

## 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

Int. Redes de Computadores - Capa de transporte 3-1

## 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



Int. Redes de Computadores - Capa de transporte 3-2

## 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

Int. Redes de Computadores - Capa de transporte 3-3

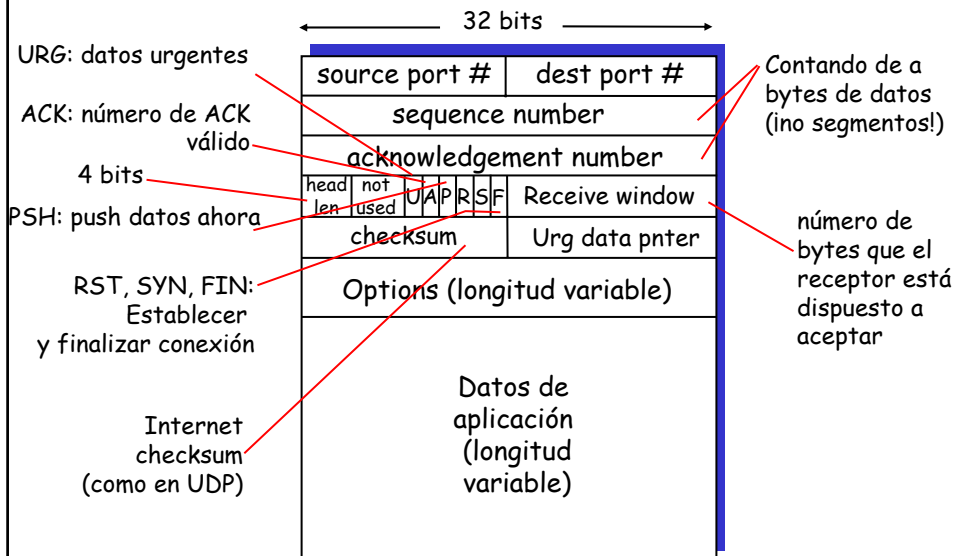
## TCP: Aspectos generales

- **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 abrumba al *receiver*



Int. Redes de Computadores - Capa de transporte 3-4

## TCP: estructura del segmento



## TCP: Nros. de secuencia y ACKs

### Nros. de secuencia:

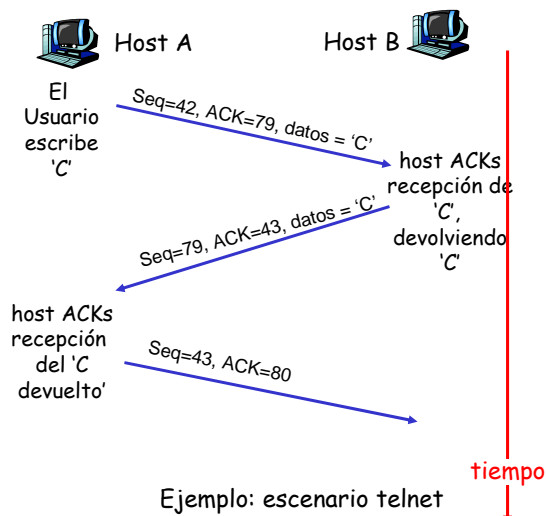
- *byte stream* número del primer byte de datos en el segmento

### 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.



## 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**

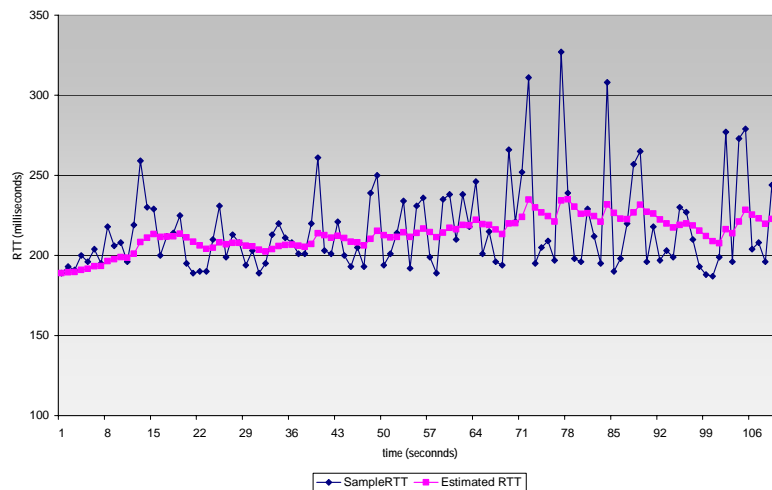
## TCP: Timeout y Round Trip Time

$$\text{EstimaciónRTT} = (1 - \alpha) * \text{EstimaciónRTT} + \alpha * \text{MuestraRTT}$$

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

## Ejemplo de la estimación del RTT

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr



Int. Redes de Computadores - Capa de transporte 3-9

## 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*:

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

Int. Redes de Computadores - Capa de transporte 3-10

## 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

Int. Redes de Computadores - Capa de transporte 3-11

## 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

Int. Redes de Computadores - Capa de transporte 3-12

## 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

Int. Redes de Computadores - Capa de transporte 3-13

```
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

Int. Redes de Computadores - Capa de transporte 3-14





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

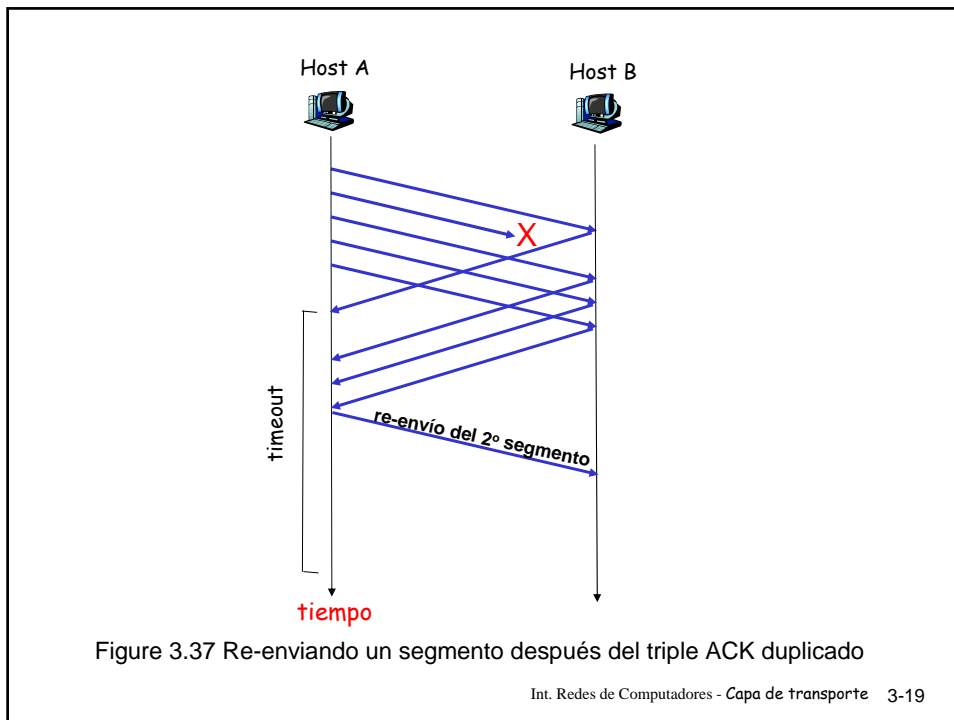
Evento en el Receptor	Acción de Receptor TCP
Llega segmento en orden, con nro. de sec. esperado. Todos los datos hasta el nro. de sec. esperado ya están ACKed	Retrasar ACK. Espera hasta 500ms por el sig. segmento. Si no hay sig. segmento en ese intervalo, enviar ACK
Llega un segmento en orden, con nro. de sec. esperado. Otro segmento en orden tiene ACK pendiente	Envía inmediatamente 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 <b>duplicate ACK</b> , 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

Int. Redes de Computadores - Capa de transporte 3-17

## 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

Int. Redes de Computadores - Capa de transporte 3-18



## 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*

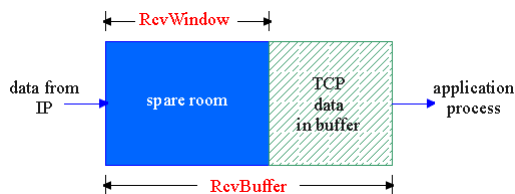
## Agenda

- 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

Int. Redes de Computadores - Capa de transporte 3-21

## 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*

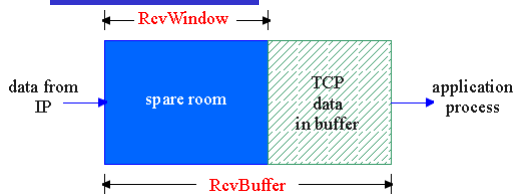
### Control de flujo

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

Int. Redes de Computadores - Capa de transporte 3-22

## 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*  
$$\text{ÚltimoByteRcvd} - \text{ÚltimoByteRead} \leq \text{RcvBuffer}$$
- espacio libre en *buffer*  
$$= \text{RcvWindow} = \text{RcvBuffer} - [\text{ÚltimoByteRcvd} - \text{ÚltimoByteRead}]$$

Int. Redes de Computadores - Capa de transporte 3-23

## 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

Int. Redes de Computadores - Capa de transporte 3-24

## 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

Int. Redes de Computadores - Capa de transporte 3-25

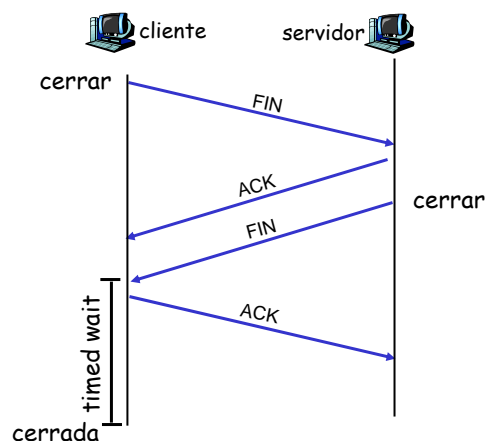
## TCP: Gestión de la conexión (cont.)

### 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*.



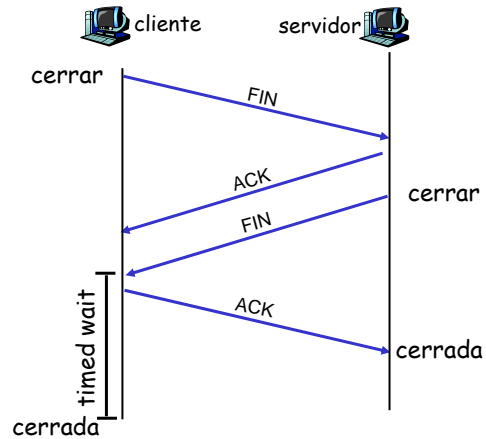
Int. Redes de Computadores - Capa de transporte 3-26

## TCP: Gestión de la conexión (cont.)

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

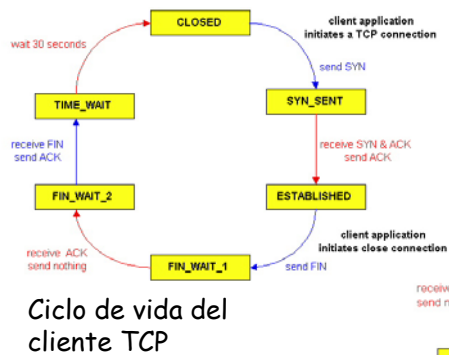
**Paso 4:** el *servidor* recibe el ACK. Cierra la conexión.

**Nota:** con pequeñas modificaciones, puede manejar FINs simultáneos.

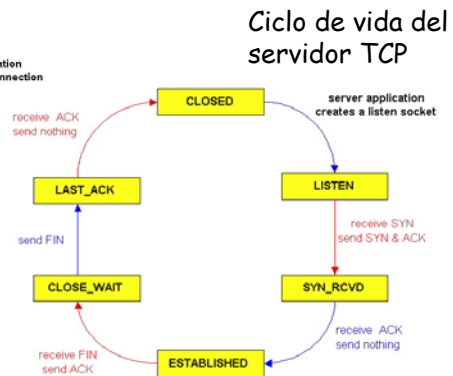


Int. Redes de Computadores - Capa de transporte 3-27

## TCP : Gestión de la conexión (cont.)



Ciclo de vida del cliente TCP



Ciclo de vida del servidor TCP

Int. Redes de Computadores - Capa de transporte 3-28

## 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*