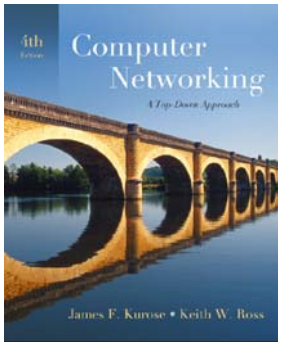
Introducción a las Redes de Computadores

Capítulo 3
Capa de Transporte



Computer Networking: A Top Down Approach 4th edition.
Jim Kurose, Keith Ross
Addison-Wesley, July 2007.

Nota acerca de las transparencias del curso:

Estas transparencias están basadas en el sitio web que acompaña el libro y han sido modificadas por los docentes del curso.

Capítulo 3: Capa de Transporte

Objetivos:

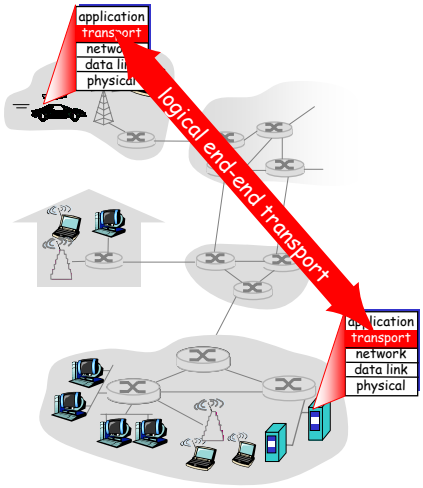
- Entender los principios detrás de los servicios de la capa de transporte:
  - multiplexación/demulti-plexación
  - transferencia de datos confiable
  - control de flujo
  - control de congestión
- Aprender acerca de los protocolos de la capa de transporte en Internet:
  - UDP: transporte no orientado a conexión
  - TCP: transporte orientado a conexión
  - control de congestión de TCP

Capítulo 3: agenda

- 3.1 Servicios de la capa de transporte
- 3.2 Multiplexación y demultiplexación
- 3.3 Transporte no orientado a conexión: UDP
- 3.4 Principios de la transferencia de datos confiable
- 3.5 Transporte orientado a conexión: TCP
  - estructura del segmento
  - transferencia de datos confiable
  - control de flujo
  - gestión de la conexión
- 3.6 Principios del control de congestión
- 3.7 Control de congestión de TCP

Protocolos y servicios de transporte

- brinda *comunicación lógica* entre procesos de Aplicación corriendo en *hosts* diferentes
- los protocolos de transporte corren en los *end systems*
  - lado del transmisor: genera segmentos a partir de los mensajes de la Aplicación, y los pasa a la capa de red
  - lado del receptor: a partir de los segmentos, los mensajes son pasados a la capa de Aplicación
- más de un protocolo de transporte disponible para las aplicaciones
  - Internet: TCP y UDP



Capa de Transporte vs. Capa de Red

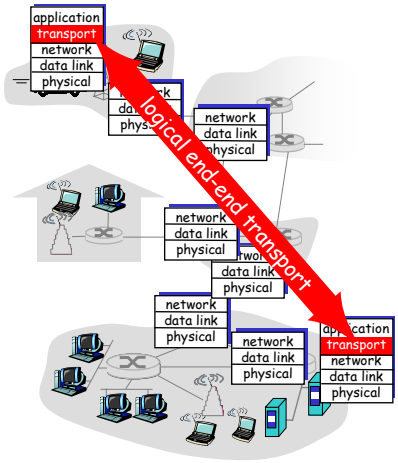
- *Capa de red:* comunicación lógica entre *hosts*
- *Capa de transporte:* comunicación lógica entre procesos ejecutándose en diferentes máquinas
- *Los dispositivos intermedios (routers por ejemplo), ¿implementan protocolos de capa de transporte y de capa de aplicación?*

Analogía familiar:

- *6 niños enviando cartas a 6 niños* (primos)
- procesos = niños
- Mensajes de aplicación = cartas (en sobres)
- *hosts* = casas
- protocolo de transporte = Ana y Guillermo
- Protocolo de capa de red = servicio de correo postal

Protocolos de la capa de Transporte de Internet

- confiable, entrega en orden: TCP
  - control de congestión
  - control de flujo
  - establecimiento de la conexión
- no confiable, entrega no ordenada: UDP
  - En relación al "mejor esfuerzo" de IP, poco aporta
- servicios no disponibles:
  - garantías de retardo
  - garantías de ancho de banda



Capítulo 3: agenda

- 3.1 Servicios de la capa de transporte
- 3.2 Multiplexación y demultiplexación
- 3.3 Transporte no orientado a conexión: UDP
- 3.4 Principios de la transferencia de datos confiable
- 3.5 Transporte orientado a conexión: TCP
  - estructura del segmento
  - Transferencia de datos confiable
  - control de flujo
  - Gestión de la conexión
- 3.6 Principios del control de congestión
- 3.7 Control de congestión de TCP

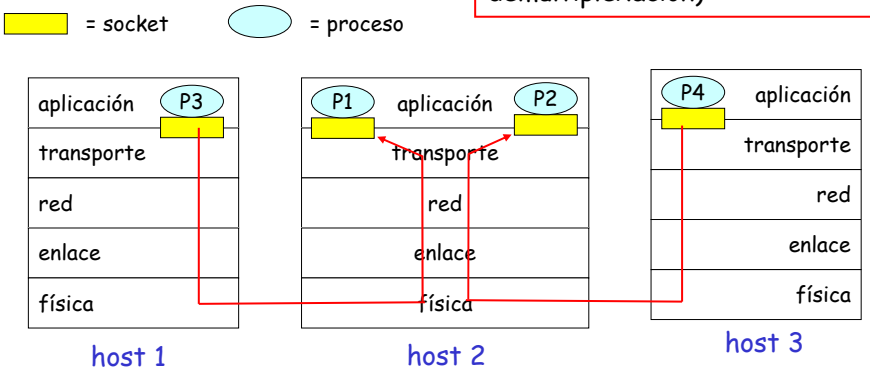
Multiplexación/Demultiplexación

Demultiplexación en el *host* receptor:

Entregando los segmentos recibidos al *socket* correcto

Multiplexación en el *host* transmisor

Recolectando datos de múltiples *sockets*, incorporando *headers* (después utilizados en la demultiplexación)



¿Cómo trabaja la demultiplexación?

- Un *host* recibe datagramas IP
  - cada datagrama tiene dirección IP de origen, dirección IP de destino
  - cada datagrama lleva 1 segmento de la capa de transporte
  - cada segmento tiene número de puertos de origen y destino
- los *host* utilizan las direcciones IP y los números de puertos para enviar el segmento al *socket* apropiado
- Nro. de puerto: **TSAP** *(Transport Service Access Point)*

Formato del segmento TCP/UDP

Un poco más sobre puertos

- IANA (Internet Assigned Numbers Authority)
  - Se agrupan en 3 rangos
  - *Well-known port numbers*: 0 – 1023
    - aplicaciones de servidor; en general ejecutados por usuarios tipo "root". Se deberían registrar
    - 53: DNS, 80: www HTTP, 20 y 21: ftp, 22: ssh, 161 y 162: SNMP, imaps: 993, imap: 143, nntps: 563
  - *Registered ports*: 1024 – 49151
    - aplicaciones de servidor, usuarios comunes. Se deberían registrar
    - 1812, 1813: radius
  - *Dynamic and/or Private ports*: 49152 – 65535
    - Para uso temporario. No se deben registrar
  - **http://www.iana.org/assignments/port-numbers**

Demultiplexación no orientada a conexión

- Crear *sockets* con números de puerto:

```
DatagramSocket elSocket1 = new
   DatagramSocket();
```

Asigna un nro. de puerto libre entre 1024 y 65535

```
DatagramSocket elSocket2 = new
   DatagramSocket(12535);
```

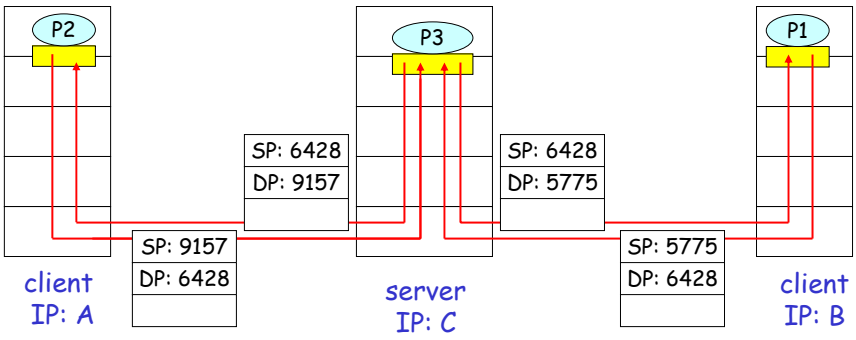
- UDP *socket* completamente identificado por la dupla:

(dirección IP destino, número puerto destino)

- Cuando el *host* recibe el segmento UDP:
  - Chequea el número de puerto destino en el segmento
  - Dirige el segmento UDP al *socket* con dicho número de puerto
- Datagramas IP con diferentes direcciones IP **origen** y/o números de puerto **origen** dirigidos al mismo *socket*

demux no orientada a conexión (continuación)

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



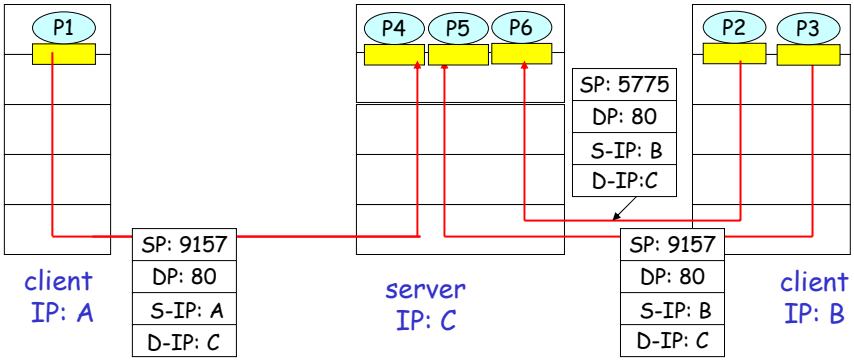
SP: *Source Port*
DP: *Destination Port*

SP proporciona "dirección de retorno"

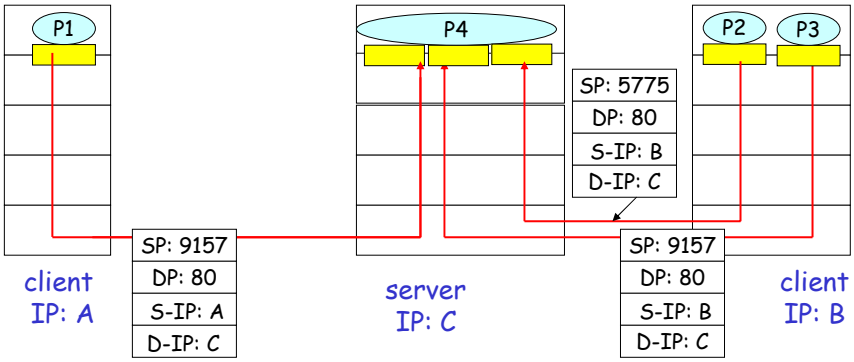
Demux orientada a conexión

- *Socket* TCP identificado por una tupla de 4 elementos:
  - dirección IP origen
  - número puerto origen
  - dirección IP destino
  - número puerto destino
- El *host* destino utiliza los 4 valores para dirigir el segmento al *socket* apropiado
- El *host* servidor debería soportar varios *sockets* TCP simultáneos:
  - cada socket identificado por su propia tupla de 4 elementos
- Los servidores Web tienen diferentes *sockets* para cada cliente que se conecta
  - HTTP no persistente debe tener diferentes *sockets* para cada request

Demux orientada a conexión (continuación)



Demux orientada a conexión: Threaded Web Server



Port scanning

- Es relativamente sencillo conocer qué aplicaciones están escuchando en qué puertos en un *end system* o en un conjunto de *end systems*
- Un programa para ello
  - Nmap (*Network Mapper*)
  - http://insecure.org/nmap
- Si detectamos algún *host* corriendo determinada aplicación de la que conocemos alguna vulnerabilidad, esto puede ser el punto de partida de un ataque

Capítulo 3: agenda

- 3.1 Servicios de la capa de transporte
- 3.2 Multiplexación y demultiplexación
- 3.3 Transporte no orientado a conexión: UDP
- 3.4 Principios de la transferencia de datos confiable
- 3.5 Transporte orientado a conexión: TCP
  - estructura del segmento
  - Transferencia de datos confiable
  - control de flujo
  - Gestión de la conexión
- 3.6 Principios del control de congestión
- 3.7 Control de congestión de TCP

UDP: User Datagram Protocol [RFC 768]

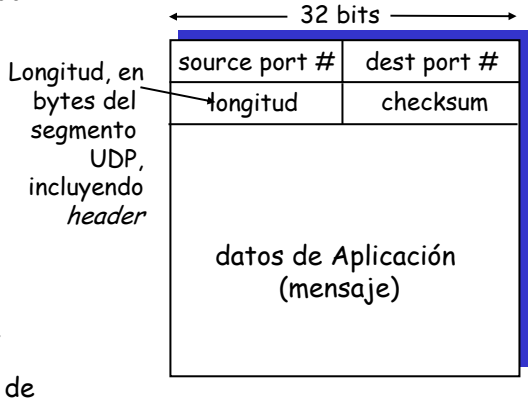
- Protocolo de transporte de Internet "esquelético"
- Servicio "*best effort*"; los segmentos UDP podrían:
  - perderse
  - entregarse fuera de orden a la aplicación
- *no orientado a conexión:*
  - no hay *handshaking* UDP entre el *sender* y el *receiver*
  - cada segmento UDP es manejado independientemente de los otros

¿Por qué existe UDP?

- no hay establecimiento de conexión (lo cual puede agregar retardo)
- simple: no hay estado de conexión ni en el *sender* ni en el *receiver*
- encabezado del segmento pequeño
- no hay control de congestión

UDP: más

- frecuentemente utilizado para aplicaciones de *streaming* multimedia
  - tolerante a las pérdidas
  - sensible a la *rate*
- otros usos de UDP
  - DNS
  - SNMP
  - NTP
- transferencia confiable sobre UDP: agregar confiabilidad en la capa de Aplicación



Formato del segmento UDP

El *checksum* de UDP

Objetivo: detectar errores (p.e., bits cambiados) en el segmento transmitido

Sender:

- trata el contenido de cada segmento como una secuencia de enteros de 16-bit
- *checksum*: complemento a 1 de la suma de los contenidos del segmento
- El transmisor pone el valor del *checksum* UDP en el campo *checksum*

Receiver:

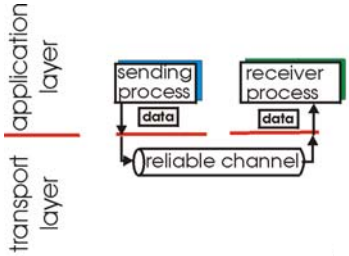
- calcula el *checksum* del segmento recibido
- Chequea si el *checksum* calculado es igual al valor del campo *checksum*:
  - NO – error detectado
  - SI - no se ha detectado error. *Pero, no obstante, ¿podrían haber errores?*

Capítulo 3: agenda

- 3.1 Servicios de la capa de transporte
- 3.2 Multiplexación y demultiplexación
- 3.3 Transporte no orientado a conexión: UDP
- 3.4 Principios de la transferencia de datos confiable
- 3.5 Transporte orientado a conexión: TCP
  - estructura del segmento
  - Transferencia de datos confiable
  - control de flujo
  - Gestión de la conexión
- 3.6 Principios del control de congestión
- 3.7 Control de congestión de TCP

Principios de la transferencia de datos confiable

- importante en app., transport, link layers
- ¡ En la lista de los temas más importantes del *networking* !

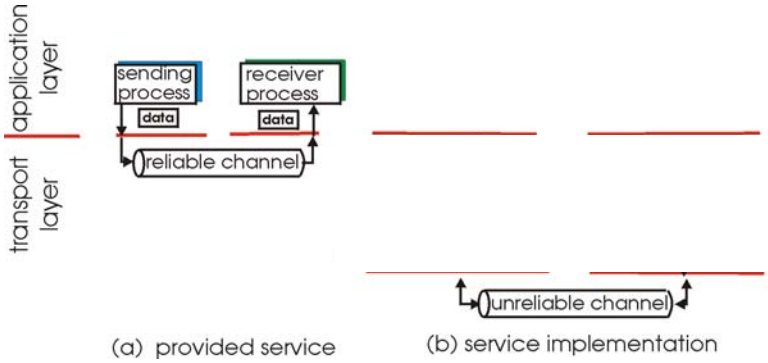


(a) provided service

- las características del canal no confiable determinan la complejidad del protocolo transferencia de datos confiable (rdt: *reliable data transfer*)

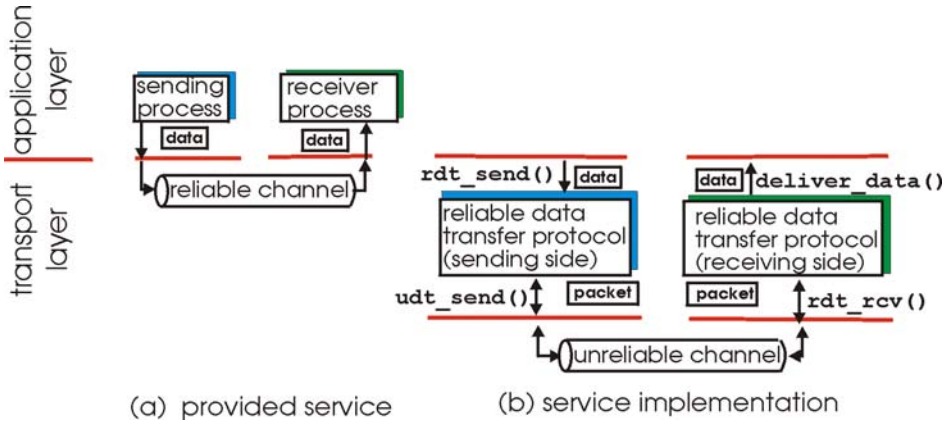
Principios de la transferencia de datos confiable

- importante en app., transport, link layers
- ¡ En la lista de los 10 temas más importantes del *networking* !



- las características del canal no confiable determinan la complejidad del protocolo transferencia de datos confiable (rdt : *reliable data transfer*)

Principios de la transferencia de datos confiable

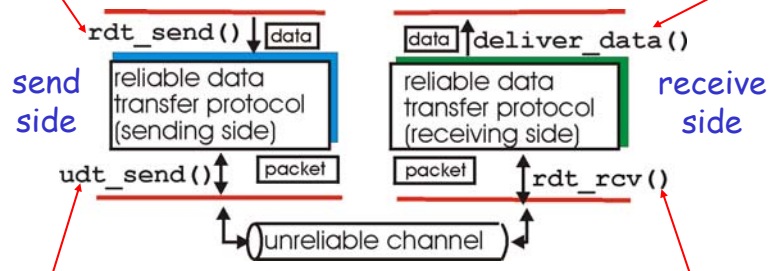


udt: *unreliable data transfer*

Transferencia de datos confiable: comenzando

`rdt_send()`: llamada desde arriba, (p.e., por aplic.). Datos pasados para entregar a la capa superior del *receiver*

`deliver_data()`: llamado por **`rdt`** para entregar los datos hacia arriba



`udt_send()`: llamado por rdt, para transferir el paquete sobre el canal no confiable hasta el *receiver*

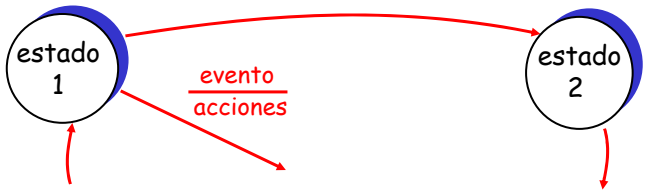
`rdt_rcv()`: llamado cuando llega el paquete al lado receptor del canal

Transferencia de datos confiable: comenzando

- Desarrollo incremental del transmisor y del receptor de un protocolo de transferencia confiable de datos (rdt)
- consideramos solamente transferencia unidireccional de datos
  - ¡ Pero la información de control y de datos fluye en ambas direcciones !
- para especificar el emisor y el receptor utilizamos una máquina de estados finita *(FSM)*

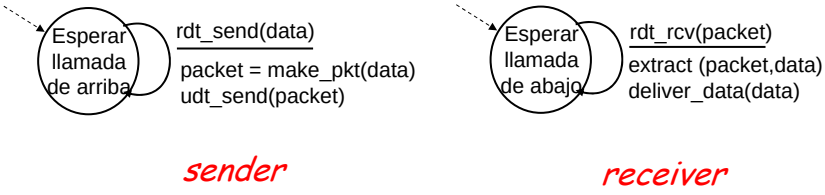
evento que causa la transición de estado
acciones tomadas en la transición de estado

estado: cuando estamos en este estado, el siguiente está determinado por el evento siguiente



rdt1.0: transferencia confiable sobre un canal confiable

- El canal subyacente es perfectamente confiable
  - Sin errores en bits
  - Sin pérdida de paquetes
- FSMs separadas para *sender* y *receiver*:
  - El emisor envía datos dentro del canal subyacente
  - El receptor lee datos del canal subyacente



sender *receiver*

Protocolos ARQ

- Implica la presencia de reconocimientos positivos ("*Correcto*") y negativos ("*Repita, por favor*")
- Los reconocimientos son mensajes de control a través de los cuales el receptor realimenta al emisor respecto a las PDU que recibe.
- Los protocolos de transferencia de datos que implementan ésto, se conocen como protocolos ARQ
  - *Automatic Repeat reQuest*

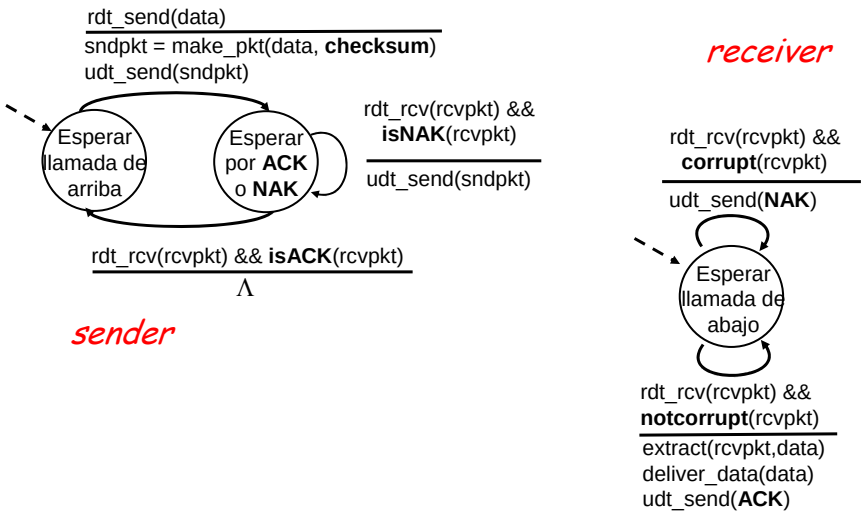
Protocolos ARQ

- ¿Qué capacidades precisan estos protocolos?
  - Detección de errores
  - Realimentación desde el receptor
  - Retransmisión
- Volveremos sobre estos protocolos en las próximas clases

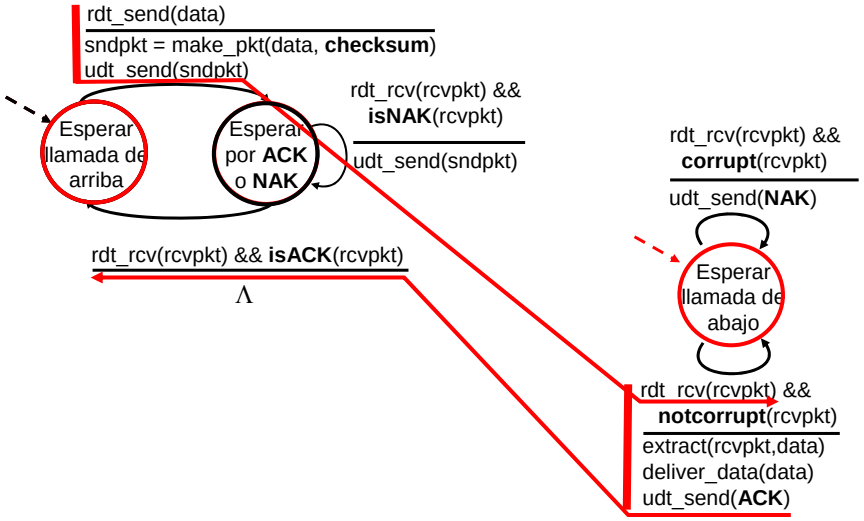
rdt2.0: canal con errores en bits

- no hay pérdida de paquetes
- el canal subyacente puede modificar bits en el paquete
  - *checksum* para detectar errores en bits
- *¿Cómo nos recuperamos de los errores?*
  - *ACKnowledgements (ACKs):* el receptor explícitamente le dice al transmisor que el paquete se ha recibido OK
  - *Negative ACKnowledgements (NAKs o NACKs):* el receptor explícitamente le dice al transmisor que el paquete tiene errores
  - El transmisor retransmite el paquete ante la recepción de un NAK
- nuevo mecanismo en **`rdt2.0`** (respecto a **`rdt1.0`**):
  - detección de error
  - realimentación desde el receptor (*receiver -> sender*): mensajes de control (ACK,NAK)

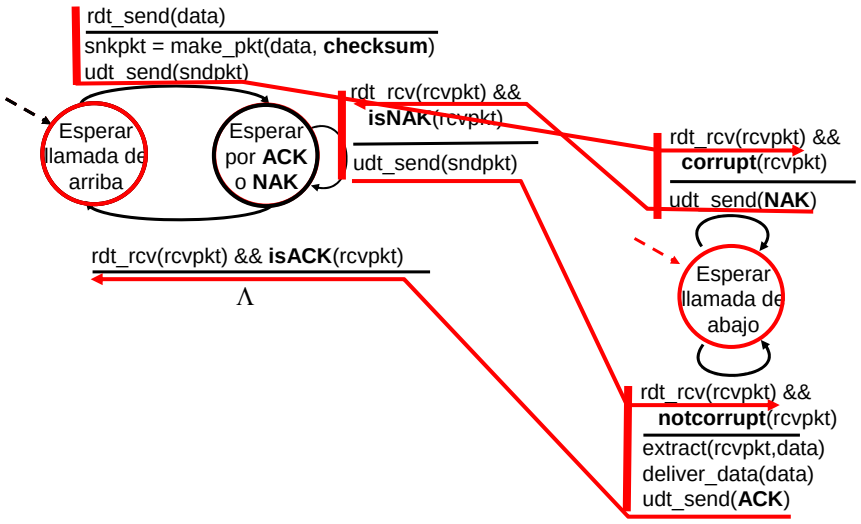
rdt2.0: Especificación FSM



rdt2.0: operación sin errores



rdt2.0: escenario de error



¡ rdt2.0 tiene un defecto fatal !

- ¿Qué ocurre si el ACK/NAK está corrupto?
- ¡ el emisor no sabe qué ha ocurrido en el receptor !
- No puede simplemente retransmitir:
  - posibles duplicados
  - el receptor debe saber distinguirlos

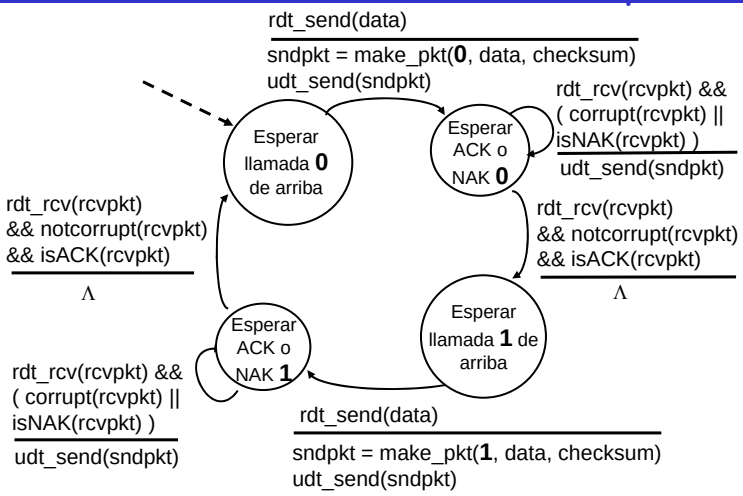
stop and wait

•el emisor envía un paquete y espera para "saber cómo le fue"
•rdt2.0 es un protocolo "parada y espera"

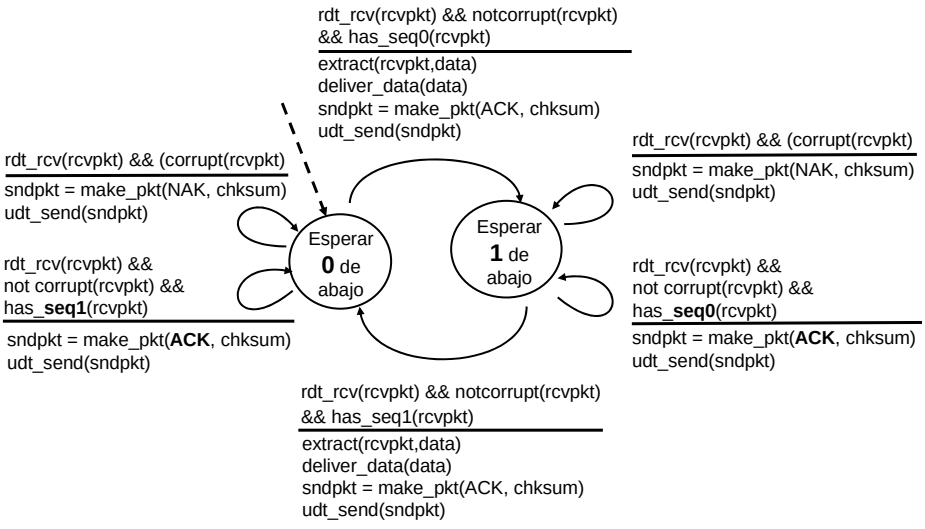
Manejando duplicados:

- el emisor retransmite el paquete actual si el ACK/NAK está corrupto
- el emisor agrega *un número de secuencia* a cada paquete
- el receptor descarta el paquete duplicado (no lo entrega hacia arriba)
- Los ACK/NAK no llevan números de secuencia
  - *¿Por qué?*

rdt2.1: el emisor maneja ACK/NAKs con errores (rdt2.1 es rdt2.0 con un parche)



rdt2.1: el receptor maneja ACK/NAKs con errores



rdt2.1: discusión

Emisor:

- número de secuencia agregado al paquete
- ¿dos números de seq. (0,1) es suficiente?
- debe chequear si el ACK/NAK recibido está corrupto

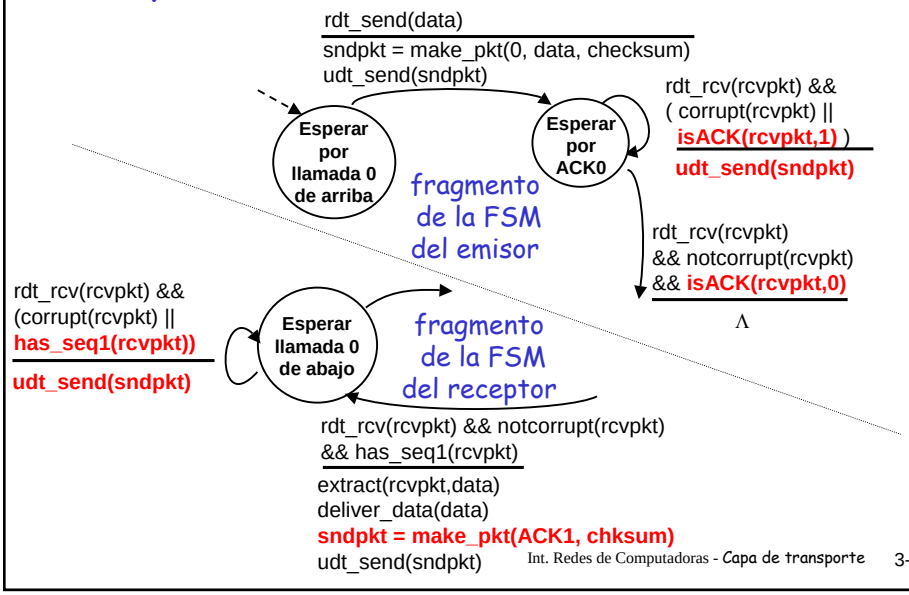
Receptor:

- debe chequear si el paquete recibido está duplicado
- nota: el receptor no puede saber si el último ACK/NAK se recibió OK en el transmisor

rdt2.2: un protocolo libre de NAK

- recordemos que los paquetes no se pierden (todavía)
- las mismas funcionalidades que rdt2.1, pero utilizando solamente ACKs
- en lugar de NAK, el receptor envía ACK para el último paquete recibido OK
  - el receptor debe incluir explícitamente el número de secuencia del paquete que está ACKed
- ACK duplicado (dos ACKs para el mismo paquete) en el emisor resulta en la misma acción que el NAK: *retransmitir el paquete actual*

rdt2.2: fragmentos del emisor y receptor



rdt3.0: canales con errores *y pérdidas*

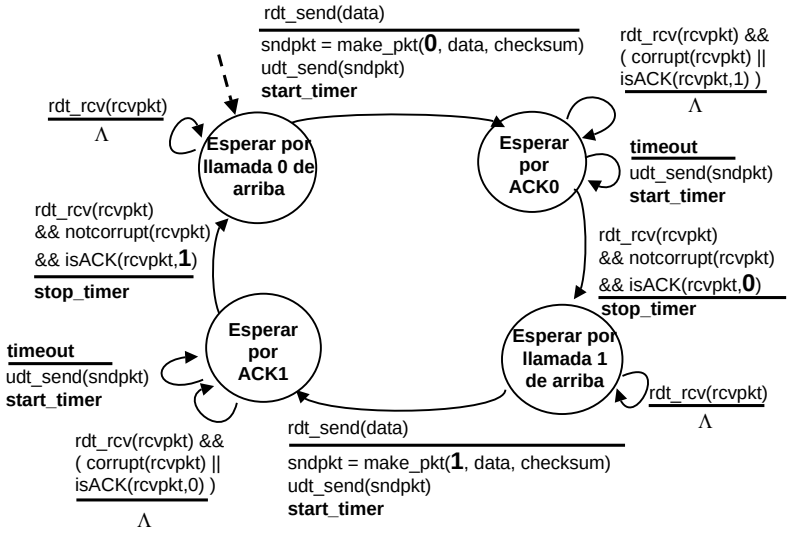
Nuevo supuesto: el canal subyacente también puede perder paquetes (datos o ACKs)

- suma de comprobación, nros. de secuencia, ACKs y retransmisiones nos ayudan, pero no lo suficiente

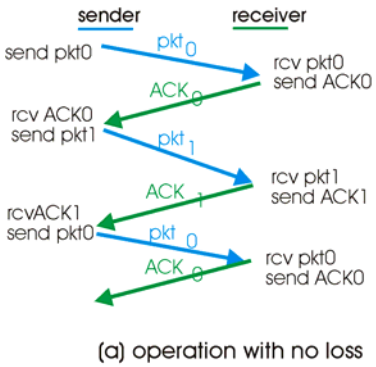
Enfoque: el transmisor espera por el ACK un tiempo "razonable"

- retransmite si no recibe el ACK en dicho tiempo
- si el paquete (o ACK) sólo está retrasado (no perdido):
  - la retransmisión será un duplicado, pero el nro. de secuencia maneja ésto
  - El receptor debe especificar el nro. de secuencia del paquete que está siendo ACKed
- requiere *countdown timer*

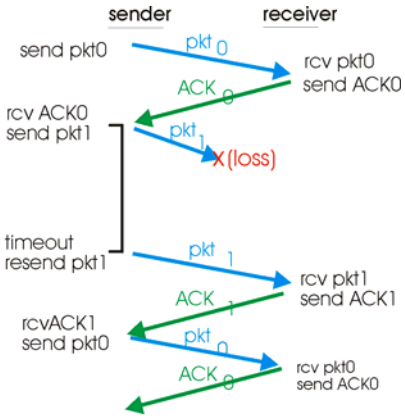
rdt3.0: emisor



rdt3.0 en acción

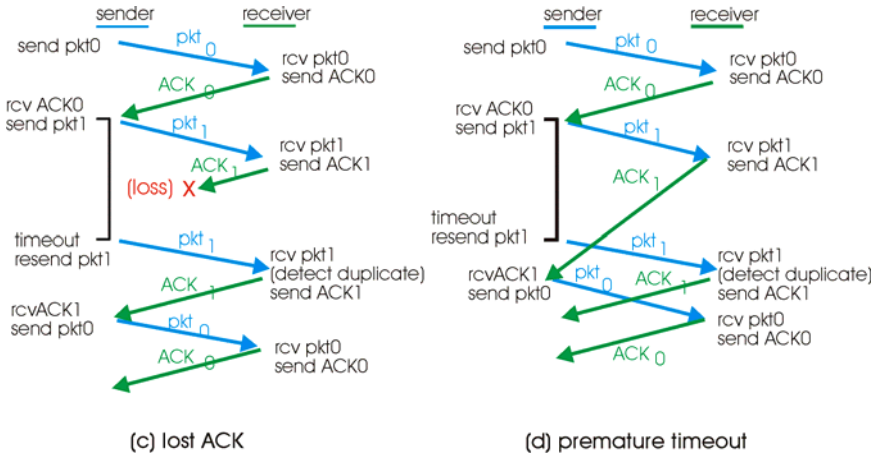


(a) operation with no loss



(b) lost packet

rdt3.0 en acción



Performance de rdt3.0

- rdt3.0 funciona, pero su *performance* no es buena
- Ejemplo: enlace de **1 Gbps**, retardo de propagación **15 ms**, paquetes de **1000 bytes** (8000 bits):

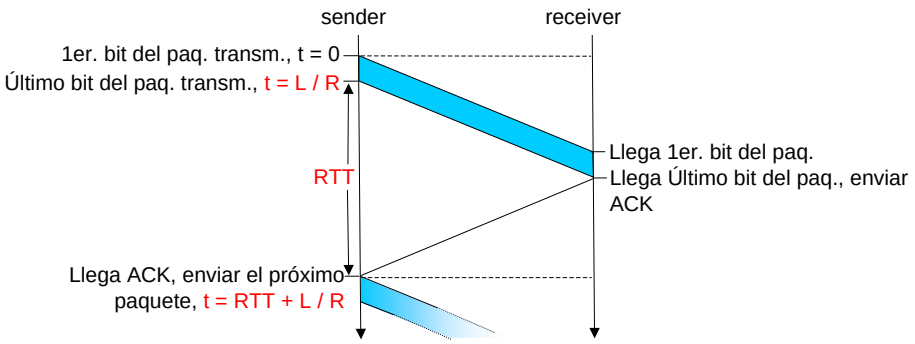
$$t_{\substack{transmitir \\ un\ paquete}} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bps}} = 8 \text{ µs}$$

  - U_{emisor}: utilización del canal – tiempo del *sender* ocupado enviando datos

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

  - Paquetes de **1KB** cada **30 msec** -> **33kB/sec** en un enlace de **1 Gbps**
  - Los protocolos de red limitan el uso de los recursos físicos

rdt3.0: operación *stop-and-wait*

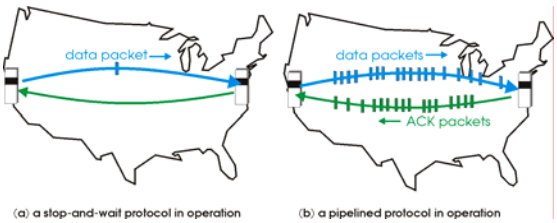


$$U_{emisor} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Pipelined protocols

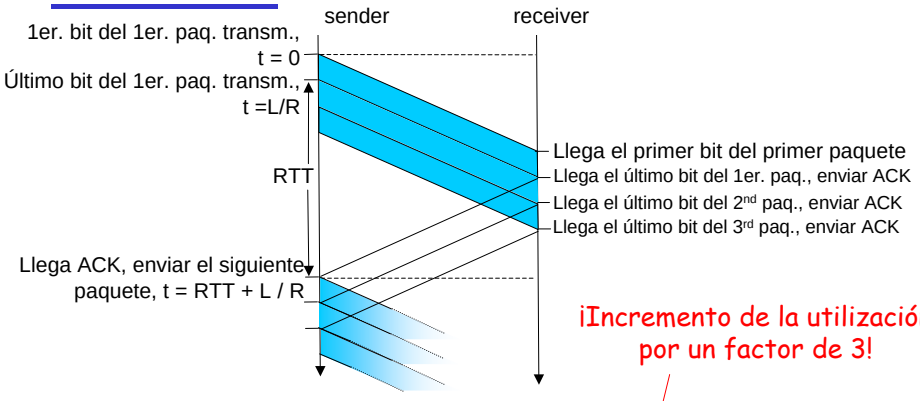
Pipelining: el emisor permite el envío de múltiples paquetes a ser reconocidos

- el rango de los números de secuencia se debe incrementar
- *buffering* en el *sender* y/o en el *receiver*



- Dos formas genéricas de *pipelined protocols*: *Go-Back-N (Retroceder N), Selective Repeat (Repetición Selectiva)*

Pipelining: incremento de la utilización



¡Incremento de la utilización por un factor de 3!

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

Pipelining Protocols

Go-back-N: titulares

- El transmisor puede tener hasta N paquetes no-ACKed en el *pipeline*
- El receptor sólo envía ACKs acumulativos
  - No envía el ACK de un paquete si hay un hueco
- El transmisor tiene un *timer* para el paquete más viejo no-ACKed
  - Si el *timer* vence, retransmite todos los paquetes no-ACKed

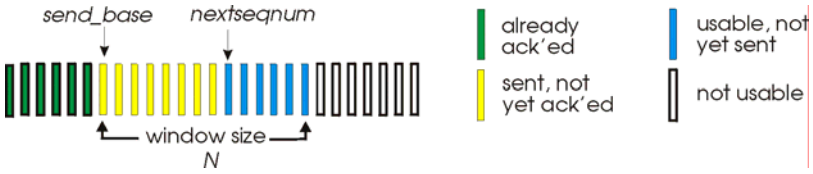
Selective Repeat: titulares

- El transmisor puede tener hasta N paquetes no-ACKed en el *pipeline*
- El receptor envía ACKs para paquetes individuales
- El transmisor tiene un *timer* para cada paquete no no-ACKed
  - Cuando el *timer* vence, retransmite sólo el paquete no-ACKed

Go-Back-N

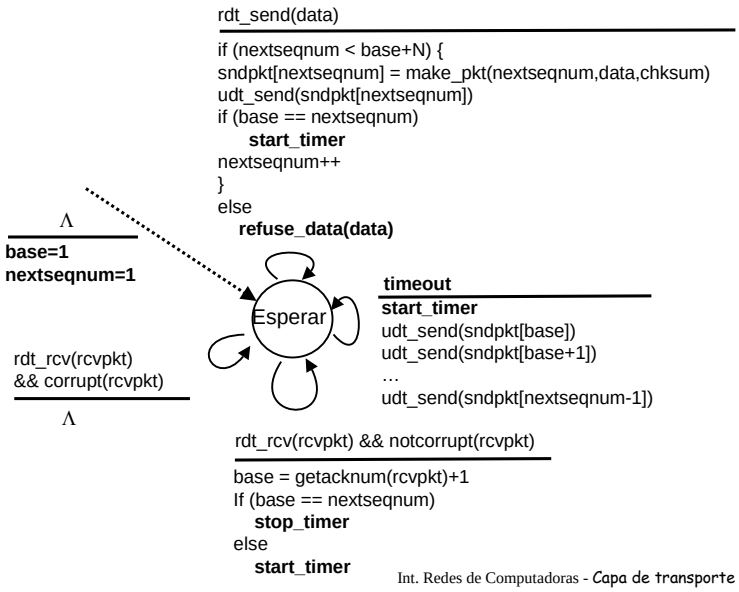
Emisor:

- Número de secuencia de **k bits** en el encabezado del paquete
- "window" permitida de hasta **N paquetes** consecutivos no-ACKed
- Si la ventana no está llena, se puede seguir transmitiendo

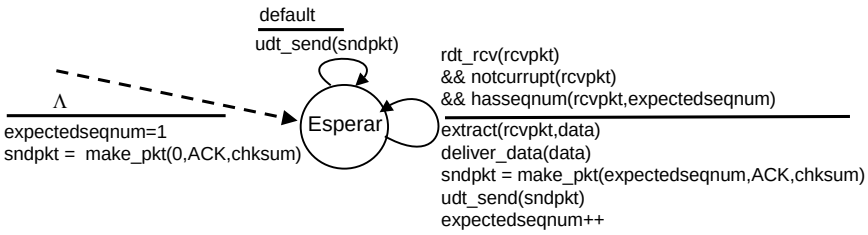


- ACK(n): ACKs de todos los paquetes hasta él, incluyendo el nro. de sec. n -- "**ACK acumulativo**"
  - podría recibir ACKs duplicados (ver receptor)
- *timer* único: para el paquete más viejo aún no-ACKed
- *timeout(n):* retransmite el paquete **n** y todos aquellos con nro. de sec. mayor y dentro de la ventana

GBN: FSM extendida del *sender*

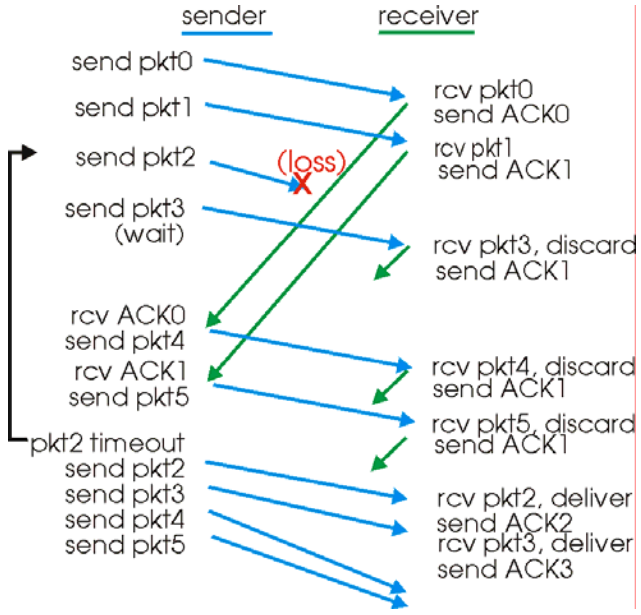


GBN: FSM extendida del *receiver*



- ACK-solamente: siempre enviar ACK para el paquete correctamente recibido y en orden con el mayor nro. de sec.
  - Se podrían generar ACKs duplicados
  - Solamente necesitamos recordar **`expectedseqnum`**
- Paquetes fuera de orden:
  - Descartar (no almacenar en *buffer*) -> sin buffer en el receptor
  - Re-ACK paquetes con el mayor nro. de sec. en orden

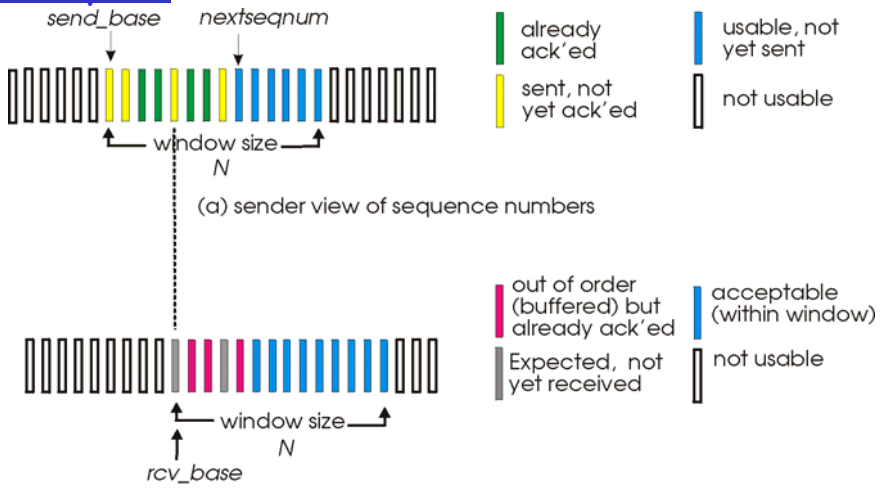
GBN en acción



Selective Repeat

- el receptor envía ACKs *individualmente* para todos los paquetes correctamente recibidos
  - *buffers* para los paquetes, para una eventual entrega en orden a la capa superior
- el emisor re-envía solamente los paquetes para los que no ha recibido su ACK
  - *timer* en el emisor para cada paquete no-ACKed
- ventana del emisor
  - N números de secuencia consecutivos
  - Nuevamente, los límites de los números de secuencia son los paquetes enviados no-ACKed

Selective repeat: ventanas en emisor y receptor



Selective repeat

emisor

datos desde arriba:

- si el próx. nro. de seq. está dentro de la ventana, enviar paquete

timeout(n):

- re-enviar el paquete **n**, re-inicio del *timer*

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

- marcar al paquete **n** como recibido
- si **n** es el paquete más pequeño no-ACKed, avanza la base de la ventana al siguiente número de secuencia no-ACKed

receptor

paq. n en [rcvbase, rcvbase+N-1]

- enviar ACK(n)
- fuera de orden: buffer
- en orden: entregar (también entregar los paquetes en orden en *buffer*), avanzar ventana al siguiente paquete no recibido aún

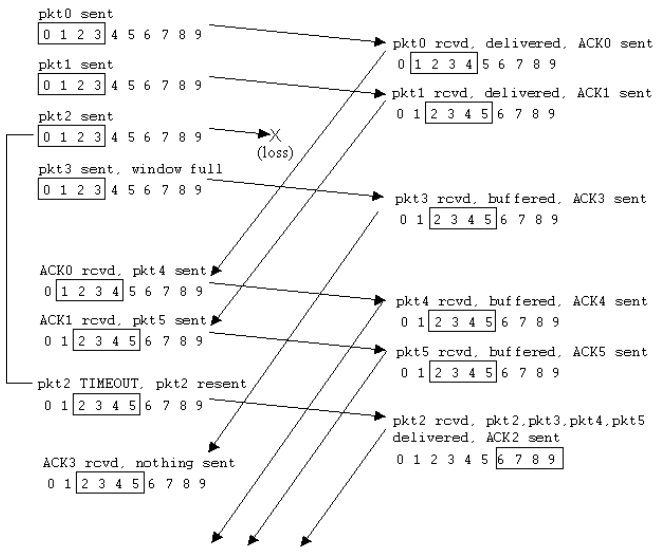
pkt n en [rcvbase-N,rcvbase-1]

- ACK(n)

en otro caso:

- ignorar

La **repetición selectiva** en acción



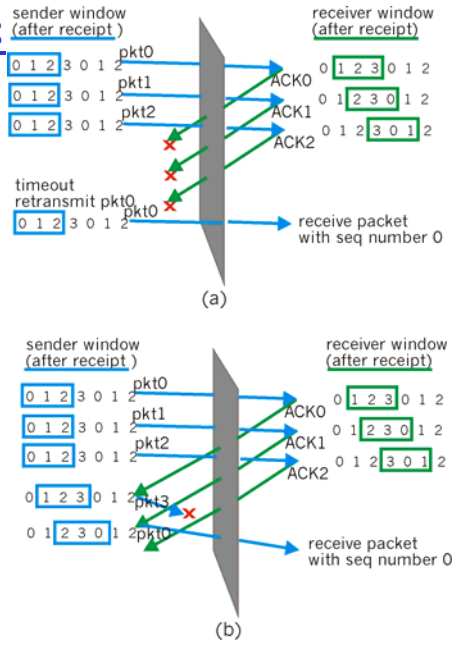
Repetición selectiva: el dilema

Ejemplo:

- seq #'s: 0, 1, 2, 3
- Tamaño de ventana = 3

- para el *receiver* no hay diferencias en los dos escenarios!
- en (a) incorrectamente se pasan datos duplicados como nuevos

P: ¿Qué relación debe haber entre seq # size y el tamaño de ventana?



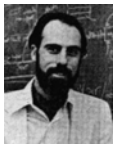
Mecanismos de transferencia confiable: resumen

- Suma de comprobación
- Temporizador
- Número de secuencia
- Reconocimiento
- Reconocimiento negativo
- Ventana deslizante

Agenda

- 3.1 Servicios de la capa de transporte
- 3.2 Multiplexación y demultiplexación
- 3.3 Transporte no orientado a conexión: UDP
- 3.4 Principios de la transferencia de datos confiable
- 3.5 Transporte orientado a conexión: TCP
  - estructura del segmento
  - transferencia de datos confiable
  - control de flujo
  - gestión de la conexión
- 3.6 Principios del control de congestión
- 3.7 Control de congestión de TCP

TCP: un poco de historia...



- *Transmission Control Protocol*
- Vinton Cerf y Robert Kahn
  - mayo del 74
    - *A Protocol for Packet Network Intercommunication*
    - Antes del PC, de las estaciones de trabajo, de Ethernet por todos lados,...
  - septiembre del 81
    - RFC **793**: Especificación de TCP
  - octubre del 89
    - RFC **1122**: Requerimientos para los *hosts* de Internet
  - mayo del 92
    - RFC **1323**: Extensiones a TCP
  - octubre del 96
    - RFC 2018: Selective ACK

TCP: un poco de historia (2)…

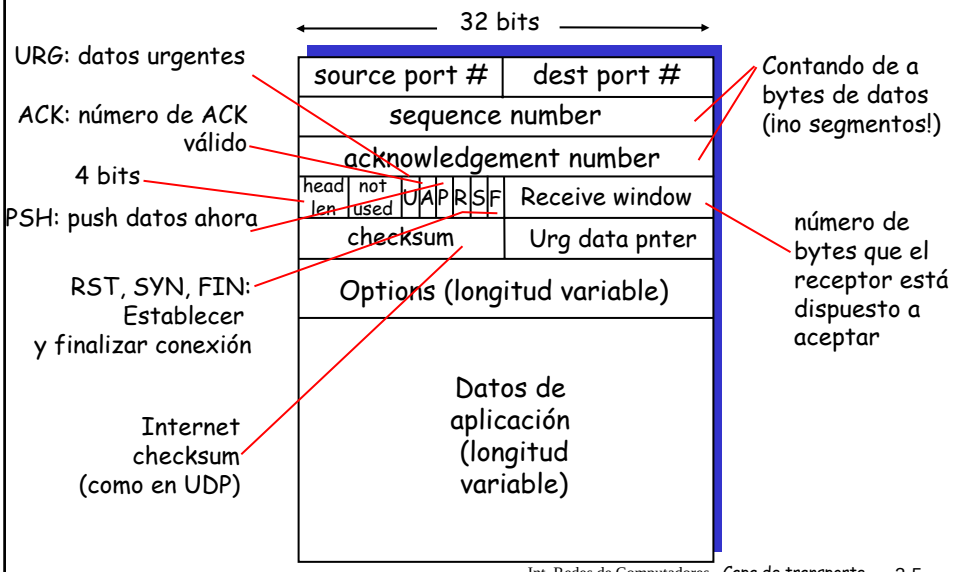
- diciembre del 98
  - RFC 2460: TCP con IPv6
- abril del 99
  - RFC **2581**: Control de congestión de TCP
- junio del 00
  - RFC 2873: Interacción de TCP y IP ("bits de precedencia")
- noviembre del 2000
  - RFC **2988**: *timer* de retransmisión
- septiembre del 06
  - RFC 4614: hoja de ruta de RFCs relacionadas con TCP

TCP: Aspectos generales

- punto a punto:
  - un *sender*, un *receiver*
  - entre *end systems*
- flujo de bytes en orden y confiable:
  - no alineado a bordes de mensajes
- *pipelined*:
  - El control de congestión y el control de flujo de TCP definen el tamaño de ventana
- *Buffers* para enviar y recibir

- servicio de datos *full duplex*
  - flujo de datos en ambas direcciones en la misma (única) conexión
  - MSS: *maximum segment size*
- orientado a conexión:
  - *handshaking* (intercambio de mensajes de control) inicializa el estado del *sender* y del *receiver* previo al intercambio de datos
- flujo controlado:
  - el *sender* no abruma al *receiver*

TCP: estructura del segmento



TCP: Nros. de secuencia y ACKs

Nros. de secuencia:

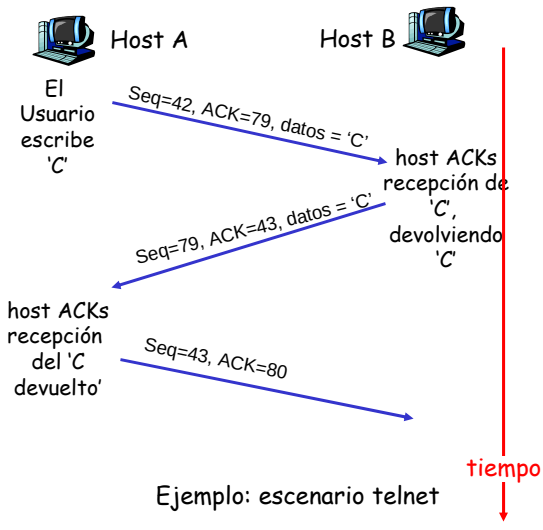
- *byte stream* número del primer byte de datos en el segmento

ACKs:

- Nro. de sec. del próximo byte esperado desde el otro lado
- ACK acumulativo

P: ¿Cómo maneja el receptor la llegada de segmentos fuera de orden?

- A: La especificación de TCP nada dice. Queda librado a la implementación de turno.



Ejemplo: escenario telnet

TCP: *Timeout* y *Round Trip Time*

P: ¿Cómo definir el valor del *timeout* de TCP?

- mayor al RTT
  - pero el RTT cambia
- demasiado corto: *timeout* prematuro
  - retransmisiones innecesarias
- demasiado largo: lenta reacción a pérdida de un segmento

P: ¿Cómo estimar el RTT?

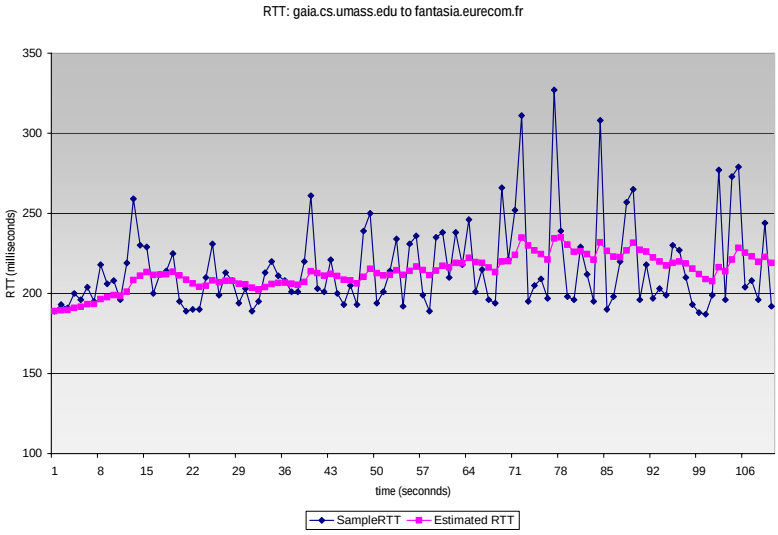
- **MuestraRTT**: tiempo medido desde la transmisión de un segmento hasta la recepción de su ACK
  - ignorando retransmisiones
- **MuestraRTT** varía, buscamos estimar un RTT "suavizado"
  - Promedio de varias medidas recientes, no solamente el valor actual de **MuestraRTT**

TCP: *Timeout* y *Round Trip Time*

EstimaciónRTT = (1- α)* EstimaciónRTT + α * MuestraRTT

- La influencia de las muestras anteriores decrece exponencialmente
- Valor típico: α = 0.125
- *"Congestion avoidance y control" – Van Jacobson & Michael Karels – November, 1988*

Ejemplo de la **estimación** del RTT



TCP: *Timeout* y *Round Trip Time*

Estableciendo el *timeout*

- **EstimaciónRTT** más "un margen de seguridad"
  - Variación amplia en **EstimaciónRTT** -> mayor margen de seguridad
- primero contempla cuánto la **MuestraRTT** se aparta de la **EstimaciónRTT**:

$$\text{DesvRTT} = (1-\beta) * \text{DesvRTT} + \beta * |\text{MuestraRTT} - \text{EstimaciónRTT}|$$

(típicamente, $\beta = 0.25$)

Entonces se establece el intervalo de *timeout*:

$$\text{IntervaloTimeout} = \text{EstimaciónRTT} + 4 * \text{DesvRTT}$$

Agenda

- 3.1 Servicios de la capa de transporte
- 3.2 Multiplexación y demultiplexación
- 3.3 Transporte no orientado a conexión: UDP
- 3.4 Principios de la transferencia de datos confiable
- 3.5 Transporte orientado a conexión: TCP
  - estructura del segmento
  - transferencia de datos confiable
  - control de flujo
  - gestión de la conexión
- 3.6 Principios del control de congestión
- 3.7 Control de congestión de TCP

TCP: transferencia de datos confiable

- TCP crea un servicio **rdt** sobre el servicio no confiable de IP
- Segmentos *pipelined*
- ACKs acumulativos
- TCP utiliza un único *timer* de retransmisión
- Las retransmisiones son disparadas por:
  - eventos *timeout*
  - ACKs duplicados
- Inicialmente consideramos un transmisor TCP simplificado:
  - ignora ACKs duplicados
  - ignora control de flujo y control de congestión
  - los datos son menores a MSS

TCP: eventos del emisor

Datos recibidos desde Aplicación

- Crea segmento con número de secuencia
- El número de sec. es el número del primer byte de datos en el segmento
- lanza *timer* si ya no estaba corriendo (pensar al *timer* asociado al segmento más viejo no-ACKed)
- Intervalo de expiración: `IntervaloTimeout`

timeout:

- Retransmite el segmento que causó el *timeout*
- Re-lanza el *timer*

Ack recibido:

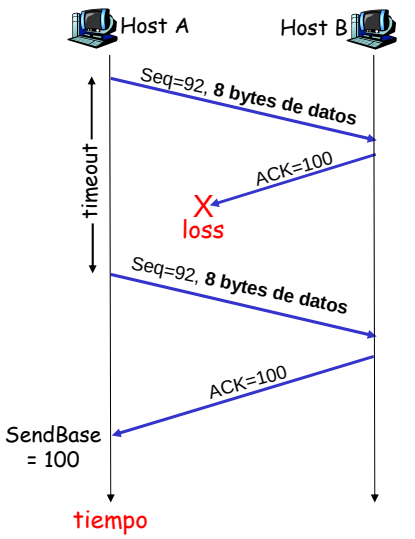
- Si reconoce segmentos no reconocidos antes
  - Actualiza la información de segmentos reconocidos
  - lanza el timer si hay segmentos no-ACKed


```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)

  event: data received from application above
      create TCP segment with sequence number NextSeqNum
      if (timer currently not running)
          start timer
      pass segment to IP
      NextSeqNum = NextSeqNum + length(data)

  event: timer timeout
      retransmit not-yet-acknowledged segment with
          smallest sequence number
      start timer

  event: ACK received, with ACK field value of y
      if (y > SendBase) {
          SendBase = y
          if (there are currently not-yet-acknowledged segments)
              start timer
          }

} /* end of loop forever */
```

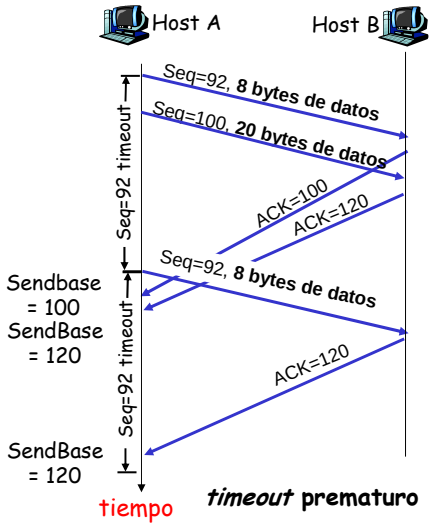
Emisor TCP (simplificado)

Comentario:
• SendBase-1: last cumulatively ack'ed byte
Ejemplo:
• SendBase-1 = 51; y= 83, entonces el receptor busca 83+ ;
y > SendBase, Nuevos datos son ACKed

TCP: escenarios de retransmisión

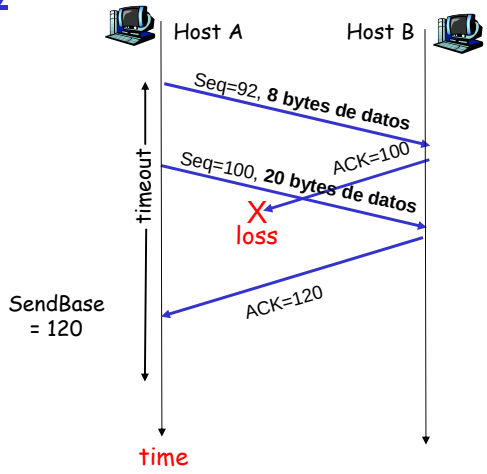


escenario de pérdida de ACK



timeout prematuro

TCP: escenarios de retransmisión (más)



Escenario ACK acumulativo

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

Evento en el Receptor	Acción de Receptor TCP
Llega segmento en orden, con nro. de sec. esperado. Todos los datos hasta el nro. de sec. esperado ya están ACKed	Retrasar ACK. Espera hasta 500ms por el sig. segmento. Si no hay sig. segmento en ese intervalo, enviar ACK
Llega un segmento en orden, con nro. de sec. esperado. Otro segmento en orden tiene ACK pendiente	Envía inmendiatamente un ACK acumulativo, Reconociendo ambos segmentos en orden
Llega un segmento fuera de orden con nro. de sec. mayor al esperado.*Gap* detectado	Envía inmediatamente ***duplicate ACK***, indicando nro. de sec. del sig. byte esperado (límite inferior del *gap*)
Llega un segmento que parcial o completamente llena el *gap*.	Envío inmediato de ACK, si dicho segmento comienza en el extremo inferior del *gap*

Fast Retransmit

- El período de *timeout* generalmente es largo:
  - Delay largo antes de re-enviar el paquete perdido
- Detectar segmentos perdidos a través de ACKs duplicados.
  - El *sender* frecuentemente envía varios segmentos de manera consecutiva
  - Si el segmento se pierde, podremos recibir varios ACKs duplicados
- Si el *sender* recibe 3 ACKs duplicados para los mismos datos, supone que el segmento de datos posterior al ACKed se perdió:
  - fast retransmit: re-enviar el segmento antes que expire el *timer* de retransmisión

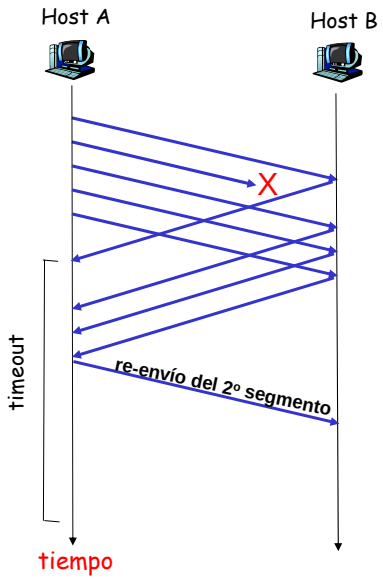


Figure 3.37 Re-enviando un segmento después del triple ACK duplicado

Algoritmo *Fast retransmit*

```
event: ACK received, with ACK field value of y
            if (y > SendBase) {
                SendBase = y
                if (there are currently not-yet-acknowledged segments)
                    start timer
              }
            else {
                increment count of dup ACKs received for y
                if (count of dup ACKs received for y = 3) {
                    resend segment with sequence number y
                 }
```

un ACK duplicado para un segmento ya ACKed

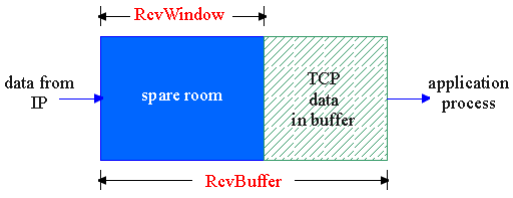
fast retransmit

Agenda

- 3.1 Servicios de la capa de transporte
- 3.2 Multiplexación y demultiplexación
- 3.3 Transporte no orientado a conexión: UDP
- 3.4 Principios de la transferencia de datos confiable
- 3.5 Transporte orientado a conexión: TCP
  - estructura del segmento
  - transferencia de datos confiable
  - control de flujo
  - gestión de la conexión
- 3.6 Principios del control de congestión
- 3.7 Control de congestión de TCP

TCP: Control de flujo

- el lado receptor de la conexión TCP tiene un *buffer* de recepción:



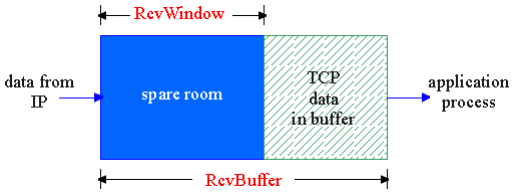
- El proceso de aplicación puede ser lento leyendo del *buffer*

Control de flujo

El emisor no debe desbordar el *buffer* del receptor transmitiendo muy rápido demasiados datos

- Servicio de equiparación de velocidad: adecuar la velocidad de envío con la velocidad de "drenado" del *buffer* por parte de la aplicación del receptor

Control de flujo de TCP: cómo funciona



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

- El receptor publica el espacio disponible en su buffer, incluyendo el valor de **RcvWindow** en los segmentos
- El emisor limita los datos no-ACKed a **RcvWindow**
  - garantiza no desbordar el *buffer* del receptor

- Para que no desborde el *buffer*
  ÚltimoByteRcvd – ÚltimoByteRead <= RcvBuffer
- espacio libre en *buffer*

= RcvWindow = RcvBuffer-[ÚltimoByteRcvd - ÚltimoByteRead]

Agenda

- 3.1 Servicios de la capa de transporte
- 3.2 Multiplexación y demultiplexación
- 3.3 Transporte no orientado a conexión: UDP
- 3.4 Principios de la transferencia de datos confiable
- 3.5 Transporte orientado a conexión: TCP
  - estructura del segmento
  - transferencia de datos confiable
  - control de flujo
  - gestión de la conexión
- 3.6 Principios del control de congestión
- 3.7 Control de congestión de TCP

TCP: Gestión de la conexión

Recordar: el emisor y el receptor de TCP establecen una "conexión" antes de intercambiar segmentos de datos

- inicializa variables de TCP:
  - nros. de sec.
  - *buffers*, información de control de flujo (p.e. **RcvWindow**)
- *cliente:* quien inicia la conexión

```
Socket clientSocket = new
Socket("hostname","port
number");
```

- *servidor:* contactado por el cliente

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

Three way handshake:

Paso 1: el *host* cliente envía un segmento SYN al servidor
- especifica el nro. de sec. inicial
- sin datos

Paso 2: el *host* servidor recibe el SYN, responde con un segmento SYN-ACK sin datos
- el servidor reserva *buffers*
- especifica el nro.de sec. inicial del servidor

Paso 3: el cliente recibe el SYN-ACK, responde con un segmento ACK, que puede tener datos

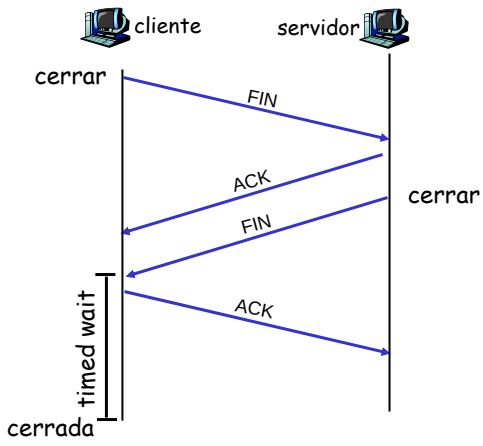
TCP: Gestión de la conexión (cont.)

Cerrando una conexión:

client closes socket:
`clientSocket.close();`

Paso 1: el sistema final *cliente* envía un segmento de control FIN al *server*

Paso 2: el *servidor* recibe el FIN y responde con un ACK. Cierra la conexión y envía un FIN.

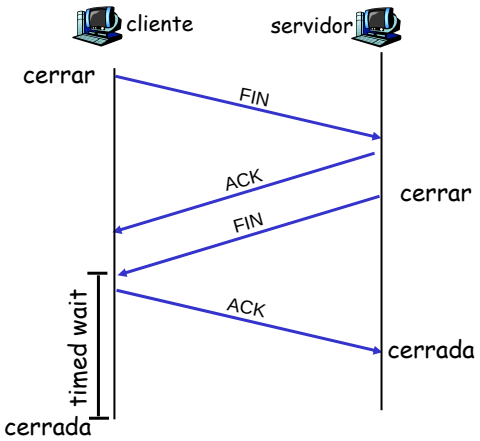


TCP: Gestión de la conexión (cont.)

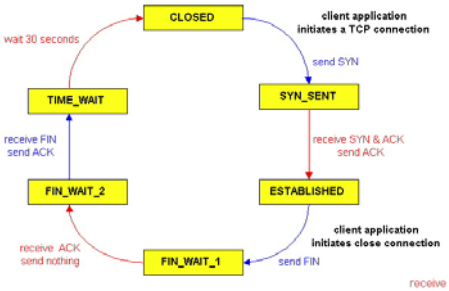
Paso 3: el *cliente recibe el* FIN y responde con un ACK.

Paso 4: el *servidor* recibe el ACK. Cierra la conexión.

Nota: con pequeñas modificaciones, puede manejar FINs simultáneos.

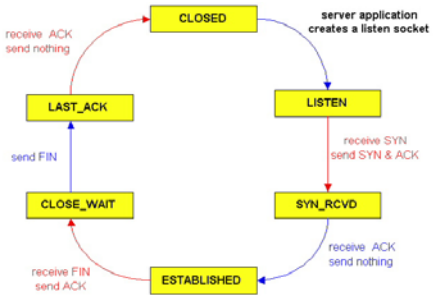


TCP : Gestión de la conexión (cont.)



Ciclo de vida del cliente TCP

Ciclo de vida del servidor TCP



Algunos comentarios más

- Bandera RST (*Reset*)
- Segmento SYN a un puerto TCP
  - Segmento SYNACK
  - Segmento RST
  - Nada
- Mensaje UDP a puerto que no está escuchando
  - Puedo recibir mensaje ICMP
- *SYN flooding attack*

Agenda

- 3.1 Servicios de la capa de transporte
- 3.2 Multiplexación y demultiplexación
- 3.3 Transporte no orientado a conexión: UDP
- 3.4 Principios de la transferencia de datos confiable
- 3.5 Transporte orientado a conexión: TCP
  - estructura del segmento
  - transferencia de datos confiable
  - control de flujo
  - gestión de la conexión
- 3.6 Principios del control de congestión
- 3.7 Control de congestión de TCP

Principios del Control de Congestión

Congestión:

- informalmente: "demasiadas fuentes enviando muy rápido demasiado tráfico a ser manejado por la red"
- ¡Diferente de control de flujo!
- manifestaciones:
  - pérdida de paquetes (*buffer overflow* en los routers)
  - grandes retardos (encolamiento en los *buffers* de los routers)

Principios del Control de Congestión

- Resulta en una injusta y pobre utilización de los recursos de la red
  - Los recursos son utilizado por paquetes posteriormente descartados
  - Retransmisiones
  - Pobre asignación de recursos
- ¡Un problema importante!

Perspectiva histórica

- *"Congestion avoidance y control" – Van Jacobson & Michael Karels – November, 1988*
- Si se respeta el "principio de conservación de los paquetes", la congestión debería ser la excepción y no la regla
- Dicho principio dice que para una conexión establecida y "en equilibrio", un nuevo paquete no puede ser introd. en la red antes que otro paquete salga de ella.
- En equilibrio: conexión estable con una ventana completa "*in-flight*"

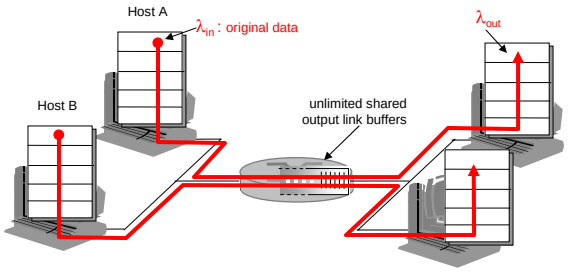
Introduction

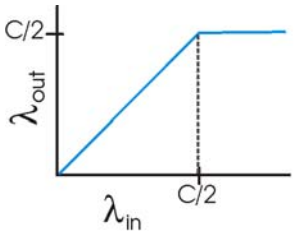
Computer networks have experienced an explosive growth over the past few years and with that growth have come severe congestion problems. For example, it is now common to see internet gateways drop 10% of the incoming packets because of local buffer overflows. Our investigation of some of these problems has shown that much of the cause lies in transport protocol implementations (*not* in the protocols themselves):

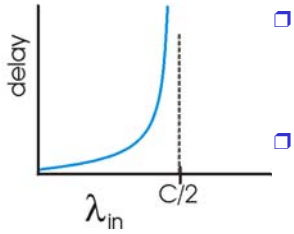
In October of '86, the Internet had the first of what became a series of 'congestion collapses'. During this period, the data throughput from LBL to UC Berkeley (sites separated by 400 yards and two IMP hops) dropped from 32 Kbps to 40 bps. We were fascinated by this sudden factor-of-thousand drop in bandwidth and embarked on an investigation of why things had gotten so bad. In particular, we wondered if the 4.3BSD (Berkeley UNIX) TCP was mis-behaving or if it could be tuned to work better under abysmal network conditions. The answer to both of these questions was "yes".

Causas/costos de la congestión: escenario 1

- 2 *senders*, 2 *receivers*
- 1 router, *buffers* infinitos
- sin retransmisiones



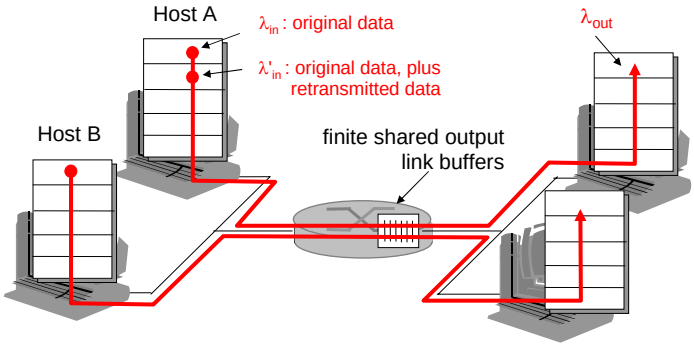




- Cuando hay congestión, tenemos grandes retardos
- máximo *throughput* alcanzable

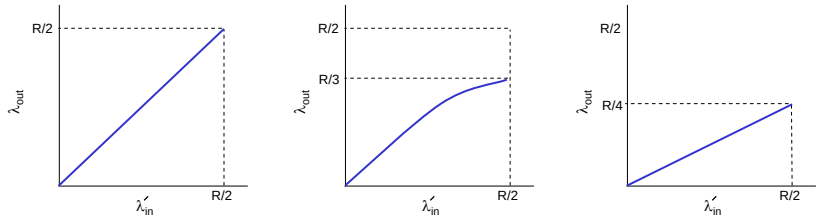
Causas/costos de la congestión: escenario 2

- 1 router, *buffers finitos*
- el *sender* retransmite paquetes perdidos



Causas/costos de la congestión: escenario 2

- siempre: $\lambda_{in} = \lambda_{out}$ (*goodput*)
- retransmisión "perfecta" sólo cuando hay pérdidas: $\lambda'_{in} > \lambda_{out}$
- La retransmisión de paquetes retrasados (no perdidos) hace λ'_{in} mayor (respecto al caso perfecto) para el mismo λ_{out}



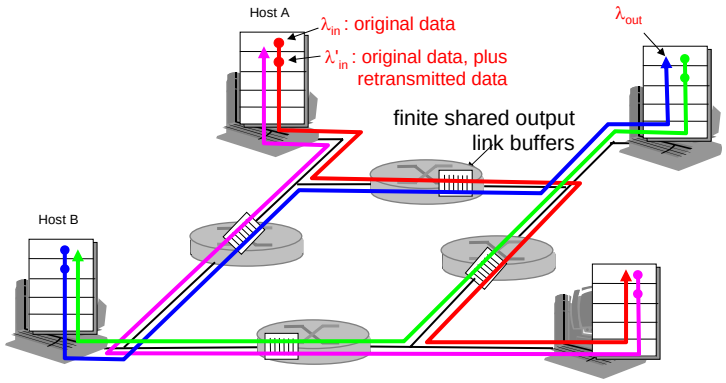
"costos" de la congestión:

- más trabajo (retransmisiones) para un *goodput* dado
- Retransmisiones innecesarias: los enlaces llevan múltiples copias de los paquetes

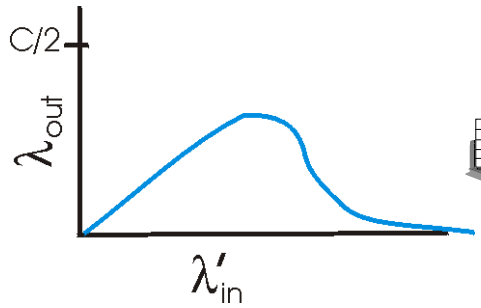
Causas/costos de la congestión: escenario 3

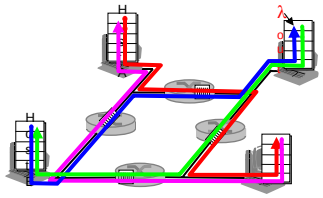
- 4 *senders*
- caminos *multihop*
- *timeout/retransmit*

P: ¿Qué ocurre si λ_{in} y λ'_{in} aumentan?



Causas/costos de la congestión: escenario 3





Otro "costo" de la congestión:

- cuando un paquete es descartado, ¡toda la capacidad de transmisión "aguas arriba" utilizada por dicho paquete fue desperdiciada!

Aproximaciones al control de congestión (1/2)

Objetivo: estrangular al emisor para asegurar que la carga en la
red es "razonable"

Dos estrategias posibles

- Entre extremos
  - No hay soporte explícito de la red
  - Los extremos la infieren
  - Tradicional de TCP

Aproximaciones al control de congestión (2/2)

Objetivo: estrangular al emisor para asegurar que la carga en la
red es "razonable"

- Asistida por la red
  - Los routers realimentan explícitamente a los extremos
  - Directa
    - Con un mensaje explícito
  - Indirecta
    - Marcando un campo en algún paquete
    - Involucra RTTs
    - *Explicit Congestion Notification* (ECN) – RFCs 3168 y 3540 y http://www.icir.org/floyd/ecn.html

Aclaración

- Si el emisor y el receptor de la conexión TCP están unidos por un cable (p.e. UTP) es muy probable que podamos llegar a tener "in-flight" la ventana de recepción
- Si entre ellos hubiera un *switch* "razonable" y con una carga "razonable", quizás también
- La congestión puede aparecer "con fuerza" cuando tenemos una red entre las entidades TCP, y debemos tratar de evitarla-controlarla

Agenda

- 3.1 Servicios de la capa de transporte
- 3.2 Multiplexación y demultiplexación
- 3.3 Transporte no orientado a conexión: UDP
- 3.4 Principios de la transferencia de datos confiable
- 3.5 Transporte orientado a conexión: TCP
  - estructura del segmento
  - transferencia de datos confiable
  - control de flujo
  - gestión de la conexión
- 3.6 Principios del control de congestión
- 3.7 Control de congestión de TCP

Control de congestión de TCP

- RFC 2581, abril 1999
- Entre sistemas finales
  - Sin asistencia de la red
- Límites de transmisión del emisor:
  LastByteSent-LastByteAcked ≤ min {CongWin, RcvWin}
- Sin mucha rigurosidad,

$$\text{tasa} = \frac{\text{CongWin}}{\text{RTT}} \text{ bytes/sec}$$

- La **CongWin** es dinámica; función de la congestión de la red percibida por el emisor

¿Cómo percibe la congestión el emisor?

- evento de pérdida = *timeout* o 3 ACKs duplicados
- El emisor TCP reduce la tasa (**CongWin**) después de de un evento de pérdida

Cuatro algoritmos:

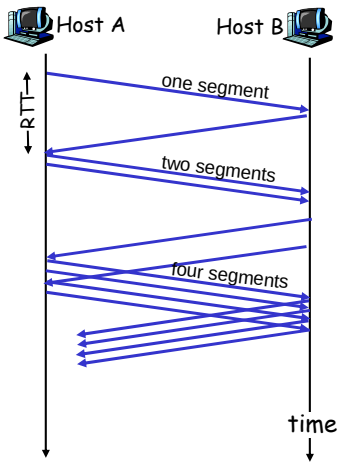
- *slow start*
- *congestion avoidance*
- *fast retransmit*
- *fast recovery*

Control de congestión de TCP: Slow Start ("arranque lento o en frío")

- Cuando la conexión comienza,
  CongWin = 1 MSS
  - Ejemplo: MSS = 500 bytes & RTT = 200 msec
  - Velocidad inicial = 20 kbps
- El ancho de banda disponible podría ser >> MSS/RTT
  - Es deseable que rápidamente lleguemos a una velocidad respetable
- Cuando la conexión comienza, se incrementa la velocidad **exponencialmente** (no linealmente) hasta el primer evento de pérdida

TCP Slow Start (más)

- Cuando la conexión comienza, se incrementa la velocidad exponencialmente hasta el primer evento de pérdida
  - La **CongWin** se incrementa en a lo sumo SMSS bytes por cada ACK "nuevo" recibido ("conservación de los paquetes")
- Resumen: la velocidad inicial es lenta pero crece exponencialmente



Refinamiento: inferiendo la pérdida

- Luego de un evento "3 ACKs duplicados":
  - **CongWin** se baja a la mitad
  - Luego la ventana crece **linealmente**, en 1 SMSS por RTT
- Pero, luego de un evento "*timeout*":
  - **CongWin** vale 1 MSS;
  - La ventana crece **exponencialmente** hasta un umbral, y luego, linealmente

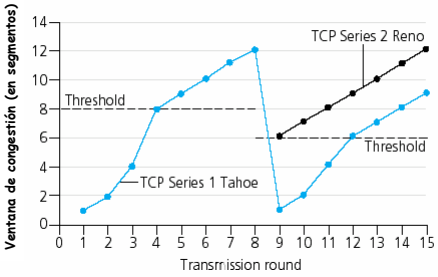
Filosofía:

- 3 ACKs duplicados indica que la red es capaz de entregar algunos segmentos
- *timeout* indica un escenario de congestión "más preocupante"

Refinamiento

P: ¿Cuándo debería el incremento **exponencial** cambiar a **lineal?**

R: Cuando **CongWin** alcanza la mitad de su valor previo al *timeout*.



Implementación:

- Variable *Threshold*
  - *ssthresh*: arbitrariamente alto (por ejemplo, *RcvWnd*)
- Reno: en un evento de pérdida, el *Threshold* es fijado a la mitad del valor de **CongWin** justo antes del evento mencionado

Algunos sabores de TCP

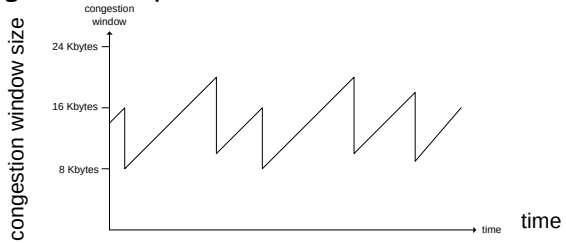
- Reactivos
  - Tahoe (Van Jacobson 1988)
    - Slow Start, Congestion Avoidance y Fast Retransmit
  - Reno (Van Jacobson 1990)
    - RFC 2581 – abril 1999
    - Tahoe + Fast Recovery
  - New Reno
    - RFC 3782 – abril 2004
    - Para contemplar el caso de pérdida de segmentos consecutivos y no poder utilizar la opción SACK
- Proactivos
  - Vegas
    - Busca evitar la congestión tratando de detectar cuando es inminente que ella ocurra (RTT) y así, reduciendo la tasa de envío. Cambios en parte de la implementación de SS y CA

Control de congestión de TCP:

Incremento Aditivo, Decremento Multiplicativo (AIMD)

- Propuesta: incrementar la tasa de transmisión (tamaño de ventana), probando por el ancho de banda utilizable, hasta que ocurra una pérdida
  - *incremento aditivo:* incrementa **CongWin** en 1 MSS cada RTT hasta que se detecta una pérdida
  - *decremento multiplicativo*: recorta a la mitad la **CongWin** luego de una pérdida

Comportamiento de "diente de sierra": Probando en busca de ancho de banda



Resumen: Control de Congestión de TCP

- Cuando **CongWin** está debajo de **Threshold**, el emisor está en la fase *slow-start*, la ventana crece **exponencialmente**.
- Cuando **CongWin** está sobre **Threshold**, el emisor está en la fase *congestion-avoidance*, la ventana crece **linealmente**.
- Cuando ocurre un triple ACK duplicado, el **Threshold** pasa a valer **CongWin/2** y **CongWin se pone al valor de Threshold**.
- Cuando ocurre un *timeout*, el **Threshold** pasa a valer **CongWin/2** y **CongWin,** 1 MSS.

TCP: control de congestión del emisor

Estado	Evento	Acción del Emisor TCP	Comentario
Slow Start (SS)	Recepción de ACK para datos previamente no-ACKed	CongWin = CongWin + MSS, If (CongWin > Threshold) pasa a estado "Congestion Avoidance"	Se dobla CongWin por cada RTT
Congestion Avoidance (CA)	Recepción de ACK para datos previamente no-ACKed	CongWin = CongWin + MSS x (MSS/CongWin)	Incremento aditivo. Aumento de CongWin en 1 MSS por cada RTT
SS or CA	Evento de pérdida detectado: Triple ACK duplicado	Threshold = CongWin/2, CongWin = Threshold pasa a estado "Congestion Avoidance"	*Fast recovery*, implementando disminución multiplicativa. CongWin no puede ser menor a 1 MSS.
SS or CA	*Timeout*	Threshold = CongWin/2, CongWin = 1 MSS, pasa a estado "Slow Start"	
SS or CA	ACK duplicado	Incrementa el contador de ACK duplicado para el segmento que es ACKed	CongWin y Threshold no cambian

¿Pérdida de paquetes = congestión?

- Respuesta parcial: "*depende*"
- Completando la respuesta
  - En medios no hostiles (redes UTP actuales, redes de fibra óptica) podríamos llegar a afirmar que sí
  - En medios hostiles (redes inalámbricas) podemos afirmar: "no necesariamente"
  - TCP para redes inalámbricas, porque las "señales de congestión" del TCP tradicional en este contexto, no siempre son adecuadas

TCP *throughput* ("para diente de sierra")

- ¿Cuál es el *throughout* promedio de TCP en función del tamaño de ventana y del RTT?
  - No tenemos en cuenta la fase *slow start*
- Sea **w** el tamaño (en bytes) de ventana cuando ocurre una pérdida.
- Cuando la ventana es w, el *throughput* es aprox. **w/RTT**
- Justo después de la pérdida, la ventana cae a **w/2**, y el *throughput* pasa a **w/2RTT**.
- Como el throughput se incrementa linealmente entre estos dos valores,
  - *throughput* promedio: **0.75 w/RTT**

TCP Futuro: TCP sobre "elefantes" (LFN: *Long and Fat Networks*)

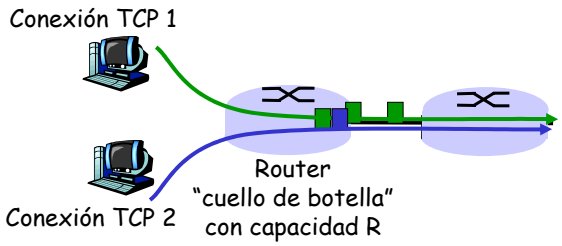
- Las aplicaciones cambian y por lo tanto los servicios requeridos a TCP también
- RFC 3649: *High Speed TCP for Large Congestion Windows*
- Ejemplo : segmentos de 1500 bytes, 100ms RTT, precisa 10 Gbps de *throughput*
- Requiere un tamaño de ventana de W = 83.333 segmentos en tránsito
- *Throughput* en función de la tasa de pérdida (L):

$$\frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

- ➜ aprox. L = 2 x 10^{-10} o sea, una pérdida cada 5 x 10^{9} segmentos, o sea, 1 pérdida cada 1 hora y 40 minutos… glup!!
- Investigación sobre nuevas versiones de TCP para alta velocidad

TCP Imparcialidad

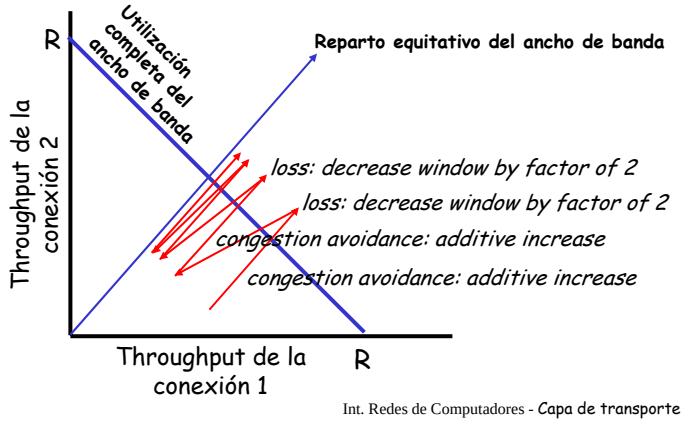
Objetivo de la imparcialidad: si **K** sesiones TCP comparten el mismo enlace "cuello de botella" de ancho de banda **R**, cada uno debería tener una tasa promedio de **R/K**



¿Por qué es imparcial TCP? Fase CA

Dos sesiones compitiendo:

- Incremento aditivo
- Decremento multiplicativo



Imparcialidad (más)

Imparcialidad y UDP

- Las aplicaciones multimedia frecuentemente no utilizan TCP
  - no quieren que la tasa se vea estrangulada por el control de congestión
- En su lugar utilizan UDP:
  - inyectan audio/video a una tasa constante, toleran pérdidas de paquetes
- Área de investigación
  - *TCP friendly*
  - Protocolos de control de congestión

Imparcialidad y conexiones TCP paralelas

- nada impide a una aplicación basada en TCP abrir varias conexiones TCP paralelas entre 2 *hosts*.
- Los *web browsers* hacen ésto
- Ejemplo: enlace de tasa R soportando 9 conexiones;
  - Si nueva aplic. requiere 1 conexión TCP, recibe tasa R/10
  - Pero, si nueva aplic. requiere 11 conexiones TCP, ¡ recibe R/2 !

Capítulo 3: Resumen

- principios detrás de los servicios de la capa de transporte:
  - Multiplexación, demultiplexación
  - Transferencia de datos confiable
  - Control de flujo
  - Control de congestión
- Instanciación e implementación en Internet
  - UDP
  - TCP

Lo que se viene:

- dejamos el borde (capas de aplicación y de transporte)
- nos metemos en el "*core*" de la red