


UNIVERSIDAD NACIONAL DE INGENIERÍA

Computación Centrada en Redes

# Entrega confiable: de UDP a la ventana deslizante

## Transporte y aplicaciones

**Gerson Garrido**

Semana 9

# Hoy

## Sesión A: transporte y entrega confiable

1. Qué agrega el transporte sobre IP
2. Demultiplexación: dos campos o cuatro
3. UDP y su checksum
4. Un protocolo, paso a paso
5. Stop-and-wait en un enlace largo
6. Dos generales

## Sesión B: ventana deslizante

7. Pipelining y BDP
8. Go-Back-N
9. Selective Repeat
10. Comparación
11. Ventana y números de secuencia
12. Laboratorio: el hackathon

**Lectura**

Kurose y Ross, secs. 3.1, 3.2, 3.3 y 3.4.1 (sesión A), 3.4.2, 3.4.3 y 3.4.4 (sesión B). CS168: *Transport Layer Principles*. Complementaria: RFC 768.

SECCIÓN 01

# Qué agrega el transporte

01

# IP entrega entre hosts; el transporte, entre procesos

**Definición**

La capa de transporte da **comunicación lógica entre procesos**: cada aplicación trabaja como si estuviera conectada directamente con la otra.

- **Corre en los extremos.** Un router hace forwarding mirando IP: no retransmite nada.
- **No lo arregla todo.** Convierte pérdidas en retransmisiones, pero no inventa ancho de banda ni baja el retardo mínimo.

**Dos cosas muy distintas**

Una **dirección más fina**, el puerto, que es barata. Y, si se la pide, **confiabilidad**, que cuesta tiempo, memoria y complejidad.

# Dos servicios

| Servicio | UDP | TCP |
|---|---|---|
| Entrega al proceso correcto | sí | sí |
| Detección de errores | sí, con checksum | sí, con checksum |
| Entrega confiable y en orden | no | sí |
| Conexión | no | sí |
| Control de flujo y de congestión | no | sí |
| Límites de mensaje | se conservan | no: stream de bytes |
| Garantías de retardo o de ancho de banda | no | no |

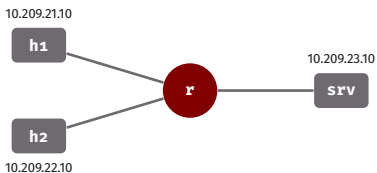
TCP no promete rapidez: promete que lo que llegue esté completo y en orden.

SECCIÓN 02

# Demultiplexación

02

# La red de la sesión A




**Demultiplexar**

En el host de destino, usar el encabezado de transporte para entregar cada segmento al **socket** correcto.

Un puerto es un número de 16 bits. Cuando un cliente no elige el suyo, recibe uno efímero.

`udp_demux` y `tcp_demux` imprimen qué socket recibe cada dato. `demux_client` elige el puerto de origen: `h1` y `h2` van a usar el mismo.

# UDP: dos campos

un servidor, tres clientes

```
$ docker exec clab-w09-par-srv cat /tmp/udp.log
socket 3  de 10.209.21.10:5000  "de h1, puerto 5000"
socket 3  de 10.209.22.10:5000  "de h2, puerto 5000"
socket 3  de 10.209.21.10:5001  "de h1, puerto 5001"
$ docker exec clab-w09-par-srv sh -c "ss -uan | grep -E 'State|9009'"
State  Recv-Q Send-Q Local Address:Port  Peer Address:Port
UNCONN 0      0            0.0.0.0:9009       0.0.0.0:*
```

- Un socket UDP sin `connect` se identifica por **IP y puerto de destino**.
- Todos caen en el mismo socket. Separarlos es trabajo de la aplicación, con lo que da `recvfrom`.

# TCP: cuatro campos

los mismos tres clientes

```
$ docker exec clab-w09-par-srv cat /tmp/tcp.log
socket 4  conexión nueva de 10.209.21.10:5000
socket 5  conexión nueva de 10.209.22.10:5000
socket 6  conexión nueva de 10.209.21.10:5001
$ docker exec clab-w09-par-srv sh -c "ss -tan | grep -E 'State|9009'"
State  Recv-Q Send-Q Local Address:Port  Peer Address:Port
LISTEN 0      8           0.0.0.0:9009      0.0.0.0:*
ESTAB  0      0       10.209.23.10:9009  10.209.21.10:5000
ESTAB  0      0       10.209.23.10:9009  10.209.21.10:5001
ESTAB  0      0       10.209.23.10:9009  10.209.22.10:5000
```

Cuatro sockets en un puerto. Las dos conexiones del puerto 5000 se separan solo por la IP de origen.

Error común: “un puerto es una conexión”. Un servidor en el 443 tiene miles de conexiones con ese puerto local: las separa el otro extremo.

# Un socket, o uno por cliente




Lo que cambia es cuántos campos usa el kernel para elegir el socket. Por eso un servidor TCP atiende a miles de clientes en un solo puerto.

# El matiz: UDP con `connect`

En UDP, `connect` no manda nada: le dice al kernel que el socket es para un solo otro extremo. Linux lo prefiere, porque es más específico.

segundo socket en el mismo puerto, conectado a h1:5000

```
$ docker exec clab-w09-par-srv cat /tmp/udp2.log
socket 4  de 10.209.21.10:5000  "de h1, puerto 5000"
socket 3  de 10.209.22.10:5000  "de h2, puerto 5000"
socket 3  de 10.209.21.10:5001  "de h1, puerto 5001"
$ docker exec clab-w09-par-srv sh -c "ss -uan | grep -E 'State|9009'"
State  Recv-Q Send-Q Local Address:Port  Peer Address:Port
ESTAB  0      0       10.209.23.10:9009  10.209.21.10:5000
UNCONN 0      0            0.0.0.0:9009       0.0.0.0:*
```

`ESTAB` sin handshake: es una marca local. El otro extremo no se entera.

# Cuando nadie escucha

puerto cerrado

```
$ demux_client tcp 10.209.23.10 9009 5002 x
connect: Connection refused
$ demux_client udp-connect 10.209.23.10 9009 5003 x
recv: Connection refused
$ demux_client udp 10.209.23.10 9009 5004 x
recv: Resource temporarily unavailable
```

- TCP: el kernel de `srv` contesta con un RST.
- UDP: contesta con ICMP de puerto inalcanzable…
- …pero ese aviso solo le llega a un socket **conectado**. El otro solo ve vencer su espera.

capturado en srv

```
10.209.23.10 > 10.209.21.10: ICMP 10.209.23.10 udp port 9009 unreachable, length 40
```

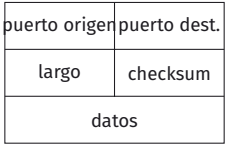
SECCIÓN 03

# UDP

03



# UDP: 8 bytes de encabezado




| Bytes | Campo | Valor |
|---|---|---|
| 1388 | Puerto origen | 5000 |
| 2331 | Puerto destino | 9009 |
| 000c | Largo | 12: 8 + 4 |
| b296 | Checksum | ahora |
| 686f 6c61 | Datos | hola |

capturado en srv

```
13:56:38.933668 IP (tos 0x0, ttl 63, id 29469, offset 0, flags [DF], proto UDP (17), length 32)
    10.209.21.10.5000 > 10.209.23.10.9009: UDP, length 4
    0x0000:  4500 0020 731d 4000 3f11 86fa 0ad1 150a
    0x0010:  0ad1 170a 1388 2331 000c b296 686f 6c61
```

# El checksum, con pseudo-encabezado

```
python3 udp_checksum.py 10.209.21.10 10.209.23.10 5000 9009 hola

pseudo-encabezado  0ad1 150a 0ad1 170a 0011 000c
encabezado UDP     1388 2331 000c 0000
datos              686f 6c61
suma               14d68
con acarreos       4d69
complemento        b296
checksum           b296
```

- **Pseudo-encabezado:** las dos IP, el protocolo y el largo. No viaja. Protege contra datagramas mal entregados.
- **Quien recibe** suma todo y debe obtener `ffff`: `4d69 + b296 = ffff`.
- Un **cero transmitido** significa “sin checksum” en IPv4. Si la cuenta da cero, se manda `ffff`. En IPv6, es obligatorio.

# Un datagrama dañado se descarta en silencio

5 % de corrupción con netem, durante una transferencia

```
$ docker exec clab-w09-par-h1 grep Udp: /proc/net/snmp
Udp: InDatagrams NoPorts InErrors OutDatagrams RcvbufErrors SndbufErrors InCsumErrors IgnoredMulti MemErrors
Udp: 919 0 0 931 0 0 0 0 0
$ docker exec clab-w09-par-h1 python3 /w09/tftp_mini.py get 10.209.23.10 50k /tmp/o
bytes=51200 tiempo=4.44s tasa=92.2 kbit/s duplicados=0 sha256=df1ba5f500ea33cb
$ docker exec clab-w09-par-h1 grep Udp: /proc/net/snmp
Udp: InDatagrams NoPorts InErrors OutDatagrams RcvbufErrors SndbufErrors InCsumErrors IgnoredMulti MemErrors
Udp: 1020 0 4 1033 0 0 4 0 0
```

**Idea clave**

Para la aplicación, el checksum convierte un **dato dañado** en un **dato perdido**, si lo detecta: es de 16 bits, y es débil.

# ¿Para qué sirve algo tan simple?

- **DNS.** Pregunta y respuesta chicas, sin gastar un handshake. Semana 12.
- **Medios en vivo.** Un paquete que llega 2 s tarde ya no le sirve a nadie.
- **Base para otro transporte.** QUIC corre sobre UDP, porque muchas middleboxes solo dejan pasar TCP y UDP. Semana 13. Y P3 construye sobre UDP lo que IP no da.

**Error común**

"UDP es más rápido que TCP". Sus bits viajan igual por el mismo enlace. No espera un handshake ni frena ante la congestión, y si manda de más, se pierde lo que sobra. El control de congestión pasa a ser responsabilidad de ustedes.

SECCIÓN 04

# Un protocolo, paso a paso

04

# Lo que queremos construir

**El servicio**

Lo que entrega el emisor llega al receptor **completo, sin daños, sin duplicados y en orden**. Solo contamos con un canal que puede fallar.

Un mecanismo a la vez, y solo cuando una falla concreta lo exija.

- Datos en un sentido; en el otro, solo confirmaciones.
- Un paquete en camino a la vez.
- `rdt_sim.py`: un simulador, con un guion de lo que le pasa a cada transmisión. 10 ms por sentido.
- El libro las llama rdt1.0, 2.0, 2.1, 2.2 y 3.0.

# Versión 1: canal perfecto, y un error de bits

```
python3 rdt_sim.py v1-errores

     0 ms  emisor   envía DATA "a"
    10 ms  receptor recibe y entrega "a"
    11 ms  emisor   envía DATA "b"  [el canal daña bits]
    21 ms  receptor recibe y entrega "?!?"
    22 ms  emisor   envía DATA "c"
    32 ms  receptor recibe y entrega "c"
    33 ms  emisor   terminó
entregado: a ?!? c   (INCORRECTO, 3 transmisiones)
```

Si el canal es perfecto, no hay nada que hacer. Basta un error para entregar basura sin enterarse.

# Versión 2: checksum, ACK y NAK

Como un dictado:

1. **Detectar:** checksum.
2. **Avisar:** ACK o NAK.
3. **Reparar:** repetir.

Mientras espera, el emisor no manda nada: **stop-and-wait**.

```
python3 rdt_sim.py v2-datos

   20 ms  emisor   envía DATA "b"  [el canal daña bits]
   30 ms  receptor recibe datos dañados
   30 ms  receptor envía NAK
   40 ms  emisor   recibe NAK: repite
   40 ms  emisor   envía DATA "b" (otra vez)
```

```
python3 rdt_sim.py v2-ack: ¿y si se daña el ACK?

    0 ms  emisor   envía DATA "a"
   10 ms  receptor recibe y entrega "a"
   10 ms  receptor envía ACK  [el canal daña bits]
   20 ms  emisor   recibe una respuesta dañada: no sabe qué era, repite
   20 ms  emisor   envía DATA "a" (otra vez)
   30 ms  receptor recibe y entrega "a"
entregado: a a b c   (INCORRECTO, 8 transmisiones)
```

# Versión 2.1: un bit de secuencia

**Número de secuencia**

Lo pone el emisor en cada paquete, para que el receptor distinga un paquete nuevo de la repetición de uno anterior.

```
python3 rdt_sim.py v21-ack

   0 ms  emisor   envía DATA 0 "a"
  10 ms  receptor recibe 0 y entrega "a"
  10 ms  receptor envía ACK  [el canal daña bits]
  20 ms  emisor   recibe una respuesta dañada: no sabe qué era, repite
  20 ms  emisor   envía DATA 0 "a" (otra vez)
  30 ms  receptor recibe 0: duplicado, no lo entrega
  30 ms  receptor envía ACK
```

El duplicado no se entrega, pero sí se confirma: el emisor evidentemente no se enteró.

# Versión 2.2: sin NAK

Si el ACK dice **qué número confirma**, un ACK repetido significa lo mismo que un NAK.

```
python3 rdt_sim.py v22-datos

  20 ms  emisor   envía DATA 1 "b"  [el canal daña bits]
  30 ms  receptor recibe datos dañados: confirma el último bueno
  30 ms  receptor envía ACK 0
  40 ms  emisor   recibe ACK 0: esperaba ACK 1, repite
  40 ms  emisor   envía DATA 1 "b" (otra vez)
  50 ms  receptor recibe 1 y entrega "b"
  50 ms  receptor envía ACK 1
```

Un tipo de mensaje menos. La idea reaparece la próxima semana, en el fast retransmit de TCP.

# Versión 3: pérdidas, con un temporizador

```
python3 rdt_sim.py v22-perdida: sin temporizador

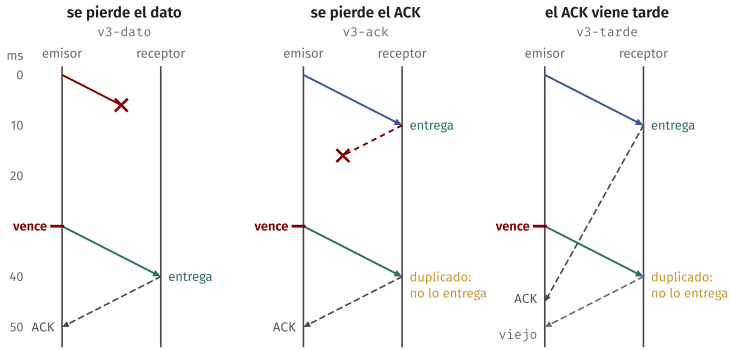
   20 ms  emisor   envía DATA 1 "b"  [el canal lo pierde]
   20 ms  emisor   se quedó esperando para siempre: nada en camino, ningún temporizador
entregado: a   (INCORRECTO, 3 transmisiones)
```

```
python3 rdt_sim.py v3-dato: con temporizador

   20 ms  emisor   envía DATA 1 "b"  [el canal lo pierde]
   50 ms  emisor   vence el temporizador
   50 ms  emisor   envía DATA 1 "b" (otra vez)
   60 ms  receptor recibe 1 y entrega "b"
   60 ms  receptor envía ACK 1
```

Lo arranca al enviar, lo apaga al recibir la confirmación y, si vence, repite.

# Tres causas, un solo síntoma




Desde el emisor los tres casos son el mismo silencio, y a los 30 ms hace lo mismo: repite.

**Idea clave.** El emisor actúa igual sea cual sea la causa. Lo que hace que eso sea correcto está en el receptor, que **reconoce lo que ya recibió**.

# ¿Para qué la secuencia, si ya hay temporizador?

v3-ack: con número de secuencia

```
     0 ms  emisor   envía DATA 0 "a"
    10 ms  receptor recibe 0 y entrega "a"
    10 ms  receptor envía ACK 0  [el canal lo pierde]
    30 ms  emisor   vence el temporizador
    30 ms  emisor   envía DATA 0 "a" (otra vez)
    40 ms  receptor recibe 0: duplicado, no lo entrega
    40 ms  receptor envía ACK 0
entregado: a b c   (correcto, 8 transmisiones)
```

v3-sin-seq: sin número de secuencia

```
     0 ms  emisor   envía DATA "a"
    10 ms  receptor recibe y entrega "a"
    10 ms  receptor envía ACK  [el canal lo pierde]
    30 ms  emisor   vence el temporizador
    30 ms  emisor   envía DATA "a" (otra vez)
    40 ms  receptor recibe y entrega "a"
    40 ms  receptor envía ACK
entregado: a a b c   (INCORRECTO, 8 transmisiones)
```

**Se necesitan los dos**

El temporizador hace que el emisor **repita**. El número de secuencia hace que esa repetición sea **inofensiva**. Un ACK tarde solo gasta red (`v3-tarde`).

# Lo que construimos

| La falla | El mecanismo | Lo que introduce |
|---|---|---|
| Bits dañados | Checksum | Detecta, no repara |
| El emisor no sabe si llegó | ACK (y NAK) | Las respuestas también se dañan |
| Hay que reparar | Retransmisión | Duplicados |
| Duplicados | Número de secuencia | Con uno en camino, basta un bit |
| Pérdida | Temporizador | Retransmisiones innecesarias |

**Error común**

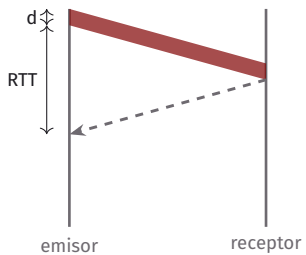
Un temporizador más corto no siempre es mejor. Más corto que el RTT, repite todo aunque nada se pierda. Muy largo, cada pérdida detiene todo. Elegirlo sobre un RTT que cambia es tema de la próxima semana.

SECCIÓN 05

# Stop-and-wait

05

# El costo de esperar




**Utilización**

Fracción del tiempo en que el emisor transmite. Con d = L / R:

$$U = \frac{d}{\mathrm{RTT} + d}$$

1500 bytes, 1 Gbps, 30 ms:

$$d = 12\ \mu\mathrm{s} \qquad U = \frac{0{,}012}{30{,}012} = 0{,}0004$$

El 0,04 %. Un enlace de 1 Gbps se comporta como uno de 0,4 Mbps.

# El simulador y la pregunta de repaso

```
python3 window_sim.py util

d = 1500 bytes a 1000 Mbps = 0.0120 ms    RTT = 30 ms
fórmula, stop-and-wait: U = d / (RTT + d) = 0.00040
simulador, ventana 1:   U = 0.00040
ventana para llenar el enlace: (RTT + d) / d = 2501 paquetes
throughput de stop-and-wait: 0.400 Mbps
```

```
python3 window_sim.py util –mbps 100 –rtt 40

d = 1500 bytes a 100 Mbps = 0.1200 ms    RTT = 40 ms
fórmula, stop-and-wait: U = d / (RTT + d) = 0.00299
simulador, ventana 1:   U = 0.00299
ventana para llenar el enlace: (RTT + d) / d = 335 paquetes
throughput de stop-and-wait: 0.299 Mbps
```

Un paquete por RTT. La velocidad del enlace casi no aparece.

# Un stop-and-wait de verdad: TFTP

RFC 1350: bloques de 512 bytes numerados, un ACK por bloque, sobre UDP. `tshark` lee la captura de `tftp_mini.py`:

bloque, ACK, bloque, ACK

```
1   0.000000 10.209.21.10 → 10.209.23.10 TFTP 54 Read Request, File: 50k, Transfer type: octet
2   0.016999 10.209.23.10 → 10.209.21.10 TFTP 558 Data Packet, Block: 1
3   0.035202 10.209.21.10 → 10.209.23.10 TFTP 46 Acknowledgement, Block: 1
4   0.051001 10.209.23.10 → 10.209.21.10 TFTP 558 Data Packet, Block: 2
```

| Velocidad | RTT | Predicción, 512 × 8 / RTT | Medido |
|---|---|---|---|
| 100 Mbps | 30 ms | 136,5 kbit/s | 123,6 kbit/s |
| 1 Gbps | 30 ms | 136,5 kbit/s | 123,2 kbit/s |
| 100 Mbps | 40 ms | 102,4 kbit/s | 90,1 kbit/s |
| 100 Mbps | 10 ms | 409,6 kbit/s | 290,0 kbit/s |

Enlace 10 veces más rápido: nada. Cada ciclo cuesta unos ms más que el RTT. Cifras de una máquina.

# En la red real: sin mirar el número de bloque

5 % de pérdida en los ACK

```
$ tftp_mini.py get 10.209.23.10 50k /tmp/o
bytes=51200 tiempo=3.77s tasa=108.7 kbit/s duplicados=2 sha256=df1ba5f500ea33cb
$ tftp_mini.py get 10.209.23.10 50k /tmp/o --ignore-block-number
bytes=52736 tiempo=3.94s tasa=107.0 kbit/s duplicados=3 sha256=14938be1c049607e
```

1536 bytes de más: tres bloques duplicados dentro del archivo, con otro hash. Lo que el simulador mostró con `a a b c` pasa también en una red.

**En la práctica: Sorcerer's Apprentice (RFC 1123, 4.2.3.1)**

Si un emisor manda el siguiente bloque por cada ACK, un ACK duplicado duplica para siempre todo lo que sigue. La regla: no se retransmite por recibir un ACK repetido.

SECCIÓN 06

# Dos generales

06

# Dos generales




**El argumento.** Tomemos el protocolo correcto más corto. Su último mensaje puede perderse, y quien lo envió debe decidir igual. Entonces quien lo recibe también podría decidir sin él: sobraba. Contradicción.

**Error común**

Sacar de ahí que la entrega confiable es imposible, o creer que se logra certeza con suficientes confirmaciones. Lo imposible es que los dos **sepan con certeza, en tiempo finito**, que ambos saben lo mismo. Lo que se logra es confiabilidad con un costo, si el canal entrega algo de vez en cuando.

SECCIÓN 07

# Ventana deslizante

07

# Sesión B: no esperar

**Pipelining**

Mandar varios paquetes sin esperar la confirmación del primero. La **ventana** W es el máximo enviado y sin confirmar.

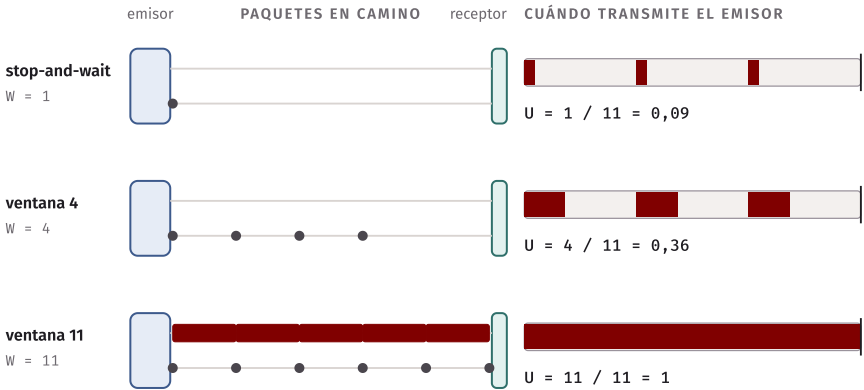
$$U = \text{mín}\left(1,\ \frac{W \times d}{\text{RTT} + d}\right)$$

**Producto ancho de banda × retardo (BDP)**

R × RTT. Con 1 Gbps y 30 ms: 3,75 MB, o 2500 paquetes de 1500 bytes. Más el que se transmite: 2501.

**El precio:** más números de secuencia, memoria para repetir y una política ante las pérdidas.

# Tres ventanas en el mismo enlace




barra: un paquete · punto: un ACK · la tira de la derecha cubre 33 ms

Con ventana 11 el primer ACK vuelve justo cuando termina de salir el último: el emisor nunca se detiene.

# La fórmula y el simulador

1 Gbps, 30 ms, 1500 bytes

| ventana | fórmula | simulador |
|---|---|---|
| 1 | 0.0004 | 0.0004 |
| 10 | 0.0040 | 0.0040 |
| 100 | 0.0400 | 0.0399 |
| 1250 | 0.4998 | 0.4936 |
| 2501 | 1.0000 | 0.9756 |
| 5002 | 1.0000 | 0.9877 |

- Crece en proporción a la ventana, hasta llenar el enlace.
- El simulador queda bajo 1 porque incluye el final de la transferencia. La fórmula es de régimen.

# En una red de verdad: TCP con `iperf3 -w`

100 Mbps, 30 ms de RTT configurado. Dos corridas seguidas.

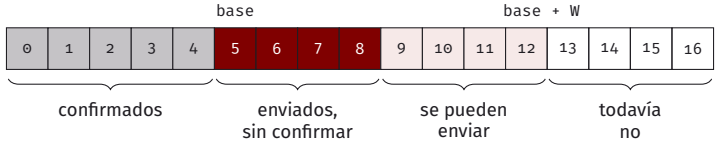
| `-w` | **Primera** | **Segunda** | **W × 8 / 32 ms** |
|---|---|---|---|
| 16 KB | 3,64 Mbps | 3,11 Mbps | 4,1 Mbps |
| 32 KB | 8,82 Mbps | 7,79 Mbps | 8,2 Mbps |
| 64 KB | 18,2 Mbps | 17,7 Mbps | 16,4 Mbps |
| 128 KB | 33,7 Mbps | 34,6 Mbps | 32,8 Mbps |
| sin límite | 92,9 Mbps | 93,0 Mbps | |

**Idea clave**

Throughput máximo: W / RTT. Si la ventana quedó chica, nada de lo que le hagan al enlace va a ayudar.

`-w` pide un tamaño de buffer, y el kernel saca la ventana con sus propias cuentas: la última columna es solo una referencia. Depende de la máquina.

# Lo que ve el emisor




La ventana se desliza a la derecha cuando llegan confirmaciones.

`window_sim.py`: los dos protocolos de libro, en un simulador. Paquetes numerados, sin conexión, sin bytes y con un temporizador fijo. d = 1 ms y RTT = 10 ms; los ACK no ocupan tiempo de transmisión.

SECCIÓN 08

# Go-Back-N

08

# Go-Back-N

**Emisor**

- Manda mientras haya espacio.
- **Un temporizador**, para el más antiguo sin confirmar.
- Si vence, **repite todo** desde `base`.

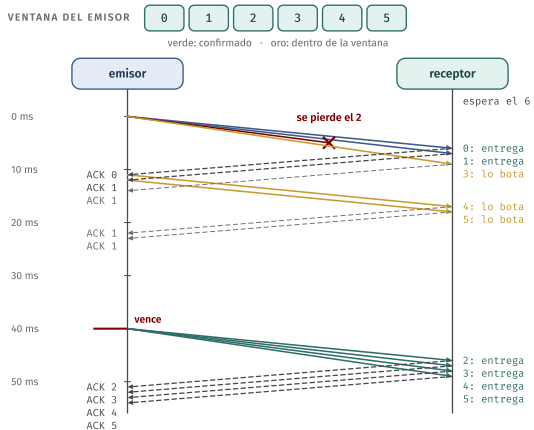
**Receptor**

- Guarda un solo número: el próximo que espera.
- Si llega ese, lo entrega y lo confirma.
- Si llega otro, lo **descarta** y repite su último ACK.

**ACK acumulativo**

"ACK n": todo hasta el n, inclusive, llegó bien. Si se pierde un ACK, el siguiente lo reemplaza.

# Go-Back-N: ventana 4, se pierde el 2




El emisor repite desde el 2: 2, 3, 4 y 5. En total, 10 transmisiones para entregar 6 paquetes.

# La traza, en extracto

```
python3 window_sim.py traza gbn –window 4 –packets 6 –drop-data 2

   0.0  emisor   envía 2  [se pierde]
   9.0  receptor recibe 3: esperaba 2, lo descarta
   9.0  receptor   ACK 1
  12.0  emisor   recibe ACK 1: confirma hasta 1
  14.0  emisor   recibe ACK 1: ya confirmado, lo ignora
  17.0  receptor recibe 4: esperaba 2, lo descarta
  18.0  receptor recibe 5: esperaba 2, lo descarta
  40.0  emisor   vence el temporizador: repite desde 2
  40.0  emisor   repite 2
  46.0  receptor recibe 2, entrega el contenido 2
entregado: [0, 1, 2, 3, 4, 5]   transmisiones: 10 (repetidas: 4)   tiempo: 54.0 ms
```

**Error común**

Aquí “ACK n” confirma el **paquete n**. En TCP y en P3 dice cuál es el **siguiente byte que se espera**.

SECCIÓN 09

# Selective Repeat

09

# Selective Repeat

**Emisor**

- Manda mientras haya espacio.
- **Un temporizador por paquete**. Si vence, repite solo ese.
- Avanza al confirmarse el más antiguo.

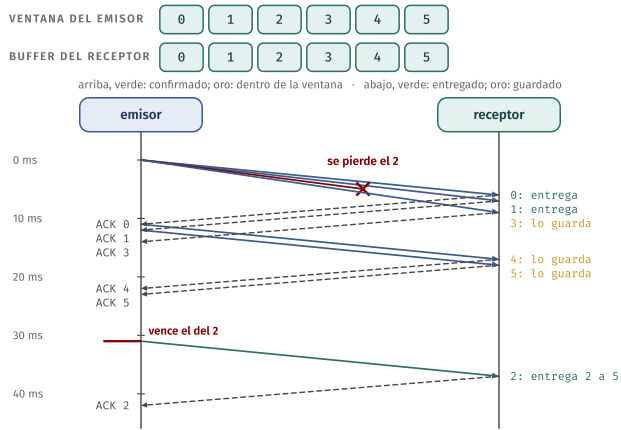
**Receptor**

- En su ventana: lo confirma y lo **guarda**. Entrega en orden lo que pueda.
- Hasta W antes: ya lo entregó, pero **igual lo confirma**.
- Cualquier otra cosa se ignora.

**La regla que siempre se olvida**

Si el receptor no vuelve a confirmar, el emisor repite ese paquete para siempre.

# Selective Repeat: ventana 4, se pierde el 2




A los 37 ms llega el 2, y 2, 3, 4 y 5 se entregan de golpe y en orden. En total, 7 transmisiones contra las 10 de Go-Back-N.

# Selective Repeat: la misma pérdida

```
python3 window_sim.py traza sr --window 4 --packets 6 --drop-data 2

   0.0  emisor   envía 2  [se pierde]
   0.0  emisor   envía 3
   9.0  receptor recibe 3, lo guarda
   9.0  receptor   ACK 3
  12.0  emisor   envía 5
  17.0  receptor recibe 4, lo guarda
  18.0  receptor recibe 5, lo guarda
  31.0  emisor   vence el temporizador de 2
  31.0  emisor   repite 2
  37.0  receptor recibe 2, entrega el contenido [2, 3, 4, 5]
  37.0  receptor   ACK 2
  42.0  emisor   recibe ACK 2
entregado: [0, 1, 2, 3, 4, 5]   transmisiones: 7 (repetidas: 1)   tiempo: 42.0 ms
```

- Siete transmisiones contra diez.
- En el instante 37 entrega de golpe y **en orden**: la aplicación nunca ve el desorden.
- Esos datos estuvieron unos 20 ms en la máquina sin poder usarse.

SECCIÓN 10

# Comparación

10

# Comparación

| | Go-Back-N | Selective Repeat |
|---|---|---|
| ACK | Acumulativo | Individual |
| Temporizadores | Uno | Uno por paquete |
| Receptor | Un número; descarta el desorden | Buffer de W paquetes |
| Ante una pérdida | Repite todo desde el perdido | Repite solo el perdido |
| Si se pierde un ACK | El siguiente lo cubre | Se repite ese paquete |
| Ventana máxima, N números | N - 1 | N / 2 |

TCP no es ninguno de los dos: usa ACK acumulativos y además guarda lo que llega fuera de orden. P3 tampoco: lean su enunciado.

# Medimos con el simulador

python3 window_sim.py barrido: W = 50, d = 1 ms, RTT = 40 ms, temporizador = 180 ms

```
pérdida en los dos sentidos:
 pérdida | GBN envíos/paq   GBN U |  SR envíos/paq    SR U
    0.0% |           1.00    0.98 |           1.00    0.98
    1.0% |           1.50    0.31 |           1.02    0.35
   10.0% |           6.71    0.04 |           1.23    0.14
   20.0% |          13.27    0.02 |           1.55    0.08
pérdida solo en los ACK:
    1.0% |           1.00    0.98 |           1.01    0.47
   20.0% |           1.00    0.98 |           1.24    0.13
```

- **Sin pérdidas, son iguales. Con muchas, GBN se derrumba:** más de 13 envíos por paquete.
- SR se puede predecir: 1 / (0,8 × 0,8) = 1,56. El simulador da 1,55.
- **Si solo se pierden ACK, gana el acumulativo.** Ninguno es mejor en todo.
- Es un modelo: sin colas ni otros flujos, sin congestión.

# El temporizador pesa

```
python3 window_sim.py barrido --timeout 60: los mismos envíos, otra utilización

ventana 50, 2000 paquetes, d = 1 ms, RTT = 40 ms, temporizador = 60 ms, promedio de 5 semillas
pérdida en los dos sentidos:
 pérdida | GBN envíos/paq   GBN U |  SR envíos/paq    SR U
    1.0% |           1.50    0.48 |           1.02    0.62
   10.0% |           6.71    0.07 |           1.23    0.33
   20.0% |          13.27    0.04 |           1.55    0.22
```

Selective Repeat al 10 %: pasó de 0,14 a 0,33, con los mismos envíos. Lo que le costaba era esperar.

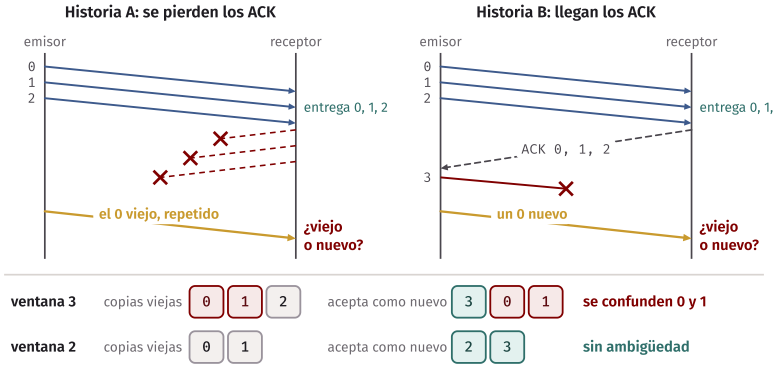
> **Idea clave**
>
> Con pocas pérdidas casi no se diferencian. Con muchas, conviene no repetir lo que ya llegó.

SECCIÓN 11

# Números de secuencia

11

# Selective Repeat: 4 números, ventana 3




Lo que ve el receptor es idéntico: entregó 0, 1 y 2, acepta 3, 0 y 1, y le llega un 0. Lo correcto es distinto en cada historia.

# Datos viejos como si fueran nuevos

```
python3 window_sim.py ambiguedad

--- sr: 4 números, ventana 3; se pierden los ACK de la primera ronda
    8.0  receptor   ACK 2  [se pierde]
   27.0  emisor   repite 0
   33.0  receptor recibe 0, lo guarda
   44.0  receptor recibe 3, entrega el contenido [3, 0, 1]
contenidos entregados: [0, 1, 2, 3, 0, 1, 6, 7, 4]   (INCORRECTO)

--- sr: 4 números, ventana 2; se pierden los ACK de la primera ronda
   31.0  receptor recibe 0: ya entregado, solo confirma
contenidos entregados: [0, 1, 2, 3, 4, 5, 6, 7]   (correcto)
```

**Sin ningún error a la vista**

Checksums correctos y emisor conforme. El 0 y el 1 se repiten, el 4 sale fuera de lugar y el 5 no se entrega.

# La regla

El receptor espera `r`. Pueden llegar copias viejas de `r - W` a `r - 1`, si no llegó ningún ACK. Aceptaría como nuevo:

- **SR:** `r` a `r + W - 1`. Son 2W números seguidos, todos distintos: $W \leq N / 2$.
- **GBN:** solo `r`. Son W + 1 seguidos: $W \leq N - 1$.

```
python3 seq_space.py

números  bits  GBN: W máx  SR: W máx
      2     1           1           1
      4     2           3           2
      8     3           7           4
     16     4          15           8
```

**Error común**

Creer que con esto basta. La cuenta supone que una copia no reaparece después de que los números dieron la vuelta. Por eso los protocolos reales usan espacios enormes (32 bits en TCP y en P3), y IP limita la vida de un paquete con el TTL.

SECCIÓN 12

# El hackathon

12

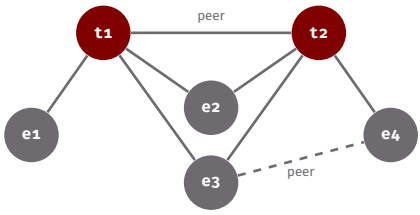
# Laboratorio: el hackathon

**Los sistemas autónomos de todos los equipos, en una sola red.**

1. Cada equipo entrega su carpeta `as/` de P2 antes de empezar.
2. El docente levanta todo. Las sesiones BGP salen de `as/vecinos.env`; ustedes revisan y depuran.
3. Objetivo: todos se alcanzan, con caminos que respetan las políticas.
4. Diagnóstico: `show bgp summary`, `show bgp ipv4`, `traceroute`.
5. El docente corta un enlace de tránsito. ¿Siguen conectados?
6. Discusión: ¿cuánto fue técnico y cuánto fue coordinación?
7. Se entrega **P2** y se lee **P3**.

> **En la práctica**
>
> Lleven escrito su plan: prefijos, AS y quién es cada vecino. Si algo falla, **hablen con el equipo vecino antes de cambiar su configuración**, y cambien una cosa a la vez.

# Para llegar con práctica: el ensayo




- Cuatro “equipos” de un router y dos proveedores.
- **Cuatro fallas puestas a propósito.**
- Sus arreglos van en `lab/`.
- No es una solución de P2: casi no tiene políticas, y `no bgp ebgp-requires-policy` sigue puesto.

# El tablero

```
make up-ensayo; python3 tablero.py

      e1  e2  e3  e4
e1     -   X   X   X
e2     X   -   X   X
e3     X   X   -  ok
e4     X   X  ok   -
2 de 12 pares se alcanzan
sesiones BGP que no están arriba:
  e1: vecino 10.209.2.1, Idle
```

- Un ping desde el prefijo de cada fila al de cada columna: exige camino **en los dos sentidos**.
- Una sola sesión caída, y casi todo en rojo.

**Ojo**

Una sesión arriba no quiere decir que las rutas estén pasando.

**¿Está arriba?** `show bgp summary`, `show bgp neighbors <ip>` (`Last reset`). **¿Qué anuncio?** `show bgp ipv4 unicast neighbors <ip> advertised-routes`. **¿Es válida?** `show bgp ipv4 unicast <prefijo>`. **¿Qué dejo entrar?** `show ip prefix-list`.

# Los pasos del ensayo

1. Antes de abrir un router, escriban una hipótesis por cada `X`. ¿Qué filas y columnas fallan juntas?
2. Encuentren las **tres fallas que se ven**: en `e1`, en `e2` y en `t2`. Una perjudica al vecino de quien la tiene.
3. Encuentren **la que no se ve**, en `e3`, con el tablero lleno.
4. Predigan cada corte, y compruébenlo con `python3 ensayo.py corta <a> <b>`.

**Error común**

Cambiar un filtro de entrada y creer que ya se aplicó. En nuestras pruebas con FRR, la ruta no apareció en más de 30 s, hasta correr `clear bgp ipv4 unicast <vecino> soft in`.

# Dos cortes que dicen mucho

```
corta t2 e4

      e1  e2  e3  e4
e1     -  ok  ok   X
e2    ok   -  ok   X
e3    ok  ok   -  ok
e4     X   X  ok   -
8 de 12 pares se alcanzan
sesiones BGP que no están arriba:
  e4: vecino 10.209.7.1, Active
```

`e4` sigue conectado a `e3`, pero un **peer no da tránsito**. Tener conexión física no es tener ruta.

```
corta t1 t2

      e1  e2  e3  e4
e1     -  ok  ok   X
e2    ok   -  ok  ok
e3    ok  ok   -  ok
e4     X  ok  ok   -
10 de 12 pares se alcanzan
todas las sesiones BGP están arriba
```

Todas las sesiones de los equipos arriba, y `e1` y `e4` no se ven. El problema está en una sesión de la que ninguno es parte.

# En `code/week-09/`

code/week-09

```
$ make up
$ python3 rdt_sim.py todos
$ python3 window_sim.py barrido
$ python3 seq_space.py
$ make up-ensayo
$ python3 tablero.py
$ make test
$ make down
```

- Las trazas de los simuladores son exactas.
- Las mediciones dependen de su máquina: comparen tendencias.
- Las pruebas de los arreglos se saltan hasta que quiten el `TODO` de los cuatro archivos de `lab/`.

# Para la próxima clase

- Leer Kurose y Ross, secs. 3.5.1 a 3.5.4, para TCP.
- Entregar **P2**. Leer el enunciado de **P3**, que se entrega en la semana 11.
- Antes de escribir código de P3, contesten por escrito: ¿cuándo avanza su ventana? ¿Qué guarda cada lado, y hasta cuándo? ¿Qué hace cada lado ante un duplicado?

**Pregunta**

Un compañero dice: “en mi protocolo, el temporizador ya resuelve las pérdidas; los números de secuencia son 2 bits y mi ventana es 3, y las pruebas pasan”. ¿Qué le responden? Muestren una secuencia de eventos que entregue datos equivocados.