

Computación Centrada en Redes

Control de congestión

Transporte y aplicaciones

Gerson Garrido

Semana 11

UNIVERSIDAD NACIONAL DE INGENIERÍA

Agenda

Sesión A

1. Qué es la congestión y cuánto cuesta
2. La ventana de congestión
3. Slow start, congestion avoidance y fast recovery
4. La misma curva en un kernel real
5. Por qué AIMD

Sesión B

6. La fórmula del throughput
7. CUBIC
8. Bufferbloat
9. BBR y ECN
10. Equidad

> **Lectura**
>
> Kurose y Ross, secs. 3.6 y 3.7.1 (Sesión A), 3.7.2, 3.7.3 y 3.7.4 (Sesión B). CS168: *Congestion Control Principles*, *Design* e *Implementation*; *TCP Throughput Model*, *Congestion Control Issues* y *Router-Assisted Congestion Control*. Complementarias: RFC 5681, RFC 9438 y RFC 3168.

SECCIÓN 01

Qué es la congestión

01

Una pregunta que TCP todavía no responde

- El control de flujo evita desbordar al **receptor**.
- ¿Quién evita desbordar a la **red**?

Congestión

Durante un tiempo sostenido, a un enlace le llega más tráfico del que puede transmitir. La diferencia se acumula en el buffer del router y, si se llena, se pierde.

Lo difícil

El receptor sabe cuánto espacio tiene. Nadie sabe cuánto ancho de banda hay disponible: millones de emisores deciden cada uno por su cuenta.

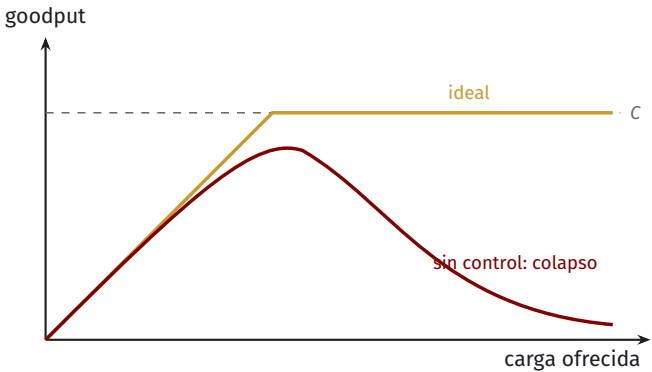
Flujo y congestión no son lo mismo

	Control de flujo	**Control de congestión**
Protege a	El receptor	La red
Quién pone el límite	El receptor, en el header	Nadie: el emisor lo estima
Variable	`rwnd`	`cwnd`
Si faltara	Se desborda un host	Se desborda una cola, y sufren todos

Lo que cuesta: 10 Mbit/s, buffer de 1,5 MB, llegan 12 Mbit/s

1. **Retardo.** La cola crece 0,25 MB por segundo. A los 6 s está llena: $1{,}5 \times 8/10 = 1{,}2$ s de espera.
2. **Pérdida.** Entran 12, salen 10: se bota uno de cada seis. Lo que ya recorrió el camino gastó esos enlaces para nada.
3. **Retransmisiones inútiles.** Si el RTO es de 500 ms, el emisor reenvía algo que sigue en la cola. Se transmiten dos copias.
4. **Colapso.** Más congestión, más copias, más congestión. El enlace sigue ocupado, y el goodput cae casi a cero.

El colapso: más carga, menos goodput



Octubre de 1986

- TCP tenía control de flujo y retransmisión, pero no control de congestión.
- Enlace entre UC Berkeley y Lawrence Berkeley Lab, a menos de 400 metros.
- El throughput cayó de **32 kbit/s a 40 bit/s**: 800 veces menos.
- Van Jacobson y Michael Karels cambiaron TCP: el emisor ajusta su ventana según las pérdidas.
- Sin tocar routers ni aplicaciones.

Realimentación

Retransmitir, por sí solo, empeora la causa de la pérdida. Con control de congestión, el emisor retransmite *y además* baja su velocidad.

Quién se encarga

End-to-end

La red no dice nada. El emisor deduce la congestión de lo que ve: pérdidas y retardo. Es lo que hizo TCP en 1986.

Con ayuda de la red

Los routers informan: con un bit en el header (ECN) o incluso con la velocidad exacta.

Con precisión

En los dos casos el router termina avisando. En uno es explícito; en el otro, el emisor lo deduce de una cola llena: es una señal implícita y tardía.

SECCIÓN 02

La ventana de congestión

02

Una variable y una regla

$$\text{bytes en vuelo} \leq \text{mín}(\texttt{cwnd},\ \texttt{rwnd})$$

- Es la ventana deslizante de la semana 9, con un tamaño que cambia.
- Hoy el receptor lee rápido: `rwnd` es grande y manda `cwnd`.

$$\text{throughput} \approx \frac{\texttt{cwnd}}{\text{RTT}}$$

Auto-reloj

Cada ACK avisa que un paquete salió de la red y autoriza a poner otro. Los ACK vuelven al ritmo del cuello de botella, sin que el emisor conozca su velocidad.

El BDP: lo que cabe en el camino

Producto ancho de banda por retardo

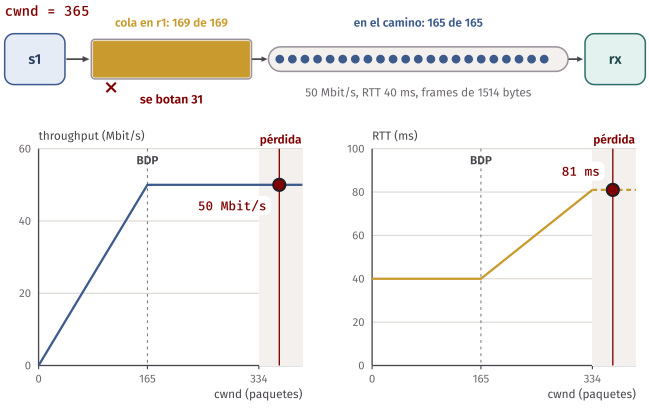
BDP $= 50$ Mbit/s $\times$ 0,040 s $= 250\,000$ bytes ≈ 167 paquetes de 1500 bytes.

	cwnd $<$ BDP	cwnd $=$ BDP	cwnd $>$ BDP
Enlace	sin llenar	lleno, sin cola	lleno, la cola crece
Throughput	cwnd/RTT	rate	rate
RTT	mínimo	mínimo	mínimo + cola/rate

Error común

Una ventana más grande no siempre da más throughput. Pasado el BDP, cada paquete agrega retardo. Y sobre BDP $+$ buffer, hay pérdida.

Más ventana no siempre es más velocidad

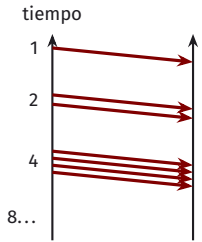


Lo que no cabe se bota. TCP basado en pérdidas sube hasta aquí, sea cual sea el tamaño del buffer.

Slow start

Regla: por cada ACK que confirma datos nuevos, `cwnd` sube un MSS.

- Vuelven `cwnd` ACK por RTT: **se duplica en cada RTT**.
- Termina en `ssthresh`, con una pérdida, o si lo limita `rwnd`.
- RFC 5681: empezar con hasta 4 segmentos. Linux: 10.



Error común

No es lento: es exponencial. Se llama así porque antes de 1986 se empezaba enviando de una vez toda la ventana anunciada.

Congestion avoidance

Regla

`cwnd` aumenta en un MSS por RTT. Sin cronómetro: por cada ACK, `cwnd` $+=$ MSS $\times$ MSS$/$`cwnd`.

Empieza cuando `cwnd` $\geq$ `ssthresh`.

No es un descuido

Sigue creciendo hasta perder. TCP clásico no tiene otra forma de descubrir que hay más capacidad: **provoca la pérdida a propósito**. Eso nos va a costar en la Sesión B.

Las dos señales de pérdida

Tres ACK duplicados: fast recovery

La red todavía entrega.

1. `ssthresh` $=$ máx(FlightSize$/2,\ 2$ MSS)
2. Retransmite. `cwnd` $=$ `ssthresh` $+\ 3$
3. Cada ACK duplicado: `cwnd` $+=1$
4. ACK nuevo: `cwnd` $=$ `ssthresh`

Timeout

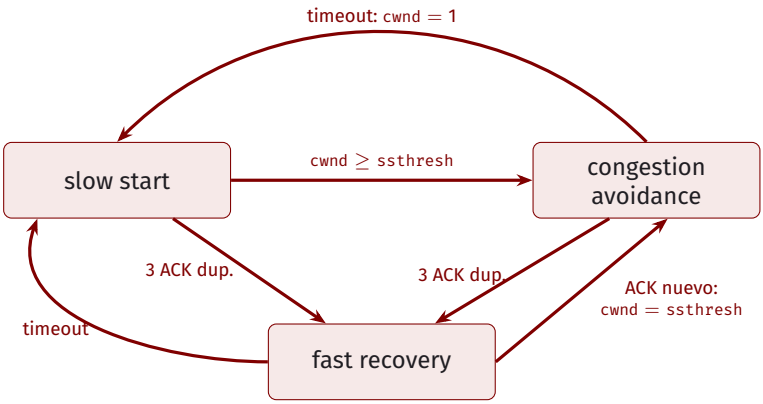
El auto-reloj se detuvo.

- `ssthresh`, igual
- `cwnd` $=1$
- Slow start

En la escala de RTT

Con ACK duplicados, `cwnd` queda en la mitad y sigue de a uno, sin slow start. Tahoe volvía siempre a 1; Reno agregó fast recovery.

Las tres fases juntas



Ejemplo resuelto: `cwnd` por RTT (filas elegidas)

```
reno_trace.py --ssthresh 16 --rounds 26 ...
```

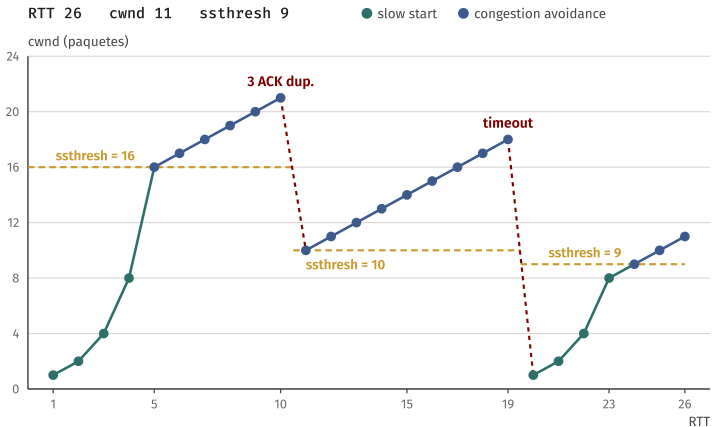
RTT	cwnd	ssthresh	fase	evento
1	1	16	slow start	
2	2	16	slow start	
3	4	16	slow start	
4	8	16	slow start	
5	16	16	congestion avoidance	
6	17	16	congestion avoidance	
9	20	16	congestion avoidance	
10	21	16	congestion avoidance	3 ACK dup.
11	10	10	congestion avoidance	
12	11	10	congestion avoidance	
18	17	10	congestion avoidance	
19	18	10	congestion avoidance	timeout
20	1	9	slow start	
21	2	9	slow start	
22	4	9	slow start	
23	8	9	slow start	
24	9	9	congestion avoidance	

- **RTT 1–5:** se duplica hasta `ssthresh`.
- **RTT 10:** tres ACK duplicados en 21. `ssthresh` = 10. Durante la recuperación vale 13 y se infla; al salir, 10.
- **RTT 19:** timeout en 18. `ssthresh` = 9, `cwnd` = 1.
- **RTT 23–24:** de 8 no dobla a 16: se corta en 9.

Es un modelo

Cuenta por RTT, en paquetes. No tiene ACK ni temporizadores: sirve para la cuenta en papel.

La misma cuenta, dibujada



ssthresh = 9 y cwnd = 1. Slow start hasta 8; el doble se pasaría del umbral, así que se corta en 9 y vuelve a sumar 1.

SECCIÓN 03

La misma curva, en un kernel real

03

La topología de la semana



- **Cola en `r1`:** 250 KiB, igual al BDP. La vamos a cambiar.
- **Retardo y pérdida en `r2`:** la pérdida de `netem` es aleatoria, no por congestión.
- **Los 40 ms solo en el sentido de los datos.** El RTT queda en 40 ms.
- `tcp_no_metrics_save=1` en los emisores, para que cada corrida empiece de cero.

Elegir y observar desde C

tcp_info.c, resumido

```c
if (setsockopt(fd, IPPROTO_TCP, TCP_CONGESTION,
       algorithm, (socklen_t)strlen(algorithm)) <
     0) {
    perror("setsockopt TCP_CONGESTION");
    return -1;
}
struct tcp_info info;
socklen_t size = sizeof info;
if (getsockopt(fd, IPPROTO_TCP, TCP_INFO,
               &info, &size) < 0) {
    perror("getsockopt TCP_INFO");
    return -1;
}
```

Quién hace qué

El control de congestión lo hace el kernel. Este programa solo elige cuál, antes de conectar, y lo observa.

- `iperf3 -C reno` hace ese mismo `setsockopt`.
- `ss -ti` muestra los mismos datos.

Slow start, en vivo

```
s1: /lab/tcp_info send 10.211.4.10 6000 reno 2 20

# algoritmo: reno
#   t_ms   cwnd ssthresh   rtt_ms retrans estado
    0.0     10        -     45.1       0 open
   61.0     20        -     46.0       0 open
  102.6     40        -     44.9       0 open
  141.6     78        -     45.3       0 open
  204.5    156        -     51.5       0 open
  241.7    251        -     56.4       0 open
  281.8    404        -     50.7       0 open
  340.7    648        -     77.3       0 open
  360.8    724        -     81.3       0 disorder
  385.4    582      365     82.4      46 recovery
  401.7    483      365     82.7      77 recovery
  426.5    365      365     82.2     175 recovery
  622.0    317      182     75.7     419 recovery
```

Filas elegidas de una corrida con muestras cada 20 ms.

- Empieza en 10 y dobla cada 40 ms.
- Sin `ssthresh`, hasta la primera pérdida.
- **Exceso:** llega a 724, cuando caben unos 335. Los ACK traen noticias de hace un RTT.
- `ssthresh` = 365, la mitad.
- `cwnd` no sube a `ssthresh`+3: baja de a poco (PRR).

Lo que el libro simplifica

El libro	Linux, en esta máquina
Ventana inicial de 1	10 segmentos
`cwnd` en bytes o en MSS	`ss` y `TCP_INFO`: segmentos
En recovery, `ssthresh` + 3, y se infla	Baja de a poco hasta `ssthresh`: PRR, RFC 6937
Solo se sabe del primer perdido	SACK dice qué llegó
CUBIC sale de slow start al perder	HyStart puede sacarlo antes

No cambia la idea

Al terminar la recuperación, `cwnd` queda en `ssthresh`, que es la mitad de lo que había. Las reglas del libro sirven para razonar, hacer la cuenta y escribir un primer protocolo.

Una línea de ss

```
s1: ss -tin "sport = :40001"

ESTAB 0      3020528   10.211.1.10:40001  10.211.4.10:5201
 reno wscale:7,7 rto:259 rtt:58.127/0.396 mss:1448 pmtu:1500 rcvmss:536 advmss:1448 cwnd:239 ssthresh:178
 bytes_sent:22609109 bytes_retrans:580648 bytes_acked:21682390 segs_out:15617 segs_in:4033 data_segs_out:15615
 send 47629776bps lastrcv:3840 pacing_rate 57155112bps delivery_rate 47147896bps delivered:14976 busy:3840ms
 unacked:239 retrans:0/401 reordering:8 reord_seen:15 rcv_space:14480 rcv_ssthresh:64088 notsent:2674456
 minrtt:40.154 snd_wnd:3881600 rcv_wnd:64256
```

La salida es una sola línea; aquí la partimos para que entre.

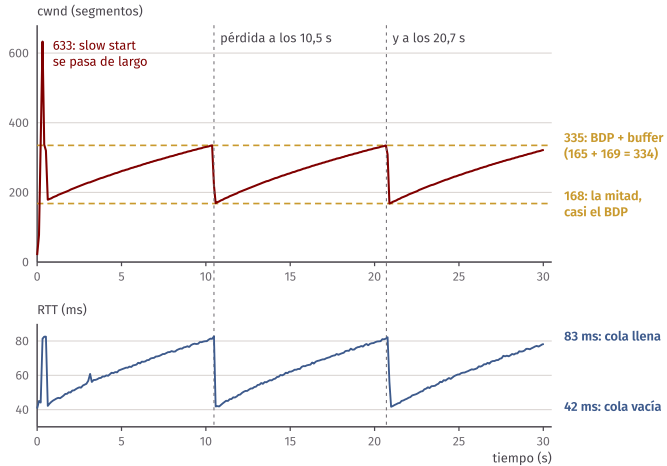
- `cwnd:239 ssthresh:178`: congestion avoidance.
- `rtt` menos `minrtt`: 18 ms de cola.

Hagan la cuenta

$239 \times 1448 \times 8 / 0{,}058127 \approx 47{,}6$ Mbit/s, justo `send`. El kernel usa `cwnd`/RTT.

Ojo: `iperf3` abre una conexión de control y otra de datos. Por eso fijamos `-cport`.

Treinta segundos de Reno



Esa curva se puede predecir

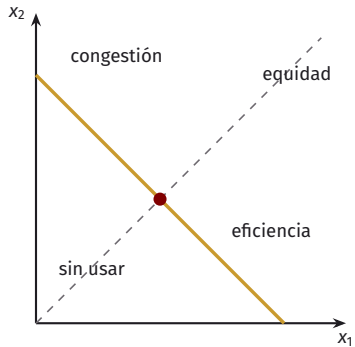
- **Dónde se pierde:** BDP + cola = 165 + 169 = 334.
- **Hasta dónde baja:** 168, casi el BDP. La cola se vacía justo.
- **El RTT:** de 42 a 83 ms.
- **Ciclo:** $167 \times 0{,}062 \approx 10{,}4$ s.
- **47,5 y no 50 Mbit/s:** $50 \times 1448/1514 = 47{,}8$.

SECCIÓN 04

Por qué AIMD

04

Dos flujos, una recta y una diagonal



Modelo de Chiu y Jain (1989)

Capacidad C. Todos reciben la misma señal: congestión si $x_1 + x_2 > C$. Cada uno aplica la misma regla, sin saber del otro.

- **Sumar:** línea de pendiente 1. La diferencia no cambia.
- **Multiplicar:** recta por el origen. La razón no cambia, y la diferencia se multiplica por ese factor.

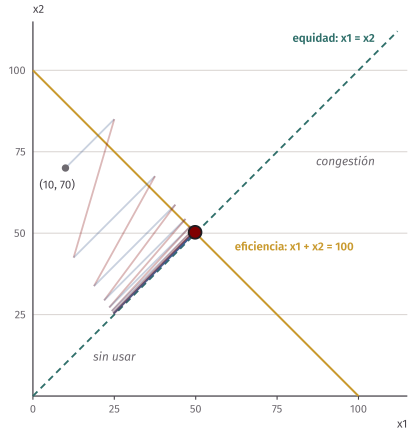
AIMD reduce la diferencia a la mitad en cada bajada

```
phase.py aimd --steps 10
```

paso	x1	x2	suma	|x1-x2|	Jain
0	10.0	70.0	80.0	60.0	0.640
2	20.0	80.0	100.0	60.0	0.735
3	25.0	85.0	110.0	60.0	0.771
4	12.5	42.5	55.0	30.0	0.771
8	32.5	62.5	95.0	30.0	0.909
9	37.5	67.5	105.0	30.0	0.925
10	18.8	33.8	52.5	15.0	0.925

Filas elegidas. $C = 100$, $+5$ al subir, $\times 0{,}5$ al bajar.

La misma trayectoria, en el plano



PHASE.PY AIMD

paso	45
x1	49,8
x2	50,3
suma	100,1
Jain	1,000

0,5

|x1 − x2|

TRAS 200 PASOS · JAIN

aimd	1,000
aiad	0,640
mimd	0,640
miad	0,500

AIMD oscila alrededor de la capacidad y se pega a la diagonal.
Las otras reglas no llegan: a los 200 pasos, su índice de Jain queda en 0,640 o 0,500.

Las otras tres no llegan

phase.py all: el punto tras 200 pasos

regla	x1	x2	suma	|x1-x2|	Jain
aimd	45.0	45.0	90.0	0.0	1.000
aiad	10.0	70.0	80.0	60.0	0.640
mimd	7.6	53.5	61.1	45.8	0.640
miad	0.0	109.7	109.7	109.7	0.500

- **AIAD:** la diferencia nunca se mueve.
- **MIMD:** la razón queda fija en 7.
- **MIAD:** un flujo termina en cero.

AIMD no reparte: converge a un reparto

Bajar multiplicando quita más al que tiene más; subir sumando le da lo mismo a los dos. Nadie necesita saber cuántos flujos hay, y por eso escala.

El precio: la sierra. Un flujo de Reno pasa su vida entre $W/2$ y W.

Cómo se mide: el índice de Jain

$$J = \frac{(x_1 + \cdots + x_n)^2}{n\,(x_1^2 + \cdots + x_n^2)}$$

x_1	x_2	J
25	25	1,000
35	15	0,862
45	5	0,610
50	0	0,500

- Va de $1/n$ a 1.
- Si k flujos se reparten todo y el resto recibe cero: $J = k/n$.

Lo que no ve

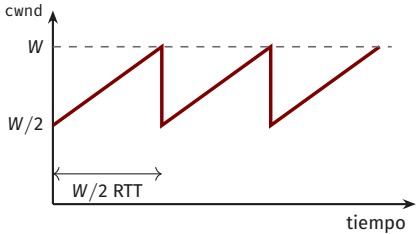
Eficiencia: dos flujos de 1 kbit/s en un enlace de 1 Gbit/s dan $J = 1$.

SECCIÓN 05

La fórmula del throughput

05

De la sierra a una fórmula



1. Ventana promedio: $3W/4$.
2. Paquetes por ciclo: $(W/2)(3W/4) = 3W^2/8$.
3. Uno perdido por ciclo: $p = 8/(3W^2)$.

$$\text{throughput} \approx \sqrt{\frac{3}{2}} \frac{\text{MSS}}{\text{RTT}\sqrt{p}} \qquad \text{con } \sqrt{3/2} \approx 1{,}22$$

- **No depende de la capacidad** del enlace, si este no es lo que limita.
- Inverso al RTT. Para duplicar, hay que reducir p a la cuarta parte.

La fórmula, contra el laboratorio

Pérdida	Fórmula, Reno	Reno medido	CUBIC medido
1 %	3,5	3,2 · 3,2 · 3,9	3,0 · 2,8 · 4,2
0,1 %	11,2	12,6 · 11,2 · 11,7	14,1 · 9,5 · 10,9
0,01 %	35,5	37,1 · 29,8 · 39,8	32,9 · 39,8 · 45,1
0 %	no aplica	47,5	47,3

Mbit/s en un enlace de 50 Mbit/s y 40 ms, con MSS de 1448 bytes. Corridas de 20 s; sin pérdida, 30 s.

La pérdida no siempre es congestión

Esta pérdida es aleatoria, con la cola casi vacía. TCP no puede distinguirlas: 1 % le quita más del 90 % del enlace.

¿Quién resiste mejor?

Con 1 % y 0,1 %, en este enlace, empatan dentro del ruido. Veremos por qué.

Por qué Reno sufre en enlaces rápidos y largos

Enlace de 1 Gbit/s, 100 ms de RTT, paquetes de 1500 bytes:

- Ventana promedio: $10^9 \times 0{,}1/(1500 \times 8) \approx 8333$ paquetes.
- Pérdida tolerable: $p = (1{,}22/8333)^2 \approx 2{,}1 \times 10^{-8}$.
- Una pérdida cada 46 millones de paquetes: una cada nueve minutos, a esa velocidad.

El problema es “uno por RTT”

Después de una pérdida, Reno recupera miles de paquetes de ventana, de a uno por RTT. Esa regla estaba bien para los enlaces de 1986, pero no escala con el BDP.

SECCIÓN 06

CUBIC

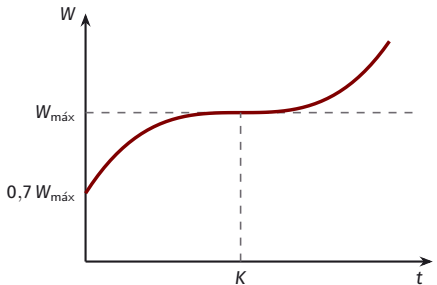
06

CUBIC: reduce menos y crece según el tiempo

$$W(t) = C\,(t - K)^3 + W_{\text{máx}}$$
$$K = \sqrt[3]{(W_{\text{máx}} - \text{cwnd}_0)/C}$$

$C = 0{,}4$, $\beta = 0{,}7$ (Reno: 0,5). t en segundos, desde la última pérdida.

- **Lejos de $W_{\text{máx}}$:** sube rápido.
- **Cerca de $W_{\text{máx}}$:** casi plano. Ahí hubo problemas.
- **Más allá:** hace probing, y acelera.



Manda el tiempo

En Reno, la ventana crece por RTT. En $W(t)$, el RTT no aparece, y tampoco en K.

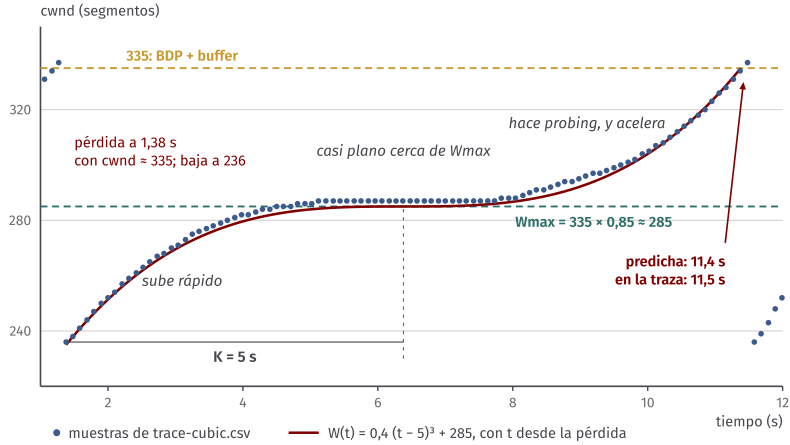
Cuánto tarda cada uno en recuperarse

```
python3 cubic_curve.py --recovery
```

enlace	RTT	BDP (paq.)	Reno	CUBIC
50 Mbps	40 ms	167	3.3 s	5.0 s
50 Mbps	200 ms	833	83.3 s	8.5 s
1000 Mbps	40 ms	3333	66.7 s	13.6 s
1000 Mbps	200 ms	16667	1666.7 s	23.2 s
10000 Mbps	100 ms	83333	4166.7 s	39.7 s

- Reno: $(W_{\text{máx}}/2) \times$ RTT. CUBIC: K. Con $W_{\text{máx}}$ igual al BDP.
- En nuestro enlace, Reno se recupera antes. En el último, tarda casi 70 minutos, frente a menos de uno.

La curva, en la traza real



La cuenta, con la fórmula

Se pierde con cwnd ≈ 335 (BDP + buffer):

- Baja a $0{,}7 \times 335 \approx 235$. ✓
- Meseta en 287, no en 335: **convergencia rápida**, $W_{\text{máx}} = 335 \times 0{,}85$.
- $K = \sqrt[3]{50/0{,}4} = 5$ s: meseta en $t \approx 6{,}4$ s.
- $W(t) = 336 \Rightarrow$ pérdida en $t \approx 11{,}4$ s. **La traza: 11,5 s.**

Ruido

A los 16,6 s hay una reducción en 256 que la fórmula no explica. Por eso comparamos tendencias.

Región amigable con Reno

- Con RTT corto y ventana chica, Reno alcanzaría a la curva.
- CUBIC calcula también W_{est}, un AIMD con $\alpha = 3(1-\beta)/(1+\beta) \approx 0{,}53$ y reducción de 0,7, que imita la ventana promedio de Reno. Usa la mayor.

Y por eso empatan con 1 % de pérdida

Ventana promedio: $1{,}22/\sqrt{0{,}01} \approx 12$ paquetes. Una pérdida cada 100 paquetes: 8 RTT, unos 0,3 s. K sería de 2 s, y la curva nunca despega. CUBIC se queda en la región amigable, **y hace lo mismo que Reno a propósito**.

Dónde gana CUBIC

Con BDP grande y pérdida rara. Donde Reno anda bien, se comporta como Reno: por eso se puede desplegar sin dañar a los demás.

Medir β en sus propias trazas

```
decrease_ratios, en las trazas de lab/

reno   [0.57, 0.28, 0.5]
cubic  [0.94, 0.66, 0.7, 0.69, 0.7, 0.71]
```

`ssthresh` nuevo sobre el `cwnd` más alto antes de la pérdida.

- Los primeros valores son del exceso de slow start: el muestreo cada 100 ms lo ve mal.
- Los de régimen son 0,5 y 0,7.

En el laboratorio

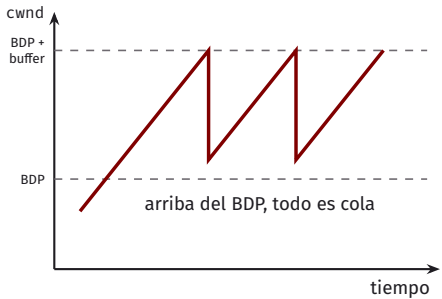
Esa función es una de las que ustedes completan en `lab/metrics.py`.

SECCIÓN 07

Bufferbloat

07

TCP llena el buffer que encuentre



Retardo de cola máximo

buffer $\times$ 8/rate

Buffer	A 50 Mbit/s	Veces BDP
50 KiB	8,2 ms	0,2
250 KiB	41 ms	1
2000 KiB	328 ms	8

Un `ping` que cruza la misma cola

buffer de 2000 KiB

```
$ python3 bench.py bloat --cc cubic --seconds 20
cubic: 46.1 Mbit/s, 90 retransmisiones
ping sin carga: 42.4 ms; con carga: mediana 300.1 ms, máximo 381.0 ms; recibidos 75 de 75
```

Buffer	`ping` Reno	`ping` CUBIC	Reno	CUBIC
50 KiB	41,1 ms	41,1 ms	22,6 Mbit/s	12,6 Mbit/s
250 KiB	60,1 ms	61,0 ms	45,6 Mbit/s	46,7 Mbit/s
2000 KiB	**196,9 ms**	**300,1 ms**	46,6 Mbit/s	46,1 Mbit/s

Bufferbloat

De 250 a 2000 KiB, el throughput no sube y el retardo se multiplica, para *todos*. El orden entre Reno y CUBIC no es ley: en otra corrida, CUBIC dio 120 ms.

El buffer chico también cuesta

- Con 50 KiB, el mínimo de Reno, (BDP + buffer)/2, queda bajo el BDP: el enlace se queda sin usar una parte del ciclo.
- Y varía mucho. Tres corridas de 30 s con el mismo buffer:

Reno	21,9 · 14,9 · 38,3 Mbit/s
CUBIC	44,7 · 23,6 · 43,7 Mbit/s

No hay un tamaño correcto para todo

Con un flujo, la regla clásica es buffer $\approx$ BDP. Pero el BDP depende del RTT de cada flujo, y el router no lo conoce.

En su casa: si la videollamada se traba cuando alguien sube un video, sospechen del buffer del módem.

SECCIÓN 08

BBR y ECN

08

BBR: medir en vez de esperar la pérdida

Óptimo: `cwnd` = BDP, enlace lleno y sin cola. BBR estima:

- **BtlBw:** máximo de la velocidad de entrega reciente.
- **RTprop:** mínimo del RTT reciente.

Envía con **pacing**, y usa la ventana como límite de seguridad.

No se mide aquí

El kernel del curso solo ofrece `reno` y `cubic`.



Las dos cantidades no se miden a la vez: el máximo pide llenar el enlace y el mínimo, vaciar la cola.

BBR y los demás

- Una pérdida aleatoria que no cambia el ancho de banda, en principio, no le cambia el modelo.
- El costo: convive mal. La versión original recibió dos críticas:

Buffers chicos

La pérdida no lo frena. CUBIC sí retrocede, y le cede el enlace.

Buffers grandes

CUBIC llena la cola, y BBR, limitado por su ventana, puede quedarse con menos.

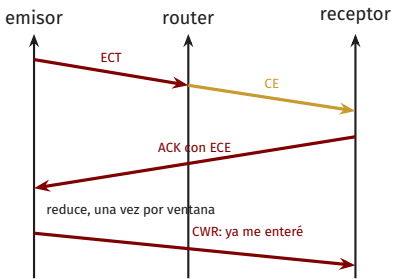
“Basado en retardo”

Buena primera aproximación, y así lo presenta CS168. Más preciso: **estima un modelo del camino** y envía según él. Las versiones nuevas usan también la pérdida.

ECN: la red avisa sin botar

Bits IP	Significado
`00`	Not-ECT
`10`, `01`	ECT: sabe usar ECN
`11`	CE: router marcó

Se negocia en el handshake: `[SEW]`, `[S.E]`.



La misma cola, con y sin ECN

RED marca o bota al azar cuando el promedio de la cola pasa de `min`, sin esperar a que se llene.

r1: tc -s qdisc show dev eth3, con RED min 30 KiB, max 90 KiB, limit 1000 KiB, en dos corridas de 20 s

```
# con ECN   cubic: 43.9 Mbit/s, 0 retransmisiones      ping con carga: mediana 41.5 ms
qdisc red 10: parent 1:1 limit 1000Kb min 30Kb max 90Kb ecn
 Sent 115325495 bytes 76271 pkt (dropped 0, overlimits 125 requeues 0)
  marked 125 early 0 pdrop 0 other 0
# sin ECN   cubic: 43.7 Mbit/s, 16 retransmisiones     ping con carga: mediana 41.9 ms
qdisc red 10: parent 1:1 limit 1000Kb min 30Kb max 90Kb ecn
 Sent 114856208 bytes 75961 pkt (dropped 16, overlimits 16 requeues 0)
  marked 0 early 16 pdrop 0 other 0
```

Las líneas con `#` resumen lo que imprimió `bench.py`; el resto es de `tc`, sin `tbf` ni `backlog`. **Dos ideas distintas:** la cola gestionada (AQM) baja el retardo, y ECN evita perder el paquete al avisar. **El precio:** con umbrales bajo el BDP, al repetir, de 17 a 44 Mbit/s con ECN y de 23 a 41 sin él.

SECCIÓN 09

Equidad

09

Dos flujos con el mismo algoritmo

Corrida	Reno (s1 · s2)	Jain	CUBIC (s1 · s2)	Jain
1	17,9 · 26,5	0,964	29,6 · 15,0	0,903
2	23,8 · 21,2	0,997	26,5 · 18,9	0,973
3	22,2 · 20,9	0,999	17,2 · 29,8	0,933

30 s, 50 Mbit/s, 40 ms, buffer de 250 KiB.

- Reparto razonable, sin coordinador, y el ganador cambia.
- Treinta segundos son pocos ciclos: el de Reno, solo, dura unos 10 s.
- CUBIC reparte algo peor aquí. El RFC 9438 advierte que reducir a 0,7 hace más lenta la convergencia.

Reno contra CUBIC

Corrida	s1 Reno · s2 CUBIC	Jain	s1 CUBIC · s2 Reno	Jain
1	28,0 · 14,6	0,910	29,7 · 16,9	0,930
2	22,4 · 23,8	0,999	30,8 · 16,0	0,909
3	27,6 · 17,7	0,955	28,0 · 19,0	0,964

- Promedios: CUBIC 24,1 y Reno 21,7 Mbit/s. Casi un empate, con este BDP.
- En cinco de seis corridas ganó el de `s1`, sin importar el algoritmo. No conocemos la causa, y podría ser casualidad.

Método

Por eso se corren las dos posiciones y se repite. Con una sola corrida, habríamos concluido cualquier cosa.

RTT distintos: 40 ms contra 120 ms

Corrida	Reno: corto · largo	Razón	CUBIC: corto · largo	Razón
1	41,0 · 6,6	6,2	37,7 · 10,1	3,7
2	38,6 · 9,1	4,2	34,8 · 12,8	2,7
3	39,0 · 8,4	4,6	34,3 · 13,4	2,6

- **Dos efectos:** a igual ventana, menos velocidad; y en Reno la ventana crece por RTT.
- RFC 9438: si pierden a la vez, Reno se acerca al **cuadrado** de la razón de RTT, y CUBIC se mantiene **lineal**.
- Con unos 20 ms de cola: $140/60 \approx 2{,}3$, y su cuadrado es 5,4. Compatible con los datos, sin ser una demostración.

Una razón más para la CDN

Acortar el RTT da más throughput a la misma conexión, aunque el enlace no cambie.

TCP reparte entre conexiones, no entre usuarios

s1 abre una conexión; s2 abre cuatro

```
$ python3 bench.py pair --cc cubic --parallel2 4 --seconds 30
s1 cubic: 13.7 Mbit/s, 166 retransmisiones
s2 cubic: 32.4 Mbit/s, 380 retransmisiones
Jain: 0.858
```

- Con 5 conexiones iguales, a `s2` le tocaría el 80 %. Se lleva cerca del 70 %.
- **Fair queuing** reparte bien aunque el emisor no coopere, a costa de más trabajo en el router.

Error común

Decir que TCP es “justo” en general. Solo tiende a serlo entre conexiones largas, con el mismo RTT, el mismo algoritmo y sin nadie que abra más conexiones.

SECCIÓN 10

Laboratorio

10

Laboratorio: Reno y CUBIC bajo distintos enlaces y buffers

1. Línea base: 50 Mbit/s y 40 ms. ¿Por qué 47 y no 50?
2. `cwnd` en el tiempo: `--csv` y `plot_cwnd.py`.
3. Con 1 % de pérdida. Comparen con la fórmula.
4. Bufferbloat: 2000 KiB y luego 50 KiB, con `ping`.
5. Dos flujos, en las dos posiciones. Índice de Jain.
6. Distinto RTT: `--extra-s2 80`.
7. Completen `lab/metrics.py`.
8. Entreguen **P3** y lean **P4**.

code/week-11

```
$ make up
$ python3 bench.py link
$ python3 bench.py flow --cc reno
$ make test
$ make down
```

Predigan antes de ejecutar. Las pruebas de `lab/` se saltan hasta que quiten los `TODO`.

P4 es otra cosa

Aquí se mide el TCP del kernel. En P4, ustedes escriben el control de congestión de `rimay`, en C y en bytes, reaccionando a cada ACK.

Resumen

- El emisor envía mín(`cwnd`, `rwnd`), y throughput $\approx$ `cwnd`/RTT. Sobre el BDP, solo hay cola.
- Slow start dobla por RTT; congestion avoidance suma uno. ACK duplicados: mitad. Timeout: 1.
- AIMD es la combinación que converge a la equidad.
- La fórmula explica por qué 1 % de pérdida hunde a Reno y por qué no escala.
- CUBIC gana con BDP grande y poca pérdida. En otros casos se parece a Reno a propósito.
- Bufferbloat: TCP basado en pérdidas llena cualquier buffer.
- BBR estima un modelo del camino; ECN marca en lugar de botar.
- El RTT corto gana, y TCP reparte entre conexiones.

Para la próxima clase

- Terminar el laboratorio y entregar **P3**.
- Leer el enunciado de **P4**: `rimay`, congestión. Entrega en la semana 13.
- Repasar con las preguntas de la guía y los ejercicios de los apuntes.

Pregunta

Con un buffer enorme, una descarga no pierde casi nada y el `ping` sube a 300 ms. Un compañero dice: “no se pierde nada, así que la red está sana”. ¿Qué le responden, y qué medirían para convencerlo?