

UNIVERSIDAD NACIONAL DE INGENIERÍA

Computación Centrada en Redes

TCP

Conexiones, retransmisión y control de flujo

Gerson Garrido

Semana 10

Hoy

Sesión A: leer TCP

1. Qué promete TCP
2. MSS y MTU
3. El segmento, byte por byte
4. Una conexión completa
5. ACK acumulativos y retardados
6. RTT, RTO y retroceso exponencial

Sesión B: hacerlo sufrir

7. Fast retransmit, SACK, RACK y TLP
8. Control de flujo y ventana cero
9. Handshake y `connect`
10. Cierre, `TIME_WAIT` y RST
11. SYN flood y SYN cookies

Lectura

Kurose y Ross, secs. 3.5.1 a 3.5.4 (sesión A), y 3.5.4 a 3.5.6 (sesión B). CS168: *TCP Design* y *TCP Implementation*. Complementarias: RFC 9293 y RFC 6298.

SECCIÓN 01

Qué promete TCP

01

La promesa

TCP

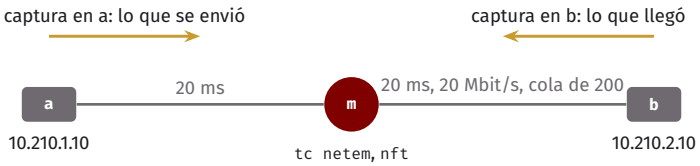
Un **stream de bytes** confiable y ordenado, **con conexión** y **full duplex**, entre dos procesos.

- **Stream:** no hay mensajes. Un `write` no corresponde a un `read`.
- **Confiable y ordenado:** los mismos bytes, en orden, o un error. Nunca un hueco.
- **Con conexión:** el estado vive en los hosts. Los routers no saben que existe.
- **Full duplex:** dos streams, cada uno con sus propios números.

Respecto a la semana pasada

Como Go-Back-N: ACK acumulativo y un temporizador. Como Selective Repeat: guarda lo que llega fuera de orden, y retransmite poco.

La red de esta semana



Por qué un router en el medio

Con `netem` en el host que captura, `tcpdump` no vería el segmento perdido. Comparar `a` con `b` es la herramienta de esta semana.

`make up` y `python3 scenarios.py clean`: 17,4 Mbit/s en el receptor. Ese escenario usa una cola de 2000, para que no se desborde.

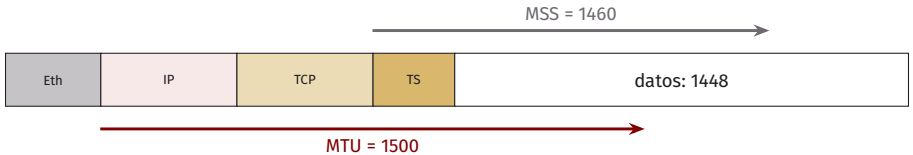
SECCIÓN 02

MSS y MTU

02

MTU y MSS

MSS *(maximum segment size)*: máximo de **datos** en un segmento, sin encabezados. Cada extremo anuncia el suyo en el SYN. No se negocia.



Lo que confunde

MSS $= 1500 - 40 = 1460$, pero se ve `Len=1448`: 12 bytes son la opción de timestamps.

Se ve en la captura

```
python3 tcpsum.py captures/clean-a.pcap

SYN           seq real 1661972085; MSS 1460, ventana 64240, escala 7, SACK sí, timestamps sí
SYN-ACK       seq real 3960613348; MSS 1460, ventana 65160, escala 7, SACK sí, timestamps sí
datos         1382 segmentos, 2000000 bytes enviados, 2000000 bytes distintos, el mayor de 1448
```

Bajamos a 1400 el MTU de `a`:

```
python3 tcpsum.py captures/mss-a.pcap

SYN           seq real 4265168592; MSS 1360, ventana 65280, escala 7, SACK sí, timestamps sí
SYN-ACK       seq real 2465781842; MSS 1460, ventana 65160, escala 7, SACK sí, timestamps sí
datos         75 segmentos, 100000 bytes enviados, 100000 bytes distintos, el mayor de 1348
```

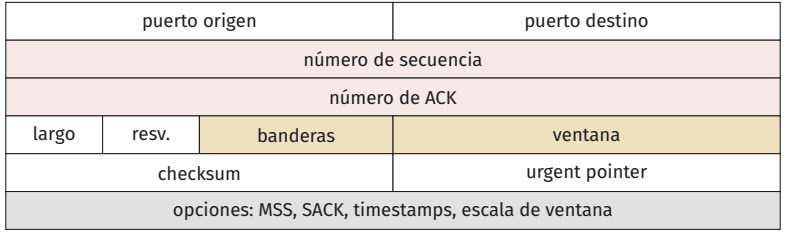
Cada uno habla de su interfaz: `b` sigue con 1460, y `a` envía $1360 - 12 = 1348$.

SECCIÓN 03

El segmento

03

El encabezado TCP



- **Banderas:** SYN (abro, este es mi número), ACK, FIN (no envío más), RST (aborto), PSH.
- **Largo:** en palabras de 4 bytes. Por eso las opciones llegan hasta 40 bytes.

Un SYN real, sin ayuda de Wireshark

```
tshark -r captures/clean-a.pcap -c 1 -x

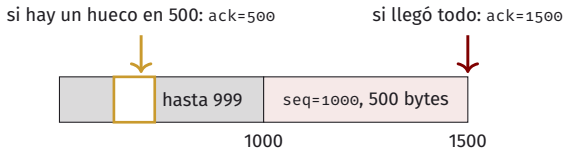
0020  02 0a 83 f0 13 89 63 0f ae 75 00 00 00 00 a0 02   ......c..u......
0030  fa f0 b1 64 00 00 02 04 05 b4 04 02 08 0a 82 de   ...d............
0040  57 16 00 00 00 00 01 03 03 07                     W.........
```

`83 f0`	puerto origen: 33776
`13 89`	puerto destino: 5001
`63 0f ae 75`	seq: 1661972085
`00 00 00 00`	ACK: nada todavía
`a0`	largo: 10 palabras, 40 bytes
`02`	banderas: solo SYN
`fa f0`	ventana: 64240

`02 04 05 b4`	opción MSS: 1460
`04 02`	SACK permitido
`08 0a …`	timestamps
`01`	NOP, relleno
`03 03 07`	escala de ventana: 7

Secuencia y ACK cuentan bytes

seq: número del primer byte que lleva el segmento. **ack:** número del siguiente byte que se espera.



SYN y FIN gastan un número

Si no, no se podrían confirmar. Tras un SYN con número x, el primer byte de datos es $x + 1$.

Los números reales

```
tshark -r captures/clean-a.pcap -c 5 -o tcp.relative_sequence_numbers:FALSE

1   0.000000  10.210.1.10 → 10.210.2.10  TCP 74 33776 → 5001 [SYN] Seq=1661972085 Win=64240 Len=0 MSS=1460
2   0.041148  10.210.2.10 → 10.210.1.10  TCP 74 5001 → 33776 [SYN, ACK] Seq=3960613348 Ack=1661972086 Win=
3   0.041166  10.210.1.10 → 10.210.2.10  TCP 66 33776 → 5001 [ACK] Seq=1661972086 Ack=3960613349 Win=64256
4   0.041191  10.210.1.10 → 10.210.2.10  TCP 1514 33776 → 5001 [ACK] Seq=1661972086 Ack=3960613349 Win=642
5   0.041195  10.210.1.10 → 10.210.2.10  TCP 1514 33776 → 5001 [PSH, ACK] Seq=1661973534 Ack=3960613349 Wi
```

El paquete 5 tiene 1661972086 + 1448. La misma captura, como la muestra Wireshark:

```
tshark -r captures/clean-a.pcap -c 2

1   0.000000  10.210.1.10 → 10.210.2.10  TCP 74 33776 → 5001 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=
2   0.041148  10.210.2.10 → 10.210.1.10  TCP 74 5001 → 33776 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_
```

Error común: `Seq=0` no viaja en el paquete. Wireshark muestra números relativos; en el paquete va 1661972085.

La ventana y su escala

El campo tiene 16 bits: hasta 65535 bytes. Con 1 Gbit/s y 40 ms de RTT hacen falta 5 MB en vuelo. La **escala de ventana** es una opción del SYN: “multipliquen por 2^N lo que yo anuncie”.

```
tcpdump -nr captures/clean-a.pcap -c 3

14:52:18.520408 IP 10.210.1.10.33776 > 10.210.2.10.5001: Flags [.], ack 1, win 502, options [nop,nop,TS val 2195
```

- `tcpdump` muestra el campo: `win 502`. Wireshark multiplica: $502 \times 128 = 64256$.
- La ventana del SYN no se escala: la escala recién se está acordando.

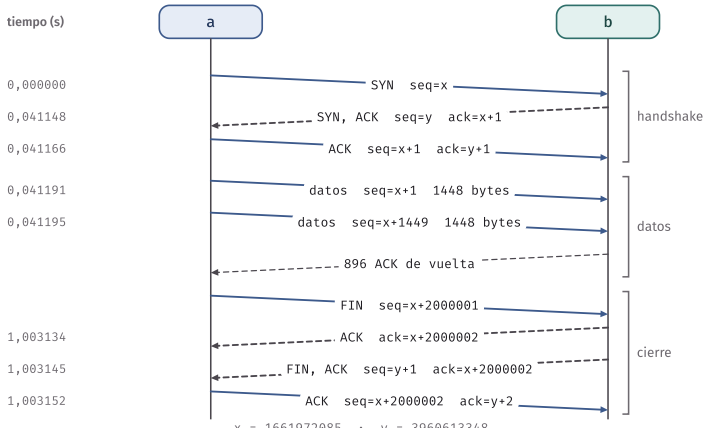
Error común: sin el handshake en la captura, Wireshark no conoce la escala, y la ventana parece diminuta.

SECCIÓN 04

Una conexión completa

04

De principio a fin



Son 2 000 000 bytes y un FIN: b confirma x+2000002. Aquí cada sentido se cierra aparte, con cuatro segmentos.

El cierre, en la captura

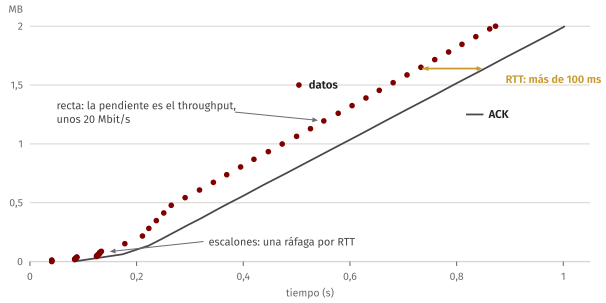
2000002 $=$ 2 000 000 bytes, $+1$ porque el stream empieza en 1, $+1$ del FIN.

Aquí `b` envió su ACK y su FIN por separado: **cuatro** segmentos. A veces el kernel los junta, y son tres. Las dos formas son correctas.

los últimos segmentos

```
2280   1.003134  10.210.2.10 → 10.210.1.10  TCP 66 5001 → 33776 [ACK] Seq=1 Ack=2000002 Win=1035136 Len=0
2281   1.003145  10.210.2.10 → 10.210.1.10  TCP 66 5001 → 33776 [FIN, ACK] Seq=1 Ack=2000002 Win=1035136 Len=0
2282   1.003152  10.210.1.10 → 10.210.2.10  TCP 66 33776 → 5001 [ACK] Seq=2000002 Ack=2 Win=64256 Len=0
```

El gráfico tiempo-secuencia



Una muestra de los puntos de `clean-a.pcap`. Wireshark: *Statistics* → *TCP Stream Graphs* → *Time Sequence*.

- **Escalones:** ráfagas separadas por un RTT. Es slow start: próxima semana.
- **Recta:** su pendiente es el throughput. 20 Mbit/s.
- **Línea de los ACK:** la distancia horizontal es el RTT. Crece porque se forma una cola en `m`.

SECCIÓN 05

ACK

05



ACK acumulativo y ACK retardado

- **ACK acumulativo:** confirma todos los bytes anteriores a ese número. Si se pierde uno, no pasa nada si llega otro mayor.
- **ACK retardado:** según el RFC 9293, al menos uno cada dos segmentos completos, y nunca después de 0,5 s.

```
captura en b: tiempo, origen, secuencia, ACK, largo

0.419021000 10.210.1.10 699385  1   1448
0.420033000 10.210.1.10 700833  1   1448
0.420042000 10.210.2.10 1   702281  0
```

$702281 = 699385 + 2 \times 1448$. En toda la transferencia: 1382 segmentos, 896 ACK.

Cuánto espera

100 bytes cada 100 ms, a un receptor que no está leyendo:

escenario delack, captura en b

```
1.638989000 10.210.1.10 1501    1   100
1.638997000 10.210.2.10 1   1601    0
1.745987000 10.210.1.10 1601    1   100
1.787576000 10.210.2.10 1   1701    0
1.846984000 10.210.1.10 1701    1   100
1.894997000 10.210.2.10 1   1801    0
```

- Al comienzo, el ACK sale en microsegundos. Después, a unos 40 a 50 ms.
- **Los 500 ms son un tope**, no lo que se usa.
- Si la aplicación está leyendo, Linux responde a cada segmento chico en microsegundos.
- Para qué esperar: el ACK puede viajar con la respuesta (piggybacking).

SECCIÓN 06

RTT y RTO

06

¿Cuánto esperar antes de retransmitir?

Muy poco: retransmite lo que no se perdió.

Mucho: una pérdida detiene la conexión sin motivo.

RFC 6298, con $\alpha = 1/8$ y $\beta = 1/4$

$$\mathrm{DevRTT} \leftarrow (1-\beta)\,\mathrm{DevRTT} + \beta\,|\mathrm{SampleRTT} - \mathrm{EstimatedRTT}|$$
$$\mathrm{EstimatedRTT} \leftarrow (1-\alpha)\,\mathrm{EstimatedRTT} + \alpha\,\mathrm{SampleRTT}$$
$$\mathrm{RTO} = \mathrm{EstimatedRTT} + 4\,\mathrm{DevRTT}$$

Antes de medir: 1 s. Primera medición R: EstimatedRTT $= R$, DevRTT $= R/2$. Si el RTO da menos de 1 s, debería subirse a 1 s. El promedio es lento a propósito. El margen crece si la red es irregular.

Ejemplo resuelto

EstimatedRTT $= 100$ ms, DevRTT $= 5$ ms, y llega un SampleRTT de 120 ms.

- DevRTT $= 0{,}75 \times 5 + 0{,}25 \times |100 - 120| = 8{,}75$ ms
- EstimatedRTT $= 0{,}875 \times 100 + 0{,}125 \times 120 = 102{,}5$ ms
- RTO $= 102{,}5 + 4 \times 8{,}75 = 137{,}5$ ms

terminal

```
$ python3 rto_table.py --start 100 5 120
  muestra  EstimatedRTT   DevRTT      RTO
   120.00        102.50     8.75   137.50
```

Error común

Usar el EstimatedRTT **nuevo** para DevRTT: da 8,125 y un RTO de 135 ms. El RFC 6298 es explícito: DevRTT va primero, con el valor anterior. Usamos el orden del RFC.

Con muestras de la captura limpia

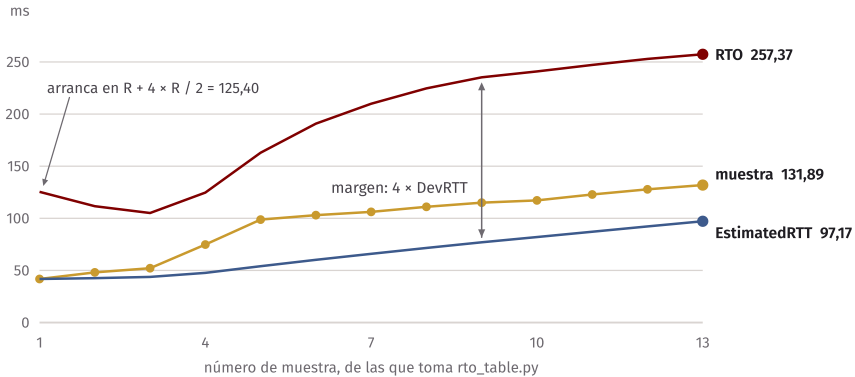
```
python3 rto_table.py -pcap ...
```

muestra	EstimatedRTT	DevRTT	RTO
41.80	41.80	20.90	125.40
52.13	43.78	15.33	105.09
98.77	54.05	27.22	162.93
106.18	65.93	36.00	209.91
117.19	82.02	39.72	240.91
131.89	97.17	40.05	257.37

Algunas filas de la salida. En ms.

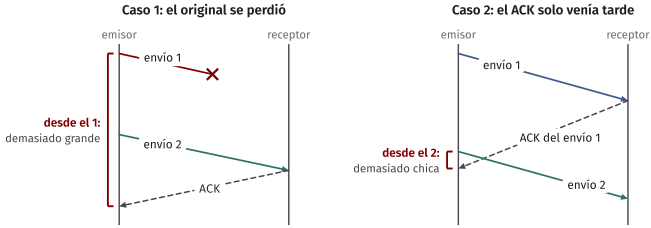
- Mismo enlace, pero el RTT se triplica: se llena la cola de `m`.
- EstimatedRTT va siempre detrás de las muestras.
- El RTO queda holgadamente por encima de todas. Esa es la idea.

Las mismas muestras, dibujadas



Las muestras suben porque se llena la cola en `m`. EstimatedRTT va detrás, y el margen deja al RTO por encima de todas.

No se mide un segmento retransmitido



El emisor ve lo mismo en los dos casos. ¿Desde cuál de los envíos mide?

Algoritmo de Karn

No tomar muestras de lo retransmitido. Excepción del RFC 6298: con timestamps no hay ambigüedad, porque el ACK devuelve el valor del envío que lo provocó.

Lo que hace Linux

```
ss -tin

cubic wscale:7,7 rto:249 rtt:48.23/11.866 mss:1448 pmtu:1500 ... cwnd:10 ...
```

RFC: $48{,}23 + 4 \times 11{,}866 = 95{,}7$ ms, y luego el piso de 1 s.

rto	rtt	diferencia
249	48,23	200,8
244	43,46	200,5
324	123,18	200,8
306	105,57	200,4

```
python3 scenarios.py rtomin

default: rto:244 rtt:43.462/3.019
500ms: rto:542 rtt:41.792/0.805
```

El margen es el mayor entre 4 DevRTT y un piso, de 200 ms por defecto. Se puede cambiar por ruta, con `rto_min`.

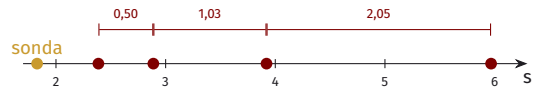
Cuál usar: en un examen y en P3, la fórmula del RFC. Al leer `ss`, recuerden el piso.

El temporizador en acción

Retroceso exponencial: al vencer el RTO, retransmitir lo más antiguo sin confirmar y **duplicar** el RTO. Flujo lento y un corte total de unos 3 s:

retransmisiones

```
1.828041000 17001
2.386997000 15001
2.887999000 15001
3.920219000 15001
5.968019000 15001
6.052147000 16001
```



`ss: rto:992`

`backoff:2`, o sea 248 × 4. Contadores: `TcpExtTCPTimeouts=4`.

Costo oculto: con el temporizador en retroceso, TCP puede quedarse callado un rato después de que la red volvió.

SECCIÓN 07

Fast retransmit

07

Una pérdida da señales antes

Fast retransmit (RFC 5681)

Al recibir el **tercer ACK duplicado**, retransmitir sin esperar el temporizador.

escenario classic: SACK y RACK apagados, y m descarta un solo paquete

```
m# nft 'add rule ip w10 forward tcp dport 5001 numgen inc mod 100000 == 40 counter drop'
```

tshark -r captures/classic-a.pcap -Y tcp.analysis.flags

```
  123   0.172808  10.210.2.10 → 10.210.1.10  TCP 66 [TCP Dup ACK 120#1] 5001 → 37986 [ACK] Seq=1 Ack=55025 Win=99840 Len=0
  125   0.173835  10.210.2.10 → 10.210.1.10  TCP 66 [TCP Dup ACK 120#2] 5001 → 37986 [ACK] Seq=1 Ack=55025 Win=99840 Len=0
  127   0.173840  10.210.2.10 → 10.210.1.10  TCP 66 [TCP Dup ACK 120#3] 5001 → 37986 [ACK] Seq=1 Ack=55025 Win=99840 Len=0
  128   0.173843  10.210.1.10 → 10.210.2.10  TCP 1514 [TCP Fast Retransmission] 37986 → 5001 [ACK] Seq=55025 Ack=1 Win=64256
```

A los 3 microsegundos del tercer duplicado.

¿Por qué tres y no uno?

Un ACK duplicado es una pista, no una prueba: también aparece si un paquete solo se atrasó.
Tres es un valor empírico, para que el desorden leve no dispare nada.
Sin ninguna pérdida, con 3 ms de jitter. Todo lo retransmitido está de más:

escenarios reorder_classic y reorder

```
recibidos 3000000 bytes en 0.52 s (46.3 Mbit/s)
TcpRetransSegs=34  TcpExtTCPTimeouts=0  TcpExtTCPLossProbes=0  TcpExtTCPSackRecovery=0  TcpExtTCPRenoRecovery=35

recibidos 3000000 bytes en 0.34 s (71.0 Mbit/s)
TcpRetransSegs=4  TcpExtTCPTimeouts=0  TcpExtTCPLossProbes=0  TcpExtTCPSackRecovery=1  TcpExtTCPRenoRecovery=0
```

python3 tcpsum.py captures/reorder-a.pcap

```
ACK duplicados 1595 (el mayor, #89); rachas que llegaron a #1: 162, a #2: 133, a #3: 115
```

Tres duplicados, 115 veces, sin pérdida. Otras corridas: 0 a 4 retransmisiones en el moderno, 34 a 40 en el clásico.

SACK: decir exactamente qué llegó

SACK *(selective acknowledgment)*

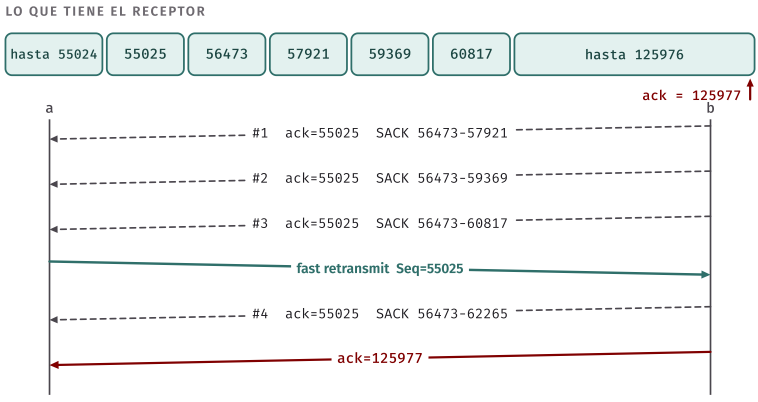
Una opción de los ACK: los bloques recibidos más allá del hueco. Se acuerda en el SYN.

captura drop: duplicado, ACK, inicio y fin del bloque

```
1   55025   56473   57921
2   55025   56473   59369
3   55025   56473   60817
```

“Espero el 55025, pero tengo desde el 56473”. Falta exactamente un segmento.

El hueco y el bloque



La retransmisión llena el hueco, y un solo ACK confirma de golpe 70 952 bytes que b había guardado.

Dos capturas de lo mismo

en a: el segmento sale dos veces

```
 57   0.128398  10.210.1.10 → 10.210.2.10  TCP 1514 37978 → 5001 [ACK] Seq=55025 Ack=1 Win=64256 Len=14
126   0.172518  10.210.1.10 → 10.210.2.10  TCP 1514 [TCP Fast Retransmission] 37978 → 5001 [ACK] Seq=55
```

en b: llega una sola vez

```
169   0.178087  10.210.1.10 → 10.210.2.10  TCP 1514 [TCP Fast Retransmission] 37978 → 5001 [ACK] Seq=55
```

El siguiente ACK que avanza salta de 55025 a 125977: 70 952 bytes de una vez.

```
python3 scenarios.py drop

TcpRetransSegs=1  TcpExtTCPTimeouts=0  TcpExtTCPLossProbes=0  TcpExtTCPSackRecovery=1  TcpExtTCPRenoRecovery=0
```

11 Mbit/s contra 17, por **un** paquete en 1383: TCP tomó la pérdida como congestión.

Linux hoy: RACK y TLP (RFC 8985)

- **RACK:** decide por tiempo. Si se confirmó algo que salió *después*, y pasó un margen, lo anterior se da por perdido.
- **TLP:** si pasan unos dos RTT sin ACK, reenvía lo último para provocar una respuesta.

Se pierden el último segmento y el FIN. No queda quién genere duplicados:

escenario tail

```
13   0.085730  10.210.2.10 → 10.210.1.10  TCP 66 5001 → 43294 [ACK] Seq=1 Ack=5793 Win=76800 Len=
14   0.177741  10.210.1.10 → 10.210.2.10  TCP 66 [TCP Retransmission] 43294 → 5001 [FIN, ACK] Seq
15   0.219737  10.210.2.10 → 10.210.1.10  TCP 78 [TCP Dup ACK 13#1] 5001 → 43294 [ACK] Seq=1 Ack=
16   0.219753  10.210.1.10 → 10.210.2.10  TCP 1274 [TCP Retransmission] 43294 → 5001 [PSH, ACK] S
17   0.261739  10.210.2.10 → 10.210.1.10  TCP 66 5001 → 43294 [ACK] Seq=1 Ack=7002 Win=79104 Len=
```

Paquete 14: la sonda, a los 92 ms y sin señal. Paquete 15: su SACK revela el hueco. Paquete 16: el dato. No siempre alcanza: en otra corrida ese dato esperó al RTO.

2 % de pérdida: Wireshark y el kernel

python3 scenarios.py loss, y luego tcpsum.py

```
recibidos 3000000 bytes en 11.53 s (2.1 Mbit/s)
TcpRetransSegs=43  TcpExtTCPTimeouts=0  TcpExtTCPLossProbes=0  TcpExtTCPSackRecovery=35  TcpExtTCPRenoRecovery=0
datos          2116 segmentos, 3062264 bytes enviados, 3000000 bytes distintos, el mayor de 1448
retransmitidos 43 segmentos, 24 de ellos marcados como fast
```

- Llegó entero. La velocidad cayó casi 90 % (de 1,6 a 3,4 Mbit/s, según la corrida).
- 19 sin marca *fast*. ¿Timeouts? **No:** `TcpExtTCPTimeouts=0`, también en las otras corridas revisadas.

Error común

Leer `[TCP Retransmission]` como timeout. Wireshark deduce desde afuera. Los contadores saben: `nstat -az TcpExtTCPTimeouts`.

SECCIÓN 08

Control de flujo

08

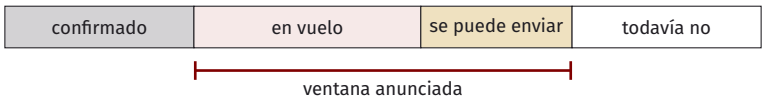
Control de flujo

Los datos esperan en un buffer hasta que la aplicación hace `read`. Si es lenta, el buffer se llena.

Ventana de recepción (`rwnd`)

Cada segmento anuncia cuántos bytes más puede recibir ese extremo. El emisor garantiza:

$$\text{último byte enviado} - \text{último byte confirmado} \leq \text{ventana anunciada}$$



La ventana deslizante de la semana 9, con una novedad: su tamaño cambia, y lo decide el receptor.

Ventana cero

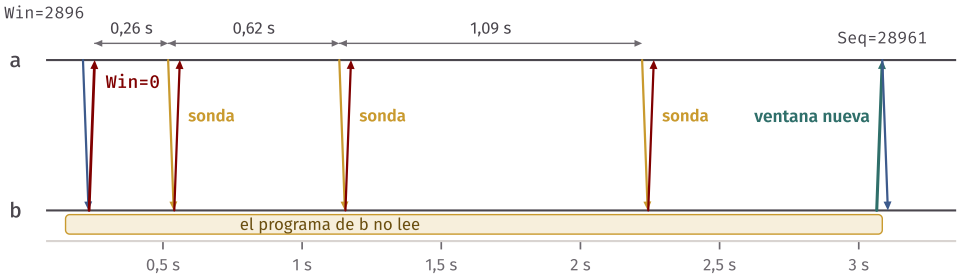
Un servidor con buffer chico, que acepta y se queda 3 s sin leer:

escenario zero

```
30   0.212588  10.210.2.10 → 10.210.1.10  TCP 66 5001 → 56426 [ACK] Seq=1 Ack=26065 Win=2896 Len=0 TSval=21
31   0.212613  10.210.1.10 → 10.210.2.10  TCP 1514 [TCP Window Full] 56426 → 5001 [ACK] Seq=27513 Ack=1 Win
32   0.254582  10.210.2.10 → 10.210.1.10  TCP 66 [TCP ZeroWindow] 5001 → 56426 [ACK] Seq=1 Ack=28961 Win=0
33   0.518604  10.210.1.10 → 10.210.2.10  TCP 66 [TCP Keep-Alive] 56426 → 5001 [ACK] Seq=28960 Ack=1 Win=64
34   0.560608  10.210.2.10 → 10.210.1.10  TCP 66 [TCP ZeroWindow] 5001 → 56426 [ACK] Seq=1 Ack=28961 Win=0
35   1.133607  10.210.1.10 → 10.210.2.10  TCP 66 [TCP Keep-Alive] 56426 → 5001 [ACK] Seq=28960 Ack=1 Win=64
36   1.175605  10.210.2.10 → 10.210.1.10  TCP 66 [TCP ZeroWindow] 5001 → 56426 [ACK] Seq=1 Ack=28961 Win=0
37   2.221656  10.210.1.10 → 10.210.2.10  TCP 66 [TCP Keep-Alive] 56426 → 5001 [ACK] Seq=28960 Ack=1 Win=64
38   2.263588  10.210.2.10 → 10.210.1.10  TCP 66 [TCP ZeroWindow] 5001 → 56426 [ACK] Seq=1 Ack=28961 Win=0
39   3.084589  10.210.2.10 → 10.210.1.10  TCP 66 [TCP Window Update] 5001 → 56426 [ACK] Seq=1 Ack=28961 Win
40   3.084611  10.210.1.10 → 10.210.2.10  TCP 1514 56426 → 5001 [ACK] Seq=28961 Ack=1 Win=64256 Len=1448 TSva
```

- **32:** “lo recibí bien, pero no me mandes más”.
- **33 a 38:** sondas de ventana, a intervalos que crecen.
- **39:** el programa leyó, y `b` avisa. `a` retoma donde quedó.

Los tres segundos, a escala



A los 3 s el programa lee, b avisa por su cuenta con una ventana nueva, y a retoma en 28961.

Las sondas de ventana

¿Por qué, si `b` ya avisa?

Ese aviso es un ACK sin datos, y esos no se retransmiten. Si se pierde, los dos se quedan esperando para siempre.

Lo que se capturó no es lo que dicen muchos textos

Hablan de “una sonda de un byte”. Aquí: `Len=0` con `Seq=2896`, uno menos que el siguiente byte. Un número ya confirmado obliga al otro a contestar.

Wireshark la etiqueta `TCP Keep-Alive`, porque tiene ese aspecto. Filtro: `tcp.analysis.keep_alive`.

Quién limita: la ventana o el enlace

$$\text{throughput} \leq \frac{\text{ventana}}{\text{RTT}}$$

SO_RCVBUF	Ventana máxima	Ventana / RTT	Medido
8192	8 688	1,7 Mbit/s	1,2 Mbit/s
32768	39 096	7,8 Mbit/s	6,8 Mbit/s
262144	261 848	52 Mbit/s	17,9 Mbit/s

40 ms de RTT, enlace de 20 Mbit/s. Una corrida: las cifras cambian, y la tendencia no.

- En las dos primeras manda la ventana. Sin pérdidas: el problema no es la red.
- En la tercera manda el enlace. Hace falta ventana $\geq$ ancho de banda $\times$ retardo $=$ 100 kB.

Linux agranda solo el buffer: la captura limpia anunció 1 035 136 bytes. Fijar SO_RCVBUF apaga ese ajuste.

Dos ventanas, dos problemas

	Control de flujo	Control de congestión
Protege a	el receptor	la red
Variable	`rwnd`	`cwnd`
Quién la decide	el receptor la anuncia	el emisor la deduce
En `ss`	`snd_wnd`	`cwnd`

Error común

Confundirlos porque los dos limitan al emisor. Una red lenta no cierra la ventana de recepción, y un receptor lento no es congestión.

El emisor respeta el menor

Bytes en vuelo $\leq$ mín(rwnd, cwnd). El segundo es el tema de la próxima semana.

Una conexión viva: `ss -tin`

python3 scenarios.py ss, recortado

```
ESTAB 0      738480   10.210.1.10:50756  10.210.2.10:5001
     cubic wscale:7,7 rto:324 rtt:123.177/0.523 mss:1448 pmtu:1500 ... cwnd:215 ssthresh:168 bytes_sent:1911360 ...
ESTAB 0      944096   10.210.1.10:50756  10.210.2.10:5001
     cubic wscale:7,7 rto:306 rtt:105.569/0.502 mss:1448 pmtu:1500 ... cwnd:178 ... retrans:0/12 ...
```

- `rtt`: promedio y desviación, en ms.
- `cwnd`: **en segmentos**.
- `retrans:0/12`: ahora, y en total.
- Tercera columna: aceptado y no confirmado.

- $215 \times 1448 \times 8/0{,}123 = 20{,}2$ Mbit/s: lo que da el enlace.
- 123 ms en un camino de 40 ms: lo demás es cola en `m`.
- Los 12 retransmitidos no vienen de `netem loss`: se llenó la cola.

SECCIÓN 09

Handshake

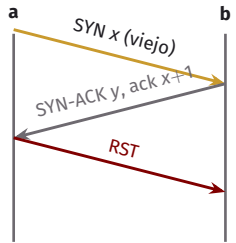
09

¿Por qué tres pasos?

Cada lado debe saber el número inicial del otro, y que el otro recibió el suyo. SYN, ACK, SYN, ACK: el segundo y el tercero viajan juntos.

¿Y con dos? Un **SYN viejo** abriría una conexión que nadie pidió.

Con tres, b espera un ACK de su número *y*, que no existía cuando salió el SYN viejo. Y a, que no reconoce la conexión, responde RST.



Idea clave

El handshake convierte “alguien me envió un paquete” en “alguien está ahí, recibe lo que le envío y sabe en qué número voy”.

Los números iniciales

Las capturas de hoy abren con 1661972085, 4265168592, 3960613348. Cambian en cada conexión.

1. **Confusión:** un segmento retrasado de una conexión anterior, con los mismos cuatro campos, podría caer en la ventana de la nueva.
2. **Seguridad:** quien adivine el número sin ver el tráfico puede inyectar segmentos falsos.

No es cifrado

Quien capture el tráfico lo lee. Protege contra quien no puede capturar. Para quien sí puede, está TLS.

Qué devuelve `connect`

Caso	Qué pasó	`connect`	Tiempo medido
Abierto	SYN-ACK	0	0,041 s
Cerrado	RST del kernel de `b`	`ECONNREFUSED`	0,042 s
Filtrado (`drop`)	Nadie responde	`ETIMEDOUT`	7,2 s (`TCP_SYNCNT` en 2)
Con `reject`	ICMP del router	`EHOSTUNREACH`	0,021 s
Sin ruta	ICMP del router	`ENETUNREACH`	0,020 s

Cerrado o filtrado

Rechazado al instante: el host existe y su kernel respondió. Colgado: algo se tragó el paquete.

Los dos últimos tardan 20 ms y no 40: responde `m`, a medio camino.

Una sorpresa: los reintentos del SYN

Esperábamos 1 s, 2 s y 4 s, y el total de 7 s parecía confirmarlo. La captura dice:

por defecto

```
0.000000000
1.059143000
2.083651000
3.107122000
4.131391000
5.155128000
```

tcp_syn_linear_timeouts=0

```
0.000000000
1.018746000
3.067214000
```

Con `tcp_syn_linear_timeouts=4`, este kernel reintenta cada 1 s antes de duplicar.

TCP_SYNCNT	SYN	se rinde a los
1	3	3,1 s
2	6	7,2 s
3	8	19,4 s

Lectura: no es un número de SYN, sino un plazo (3, 7 y 15 s) que se revisa cuando vence el temporizador.

SECCIÓN 10

Cierre

10

Cada sentido se cierra por separado

FIN dice “no voy a enviar más”, no “no quiero recibir más”. Con `shutdown(fd, SHUT_WR)`, la conexión queda **a medio cerrar**.

escenario halfclose

```
 7   0.055415  10.210.1.10 → 10.210.2.10  TCP 66 48000 → 5001 [FIN, ACK] Seq=3001 Ack=1
10   0.097516  10.210.2.10 → 10.210.1.10  TCP 66 5001 → 48000 [ACK] Seq=1 Ack=3001 Win=
11   0.098352  10.210.2.10 → 10.210.1.10  TCP 77 5001 → 48000 [PSH, ACK] Seq=1 Ack=3002
12   0.098372  10.210.1.10 → 10.210.2.10  TCP 66 48000 → 5001 [ACK] Seq=3002 Ack=12 Win
13   0.098373  10.210.2.10 → 10.210.1.10  TCP 66 5001 → 48000 [FIN, ACK] Seq=12 Ack=300
14   0.098375  10.210.1.10 → 10.210.2.10  TCP 66 48000 → 5001 [ACK] Seq=3002 Ack=13 Win
```

- El cliente mandó su FIN en el 7, y en el 11 **todavía recibe** 11 bytes.
- Aquí sí está el cierre de cuatro pasos: FIN (7), ACK (10), FIN (13), ACK (14).
- Útil cuando el servidor necesita saber que el pedido terminó.

TIME_WAIT

El que cierra primero espera. El RFC 9293 pide 2 MSL, con MSL de 2 minutos: cuatro minutos. Linux usa 60 s fijos.

```
ss -tano state time-wait sport = :5003

0      0         10.210.2.10:5003   10.210.1.10:38724 timer:(timewait,59sec,0)
```

1. **Para poder repetir el último ACK.** Si se pierde, el otro retransmite su FIN, y sin este estado recibiría un RST.
2. **Para que mueran los segmentos viejos.** No se puede crear otra conexión con esos cuatro campos.

Error común: creer que es un error. Es una conexión que terminó bien, y no impide atender a otros clientes.

Se pierde el último ACK

escenario timewait: m descarta el paquete 6

```
4   0.046540  10.210.1.10 → 10.210.2.10  TCP 66 54188 → 5001 [FIN, ACK] Seq=1 Ack=1 Win=642
5   0.090426  10.210.2.10 → 10.210.1.10  TCP 66 5001 → 54188 [FIN, ACK] Seq=1 Ack=2 Win=652
6   0.090445  10.210.1.10 → 10.210.2.10  TCP 66 54188 → 5001 [ACK] Seq=2 Ack=2 Win=64256 Le
7   0.367620  10.210.2.10 → 10.210.1.10  TCP 66 [TCP Retransmission] 5001 → 54188 [FIN, ACK
8   0.367712  10.210.1.10 → 10.210.2.10  TCP 66 [TCP Dup ACK 6#1] 54188 → 5001 [ACK] Seq=2
```

Quién contesta

Del lado de `a` ya no hay programa: `blast` terminó. El paquete 8 lo envía el estado que `TIME_WAIT` conservó en el kernel.

El costo: `bind` falla

Lo paga quien cierra primero. Si es el servidor, aparece un error famoso:

python3 scenarios.py bindfail, sin los comandos del cliente

```
b# timeout 1 /w10/sink 5003 --no-reuseaddr
  sin SO_REUSEADDR: bind: Address already in use
b# timeout 1 /w10/sink 5004
  con SO_REUSEADDR: bind funciona
```

- Por eso los servidores del curso ponen `SO_REUSEADDR`.
- La opción debe estar en los **dos** sockets: el que dejó el `TIME_WAIT` y el nuevo. Probado a mano: si solo la pone el nuevo, el `bind` también falla.

RST: abortar

RST no cierra, aborta: no se confirma, y el otro se entera con un error en su `read`.

un puerto cerrado

```
1   0.000000  10.210.1.10 → 10.210.2.10  TCP 74 55194 → 5999 [SYN] Seq=0 Win=64240 Len=0 MS
2   0.041334  10.210.2.10 → 10.210.1.10  TCP 54 5999 → 55194 [RST, ACK] Seq=1 Ack=1 Win=0 L
```

close con SO_LINGER en cero

```
7   0.041594  10.210.1.10 → 10.210.2.10  TCP 66 56884 → 5001 [RST, ACK] Seq=3001 Ack=1
8   0.083510  10.210.2.10 → 10.210.1.10  TCP 66 5001 → 56884 [ACK] Seq=1 Ack=1449 Win=
9   0.083525  10.210.1.10 → 10.210.2.10  TCP 54 56884 → 5001 [RST] Seq=1449 Win=0 Len=
```

- Servidor: `read: Connection reset by peer`.
- Cualquier segmento que llegue a una conexión que no existe recibe un RST: paquetes 8 y 9.
- Un tercero que falsifique un RST válido corta una conexión ajena (CS168 lo menciona como censura).

SECCIÓN 11

SYN flood

11

SYN flood

SYN flood (RFC 4987)

Muchos SYN sin completar el handshake, normalmente con origen falso. La cola se llena, y los clientes legítimos no entran.

Sin falsificar nada: `backlog` 16, 100 SYN desde otra dirección de `a`, y `m` descarta sus SYN-ACK.

```
python3 scenarios.py syn, recortado: con cookies apagadas

cookies off: 17 en SYN-RECV, connect devolvió -1 en 7.154 s: ETIMEDOUT (Connection timed out), {'TcpExtTCPRe
```

```
con cookies encendidas

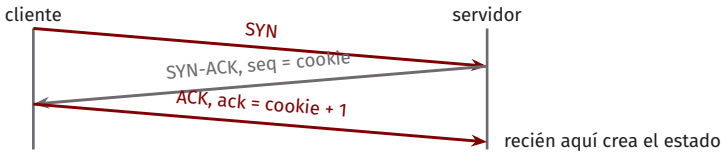
cookies on: 17 en SYN-RECV, connect devolvió 0 en 0.041 s, {'TcpExtTCPReqQFullDrop': 0, 'TcpExtSyncookiesSen
```

La misma cola llena, y el cliente legítimo entra en un RTT.

SYN cookies

SYN cookies

Con la cola llena, el servidor no guarda estado: lo codifica **en el número de secuencia del SYN-ACK**. Un hash de los cuatro campos y un secreto, un contador de tiempo y el MSS.



- El atacante con dirección falsa nunca vio el SYN-ACK, y no puede responder ese número.
- Costo: no hay dónde guardar todo lo del SYN. Por eso solo se activan con la cola llena.
- Protegen la memoria del servidor, no el ancho de banda de su enlace.

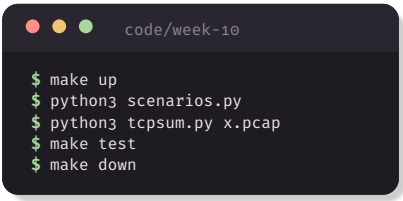
SECCIÓN 12

Laboratorio

12

Laboratorio: leer TCP en capturas

1. `clean`: handshake, cierre, MSS.
2. Gráfico tiempo-secuencia: expliquen la forma.
3. `loss`: duplicados y fast retransmit. Un timeout real no va a aparecer: revisen el contador, y úsenlo en `rto`.
4. Completen `lab/slow_reader.c`, y encuentren la ventana cero.
5. `ports`: cerrado y filtrado.
6. `ss`: `rtt`, `rto` y `cwnd`.
7. Sigan con **P3**.



Basado en parte en el laboratorio de TCP de Kurose y Ross.

La prueba de `slow_reader` se salta hasta que quiten el `TODO`.

Resumen

- TCP numera bytes. `seq`: el primero que lleva. `ack`: el siguiente que espera. SYN y FIN gastan un número.
- RTO $=$ EstimatedRTT $+4$ DevRTT, DevRTT primero. Linux: piso de 200 ms sobre el RTT. Al vencer, se duplica.
- Tres ACK duplicados: fast retransmit. SACK dice qué llegó, RACK decide por tiempo y TLP hace probing.
- Las etiquetas de Wireshark son deducciones. Los contadores del kernel son la fuente.
- El control de flujo protege al receptor; el de congestión, a la red. Throughput $\leq$ ventana / RTT.
- Tres pasos, para descartar SYN viejos. Cerrado: `ECONNREFUSED`. Filtrado: `ETIMEDOUT`.
- Quien cierra primero queda en `TIME_WAIT`. Un SYN flood llena la cola; las SYN cookies guardan el estado en el número de secuencia.

Para la próxima clase

- Terminar el laboratorio y avanzar con **P3**, que se entrega la próxima semana. Es otro protocolo, sobre UDP: nada de aquí se copia.
- Próxima semana: control de congestión. Slow start, AIMD, CUBIC y BBR.
- Resolver los ejercicios de los apuntes.

Pregunta

Una transferencia va lenta. En la captura hay muchos `[TCP Window Full]` y ninguna retransmisión, y `ss` muestra un `cwnd` grande. ¿Dónde está el problema? ¿Qué cambiarían? ¿Y si vieran lo contrario: `snd_wnd` grande y muchas retransmisiones?