Capítulo 3
La capa de transporte

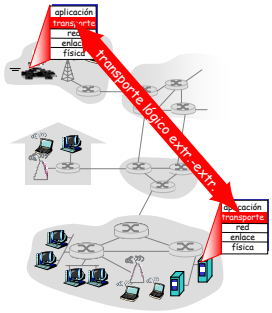
Redes de computadoras: Un enfoque descendente, 5ª edición.
Jim Kurose, Keith Ross
Pearson Educación, 2010.

A note on the use of these ppt slides:
We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:
- If you use these slides (e.g., in a class) in substantially unaltered form, that you mention their source (after all, we'd like people to use our book!)
- If you post any slides in substantially unaltered form on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

servicios y protocolos de transporte

- proporcionar *comunicación lógica* entre procesos en ejecución en diferentes hosts
- los protocolos de transporte corren en sistemas terminales
  - emisor: divide mensajes en segmentos, los pasa a la capa de red
  - receptor: reensambla segmentos en mensajes, los pasa a la capa de aplicación
- más de un protocolo disponible para las aplicaciones
  - Internet: TCP y UDP



Capítulo 3: La capa de transporte

Objetivos:

- comprender los principios que están tras los servicios de la capa de transporte
  - multiplexar/des-multiplexar
  - transferencia de datos fiable
  - control de flujo
  - control de congestión
- conocer los protocolos de transporte de Internet:
  - UDP: transporte sin conexión
  - TCP: transporte orientado a conexión
  - control de flujo TCP
  - control de congestión TCP

capa de transporte / capa de red

- *encapsulación:* arquitectura en capas
- *capa de red:* comunicación lógica entre hosts
- *capa de transporte:* comunicación lógica entre procesos
  - se basa en, y amplía, los servicios de la capa de red

analogía doméstica:

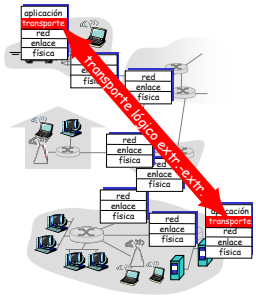
12 chicos envían cartas a 12 chicos

- procesos = chicos
- mensajes = cartas en sobres
- hosts = casas
- protocolo de transporte = Ana y Juan, que reparten a sus hermanos respectivos
- protocolo de red = Correos

Capítulo 3: índice

protocolos de capa de transporte de Internet

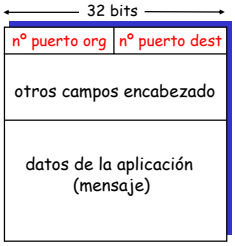
- distribución fiable en orden (TCP)
  - control de congestión
  - control de flujo
  - establecimiento de conexión
- distribución no fiable, fuera de orden: UDP
  - extensión "sin virguerías" de IP "haz lo que puedas"
- servicios no disponibles:
  - garantía de retardo mínimo
  - garantía de ancho de banda mínimo



Capítulo 3: índice

Cómo funciona la desmultiplexación

- el host recibe datagramas IP
  - cada datagrama tiene IP de origen e IP de destino
  - cada datagrama lleva un segmento de la capa de transporte
  - cada segmento tiene nº de puerto de origen y de destino
- el host usa IP y nº de puerto para dirigir el segmento al socket apropiado



formato de segmento TCP/UDP

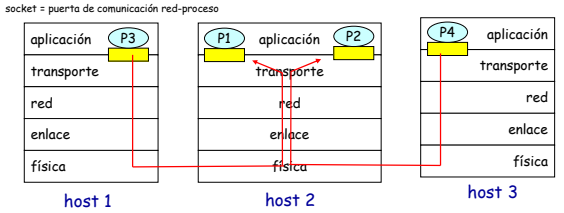
Multiplexación/desmultiplexación

Desmultiplexación en el destino:
entregar segmentos recibidos al socket correcto

Multiplexación en el emisor:
reunir datos de múltiples sockets, empaquetarlos con el encabezado (usado luego para desmultiplexar)

= socket = proceso

socket = puerta de comunicación red-proceso



desmultiplexación sin conexión

- *recordatorio:* crear sockets con números de puerto locales:

```
DatagramSocket mySocket1 = new
   DatagramSocket(12534);
DatagramSocket mySocket2 = new
   DatagramSocket(12535);
```

- *recordatorio:* al crear un datagrama para enviar por un socket UDP, hay que especificar

(IP dest ,nº puerto dest)

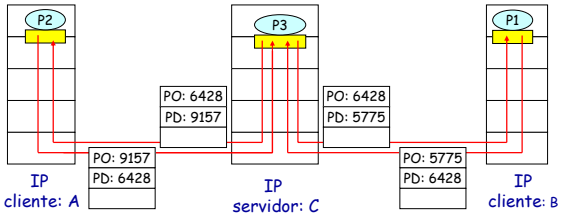
- cuando un host recibe un segmento UDP
  - comprueba el nº de puerto destino del segmento
  - redirige el segmento UDP al socket con ese nº de puerto
- datagramas IP con diferente IP origen y/o nº puerto origen se dirigen al mismo socket

Protocolo de red IP

- El protocolo de Internet para la capa de red se llama **IP**.
- Se encarga de dar una conexión lógica entre hosts.
- Entrega datagramas de un host a otro, pero sin garantías.
- Cada host se identifica con una dirección de red, que llamamos dirección IP.

desmultiplexación sin conexión (cont)

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```

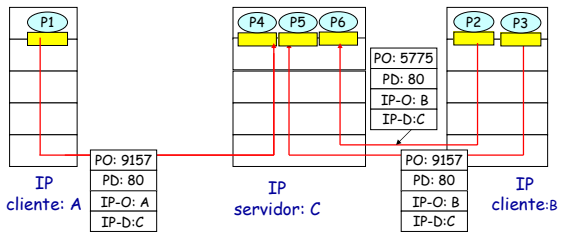


PO proporciona "dirección de retorno"

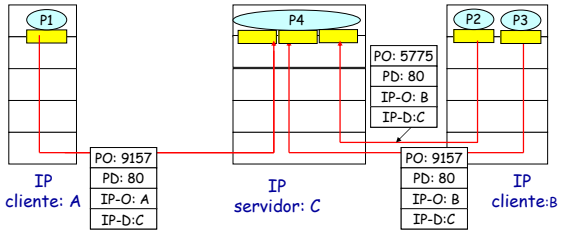
Desmultiplexación orientada a conexión

- un socket TCP se identifica por una 4-upla:
  - IP origen
  - nº puerto origen
  - IP destino
  - nº puerto destino
- el receptor usa los 4 valores para redirigir el segmento al socket adecuado
- el host servidor debe soportar varios sockets TCP simultáneos
  - cada socket identificado por su propia 4-upla
- los servidores web tienen sockets diferentes para cada cliente que se conecta
  - HTTP no persistente tendrá un socket para cada solicitud

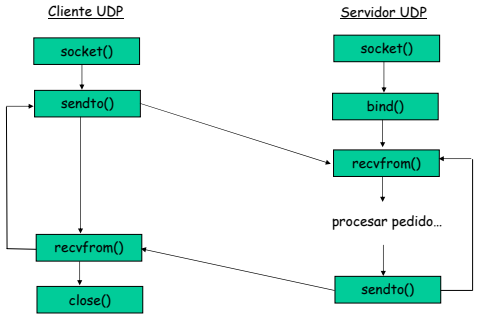
Desmultiplexación orientada a conexión (cont)



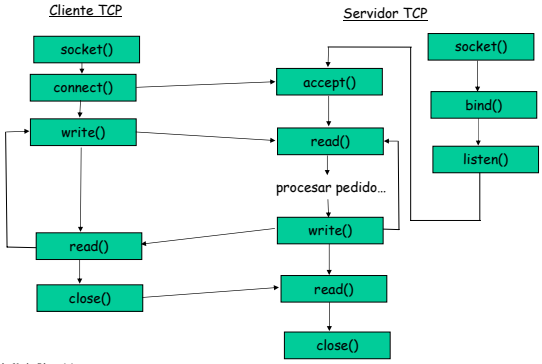
desmultiplexación orientada a conexión: Web Server con hebras



Sockets en cliente/servidor UDP



Sockets en cliente/servidor TCP



Capítulo 3: índice

UDP: User Datagram Protocol [RFC 768]

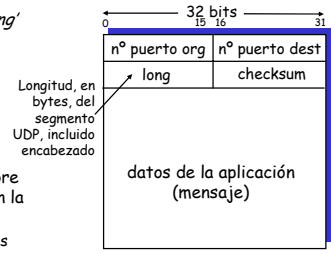
- protocolo de transporte de Internet sin adornos, "con lo puesto"
- al ser un servicio de "haz lo que puedas", los segmentos UDP pueden:
  - perderse
  - ser entregados fuera de orden a la aplicación
- *sin conexión:*
  - sin establecimiento de conexión entre el emisor y el receptor UDP
  - cada segmento UDP se trata de forma independiente de los otros

¿Por qué existe UDP?

- no hay establecimiento de la conexión (que puede añadir retardo)
- sencillo: no hay estado ni en el emisor ni en el receptor
- encabezado pequeño
- no hay control de congestión: UDP puede disparar todo lo rápido que se quiera

UDP: más

- a menudo usado para aplicaciones de *'streaming'* multimedia
  - tolerante a pérdidas
  - sensible a la velocidad
- otros usos de UDP
  - DNS
  - SNMP
- transferencia fiable sobre UDP: añadir fiabilidad en la capa de aplicación
  - recuperación de errores específica para la aplicación



formato de segmento UDP

UDP: checksum

Objetivo: detectar "errores" (p.ej.: bits alterados) en el segmento transmitido

Emisor:

- trata contenidos del segmento como secuencia de enteros de 16 bits
- checksum: suma (en compl. a 1) del contenido del segmento
- el emisor pone el checksum en el campo UDP correspondiente

Receptor:

- calcula el checksum del segmento recibido
- comprueba si el valor calculado = campo checksum
  - NO - error detectado
  - SÍ - error no detectado

¿Puede haber errores aun así? Lo veremos más adelante

Ejemplo de Checksum Internet

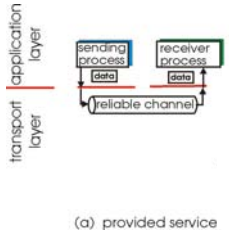
- Nota: isuma en complemento a 1!
- Ejemplo: sumar dos enteros de 16 bits

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
acarreo	(1)	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
suma		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Capítulo 3: índice

Principios de transferencia de datos fiable

- es importante en las capas de aplicación, transporte y enlace
- ¡en el "top-10" de las cuestiones importantes en redes!

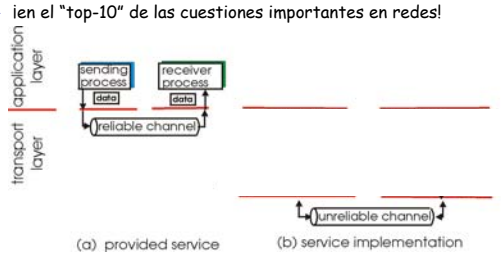


(a) provided service

- las características del canal no fiable determinarán la complejidad del protocolo de transferencia de datos fiable (rdt: *'reliable data transfer protocol'*)

Principios de transferencia de datos fiable

- es importante en las capas de aplicación, transporte y enlace
- ien el "top-10" de las cuestiones importantes en redes!



- las características del canal no fiable determinarán la complejidad del protocolo de transferencia de datos fiable (rdt: *'reliable data transfer protocol'*)

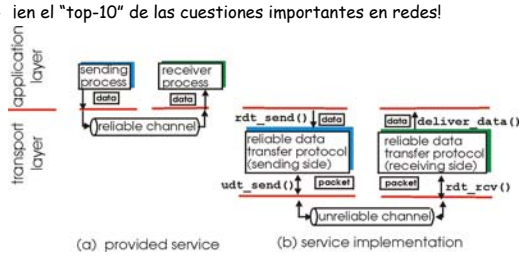
transferencia de datos fiable: preliminares

Vamos a:

- desarrollar el emisor y el receptor de un protocolo de transf. de datos fiable (rdt) paso a paso
- considerar sólo transferencia de datos unidireccional
  - ipero el control se transmitirá en ambas direcciones!
- usar máquinas de estados finitos (MEFs) para definir emisor y receptor

Principios de transferencia de datos fiable

- es importante en las capas de aplicación, transporte y enlace
- ien el "top-10" de las cuestiones importantes en redes!

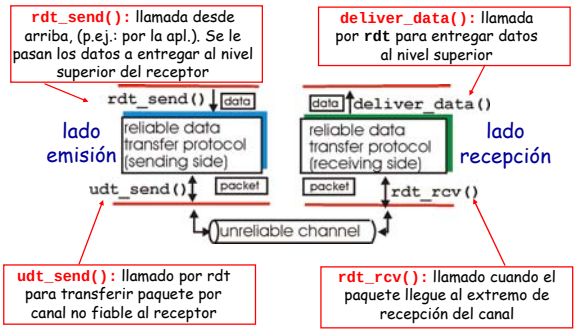


- las características del canal no fiable determinarán la complejidad del protocolo de transferencia de datos fiable (rdt: *'reliable data transfer protocol'*)

Rdt1.0: transferencia fiable en canal fiable

- canal subyacente perfectamente fiable
  - no hay errores de bit
  - no hay pérdida de paquetes
- MEF diferente para emisor y receptor
  - el emisor envía dato al canal subyacente
  - el emisor lee datos del canal subyacente

transferencia de datos fiable: preliminares



Rdt2.0: canal con errores de bit

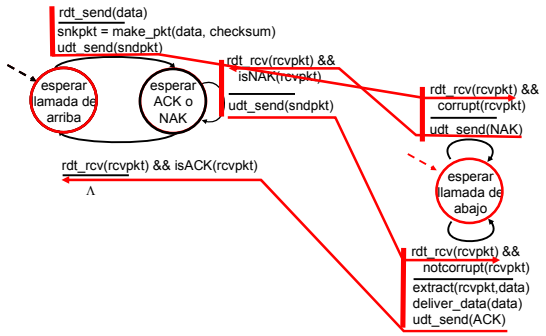
- el canal subyacente puede alterar bits del paquete
  - usar el checksum para detectar errores de bit
  - *la cuestión*: ¿cómo recuperarse de errores?

¿Cómo se recuperan los humanos de "errores" durante la conversación?

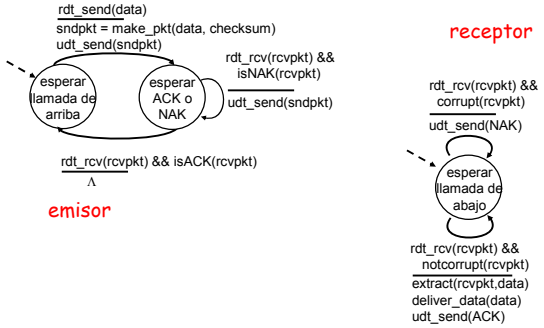
Rdt2.0: canal con errores de bit

- el canal subyacente puede alterar bits del paquete
  - usar el checksum para detectar errores de bit
  - *la cuestión*: ¿cómo recuperarse de errores?
  - *reconocimientos('acknowledgements', ACKs):* el receptor indica explícitamente que la recepción fue buena
  - *reconocimientos negativos (NAKs):* el receptor indica explícitamente que el paquete tenía errores
  - el emisor retransmite el paquete si recibe un NAK
- nuevos mecanismos en **`rdt2.0`** (sobre **`rdt1.0`**):
  - detección de errores
  - realimentación del receptor:mensajes de control (ACK,NAK) del receptor al emisor

rdt2.0: caso de error



rdt2.0: especificación de la MEF



¡ rdt2.0 tiene un defecto fatal !

¿Qué ocurre si ACK o NAK se corrompen?

- ¡¡el emisor no sabe qué ocurrió en el receptor!!
- no es posible simplemente retransmitir: se pueden crear duplicados

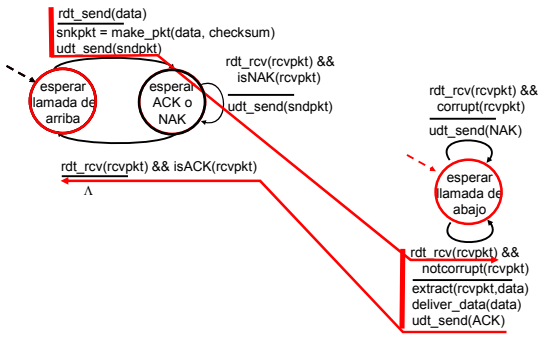
Manejo de duplicados:

- el emisor retransmite el paquete actual si ACK o NAK no llegan bien
- el emisor añade un número de secuencia a cada paquete
- el receptor descarta (no entrega hacia arriba) el paquete duplicado

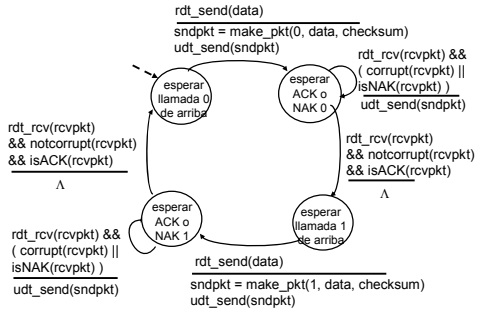
parada y espera

el emisor envía un paquete y espera la respuesta del receptor

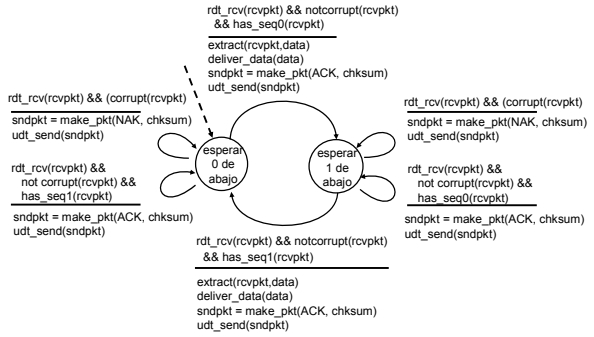
rdt2.0: funcionamiento sin errores



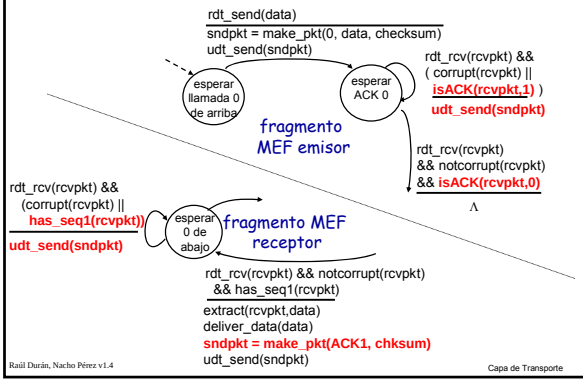
rdt2.1: el emisor maneja ACK/NAKs erróneos



rdt2.1: el receptor maneja ACK/NAKs erróneos



rdt2.2: fragmentos del emisor y el receptor



rdt2.1: discusión

Emisor:

- nº secuencia añadido al paquete
- 2 números valen (0,1). ¿Por qué?
- comprobar si el ACK/NAK recibido corrupto
- el doble de estados
  - el estado debe "recordar" si el paquete "actual" tiene nº sec. 0 ó 1

Receptor:

- debe comprobar si el paquete recibido es duplicado
  - el estado indica si se espera el paquete 0 ó 1
- nota: el receptor *no* puede saber si su último ACK/NAK se recibió bien en el emisor

rdt3.0: canales con errores y pérdidas

Nueva suposición: canal subyacente también puede perder paquetes (datos o ACKs)

- las retransmisiones de checksum, nº secuencia, ACKs, ayudan, pero no son suficiente

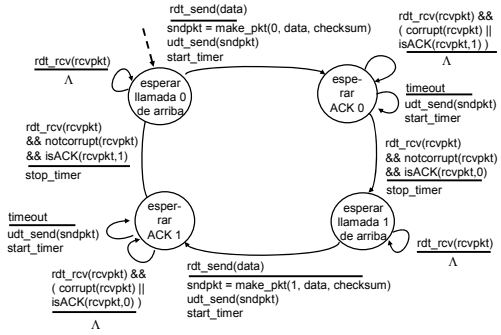
línea de trabajo: emisor espera un tiempo "razonable" al ACK

- retransmite si no recibe ACK en ese tiempo
- si el paquete (o ACK) sólo se retrasó (no se perdió):
  - las retransmisiones estarán repetidas, pero con el nº secuencia esto está resuelto
  - el receptor debe indicar nº secuencia del paquete al que se aplica el ACK
- requiere un temporizador

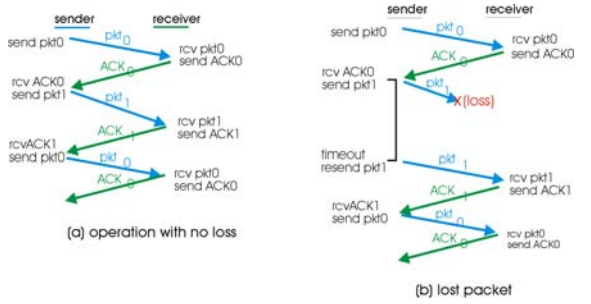
rdt2.2: un protocolo sin NAK

- la misma funcionalidad que rdt2.1, usando sólo ACKs
- en lugar de NAK, el receptor envía ACK para el último paquete recibido bien
  - el receptor debe incluir explícitamente el nº de secuencia del paquete al que se refiere el ACK
- un ACK duplicado en el emisor resulta en la misma acción que un NAK: *retransmitir paquete actual*

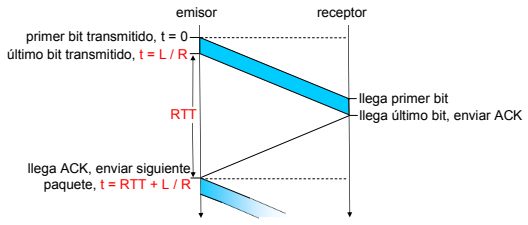
rdt3.0 emisor



rdt3.0 funcionando

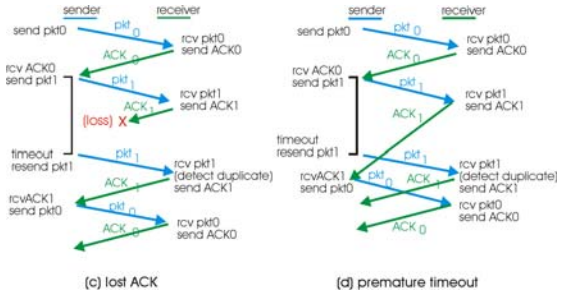


rdt3.0: funcionamiento de parada y espera



$$U_{emisor} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

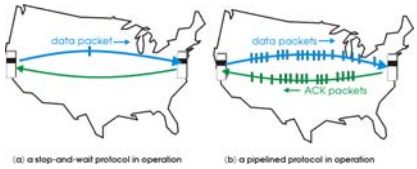
rdt3.0 funcionando



Protocolos segmentados (en cadena)

segmentar: el emisor permite que haya múltiples paquetes "en camino", pendientes de ACK

- el rango de nº de secuencia debe aumentarse
- hay que añadir búfferes en el emisor y/o el receptor



- hay dos formas genéricas de protocolos segmentados: *retroceder N ('go-Back-N') y repetición selectiva ('selective repeat')*

Rendimiento del rdt3.0

- rdt3.0 funciona, pero el rendimiento es muy malo
- p.ej.: enlace de 1Gb/s, 15ms retardo prop., paquete 8k bits

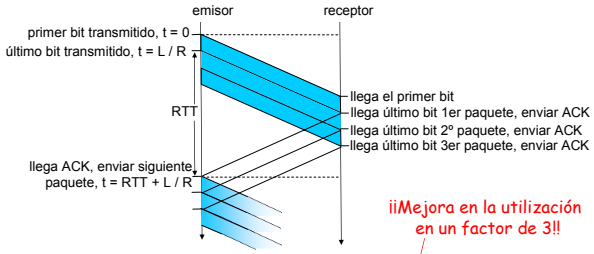
$$d_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bps}} = 8 \text{ microsegundos}$$

- U_{emisor}: utilización – fracción de tiempo que el emisor está ocupado emitiendo

$$U_{emisor} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

- si RTT=30 ms, 1 paquete de 1KB cada 30 sg -> 33KB/s de 1Gbps
- ¡¡el protocolo de red limita el uso de los recursos físicos!!

segmentación: uso mejorado



¡¡Mejora en la utilización en un factor de 3!!

$$U_{emisor} = \frac{3 * L/R}{RTT + L/R} = \frac{.024}{30.008} = 0.0008$$

Protocolos segmentados

Retroceder N: vista global

- el emisor puede tener hasta N paquetes pendientes de ACK
- el receptor sólo envía ACKs *acumulativos*
  - no lo envía para un paquete si hay una laguna
- el emisor tiene un temporizador para el paquete más antiguo sin ACK
  - si llega a 0, retransmitir paquetes sin ACK

Repetición selectiva: vista global

- el emisor puede tener hasta N paquetes pendientes de ACK
- el receptor envía ACK para cada paquete
- el emisor mantiene un temporizador para cada paquete sin ACK
  - si llega a 0, retransmitir sólo paquete sin ACK

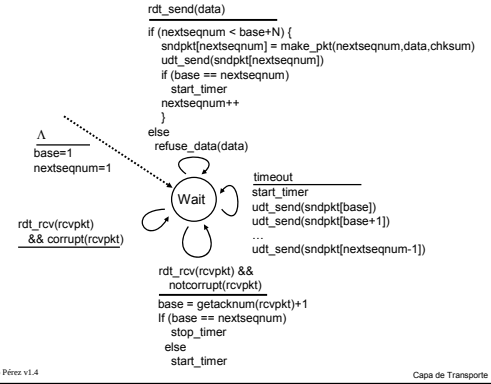
Retroceder N (GBN)

Emisor:

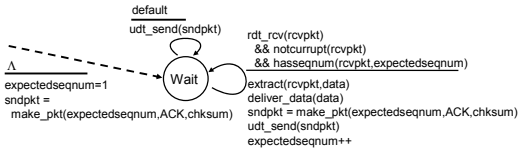
- nº de secuencia de *k* bits en cabecera del paquete
- ventana de hasta N paquetes consecutivos sin ACK

- ACK(*n*): ACK para todos los paquetes hasta nº sec. *n* (inclusive): "ACK acumulativo"
  - puede recibir ACKs duplicados (ver receptor)
- temporizador para cada paquete en camino
- timeout(*n*): retransmitir paquete *n* y todos los de mayor nº sec. en la ventana

GBN: MEF ampliada para el emisor



GBN: MEF ampliada del receptor



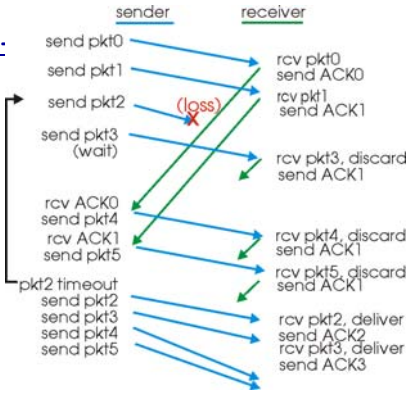
Enviar ACK para paquete correcto con mayor nº de secuencia en orden

- se pueden generar ACKs duplicados
- sólo hay que recordar **`expectedseqnum`**

paquete fuera de orden

- descartar (no se guarda) -> ¡¡no hay buffer en el receptor!!
- Reenviar ACK para paquete con mayor nº secuencia en orden

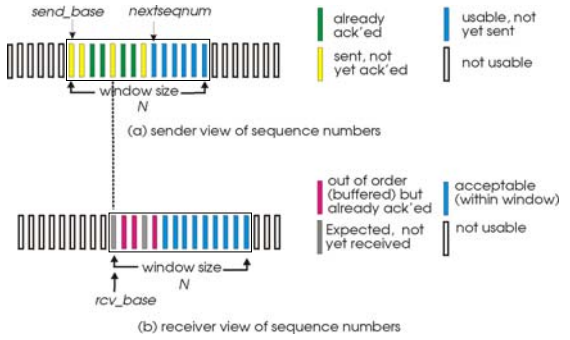
GBN en funcionamiento



Repetición selectiva (SR)

- el receptor envía ACK *individual* para cada paquete correcto
  - se deben guardar los paquetes en buffers según sea necesario, para entregarlos en orden a la capa superior
- el emisor sólo reenvía paquetes para los que no reciba ACK
  - un temporizador para cada paquete en camino
- ventana de emisor
  - hay N nº de secuencia consecutivos
  - de nuevo limita nºsec. de los paquetes en camino

Repetición selectiva: ventanas de emisor y receptor



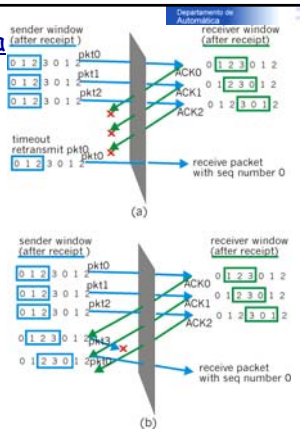
Repetición selectiva: dilema

Ejemplo:

- nos. sec: 0, 1, 2, 3
- tamaño ventana=3

- ¡el receptor no ve diferencia entre ambos casos!
- en (a) pasa datos repetidos por nuevos de forma incorrecta

P: ¿relación entre nº de nos. de secuencia válidos y tamaño de la ventana?



Repetición selectiva

emisor

datos de arriba:

- si sig. nº sec. en la ventana vacío, enviar paquete

timeout(*n*):

- reenviar paquete *n*, reiniciar temporizador

ACK(*n*) en [sendbase,sendbase+N]:

- marcar paquete *n* como recibido
- si *n* era el paquete sin ACK con menor nº sec., avanzar el inicio de la ventana al siguiente nº sec. sin ACK

receptor

paquete *n* en [rcvbase, rcvbase+N-1]

- enviar ACK(*n*)
- fuera de orden: guardar en buffer
- en orden: entregar (junto con los previamente guardados por fuera de orden), avanzar ventana al siguiente pendiente de recibir

paquete *n* en [rcvbase-N,rcvbase-1]

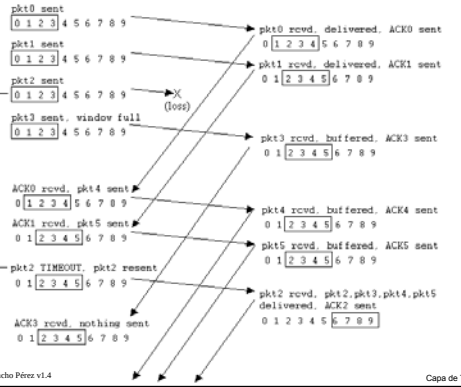
- ACK(*n*)

otro caso:

- ignorar

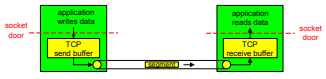
Capítulo 3: índice

repetición selectiva en funcionamiento



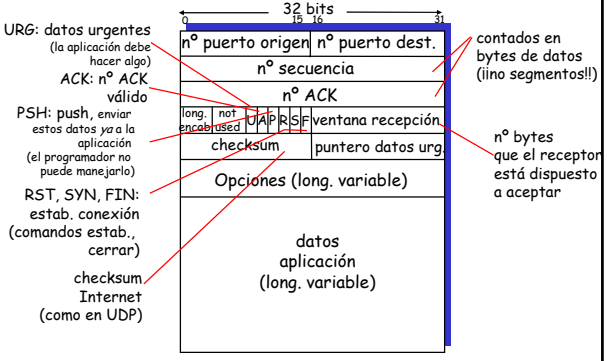
TCP: Visión global RFCs: 793, 1122, 1323, 2018, 2581

- punto a punto:
  - un emisor, un receptor
- flujo de bytes fiable, en orden:
  - no hay "límites de mensaje"
- segmentado:
  - el control de flujo y congestión de TCP fijan el tamaño de la ventana
  - *buffers de emisión y recepción*



- datos full duplex:
  - flujo de datos bidireccional en la misma conexión
  - MSS: Máximo tamaño de segmento (*'maximum segment size'*)
- 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



TCP: gestión de la conexión

Recordatorio: en TCP, emisor y receptor establecen una "conexión" antes de intercambiar segmentos de datos

- inicializar variables TCP:
  - nos. de secuencia
  - buffers, info. de control de flujo (p.ej.: **RcvWindow)**
- *el cliente:* inicia la conexión

```
Socket clientSocket = new
Socket("nombrehost","numero
puerto");
```

- *el servidor:* el cliente contacta con él

```
Socket connectionSocket =
welcomeSocket.accept();
```

Establecimiento en 3 pasos:

Paso 1: el cliente envia segmento SYN al servidor
- especifica nº secuencia inicial
- sin datos

Paso 2: el servidor recibe SYN, responde con segmento SYNACK
- el servidor crea buffers
- especifica el nº sec. inicial del servidor

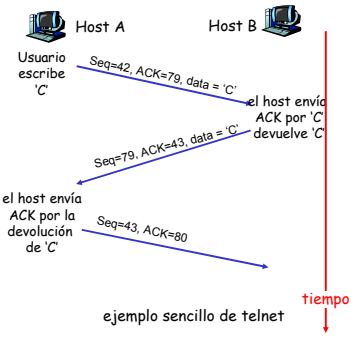
Paso 3: el cliente recibe SYNACK, responde con segmento ACK, que puede contener datos

TCP: núms. secuencia y ACKs

nos. secuencia:
- "nº" flujo de bytes del 1er byte de datos del segmento

ACKs:
- nº sec. del siguiente byte esperado de la otra parte
- ACK acumulativo

P: cómo se tratan los segmentos fuera de orden
- R: la especificación de TCP no lo dice: lo que haga el implementador



TCP: gestión de la conexión (cont.)

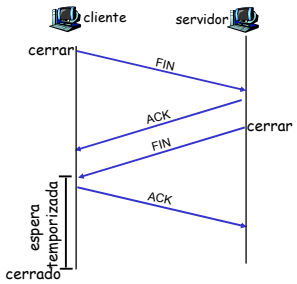
Cerrar una conexión:

el cliente cierra el socket:

```
clientSocket.close();
```

Paso 1: el sistema terminal del cliente envía el segmento de control TCP FIN al servidor

Paso 2: el servidor recibe FIN, responde con ACK. Cierra la conexión, envía FIN



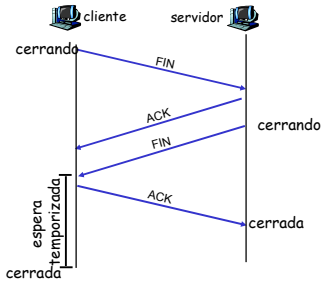
Capítulo 3: índice

TCP: gestión de la conexión (cont.)

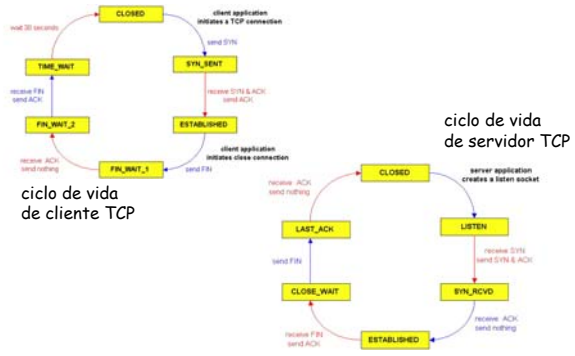
Paso 3: el cliente recibe FIN, responde con ACK.
- entra en "espera temporizada" – responderá con ACK a los FINs que reciba

Paso 4: el servidor recibe ACK. Conexión cerrada.

Nota: con una pequeña modificación, puede manejar FINs simultáneos.



TCP: gestión de la conexión (cont.)



Capítulo 3: índice

3.1 Servicios de la capa de transporte
3.2 Multiplexación y desmultiplexación
3.3 Transporte sin conexión: UDP
3.4 Principios de transferencia de datos fiable
3.5 Transporte orientado a conexión: TCP
- estructura de segmento
- gestión de conexión
- transferencia de datos fiable
- control de flujo
- estimación de RTT y temporización

3.6 Principios de control de congestión
3.7 Control de congestión TCP

TCP transferencia de datos fiable

- TCP crea servicio **rdt** sobre el servicio no fiable de IP
- segmentos en cadena
- **acks** acumulativos
- TCP usa un único temporizador de retransmisión
- retransmisiones disparadas por:
  - eventos de temporizador a cero
  - ACKs duplicados
- inicialmente considerar emisor TCP simplificado
  - ignorar ACKs duplicados
  - ignorar control de flujo, congestión de flujo

eventos de emisión TCP:

datos recibidos de la aplicación:
- crear segmento con nº sec.
- nº sec. es el nº del 1er byte del segmento dentro del flujo de bytes
- iniciar temporizador si no lo está
- intervalo de expiración: `TimeOutInterval`

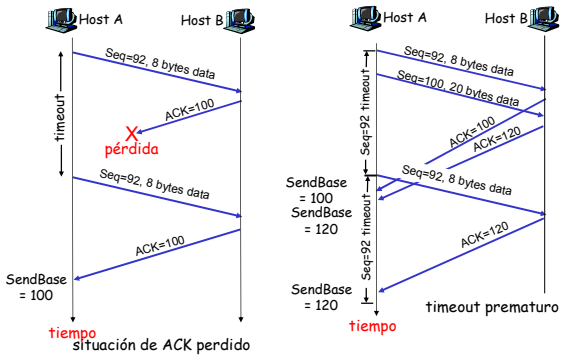
'timeout' (expiración):
- retransmitir segmento que la provocó
- reiniciar temporizador

ACK recibido:
- si se refiere a segmentos sin ACK previo
  - actualizar aquellos a los que les falta el ACK
  - iniciar temporizador si hay segmentos pendientes

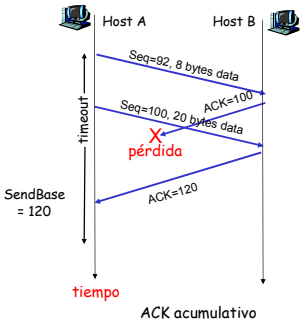
emisor TCP (simplificado)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (siempre) {
  switch(suceso)

  suceso: datos recibidos de la aplicación de capa superior
     crear segmento TCP con nº sec. NextSeqNum
     if (temporizador no en marcha)
          iniciar temporizador
     pasar segmento a IP
     NextSeqNum = NextSeqNum + length(data)

  suceso: temporizador expiró
     retransmitir segmento sin ACK con el menor nº sec.
     iniciar contador

  suceso: recibido ACK con valor del campo ACK == y
     if (y > SendBase) {
          SendBase = y
          if (hay segmentos sin ACK)
               iniciar temporizador
     }

} /* fin de loop siempre */
```


TCP: situaciones para retransmisión



TCP situaciones para retransmisión (más)



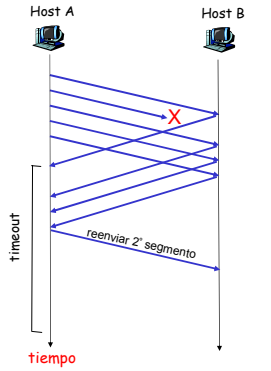


Figura 3.37 Reenviar segmento tras triple ACK

generación de ACK en TCP [RFC 1122, RFC 2581]

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

Algoritmo de retransmisión rápida:

```
suceso: recibido ACK, con campo ACK con valor == s
            if (s > SendBase) {
                SendBase = s
                if (hay segmentos pendientes de ACK)
                    iniciar temporizador
            }
            else {
                incrementar cuenta de ACKs duplicados recibidos para s
                if (cuenta de ACKs duplicados para s == 3)
                        reenviar segmento con nº sec. s
            }
```

un ACK duplicado para un segmento ya con ACK

retransmisión rápida

Retransmisión rápida

- período de expiración a menudo relativamente largo
  - largo retardo antes de reenviar el paquete perdido
- se detectan segmentos perdidos por ACKs repetidos
  - 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 datos con ACK se perdió:
  - retransmisión rápida: reenviar segmento antes de que expire el temporizador

Algoritmo de Nagle [RFC896]

- Las conexiones interactivas (ssh, telnet) suelen enviar segmentos con muy pocos datos (uno, dos bytes).
  - ¡Pérdida de eficiencia!
- Es más 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

Algoritmo de Nagle [RFC896]

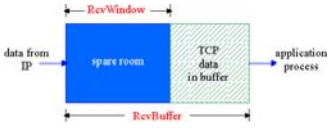
Evento en emisor	Acción en emisor
Llegada de datos de la aplicación. Hay ACKs pendientes.	Acumular datos en el buffer del emisor.
Llegada de un ACK pendiente.	Inmediatamente enviar todos los segmentos acumulados en buffer.
Llegada de datos de la aplicación. No hay ACKs pendientes.	Inmediatamente enviar 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 ACKs previos.

Capítulo 3: índice

TCP: Control de flujo

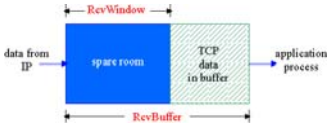
control de flujo
el emisor no saturará el buffer del receptor a base de enviar mucho muy seguido

- en TCP, el receptor tiene un buffer de recepción



- la aplicación puede ser lenta leyendo del buffer
- servicio de equilibrado de velocidad: equilibrar la velocidad de envío a la de la aplicación vaciando el buffer de recepción

TCP control de flujo: cómo funciona



(suponer que el receptor TCP descarta segmentos fuera de orden)

- hay sitio en el buffer

```
RcvWindow = RcvBuffer -
  [LastByteRcvd - LastByteRead]
```

- el receptor anuncia el espacio libre incluyendo el valor **RcvWindow** en los segmentos
- el emisor limita los datos sin ACK a **RcvWindow**
  - garantiza que el buffer de recepción no se desborda

TCP: *'Round Trip Time'* y *'Timeout'*

P: ¿cómo fijar el tiempo de '*timeout*' de TCP?

- más que RTT
  - pero RTT varía
- si demasiado corto: '*timeout*' prematuro
  - retransmisiones innecesarias
- si demasiado largo:
  - reacción lenta a pérdidas

P: ¿cómo estimar RTT?

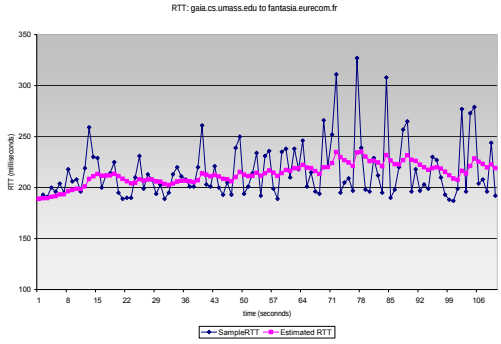
- **SampleRTT**: tiempo medido desde transmisión de un segmento hasta recepción de ACK
  - ignorar retransmisiones
- **SampleRTT** variará, queremos un valor más "estable"
  - promedio de varias mediciones recientes, no el valor actual

TCP: *'Round Trip Time'* y *'Timeout'*

```
EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT
```

- media móvil ponderada exponencial
- la influencia de una muestra pasada decrece exponencialmente
- valor típico: α = 0,125

Ejemplo de estimación de RTT:



Capítulo 3: índice

TCP: *'Round Trip Time'* y *'Timeout'*

Fijar el tiempo de expiración (*'timeout'*)

- **EstimatedRTT** más "margen de seguridad"
  - gran variación en **EstimatedRTT** -> mayor margen de seguridad
- primero, estimar cuánto **SampleRTT** se desvía de **EstimatedRTT**:

```
DevRTT = (1-β)*DevRTT +
          β*|SampleRTT-EstimatedRTT|

(típicamente, β = 0,25)
```

Entonces fijar el tiempo de expiración:

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```


Principios de control de la congestión

Congestión:

- informal: "demasiadas fuentes enviando demasiados datos demasiado deprisa para que la red lo pueda asimilar"
- ¡¡diferente a control de flujo!!
- síntomas:
  - paquetes perdidos (desbordamiento de bufferes en los routers)
  - grandes retardos (encolado en los bufferes de los routers)
  - ¡¡un problema "top-10"!!

Algoritmo de Karn

- Si recibimos el reconocimiento de un paquete retransmitido, no tenemos forma de saber a cuál de las retransmisiones corresponde ese reconocimiento.
- Por ello, se ignoran los paquetes retransmitidos a la hora de computar el RTT.
- Este procedimiento se denomina algoritmo de Karn.

Formas de abordar el control de congestión

Dos formas principales de abordarla:

control de terminal a terminal:

- no hay realimentación explícita de la red
- la congestión se deduce por el retardo y las pérdidas observadas por los terminales
- este es el método de TCP

control asistido por la red:

- los routers proporcionan realimentación a los terminales
  - un bit que indica la congestión (SNA, DECnet, TCP/IP ECN, ATM)
  - indicación explícita de la tasa a la que el emisor debería enviar

Caso de estudio: servicio ABR de las redes ATM

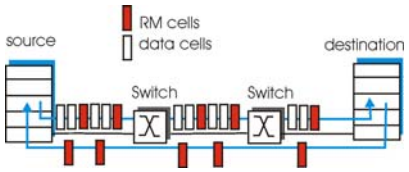
ABR: *'Available Bit Rate'* (Tasa de bits disponible):

- "servicio elástico"
- si la ruta del emisor "infra-cargada":
  - el emisor debería usar el ancho de banda disponible
- si la ruta del emisor está congestionada:
  - el emisor se limita a la tasa garantizada

células RM (*'resource management'*, gestión de recursos) :

- enviado por el emisor, intercalado con las celdas de datos
- los bits en las celdas RM se rellenan por los switches ("asistido por la red")
  - bit NI : no hay mejora en la velocidad (congestión suave)
  - bit CI : indica congestión
- las celdas RM se devuelven al emisor por el receptor, sin modificar

Caso de estudio: servicio ABR de las redes ATM



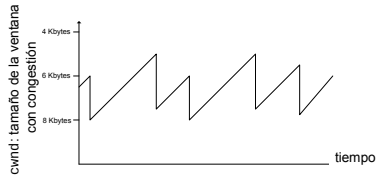
- campo ER (*'explicit rate'*, tasa explícita) de 2 bytes en celda RM
  - un switch congestionado puede rebajar el valor ER
  - la tasa del emisor es así la máxima que puede aguantar la ruta
- bit EFCI en celdas de datos: se pone a 1 en switch congestionado
  - si la celda que precede a la RM tiene EFCI a 1, el emisor pone a 1 el bit CI en la celda RM devuelta

Capítulo 3: índice

control de congestión en TCP : incremento aditivo, decremento multiplicativo

- *filosofía:* incrementar la tasa de transmisión (tamaño de la ventana), sondeando el ancho de banda accesible, hasta que hay pérdidas
  - *incremento aditivo:* incrementar **cwnd** en 1 MSS cada RTT hasta que haya pérdidas
  - *decremento multiplicativo:* dividir **cwnd** por 2 cuando las haya

diente de sierra: sondeo del ancho de banda



Control de congestión TCP: detalles

- el emisor limita la transmisión: `LastByteSent-LastByteAcked` ≤ **cwnd**
- *'grosso modo'*,

$$\text{tasa} = \frac{\text{cwnd}}{\text{RTT}} \quad \text{Bytes/s}$$

- **cwnd** es dinámica, función de la congestión de la red percibida

¿Cómo percibe el emisor la congestión?

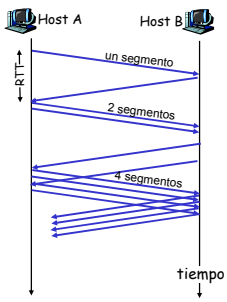
- evento de pérdida = expiración o 3 ACKs duplicados
- el emisor reduce la tasa (**cwnd**) tras un evento de pérdida

3 mecanismos:

- AIMD
- arranque lento
- conservador tras eventos de expiración

TCP: Arranque lento

- cuando se inicia la conexión, la tasa se incrementa exponencialmente hasta la primera pérdida:
  - inicialmente **cwnd** = 1 MSS
  - **cwnd** se dobla cada RTT
  - se incrementa **cwnd** con cada ACK recibido
- resumen: la tasa inicial es baja, pero crece exponencialmente rápido



Refinado: deducción de pérdidas

- tras 3 ACKs duplicados
  - **cwnd** se divide por 2
  - la ventana ya crece linealmente
- pero tras una expiración:
  - **cwnd** en cambio se pone a 1 MSS;
  - la ventana entonces crece exponencialmente
  - hasta un umbral, luego linealmente

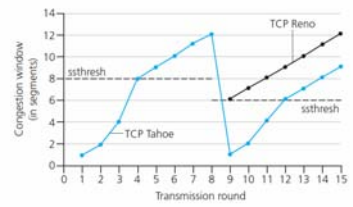
Filosofía:

- 3 ACKs duplicados indica que la red es capaz de entregar algunos segmentos
- expiración indica una situación de congestión "más alarmante"

Refinado

P: ¿cuándo debería pasarse de incremento exponencial a lineal?

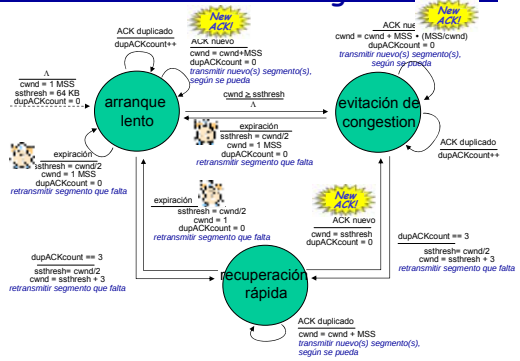
R: cuando **cwnd** llegue a 1/2 de su valor antes de la expiración



Implementación:

- variable **ssthresh**
- con una pérdida, **ssthresh** se pone a 1/2 de **cwnd** justo antes de la pérdida

Resumen: Control de congestión TCP



eficiencia de TCP

- ¿cuál es la tasa media de TCP en función del tamaño de ventana y RTT?
  - ignorar el arranque rápido
- sea W el tamaño de la ventana cuando ocurre una pérdida
  - cuando la ventana es W, la tasa es W/RTT
  - justo tras la pérdida, la ventana pasa a W/2, y la tasa a W/2RTT
  - la tasa media es: 0,75 W/RTT

Futuros de TCP: TCP sobre "tubos largos y gruesos"

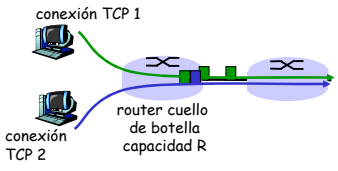
- ejemplo: segmentos de 1500 bytes, RTT 100ms, se quiere tasa de 10 Gbps
- requiere tamaño de ventana W = 83.333 segmentos en tránsito
- tasa de transferencia en función de la tasa de pérdidas:

$$\frac{1{,}22 \cdot MSS}{RTT\sqrt{L}}$$

- L = $2 \cdot 10^{-10}$ *¡¡una tasa de pérdidas muy baja!!*
- nuevas versiones de TCP para alta velocidad

Equidad en TCP

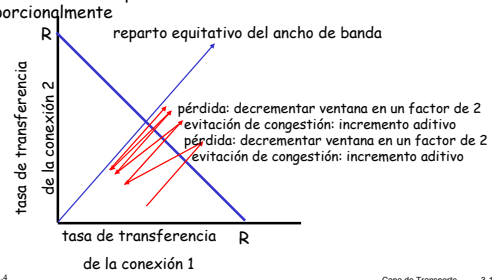
objetivo de equidad: si K sesiones TCP comparten el mismo enlace cuello de botella de ancho de banda R, cada uno debería tener una tasa media de R/K



¿Por qué es equitativoTCP?

2 sesiones que compiten:

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



Equidad (más)

Equidad y UDP

- las aplicaciones multimedia a menudo no usan TCP
  - no quieren que la tasa de transf. se limite por el control de congestión
- por eso usan UDP:
  - envían audio/video a tasa constante, toleran pérdida de paquetes

Equidad y conexiones TCP paralelas

- nada impide a una aplicación abrir conexiones paralelas entre 2 hosts
- los navegadores lo hacen
- ejemplo: enlace de tasa R permite 9 conexiones
  - una aplicación pide 1 TCP, obtiene tasa R/10
  - una aplicación pide 11 TCPs, ¡¡obtiene tasa R/2!!

Capítulo 3: Resumen

- principios detrás de los servicios de la capa de transporte
  - multiplexación, desmultiplexación
  - transferencia de datos fiable
  - control de flujo
  - control de congestión
- instanciación e implementación en Internet
  - UDP
  - TCP