TEMA 3: LA CAPA DE TRANSPORTE.

La capa de transporte y sus servicios.

La capa de transporte proporciona directamente servicios de comunicación a los procesos de aplicación que se ejecutan en host diferentes.

Los protocolos de la capa de transporte están implementados en los sistemas terminales. El emisor, convierte los mensajes que recibe procedentes de un proceso de aplicación emisor en paquetes de la capa de transporte, conocidos como segmentos de la capa de transporte. (Divide mensajes en segmentos y los pasa a la capa de red).

El receptor, extrae el segmento de la capa de transporte del datagrama y lo sube a la capa de transporte (Reensambla segmentos en mensajes y los pasa a la capa de aplicación).

La capa de transporte es la comunicación lógica entre procesos y se basa de los servicios de la capa de red, a la vez que los amplia.

Relaciones entre las capas de transporte y de red.

Un proceso de la capa de transporte proporciona una comunicación lógica entre procesos que se ejecutan en host diferentes, un protocolo de la capa de red proporciona una comunicación lógica entre hosts.

La capa de transporte en Internet.

UDP proporciona un servicio sin conexión no fiable a la aplicación que invoca.

TCP proporciona a la aplicación que invoca un servicio orientado a la conexión fiable.

Los paquetes de la capa de red los llamamos segmentos y a los paquetes de UDP, datagramas.

IP proporciona una comunicación lógica entre host. Hace todo lo que puede por entregar los segmentos entre los dos hosts que se están comunicando, pero no garantiza la entrega, por lo que se dice que no es un servicio fiable.

UDP y TCP amplían el servicio de entrega de IP entre dos sistemas terminales a un servicio de entrega entre dos procesos. Extender la entrega se denomina multiplexación y demultiplexación de la capa de transporte. También incluyen campos de detección de errores en las cabeceras de sus segmentos.

TCP ofrece transferencia de datos fiable y mecanismo de control de congestión.

Multiplexación y demultiplexación.

La capa de transporte del host receptor realmente entrega los datos directamente a un proceso sino a un socket intermedio. Cada socket tiene asociado un identificador único.

Demultiplexación es entregar los datos contenidos en un segmento de la capa de transporte al socket correcto.

Multiplexación es reunir los fragmentos de datos en el host de origen desde los diferentes sockets, encapsulando cada fragmento de datos con la información de cabecera para crear los segmentos y pasarlos a la capa de red.

¿Cómo funciona la demultiplexación?

El programa recibe datagramas IP, estos tienen una IP de origen, otra de destino y un segmento de la capa de transporte, que cada uno tiene un numero de puerto de origen y destino. El host usa IP y el numero de puerto para dirigir el segmento al socket apropiado.

Multiplexación y demultiplexación sin conexión.

Al crear un socket UDP, la capa de transporte asigna automáticamente un numero de puerto al socket. Una vez asignados podemos describir de forma precisa las tareas de multiplexación y demultiplexación en UDP.

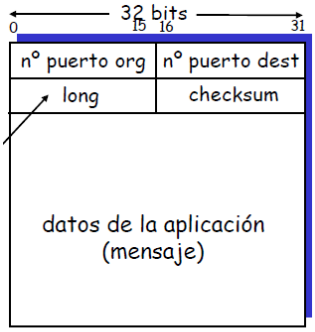
La capa de transporte del host emisor crea un segmento de la capa de transporte que incluye los datos de aplicación, el numero de puerto origen, el de destino y otros dos valores más. La capa de transporte pasa a la capa de red. Esta encapsula el segmento en un datagrama IP y entrega el segmento al host receptor. Si llega, la capa de transporte examina el numero de puerto destino y entrega el segmento a su socket.

Multiplexación y demultiplexación orientadas a la conexión.

Un socket TCP es identificado por una tupla de: dirección IP de origen, numero de puerto de origen, dirección IP de destino y numero de puertos de destino. Cuando un segmento TCP llega a un host procedente de la red, el host emplea los cuatro valores para demultiplexar el segmento al socket.

Transporte sin conexión: UDP.

Es un servicio “haz lo que puedas”, por tanto, los segmentos UDP pueden perderse o bien ser entregados fuera de orden a la aplicación.



formato de segmento UDP

UDP toma los mensajes precedentes del proceso de la aplicación, asocia los campos correspondientes a los números de puerto origen y de destino para proporcionar el servicio de multiplexación/demultiplexación, añade dos campos pequeños mas y pasa el segmento resultante a la capa de red. Esta encapsula el segmento en un datagrama IP y hace el mejor esfuerzo por entregar el segmento al host receptor. Si llega, UDP utiliza el numero de puerto de destino para entregar los datos del segmento al proceso apropiado de la capa de aplicación. UDP no tiene lugar en una fase de establecimiento de la conexión entre entidades de la capa de transporte emisora y receptora previa al envío del segmento. Aplicaciones utilizas UDP por:

- Mejor control en el nivel de aplicación.
- Sin establecimiento de la conexión. Por ello no añade ningún retardo.
- Sin información del estado de la conexión.
- Poca sobrecarga, gracias a que la cabecera contiene solo 8 bytes (TCP 20 bytes).

Suma de comprobación UDP (Checksum).

La suma de comprobación proporciona un mecanismo de detección de errores. El emisor calcula el complemento a 1 de la suma de todas las palabras de 16 bits. El desbordamiento obtenido se almacena en el campo suma de comprobación del segmento UDP.

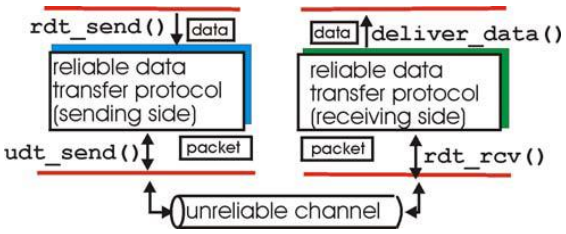
El receptor calcula el checksum del segmento recibido, y comprueba si es igual, si no entonces detecta un error y si es igual no detecta error.

Al no estar garantizadas ni la fiabilidad enlace a enlace, ni la detección de errores durante el almacenamiento de memoria, UDP tiene que proporcionar un mecanismo de detección de errores de la capa de transporte.

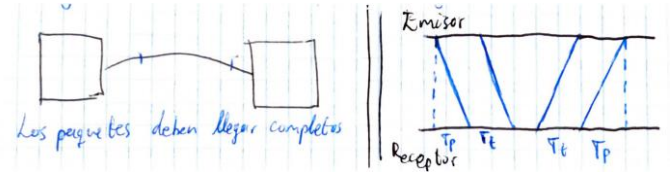
Principios de un servicio de transferencia de datos fiable.

Los problemas de implementar estos servicios aparecen en la capa de transporte, capa de enlace y capa de aplicación.

El emisor del protocolo de transferencia de datos es invocado desde la capa superior con *rdt_enviar()*, que pasara los datos que haya que entregar a la capa superior en el lado receptor. El receptor *rdt_recibir()*, es llamado cuando llegue un paquete desde el lado del receptor del canal. Y cuando rdt desea suministrar datos a la capa superior, llama *entregar_datos()*.



TCP garantiza una transferencia fiable sobre un canal con perdidas/errores.



Tiempo transmisión: Capacidad que tiene el emisor (interfaz de red) que es capaz de emitir los bits (bit a bit, no pueden en paralelo) a una velocidad.

Tiempo de propagación: Tiempo que tarda en llegar cada bit al otro extremo.

Tr: Tiempo de procesamiento.

Longitud del paquete: Nº de bits que contiene un paquete.

Construcción de un protocolo de transferencia de datos fiable.

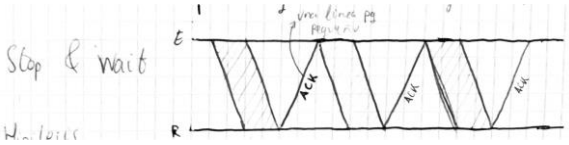
1. **TRANSFERENCIA DE DATOS FIABLE SOBRE UN CANAL TOTALMENTE FIABLE: rdt1.0.**

El lado emisor acepta los datos de la apa superior con *rdt_enviar(datos)*, crea un paquete con *crear_paquete(datos)* y envía el paquete al canal. El lado receptor, recibe un paquete del canal subyacente con *rdt_recibir(paquete)*, extrae los datos del paquete con *extraer(paquete, datos)* y pasa los datos a la capa superior con *entregar_datos(datos)*.

2. **TRANSFERENCIA DE DATOS FIABLE SOBRE UN CANAL DE ERRORES DE BIT: rdt2.0.**

Los protocolos ARQ requieren tres capacidades:

- Detección de errores→Necesitan un mecanismo que permita al receptor detectar errores de bit. Requieren que el emisor envíe al receptor bits adicionales que se tendrán en cuenta para el calculo de la suma de comprobación del paquete de datos.
- Realimentación del receptor→Dado que el emisor y el receptor normalmente se ejecutan en sistemas diferentes. Para que el emisor sepa lo que ocurre en el receptor, el receptor envía información de realimentación al emisor. Los ACK y los NAK.
- Retransmisión→Un paquete que se recibe con errores en el receptor será retransmitido por el emisor.



El lado emisor tiene dos estados:

- Espera datos procedentes de la capa superior. Cuando llega *rdt_enviar(datos)* el emisor crea un paquete que contendrá los datos a transmitir junto con la suma de comprobación y se envía con *rdt_enviar(paquete)*.
- El emisor está a la espera de un paquete de reconocimiento procedente del receptor. Si se recibe ACK, se sabe que el paquete mas recientemente transmitido ha sido recibido

correctamente, y el protocolo vuelve a la espera de datos. Si recibe NAK, el protocolo retransmite el ultimo paquete y espera a recibir un nuevo paquete de reconocimiento en respuesta al paquete retransmitido.

El receptor solo tiene un único estado. Cuando llega un paquete, responde con un reconocimiento positivo o negativo, según si el paquete recibido es correcto o esta corrompido.

Lo mas complicado es como puede recuperarse el protocolo de los errores en los paquetes ACK o NAK. Si un paquete ACK o NAK esta corrompido, el emisor no tiene forma de saber si el receptor ha recibido o no correctamente el ultimo fragmento de datos transmitido.

Entonces el emisor numera sus paquetes de datos colocando un numero de secuencia en este campo. Entonces solo bastara con que el receptor compruebe ese número.

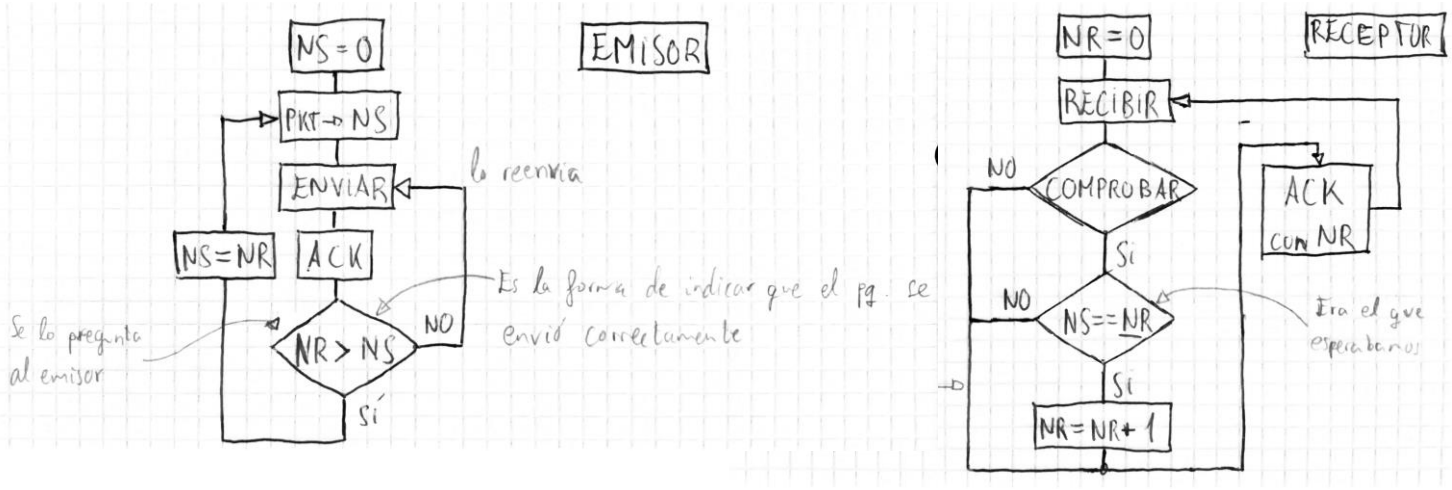
3. TRANSFERENCIA DE DATOS FIABLE SOBRE UNA CANA CON PERDIDAS Y ERRORES DE BIT: rdt3.0.

Ahora hay dos problemas más: cómo detectar la perdida de paquetes y que hacer cuando se pierde un paquete.

Para el primero hay que añadir un nuevo método de protocolo. El emisor selecciona un intervalo de tiempo en el cual es probable que el paquete se ha perdido. Si dentro de ese intervalo no se recibe ningún ACK entonces el paquete se retransmite. Hay posibilidad de que existan paquetes de datos duplicados en el canal emisor-receptor (rdt2.2).

Esto requiere un temporizador de cuenta atrás que pueda interrumpir al emisor. El emisor necesitara poder iniciar el temporizador cada vez que envíe un paquete, responder a una interrupción del temporizador y detener el temporizador.

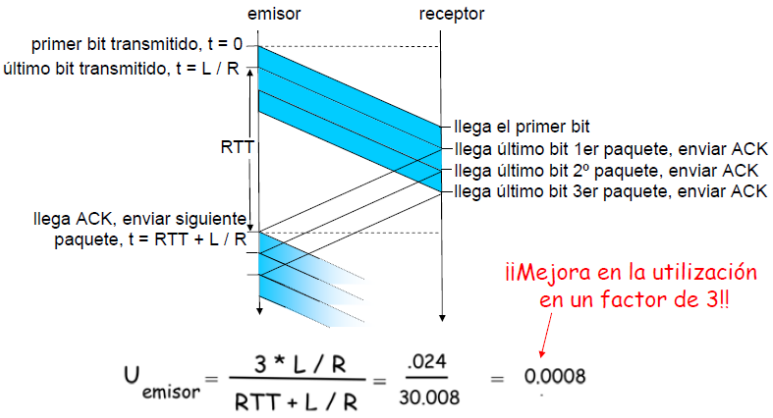
NS es el nº de secuencia y **NR** es el nº de reconocimiento, lo utilizamos para distinguir las situaciones de llegada correcta o incorrecta. **NR≥NS**



Si le añadimos temporizador al emisor entre enviar y ACK si el temporizador se agota, entonces vuelve a enviar.

Protocolos segmentados.

Los protocolos segmentados sirven para conseguir un rendimiento mas alto, permitiendo que haya varios paquetes a lo largo del canal sin esperar el reconocimiento. Obtiene un rendimiento n veces mayor.



Retroceder N (GBN): Permite enviar un máximo de N paquetes antes de tener que esprar el reconocimiento. El receptor solo da por buenos los paquetes que lleguen en el orden y rechaza los demás. El emisor va a aceptar reconocimientos acumulativos. Tiene un temporizador para el paquete mas antiguo sin ACK, si llega a 0, retransmite los paquetes sin ACK.

NS_{min}: nº de secuencia mas bajo de paquete no reconocido.

NS_{max}: nº de secuencia mas alto.

N: nº máximo de paquetes.

$NS_{max} - NS_{min} \leq N$

Ante la llegada de un NR el emisor actualiza **NS_{min} : NS_{min}** ← NR (siempre que NR>NS)

Window size (desplazamiento de venatana): Cuando van llegando reconocimientos los nº de secuencia menores quedan reconocidos y por tanto tenemos mas hueco (se desliza). Cuando expira el temporizador (se produce un perdida), el emisor repite la emisión en orden, es decir, reenvia todos.

$$NSmin \leq NR \leq NSmin + N \qquad 0 \leq NR - NSmin \leq N$$

Repetición Selectiva (SR).

Cada vez que manda un paquete activa el temporizador. Cuando se pierde un paquete envía el paquete del temporizador que haya expirado (de esta manera es selectiva). Este protocolo mejora notablemente el uso del ancho de banda.

Entrega datos a la aplicación una serie de paquetes consecutivos en orden y correctos. Este protocolo necesita más memoria, ya que los almacena antes de enviarlos (posteriormente se libera).

Transporte orientado a conexión TCP.

TCP visión global.

Punto a punto: un emisor y un receptor.

Flujo de bytes fiable, en orden: no hay limites de mensajes.

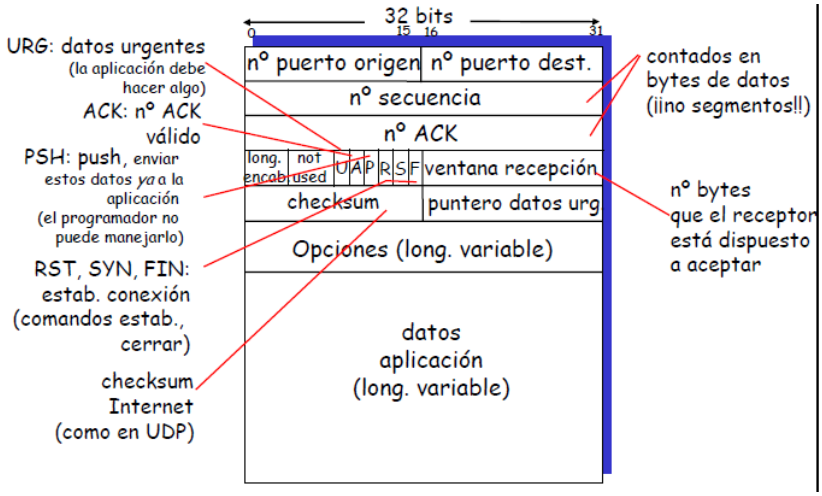
Segmentado: El control de flujo y congestion de TCP fijan el tamaño de la ventana y hay buffers de emisión y recepción.

Datos full dúplex: Flujo de datos bidireccional en la misma conexión y MSS (máximo tamaño de segmento).

Orientado a conexión: Establecimiento conexión (intercambio de mensajes) inicializa estados antes del intercambio de datos.

Con control de flujo: El emisor no satura al receptor.

Estructura del segmento TCP.



Los **nº secuencia** en TCP son el nº de flujo de bytes del primer byte de datos del segmento.

Los **ACKs** son el nº de secuencia del siguiente byte esperado de la otra parte. ACK acumulativo.

La especificación de TCP no especifica como se tratan lso segmentos fuera de orden.

TCP decide de que tamaño van a ser los paquetes. El nº de secuencia es mayor (estrictamente creciente), pero no necesariamente de 1 en 1.

$$NS(i+1) = NS(i) + datos(i) = NR(i)$$

Gestión de la conexión.

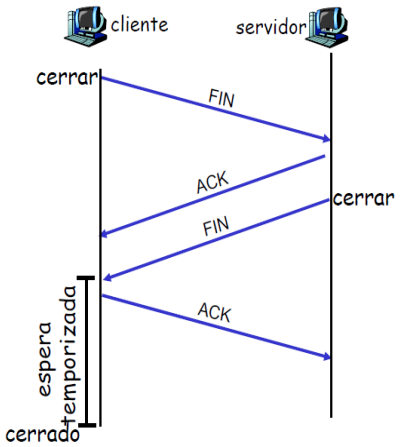
Establecimiento en 3 pasos:

1. El cliente envía segmento SYN al servidor, especifica nº secuencia inicial y no tiene datos.
2. El servidor recibe SYN, responde con segmento SYNACK. El servidor crea buffers y especifica el nº secuencia inicial del servidor.
3. El cliente recibe SYNACK y responde con segmento ACK, que puede contener datos.

Cerrar una conexión:

El cliente cierra el socket: *clientSocket.close()*.

1. El sistema terminal del cliente envía el segmento de control TCP FIN al servidor.
2. El servidor recibe FIN, responde con ACK. Cierra la conexión y envía FIN.
3. El cliente recibe FIN, responde con ACK. Este entra en “espera temporizada” y responderá con ACK a los FINs que reciba.
4. El servidor recibe ACK. Conexión cerrada.



TCP transferencia de datos fiables.

TCP crea servicio rdt sobre el servicio no fiable de IP. Tiene segmentos en cadena, ACKs acumulativos, y TCP usa un único temporizador de retransmisión.

Las retransmisiones pueden ser diparadas por eventos de temporizador a cero o ACKs duplicados.

Inicialmente considera el emisor TCP simplificado. Ignora ACKs duplicados, el control de flujo y la congestion de flujo.

Eventos de emisión TCP.

Datos recibidos de la aplicación: Crea el segmento con NS, donde NS es el nº del primer byte del segmento dentro del flujo de bytes. Inicia el temporizador si no lo esta y recibe el intervalo de expiración TimeOutInterval.

Timeout (expiración): Retransmite el segmento que la provoco y reinicia el temporizador.

ACK recibido: Si se refiere a segmentos sin ACK previo, actualiza aquellos a los que les falte el ACK. Inicia el temporizador si hay segmentos pendientes.

Generación de ACK en TCP.

Evento en Receptor	Receptor TCP: acción
Llegada de segmento en orden con NS esperado. Todos hasta el NS esperado ya tienen ACK	ACK retardado. Esperar hasta 500ms al siguiente segmento. Si no llega, enviar ACK
Llegada de segmento en orden con NS esperado. Hay otro segmento en orden esperando transmisión de ACK	Inmediatamente enviar ACK acumulativo para ambos segmentos.
Llegada de NS fuera de orden mayor que el esperado. Detectada laguna	Inmediatamente enviar ***ACK duplicado*** indicando NS del siguiente byte esperado
Llegada de segmento que complete parcialmente una laguna	Inmediatamente enviar ACK, suponiendo que el segmento empieza en el límite inferior de la laguna

Retransmisión Rápida.

El periodo de expiración a menudo es relativamente largo debido al largo retardo antes de reenviar el paquete perdido.

El emisor a menudo envía varios segmentos seguidos, si se pierde un segmento seguramente habrá varios ACKs repetidos. Si el emisor recibe 3 ACKs por los mismos datos, supone que el segmento de después de los daos con ACK se perdió.

La **retransmisión rápida** consiste en reenviar el segmento antes de que expire el temporizador.

Algoritmo de Nagle

Las conexiones interactiva (ssh, telnet) suelen enviar segmentos con muy pocos datos (uno o dos bytes). Perdida de eficiencia. Es mas interesante reunir varios datos procedentes de la aplicación y mandarlos todos juntos.

El algoritmo de Nagle indica que no se envíen nuevos segmentos mientras queden reconocimientos pendientes.

Evento en Emisor	Acción en emisor
Llegada de datos de la aplicación. Hay ACKs pendientes	Acumular datos en el buffer del emisor
Llegada de un ACK pendiente	Inmediatamente enviar todos los segmentos acumulados en el buffer
Llegada de datos de la aplicación. No hay ACKs pendientes	Inmediatamente enviar datos al receptor
Llegada de datos de la aplicación. No queda sitio en el buffer del emisor.	Inmediatamente enviar datos si lo permite la ventana, aunque no se hayan recibido los ACKs previos

TCP: Control de flujo.

El emisor no saturara el buffer del receptor (en TCP el receptor tiene un buffer de recepción) a base de enviar mucho y muy seguido. El servicio equilibrado de velocidad es equilibrar la velocidad de envio a la de aplicación vaciando el buffer de recepción.

Ventana de recepción: nº de bytes no leidos. Va a ser enviada con cada reconocimiento de los bytes recibidos.

$$RcvWindow = RcvBuffer - (LastByteRcvd - LastByteRead)$$

El receptor anuncia el espacio libre incluyendo el valor *RcvWindow* en los segmentos y limita los datos sin ACK a *RcvWindow*, asi garantiza que el buffer de recepción no se desborda.

Si el receptor no tiene nada que reconocer, no hay forma de informar al emisor → bloqueo.

Para solucionar esto existe un "informador (temporizador) persistente".

Si la ventana = 0. El emisor tiene un temporizador persistente, manda periódicamente un byte (para evitar la situación de bloqueo, este byte no se da por recibido , siempre se descarta). Y nunca para.

El **síndrome de la ventana tonta** se produce al mandar mensajes de petición cuando una ventana tiene espacio disponible, aunque sea muy pequeño (siempre que sea posible). Se puede comparar a llamar a un tráiler para llevar una caja de zapatos.

En el receptor se envía cuando el tamaño es MSS o ½buffer

En el emisor se envía cuando el tamaño es MSS o ½buffer. El algoritmo de Nagle intenta juntar todos los datos posibles y los envía solo cundo hay un reconocimiento por parte del receptor (ACK).

TCP: 'Round Trip Time' y Timeout

Timeout: Si el timeout tiene un tiempo muy corto puede suponer que expire antes de que lleguen otros reconocimientos. Esto también supone que se reenvíen paquetes que se podrían haber enviado (es poco efectivo) ya que cuando expira se reenvían. Si el tiemout es demasiado largo la reacción a perdidas es lenta.

SimpleRTT: Tiempo medio desde transmisión de un segmento hasta recepción de ACK.

$$EstimatedRTT = (1 - \alpha) * EstimatedRTT + \alpha * SampleRTT$$

El valor típico asignado a α es 0.125, este valor oscila entre 0 (no cambia) y 1 (a la estimación anterior no le damos ningún peso). Este valor nos dice el peso que le damos a las estimaciones anteriores.

¡Esto no se hace para todos los paquetes, solo para algunos. Aumentamos el valor de la estimación como método de protección!. Cuanto mas EstimatedRTT mas margen de seguridad.

$$DevRTT = (1 - \beta) * DevRTT + \beta * |SampleRTT - EstimatedRTT| (tipicamente, \beta = 0.25)$$

$$TimeoutInterval = EstimatedRTT + 4 * DevRTT$$

Algoritmo de Karn

Si recibimos el reconocimiento de un paquete retransmitido, no tenemos forma de saber a cual de las retransmisiones corresponde ese reconocimiento. Por ello, se ignoran los paquetes retransmitidos a la hora de computar el RTT y solo se atienden a los que no hayan sido transmitidos.

Principios de control de congestion.

La congestion es cuando hay demasiadas fuentes enviando demasiados datos demasiado deprisa para que la red lo pueda asimilar. Es diferente al control de flujo.

Control de congestion TCP.

1. Percibir la congestion.
2. Como responder: limitar la velocidad.
3. Como limitamos la velocidad.

En el TCP tradicional no tenemos información de la capa de red, por tanto, no podemos conocer cuan cerca de la congestion estamos (de producirla). Esta información la recibimos de manera indirecta cuando empiezan a aparecer perdidas. Hay dos tipos de congestiones, **la congestion suave**, cuando se recibe un triple ACK y la **congestion grave**, cuando el tiempo del temporizador expira.

Limitamos la velocidad en forma de cantidad de segmentos que mandamos (este lo controla la ventana de emisión).

La **ventana de recepción** es controlada por el receptor, pero es una variable del emisor. Y la **ventana de congestion** cuyo valor va a estar controlado por el emisor, también es una variable del emisor.

La cantidad de elementos que puede enviar será el minimo de esas dos ventanas.

$$Ventana\ emisor = \min(VR, VC)$$

Arranque lento. (VC=cwnd)

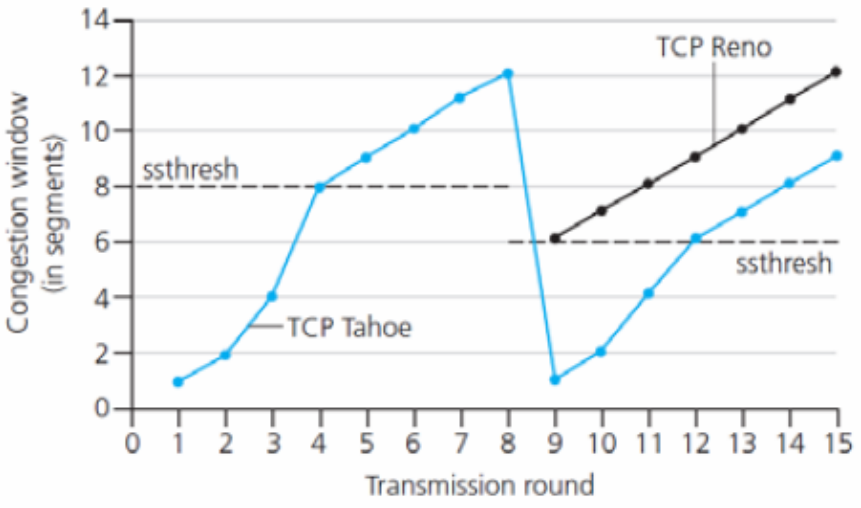
3 ACK→VC=VC/2 y Umbral=VC.

EXP→Umbral=VC/2 y VC=1

$$tasa = \frac{cwdn}{RTT} Bytes/s$$

Para arranque lento inicialmente cwdn=1 MSS, este se dobla cada RTT. Se incrementa con cada ACK recibido.

Tras 3 ACKs duplicados cwnd se divide por 2.



Tras una expiración cwnd se pone a 1 y la ventana crece exponencialmente hasta un umbral, luego linealmente.

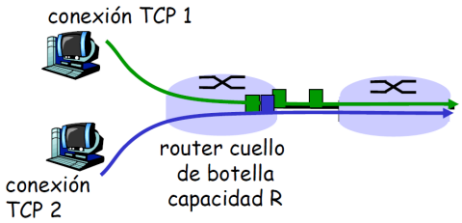
Eficiencia TCP.

Siendo W el tamaño de la ventana cuando ocurre una perdida. Cuando la ventana es W, la tasa es $\frac{W}{RTT}$. Justo tras la perdida, la ventana pasa a $\frac{W}{2}$ y la tasa a $\frac{W}{2RTT}$. La tasa media es: 0.75 W/RTT.

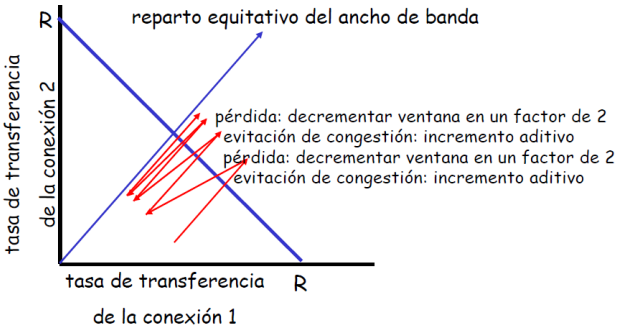
$$Tasa\ de\ transferencia\ en\ funcion\ de\ la\ tasa\ de\ perdidas: 1.22 * \frac{MSS}{RTT\sqrt{L}}$$

Equidad en TCP.

Su objetivo es: si K sesiones TCP comparten el mismo enlace cuello de botella de ancho de banda R, cada uno debería tener una tasa media de $\frac{R}{K}$.



¿Por qué es equitativo TCP?.



Hay 2 sesiones que compiten:

- El incremento aditivo da una pendiente de 1 en los incrementos.
- El decremento multiplicativo decrementa la tasa proporcionalmente.

FÓRMULAS REDES

$NS_{max} - NS_{min} \leq N$ nº max paquetes

$$\vec{V} = \frac{W + (W+1) + (W+2) + \ldots + 2W}{W \cdot RTT} \cdot MSS \simeq \frac{(W+2W)\cdot W}{2W \cdot RTT} \cdot MSS = \boxed{\frac{3}{2} W \cdot \frac{MSS}{RTT}}$$

velocidad promedio del trozo repetido

2W + tamaño de la ventana cuando ocurre una pérdida

$$\vec{V} = \frac{\text{nº total bytes transmitidos}}{\text{tiempo en transmitirlos}}$$

$$\text{tasa perdidos} = \frac{\text{nº paquetes perdidos}}{\text{nº paquetes transmitidos}}$$

$$V_{inst} = \frac{W \cdot MSS}{RTT}$$

$$NS(i+1) = NS(i) + datos(i) = NR(i)$$

$$L = MSS + H$$ (o cabeceras)

Velocidad promedio: $\vec{V} = \frac{L}{2T_p + \frac{L}{R}}$

Paquete L bits

Transmisor R bits/s

$RTT = 2T_p$

Eficiencia: $E = \frac{T_t}{RTT + T_t}$

$RTT = Clk - T_{secr}$

$TIME\_WAIT = 2 \cdot MSS$

Tiempo hasta que desaparece de la red

Tiempo medio desde transmisión de un segmento hasta recepción de ACK

$$EstimatedRTT = (1-\alpha) \times EstimatedRTT + \alpha \cdot SampleRTT$$

$0 \leq \alpha \leq 1$ → la estimación anterior sin peso

α: no cambia; $\alpha = 0{,}125$

$$DevRTT = (1-\beta) \times DevRTT + \beta \times |SampleRTT - EstimatedRTT|$$

(típicamente $\beta = 0{,}125$)

$$TimeOutInternet = EstimatedRTT + 4 \times DevRTT$$

$$L = \frac{1}{\frac{3W^2}{2}} = \frac{2}{3W^2} \Rightarrow W = \sqrt{\frac{2}{3}} \cdot \frac{1}{\sqrt{L}}$$

$$tasa = \frac{W \cdot d}{RTT} \text{ Bytes/s}$$

tasa transf. en func. tasa pérdidas $= \frac{1{,}22 \cdot MSS}{RTT \cdot \sqrt{L}}$

$$V = \frac{3}{2}\sqrt{\frac{2}{3}} \cdot \frac{1}{\sqrt{L}} \cdot \frac{MSS}{RTT} = \sqrt{\frac{3}{2}} \frac{MSS}{RTT \cdot \sqrt{L}} \Rightarrow \boxed{\frac{1{,}22 \cdot MSS}{RTT\sqrt{L}}}$$

$$\% = \frac{T_t}{RTT}$$

R = Velocidad transmisor

F = tamaño fichero $\Rightarrow V = \frac{F}{T_{total}}$

$N \leq \left\lfloor \frac{N}{MSS} \right\rfloor + 1 \rightarrow F \,/\!/\, MSS + H$