conoce como handshaking. Permite la entrega ordenada y el full-duplex, ésto significa que la conexión TCP permite el envío de datos en ambos sentidos de la comunicación.
- Mecanismo de detección y recuperación de errores (ARQ):
  - Confirmaciones positivas, que son paquetes de control que permiten controlar cuándo ha llegado un segmento a la otra dirección, los paquetes que se usan para confirmar se denominan ACK.
  - No se usan confirmaciones negativas para indicar que un paquete ha llegado mal o no ha llegado, sino que el emisor cada vez que manda un paquete inicia un temporizador, si al finalizar dicho temporizador no ha recibido confirmación vuelve a enviar el paquete. *De aquí viene el hecho de decir que TCP es un servicio fiable.
- Incorporación de confirmaciones (piggybacking), minimiza el número de paquetes enviados de señalización de control, con esta técnica, en vez de enviar ACK en un paquete individual, éste es incluido dentro del próximo paquete a enviar.



El principio es el mismo que en UDP: puerto origen y puerto destino, tras esto tenemos dos números de secuencia:

-Número de secuencia, para llevar la cuenta de los paquetes enviados.

-Número de "acuse" de recibo, para llevar la cuenta de los paquetes que hemos confirmado.

Ambos llevan la cuenta de por donde va la información en ambas direcciones, en ambos sentidos de la comunicación.

Después tenemos un campo para almacenar la longitud de la cabecera TCP, una zona reservada que se usa para temas de investigación. A continuación un campo de flags que son bits que marcan segmentos especiales con algún tipo de infomación especial y es uno de los primeros campos que se mira para saber el valor del resto.

Tras este campo tenemos un parámetro "ventana", que se utiliza en el control de flujo para poder enviar varios segmentos a la vez, seguidamente la comprobación y un puntero que permite diferenciar en el payload datos que deben tener máxima prioridad de datos que no.

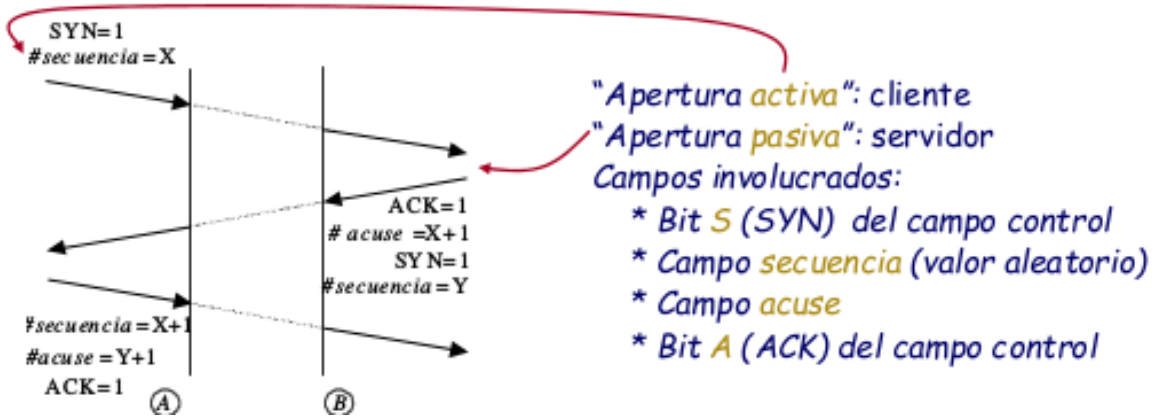
Por último las opciones, que es variable, pueden o no tener opciones, entre las opciones nos podemos encontrar que la longitud de la cabecera sea variable.

Cada segmento TCP se encapsula en un datagrama IP.

Puerto	Aplicación/Servicio	Descripción
20	FTP-DATA	Transferencia de ficheros: datos
21	FTP	Transferencia de ficheros: control
22	SSH	Terminal Seguro
23	TELNET	Acceso remoto
25	SMTP	Correo electrónico
53	DNS	Servicio de nombres de domino
80	HTTP	Acceso hipertexto (web)
110	POP3	Descarga de correo

<u>Control de conexión:</u>

Implica un inicio de la conexión y una desconexión, entre medias de ambos se realiza el envío de datos. En la handshake vemos el esquema de conexión inicial en TCP. Dicho diagrama se denomina three way handshake porque casi siempre se utilizan tres segmentos: uno del origen al destino, otro del destino al origen y otro final del origen al destino. En resumen, se usan tres paquetes para iniciar la conexión.



En un primer paquete se inicializa el número de secuencia (con un número pseudo-aleatorio que sea difícil de predecir para no tener problemas con distintos tipos de conexiones, por ejemplo al iniciar varias conexiones en paralelo que es lo que hacemos al iniciar un navegador web) que irá en la dirección destino-origen y se inicializa el flag SYN (sincronización) a uno. Tras esto, el sistema operativo asigna un puerto a la conexión y se envía dicho paquete sin payload, ya que no es más que un paquete de control.

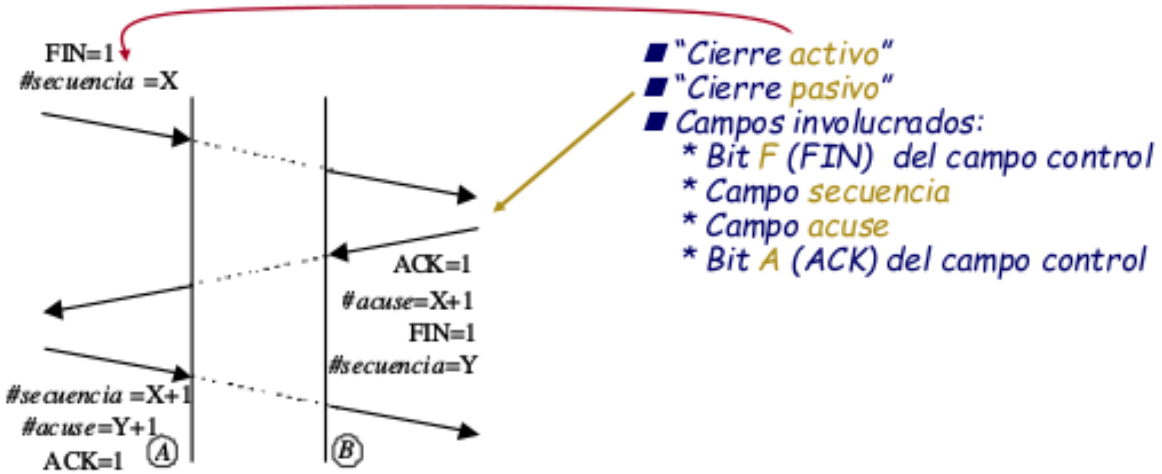
Cuando dicho paquete llega al destino, con un determinado puerto asignado al receptor (servidor) y si consiguen conectarse, el destino enviará un paquete con el flag ACK será establecido a uno. Cuando el flag ACK se establece a uno significa que hay un número válido en el acuse de recibo: el número de secuencia más uno. Esto le deja claro al emisor original que dicha confirmación se refiere a su paquete, así realizamos una doble comprobación.

Al tener una conexión full-duplex, el destino también inicializa su dirección de la conexión haciendo lo mismo que el emisor: inicializa su número de secuencia con un valor pseudo-aleatorio independiente del número de secuencia del emisor y establece el flag de sincronización SYN a uno.

Cuando éste último paquete llega al emisor original, establece el flag ACK a uno y da el acuse de recibo de lo que ha recibido.

*TCP incrementa el número de secuencia de cada segmento según los bytes que tenía el segmento anterior, con una sola excepción: los flags SYN y FIN, cuando están puestos, incrementan en 1 el número de secuencia. ACK no incrementa el número de secuencia. Son campos de 32 bits.

<u>Cierre de la conexión:</u>
Es muy parecida a la conexión, con la diferencia de que tenemos un flag denominado FIN. Ocurre lo mismo que en la conexión, con la diferencia de que aquí ya ha habido intercambio de datos entre finales anteriormente, es decir, entre la conexión y la desconexión está todo el servicio proporcionado.



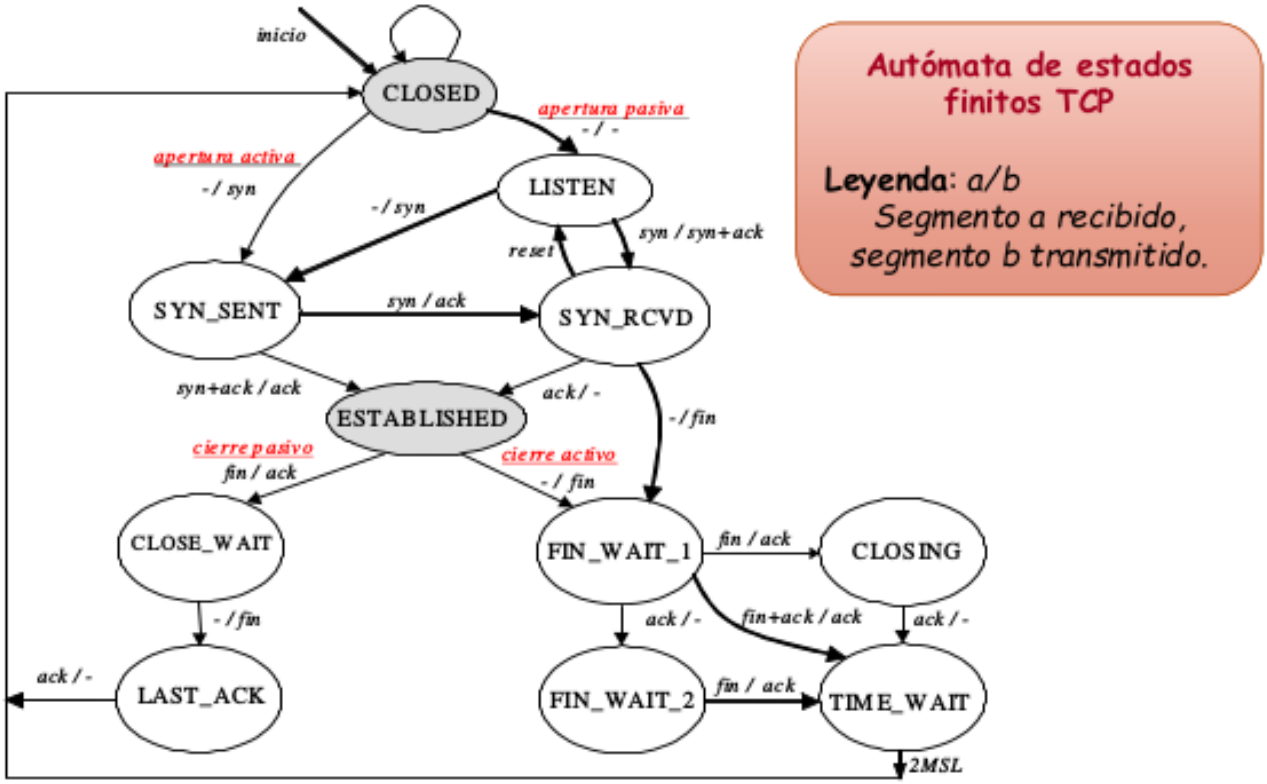
Para hacer la desconexión, el que quiere desconectar manda un segmento en el que establece el flag FIN a uno y va con un determinado número de secuencia que será el número de secuencia original más el número de bytes enviados.

Cuando dicho segmento llega al destino, éste lo confirma y si quiere cerrar también la conexión, establece también su flag FIN a uno e incrementa su número de acuse de recibo.

*Un detalle importante es que el que inicia la desconexión, tras mandar el último paquete de desconexión, inicializa un temporizador antes de cerrar definitivamente.

Autómata de estados finitos TCP

Cada nodo representa un estado en el que puede estar cualquiera de los dos fines de TCP (tanto cliente como servidor). Sobre cada flecha que une dos nodos vemos una / que separa las acciones que hace cada parte para cambiar de estado, si vemos un símbolo "-" significa que esa parte no ha hecho nada para cambiar de estado sino que cambia de estado por una acción realizada por la otra parte.



En el lado del cliente, el que inicia la conexión, se envia en primer lugar un syn (estado SYN_SENT), tras esto, esperará un segmento con syn + ack y finalmente, mandará un ack, pasando al estado ESTABLISHED tras el cual podemos empezar a mandar datos. Este proceso sería una apertura activa.

En el lado del servidor se realiza una apertura pasiva. Inicialmente el servidor se encuentra en estado LISTEN, en el cual se encontrará escuchando peticiones en un determinado puerto. En esa escucha, recibe un syn por parte de un cliente, a lo que le responde con un syn + ack y pasa al estado SYN_RCVD. Cuando reciba un ack sin decir él nada, pasará al estado ESTABLISHED.

Una vez en ESTABLISHED, podemos cerrar la conexión mediante el cierre activo. Para ello, debemos enviar un fin y, sin nosotros recibir nada, pasaremos al estado FIN_WAIT_1. Cuando nos llegue un fin + ack, nosotros mandaremos un ack y pasaremos al estado TIME_WAIT donde esperaremos el intervalo de tiempo que debía esperar el que iniciaba la desconexión antes de cerrarla del todo.

En el cierre pasivo, recibiremos un fin y nosotros responderemos con un ack pasando al estado CLOSE_WAIT. Cuando estemos preparados para cerrar, responderemos con un fin pasando al estadoLAST_ACK y cuando recibamos un ack volveremos al estado inicial, CLOSED.

Ejercicio:

9. Se desea transferir con protocolo TCP un archivo de L bytes usando un MSS de 536.

El MSS son las siglas de Maximum Segment Size que en español significa Tamaño Máximo de Segmento y se corresponde con el número máximo de bytes que pueden ir en el payload (sin incluir

el tamaño de la cabecera). El MSS se mide en bytes y, por tanto, el tamaño máximo de un paquete será el tamaño de la cabecera más el MSS.

a) ¿Cuál es el valor máximo de L tal que los números de secuencia de TCP no se agoten?
Como el número de secuencia mide el tamaño en bytes transmitidos y debemos tener en cuenta que enviar un paquete con los flags SYN o FIN a uno cuenta como enviar un byte, el máximo de L sería:

L = 2^32 – 2 bytes.

b) Considerando una velocidad de transmisión de 155 Mbps y un total de 66 bytes para las cabeceras de las capas de transporte, res y enlace de datos, e ignorando limitaciones debidas al control de flujo y congestión, calcules el tiempo que se tarda en transmitir el archivo en A.
Velocidad de transmisión = 155 Mbps
al estar el tamaño del paquete en bytes (536+66), debemos pasarlo antes a bits y dividir el tamaño del paquete entre el tiempo de transmisión:

Tt = ((536+66) * 8 / 155 * 10^6) = 3,107097 * 10^-5

Después, debemos calcular el número total de segmentos que enviaremos. Cada segmento tiene 66+536 bytes. Asumiendo como L el tamaño del fichero, el número total de segmentos transmitidos será:

Ns = L / MSS = 2^32 - 2 / 536 = 8012999

Como el resultado de esta división casi siempre será un número real, debemos redondearlo al siguiente entero.

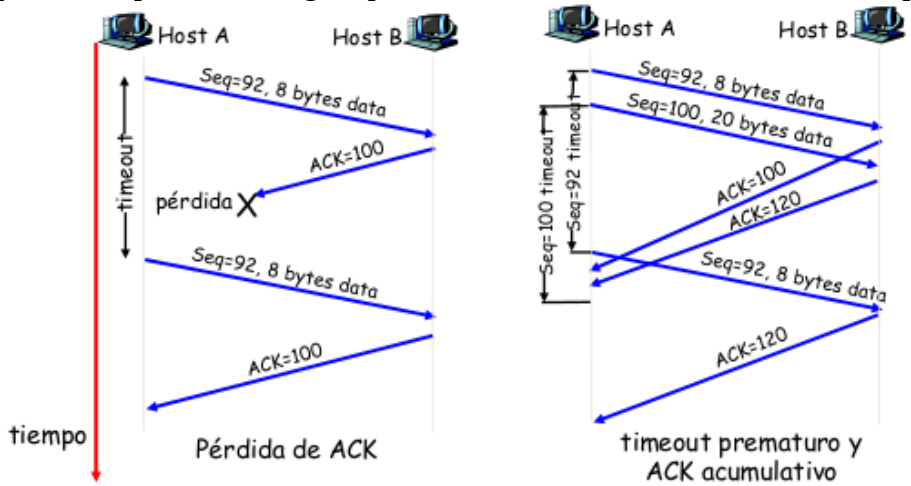
Asumiendo que el último paquete tiene 536 bytes, el tiempo total de transmisión del paquete se calculará multiplicando el número de segmentos a enviar por el tiempo empleado para enviar cada uno:

Ttotal = Ns * Tt = 8012999 * 3,107097 * 10^-5 = 248.97163344516127s

Si no asumimos que todos los paquetes tienen el mismo tamaño, deberíamos calcular el tamaño del último paquete diviendo L entre el MSS y quedándonos con el resto. Después deberíamos de calcular el tiempo de transmisión para dicho segmento y hacer Ns - 1 * Tt + Tultimopaquete.

<u>Control de errores y flujo:</u>
Tanto el control de flujo como el control de congestión reducen la velocidad de transmisión cuando sea necesario. El control de congestión lo hace para no congestionar la red y el control de flujo, para no saturar al receptor. Además, también comprueba que todos los paquetes han sido recibidos y que además, se hayan recibido en orden.

Sería natural pensar que TCP, al enviar un segmento, espera a recibir la confirmación (ack) del receptor para mandar el siguiente segmento (esta estrategia se denomina stop&wait) pero tiene un problema: presenta una gran pérdida de velocidad de transmisión, tiempo muerto (timeout).



En el caso de la izquierda, Host A está enviando un segmento de 8 bytes de longitud. El número de secuencia que se ve es el segmento por el que va en ese momento, el número de secuencia se puede haber inicializado a 91, por eso espera el 92. Cuando Host B recibe el paquete, realiza el checksum, para ver que no hay problemas y responde con un ack indicando el siguiente segmento que espera (en este caso, 92 + 8, es decir, Seq + NumBytes = 100). Si se pierde ese ack, se acabará el timeout y el emisor volverá a mandar el segmento. Entonces, Host B verá que es lo mismo que ha recibido antes y volverá a mandar el ack. En este escenario estámos mandando una ventana de un único segmento, por lo que cuando llegue el ack se enviará el siguiente incrementándose el número de secuencia.

En el segundo caso, tenemos una ventana con dos segmentos, por lo tanto se pueden enviar dos segmentos esperando solo una confirmación. Se envía el primer segmento con número de secuencia 92, 8 bytes de datos y se inicializa su timeout y acto seguido, se envía el segundo segmento, inicializando también su timeout. Cuando llega el primero, se confirma indicando que lo siguiente que espera es 100, es decir el segundo segmento que hemos mandado y cuando llega el segundo, se manda el ack indicando que lo siguiente que espera es 120. El problema es que el primer ack llega tarde y expira el primer timeout. Una vez expirado, se vuelve a reenviar el primer segmento, pero justo después llega el ack del segundo segmento dentro del timeout. Por lo tanto, el Host A sabe que han llegado ambos segmentos y sabe que el siguiente segmento que espera es el 120.

Evento	Acción del TCP receptor
Llegada ordenada de segmento, sin discontinuidad, todo lo anterior ya confirmado.	Retrasar ACK. Esperar recibir al siguiente segmento hasta 500 mseg. Si no llega, enviar ACK.
Llegada ordenada de segmento, sin discontinuidad, hay pendiente un ACK retrasado.	Inmediatamente enviar un único ACK acumulativo.
Llegada desordenada de segmento con # de sec. mayor que el esperado, discontinuidad detectada.	Enviar un ACK duplicado, indicando el # de sec. del siguiente byte esperado.
Llegada de un segmento que completa una discontinuidad parcial o totalmente.	Confirmar ACK inmediatamente si el segmento comienza en el extremo inferior de la discontinuidad.

Ejercicio:
11. Los hosts A y B se están comunicando a través de una conexión TCP y B ya ha recibido y confirmado todos los bytes hasta el byte 126. Suponga que a continuación el host A envía dos segmentos seguidos a B que contienen, respectivamente, 70 y 50 bytes de datos. El envío de A es ordenado, el número de puerto origen en dichos segmentos es 302 y el de destino el 80. El host B envía una confirmación inmediata a la recepción de cada segmento de A, sin esperar el retardo de 500 ms del estándar.

a) Especifique los números de secuencia de ambos segmentos.
En el segmento de 70 bytes de datos, su número de secuencia será 127, ya que se pone el siguiente que se espera. Es decir, como se había confirmado hasta el 126, el número de secuencia se pone a 127. Para el segmento de 50 bytes, será de 197. Esto no se incrementa en uno, sino que se pone el siguiente que se espera. Cuando se había confirmado el 126, se pone a 127 ya que es el siguiente que se espera. Cuando llega el segmento de 70 bytes, se confirma hasta el 196, y como el siguiente que se espera es el 197, pues el número de secuencia se pone a este número.

b) Si el primer segmento llega antes que el segundo ¿cuál es el número de acuse y los puertos origen y destino en el primer ACK que se envía?

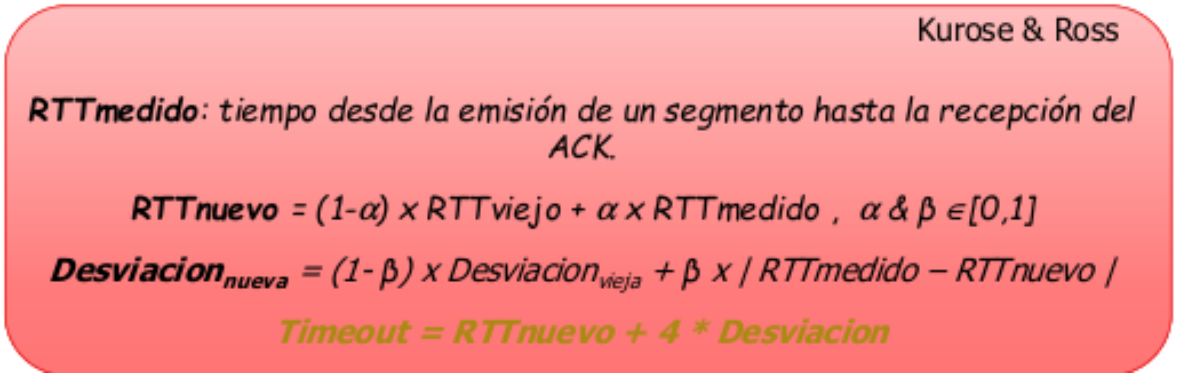
El número de acuse es 197, y el puerto origen es el 80 y el destino 302. Si A tiene como puerto origen el 302 y envía al 80, las comunicaciones de B a A, tendrán como puerto origen el 80 y puerto destino el 302.

c) Si el segundo segmento llega antes que el primero ¿cuál es el número de acuse y los puertos origen y destino en el primer ACK que envía?
En este caso, el número de acuse es el 127, porque hemos dejado una discontinuidad. Es decir, del 127 al 197 se ha quedado vacío, por lo que necesitamos recibir el segmento de 70.

d) Imagine que los segmentos llegan en orden pero se pierde el primer ACK.
No pasa nada ya que como los ACK son acumulativos, llegando el segundo ACK, queda confirmado el primer segmento y el segundo.

¿Cómo estimar los timeout?
No podemos poner un tiempo ni demasiado grande ni demasiado pequeño, ya que si se pone demasiado grande, aunque parezca que le puede dar tiempo a mandar el segmento y recibir el ack, podemos mandar los paquetes de forma muy lenta. Y si se pone demasiado pequeño, no le dará tiempo a ningún segmento a ser confirmado, porque no le dará tiempo al ack a llegar, por lo que podríamos estar siempre mandando el mismo segmento.

Es algo que hay que estimar en función de la velocidad de red, por lo tanto debe estar ajustado a la velocidad de esta, y debe ser ajustado en el momento, es decir, de forma dinámica. Hay que ver como va la red y fijar un valor de acuerdo al estado de esta.



Ejercicio:
15. Si el RTT es 30 ms, la Desviación es 2 ms y se reciben ACKs tras 26, 32 y 24 ms, ¿Cuál será el nuevo RTT, Desviación y timeout? Usar α=0,125 y β=0,25.
En este caso, cogemos el valor 30 y lo tomaremos como el RTTviejo, cogemos el primer valor que hemos medido 26ms y lo tomamos como el RTTmedido y con el valor de α=0,125 y β=0,25.

RTTnuevo = (1-α) * RTTviejo + α * RTTmedido = (1-0,125) * 30 + 0,125 * 26 = 29,5ms

Una vez actualizada la media, actualizamos la desviación típica.
Desviación_nueva = (1-β) * Desviación_vieja + β * | RTTmedido – RTTnuevo | =
= (1-0,25) * 2 + 0,25 * | 26 – 29,5 | = 2,375ms

Con esto, nuestro RTTnuevo pasa a ser 29.5ms y la Desviación_nueva pasa a ser 2,375 y su timeout se calcula como:

Timeout = RTTnuevo + 4 * Desviación = 29,5 + 4 * 2,375 = 39ms

Para los demás ACKs, para cada valor que llegue, el RTTviejo será el que calculamos anteriormente y el RTTnuevo el que acaba de llegar.

RTTnuevo = (1-α) * RTTviejo + α * RTTnuevo = (1-0,125) * 29,8125 + 0,125 * 32 = 30,0859375ms

Desviación_nueva = (1-β) * Desviación_vieja + β * | RTTmedido – RTTnuevo | = = (1-0,25) * 2,328125 + 0,25 * | 32 – 30,0859375 | = 2,224609375ms

Timeout = RTTnuevo + 4 * Desviación = 29,8125 + 4 * 2,328125 = 39,125ms

Para el último segmento, volvemos a calcular el tiempo igual que antes:

RTTnuevo = (1-α) * RTTviejo + α * RTTnuevo = (1-0,125) * 30,0859375 + 0,125 * 24 = 29,3251953125ms

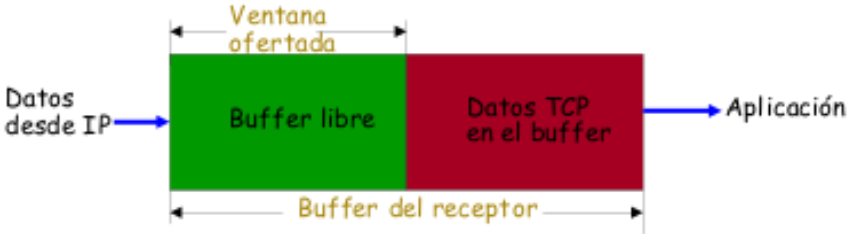
Desviación_nueva = (1-β) * Desviación_vieja + β * | RTTmedido – RTTnuevo | = = (1-0,25) * 2,224609375 + 0,25 * | 32 – 30,0859375 | = 2,999755859375ms

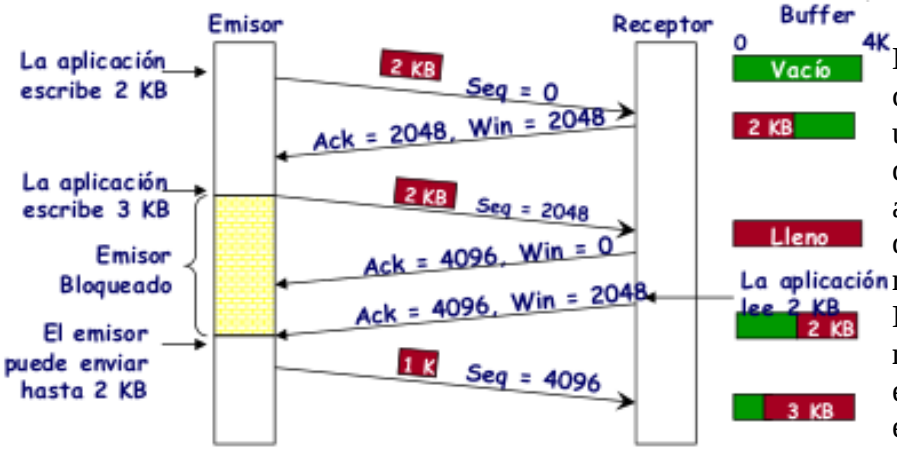
Timeout = RTTnuevo + 4 * Desviación = 29,3251953125 + 4 * 2,999755859375= 41,32421875ms

¿Y si los dos primeros ACKs tienen el mismo número de acuse y se usa el algoritmo de Karn?
En el caso de usar el algoritmo de Karn, es todo igual, pero los dos últimos segmentos, que como están repetidos, se rechazan para el cálculo y no se tiene en cuenta.

Control de errores y de flujo (esquema crediticio):
El objetivo del control de flujo era no saturar al receptor (evitar unbuffer overfow), para implementar esto el receptor avisa al emisor de cuando está dispuesto a recibir datos. La información que puede recibir el receptor se mide en créditos: si el receptor puede recibir 4KB, el crédito será de 4KB y si no puede recibir nada, el crédito será de 0KB y se detendría el envío de datos TCP.

ventana útil emisor = ventana ofertada receptor – bytes en tránsito





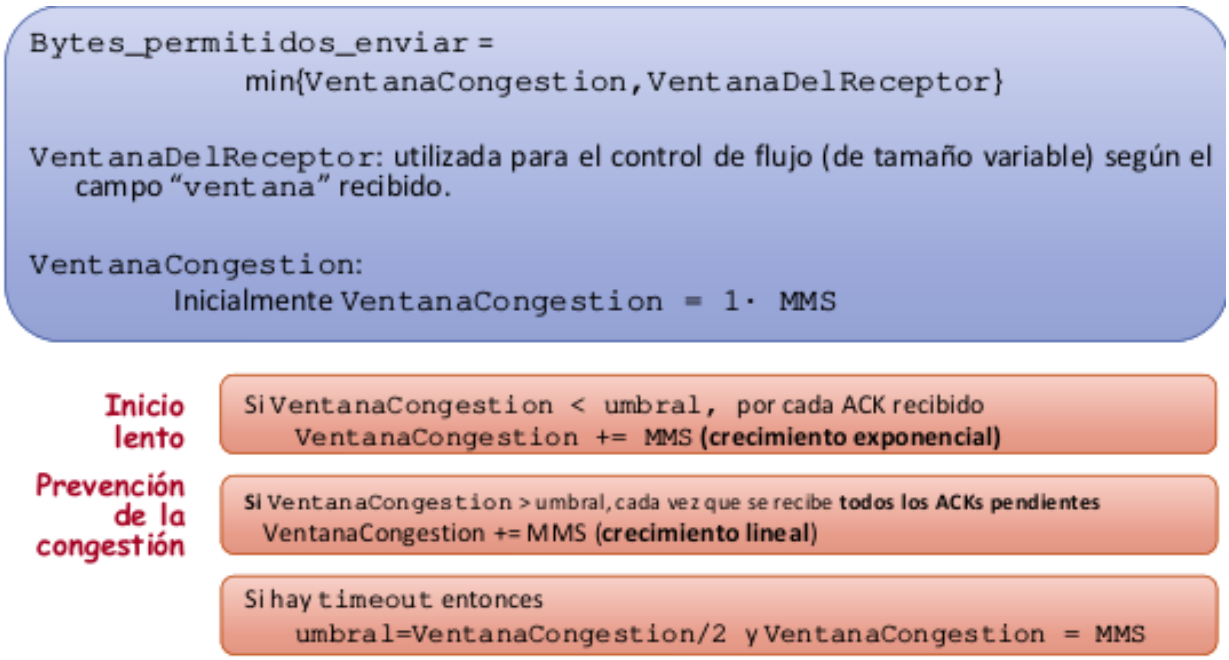
En la imagen tenermos una comunicación entre un emisor y un receptor, en el emisor la capa de aplicación envía 2KB a TCP y al ser su Maximum Segment Size de 2KB lo envía directamente al receptor en un sólo segmento. El segmento es recibido por el receptor, cuyo buffer de recepción es de 4KB y en ese momento se encuentra vacío, y tras almacenarlo en dicho buffer, envía un ACK de esos 2048 bytes indicando que le quedan libres otros 2KB en su ventana.
El emisor, al no tener más información que enviar, no hace nada más que hasta que la aplicación vuelve a enviar información, 3KB, a TCP. Dicha infomación se divide en segmentos y TCP se da

cuenta de que no puede enviarlo todo, pues sólo le han dado crédito para enviar 2KB, por tanto, envía esos 2KB y se queda esperando a poder enviar los 1024 bytes restantes.
El receptor al recibir dicho segmento, lo almacena en su buffer y le envía la confirmación correspondiente indicando un crédito de 0 bytes, es decir, que no hay espacio en el buffer.
En algún momento la aplicación del receptor preocesará 2KB del buffer y volverá a dejar espacio libre. En ese instante, el receptor envía un ACK duplicado al amisor con el mismo número de secuencia pero dándole 2KB de crédito para enviar, como el emisor tenía pendiente enviar 1KB, lo envía y finaliza la comunicación entre ambos.

**Este método tiene un problema, si el emisor envía datos a TCP muy rápido y el receptor los procesa muy lentamente, al final el emisor enviará muchos paquetes con segmentos pequeños ya que al recibir crédito del receptor, por mínimo que sea, enviará inmeditamente segmentos con ese tamaño. Esto se conoce como ventana tonta.
Una posible solución es la ventana optimista.

Ventana optimista, el receptor asume que el emisor va procesando datos conforme éste se los va enviando y por tanto, no hace falta considerar los bytes en tránsito en la ventana útil, porque conforme los ha ido enviando el emisor, los ha ido procesando el emisor. Así, no se reduce el tamaño de la ventana y se usan todos los créditos que ha concedido el emisor. Si se llega a un buffer overflow, el emisor no enviará confirmación y el receptor tendrá que volver a enviar el segmento de acuerdo a su temporizador.

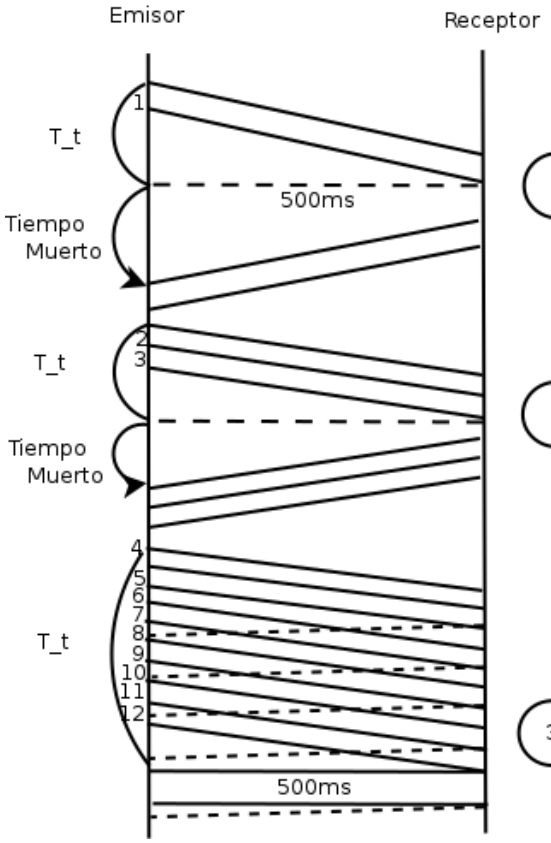
Control de congestión:
El objetivo del control de congestión es el mismo del control de flujo pero cambiando al agente, es decir, a quien no se quiere saturar. En el control de flujo dejábamos que el receptor nos dijese cuanta información podíamos enviarle, sin embargo en el control de congestión no tiene a nadie que le diga cuando está la red congestionada, sino que tiene que darse cuenta el mismo. Para ello el emisor debe medir el tiempo de alguna forma y la forma más sencilla es contabilizar la pérdida o el retraso de ACKs: si el ACK no ha llegado dentro del temporizador, significa que ha pasado algo. Puede ser que el receptor se haya saturado, que se haya perdido el paquete sin haber congestión en la red…



Ejercicio:
16. Teniendo en cuenta el efecto del inicio lento, en una línea sin congestión con 10 ms de tiempo de propagación, 1 Mbps de velocidad de transmisión y un MSS de 2KB, ¿cuánto tiempo se emplea en enviar 24 KB?
TCP cuando recibe el stream de datos de la capa de aplicación tras el handshake, lo primero que hace es inicializar la ventana de congestión a 1 MSS, y divide el stream en segmentos. El número de segmentos se calcula utlizando el MSS (o tamaño máximo de bytes que se puede llevar en el payload) y el tamaño del stream, sin contar las cabeceras.

Ns = M / MSS = 24KB / 2KB = 12 segmentos.

Estos 12 segmentos, lo enviaremos de acuerdo a nuestro control de congestión. Como nuestra ventana está a 1 MSS, pues enviaremos un único segmento. Empezamos enviando el primer segmento y calculamos nuestro tiempo de transmisión:



(1) podemos ver el tiempo que tarda en transmitir desde el primer al último but del primer segmento.
Tt = N.º bits / Velocidad de transmisión
= (2048bytes * 8bits) / 1 * 10^6 = 0,016s = 16ms

1 Desde que se ha enviado el primer segmento hasta que se ha confirmado, han pasado:
Tt + 2 * Tp + 500ms

(2) podemos enviar 2 segmentos a la vez, pero esta vez cuando llega el segundo, se confirma inmediatamente

2 2 * Tt + 2 * Tp

(3) para el resto de segmentos, están preparados 4, 5, 6 y 7, se envian de dos en dos, cuando llegan sus confirmaciones se libera espacio.
9 * Tt

3 En este caso, solo se cuentan los tiempos de transmisión, ya que todos los segmentos se propagan en paralelo por lo que no se contabilizan. En total, el tiempo en mandar los 24KB es:
12 * Tt + 4 * Tp + 2 * 500ms

4. Extensiones TCP.

Existen muchos tipos de TCP, lo que se conoce como sabores de TCP, es decir, versiones. Todos ofrecen una interoperabilidad, ya que tiene que convivir con el resto de versiones de TCP que existen.

Para linux se usa la version de TCP CuBIC, que tiene una especie de temporizador para pegarse los más posible al límite de la red.

<u>Adaptación de TCP a redes actuales</u>

- Ventana escalada: Con las redes muy amplias, se necesitan de ventanas muy amplias. Por eso, una de las opciones que se añaden durante el handshake es habilitar la ventana escalada en la cabecera, lo que contiene un factor de escala por el que se multiplica la ventana máxima (parámetro window). Este factor de escala utiliza una ley exponencial. Este valor de escala puede llegar a 2^14. Esto junto a los 2^16 bytes que había inicialmente, hacen un

total de 2^14 * 2^16=2^30 bytes, es decir, se puede llegar a conseguir una ventana de 1GB. Con este tamaño de ventana, es más fácil llegar al rendimiento de la eficiencia unidad.

- Estimación RTT: esta estimación del RTT consiste en que al enviar un segmento, se mete dentro del paquete un sello de tiempo (time stamp), es decir, el momento de tiempo justo en el que lo está enviando. Cuando se responda con un ack, este ack también tendrá un sello de tiempo, aunque también se puede tener registrado este tiempo, en vez de tener que llevarlo el ack. Si se manda un paquete con un sello de tiempo X, el ack que se espera tendrá también el sello de tiempo X_1. Cuando llegue el ack, lo único que hay que hacer es restar ambos sellos de tiempo y ya se sabe cuánto ha tardado el paquete en llegar y ser confirmado, con lo que se consigue estimar el RTT, sin necesidad de comprobar que llega el ack del siguiente, etc.

- PAWS: Sirve para que cuando llegue un segmento con un time stamp que ya ha pasado, se desprecie directamente, ya que no es información útil.