

UNIVERSIDAD NACIONAL DE INGENIERÍA

Computación Centrada en Redes

Qué es Internet y cómo medirla

Piezas, conmutación de paquetes, retardo, throughput y capas

Gerson Garrido

Semana 1

Hoy

1. **Sesión A.** El curso. Qué es Internet: piezas, servicio y protocolos. Redes de acceso. Conmutación de paquetes y de circuitos. La red de redes.
2. **Sesión B.** Retardo, colas y pérdida. `traceroute`. Throughput. Capas y encapsulamiento.
3. **Laboratorio.** Docker, Wireshark y `traceroute`.

> **Lectura**
>
> Kurose y Ross (9.ª ed.), secs. 1.1–1.5. CS168: *Introduction*, *Designing Resource Sharing*, *Layers* y *Headers*. Complementaria: CS168, *Links*.

SECCIÓN 01

El curso

01

Cinco partes

Fundamentos

arquitectura,
retardo,
capas, sockets

sem. 1–2

Cinco partes

Fundamentos

arquitectura, retardo, capas, sockets

sem. 1–2

Redes locales e IP

Ethernet, VLANs, forwarding, NAT

sem. 3–4

Cinco partes

Fundamentos

arquitectura, retardo, capas, sockets

sem. 1–2

Redes locales e IP

Ethernet, VLANs, forwarding, NAT

sem. 3–4

Routing

OSPF, BGP, políticas entre redes

sem. 5–7

Cinco partes

Fundamentos

arquitectura, retardo, capas, sockets

sem. 1–2

Redes locales e IP

Ethernet, VLANs, forwarding, NAT

sem. 3–4

Routing

OSPF, BGP, políticas entre redes

sem. 5–7

Transporte y aplicaciones

TCP, congestión, DNS, HTTP, TLS, QUIC

sem. 9–13

Cinco partes

Fundamentos	**Redes locales e IP**	**Routing**	**Transporte y aplicaciones**	**Redes modernas**
arquitectura, retardo, capas, sockets	Ethernet, VLANs, forwarding, NAT	OSPF, BGP, políticas entre redes	TCP, congestión, DNS, HTTP, TLS, QUIC	data centers, SDN, 5G, seguridad
sem. 1–2	sem. 3–4	sem. 5–7	sem. 9–13	sem. 14–15

Otro orden que el libro: primero cómo llega un paquete; después, lo que corre encima. Para probar transporte, hace falta una red donde se pierdan paquetes.

Cómo funciona el curso

- **Teoría** (4 h): dos sesiones por semana.
- **Laboratorio** (2 h): práctica guiada.
- **Proyecto**: Qhapaq Ñan, un solo proyecto que crece.
- **Lectura**: Kurose y Ross, y lecturas gratuitas cada semana.
- **Lenguaje**: C, con sockets POSIX. Routers con FRRouting.
- **Exámenes**: parcial en la semana 8 y final en la semana 16.

Qhapaq Ñan: lo que van a construir

	Entrega	Lo que agrega
P0	Calentamiento	Un servicio con sockets y un `traceroute` propio
P1	Tu red	Direccionamiento, VLANs, NAT y OSPF
P2	Qhapaq Ñan	BGP con políticas y defensa ante un hijack
P3	`rimay`: entrega confiable	Ventana deslizante sobre UDP
P4	`rimay`: congestión	Control de congestión y mediciones
P5	`wasi`	Servidor HTTP sobre `rimay`, publicado en la red

P0 se publica esta semana y se entrega en la semana 3.

El entorno, y una regla

- Todo corre en su laptop: un contenedor con el compilador y las herramientas, y Containerlab.
- Cada router es otro contenedor, con FRRouting.
- Se instala en el laboratorio de esta semana.

La regla del curso

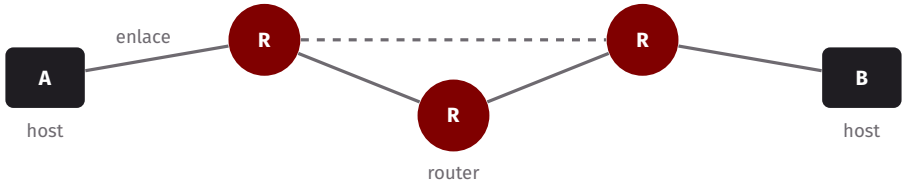
En redes, **una afirmación vale lo que vale su medición**. Toda salida de estas láminas viene de una ejecución real. Las suyas tendrán otras cifras: dependen de su máquina y de su conexión.

SECCIÓN 02

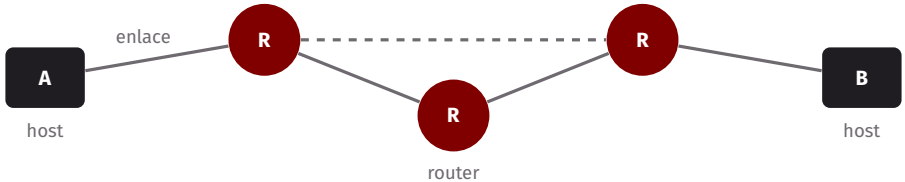
Qué es Internet

02

Primera descripción: sus piezas

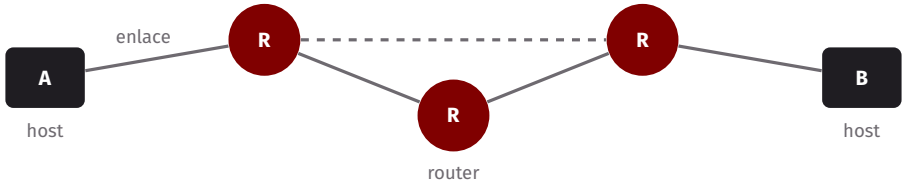


Primera descripción: sus piezas



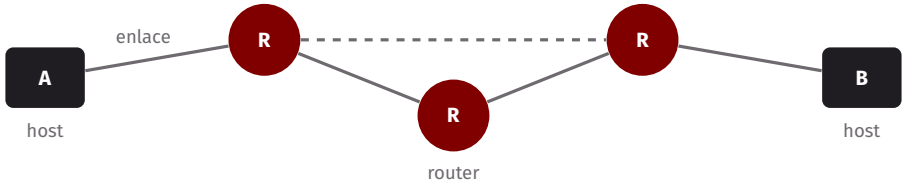
- **Hosts**: donde corren las aplicaciones. Laptop, celular, servidor.

Primera descripción: sus piezas



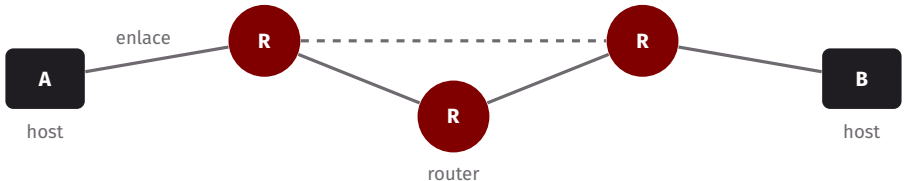
- **Hosts**: donde corren las aplicaciones. Laptop, celular, servidor.
- **Enlaces** *(links)*: unen nodos. Tienen una velocidad, en bits por segundo.

Primera descripción: sus piezas



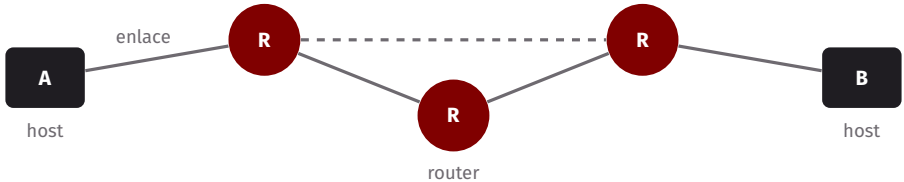
- **Hosts**: donde corren las aplicaciones. Laptop, celular, servidor.
- **Enlaces** *(links)*: unen nodos. Tienen una velocidad, en bits por segundo.
- **Routers y switches**: reciben un paquete por un enlace y lo reenvían por otro.

Primera descripción: sus piezas



- **Hosts**: donde corren las aplicaciones. Laptop, celular, servidor.
- **Enlaces** *(links)*: unen nodos. Tienen una velocidad, en bits por segundo.
- **Routers y switches**: reciben un paquete por un enlace y lo reenvían por otro.
- **ISPs**: las organizaciones que operan todo lo anterior.

Primera descripción: sus piezas



- **Hosts**: donde corren las aplicaciones. Laptop, celular, servidor.
- **Enlaces** *(links)*: unen nodos. Tienen una velocidad, en bits por segundo.
- **Routers y switches**: reciben un paquete por un enlace y lo reenvían por otro.
- **ISPs**: las organizaciones que operan todo lo anterior.

Los routers no ejecutan su aplicación: el núcleo solo mueve paquetes. Por eso se pudo inventar la Web sin cambiar ningún router.

Segunda descripción: su servicio

- Para un programador, Internet es una infraestructura que da servicios a las aplicaciones.
- El pedido: “lleva estos bytes a ese programa, en ese host”.
- Se pide por una interfaz: la **API de sockets** (semana 2).

Best effort

La red intenta entregar cada paquete, pero **no promete** que llegue, ni cuándo, ni en qué orden.

Segunda descripción: su servicio

- Para un programador, Internet es una infraestructura que da servicios a las aplicaciones.
- El pedido: “lleva estos bytes a ese programa, en ese host”.
- Se pide por una interfaz: la **API de sockets** (semana 2).

Best effort

La red intenta entregar cada paquete, pero **no promete** que llegue, ni cuándo, ni en qué orden.

¿Un defecto?

Es la decisión que hizo posible todo lo demás. Si una aplicación necesita más, lo construye en los hosts: TCP da bytes confiables sobre una red que no promete nada. Ustedes harán lo mismo con `rimay`.

Qué es un protocolo

Protocolo

Define el **formato** y el **orden** de los mensajes entre dos o más entidades, y las **acciones** que se toman al enviar o recibir un mensaje, o ante otro evento.

h1

```
$ curl -sv -o /dev/null http://10.2.0.2/ ...
> GET / HTTP/1.1
> Host: 10.2.0.2
> User-Agent: curl/8.14.1
> Accept: */*
>
< HTTP/1.0 200 OK
< Server: SimpleHTTP/0.6 Python/3.12.10
< Date: Mon, 02 Nov 2026 06:48:13 GMT
< Content-type: text/html
< Content-Length: 97
< Last-Modified: Mon, 02 Nov 2026 06:32:09 GMT
<
```

Formato: línea, encabezados, línea en blanco. **Orden:** primero habla el cliente.

Las acciones: distintos mensajes, distintas respuestas

HTTP es texto: se puede hablar a mano con `nc`.

h1

```
$ printf 'GET /nada.html HTTP/1.0\r\n\r\n' | nc -w 2 10.2.0.2 80 | head -1
HTTP/1.0 404 File not found

$ printf 'BORRAR / HTTP/1.0\r\n\r\n' | nc -w 2 10.2.0.2 80 | head -1
HTTP/1.0 501 Unsupported method ('BORRAR')
```

Error común

"El protocolo TCP/IP" no existe. TCP e IP son dos protocolos, de capas distintas. Un router usa IP y no necesita saber nada de TCP. Lo veremos en una captura.

SECCIÓN 03

El borde de la red

03

Redes de acceso

Tecnología	Medio	¿Compartido?
DSL	Par de cobre telefónico	No hasta la central; cae con la distancia
Cable (HFC)	Fibra y coaxial	Sí, entre vecinos del tramo
Fibra (FTTH)	Fibra óptica	Sí, entre casas del mismo splitter
Wi-Fi	Radio, decenas de metros	Sí, en el mismo access point
Móvil (4G, 5G)	Radio, kilómetros	Sí, en la misma celda

- Un plan de 500 Mbps no está reservado para ustedes.
- Muchos planes son **asimétricos**: más bajada que subida.
- Lo común: Wi-Fi hasta el equipo de la casa, y fibra hasta el ISP.

Dos velocidades que no son lo mismo

Velocidad de transmisión (R)

Cuántos bits por segundo se ponen en el medio. Depende de la **tecnología**. Se mejora cambiando equipos.

Velocidad de propagación (s)

Cuán rápido viaja cada bit. Depende de la **física**: unos 2×10^8 m/s en fibra y cobre, cerca de 3×10^8 m/s en el aire.

Dos velocidades que no son lo mismo

Velocidad de transmisión (R)

Cuántos bits por segundo se ponen en el medio. Depende de la **tecnología**. Se mejora cambiando equipos.

Velocidad de propagación (s)

Cuán rápido viaja cada bit. Depende de la **física**: unos 2×10^8 m/s en fibra y cobre, cerca de 3×10^8 m/s en el aire.

Bits y bytes

Enlaces en bits/s, con prefijos decimales; archivos en bytes. Un plan de 100 Mbps baja, como máximo, 12,5 MB por segundo.

SECCIÓN 04

El núcleo de la red

04

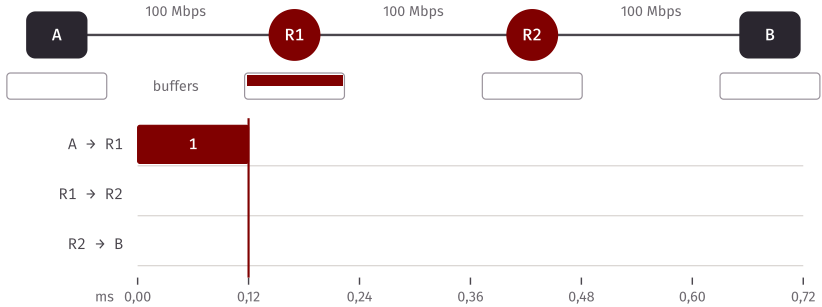
Store-and-forward

Store-and-forward

Un router debe **recibir el paquete completo** antes de empezar a reenviarlo: tiene que leer el encabezado, comprobar que llegó bien y decidir por dónde sale.

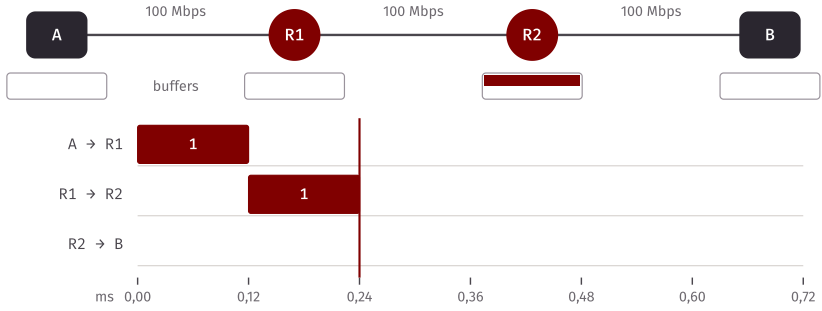
N enlaces iguales: $d = N \cdot L/R$. 1500 bytes, 3 enlaces de 100 Mbps: $3 \times 0{,}12 = 0{,}36$ ms.

Store-and-forward, paso a paso



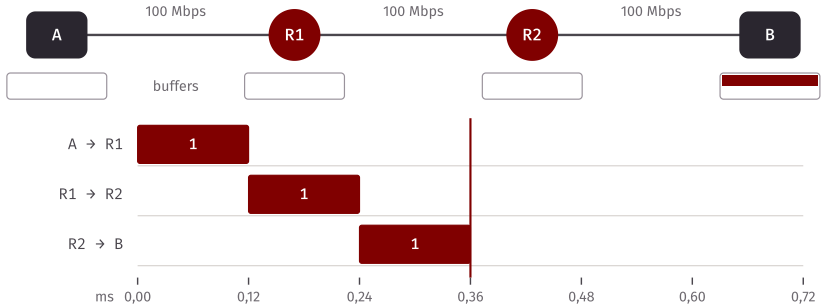
El enlace 1 tarda L/R = 12 000 bits / 100 Mbps = 0,12 ms. R1 guarda los bits hasta tener el paquete completo.

Store-and-forward, paso a paso



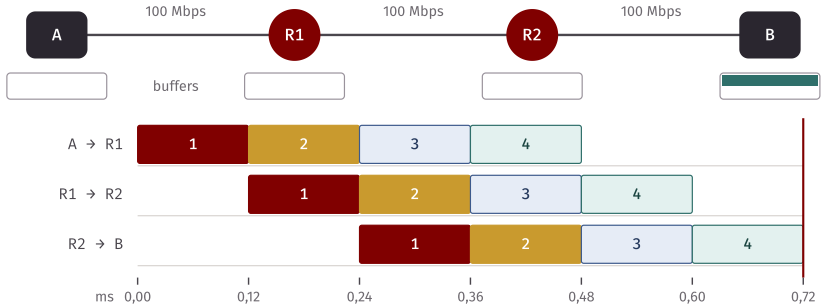
Recién entonces R1 lo reenvía, y tarda otros 0,12 ms. R2 hace lo mismo: para decidir necesita el paquete entero.

Store-and-forward, paso a paso



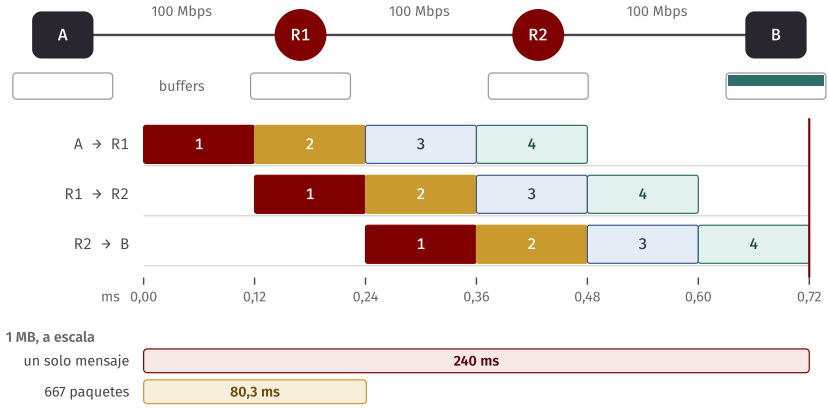
Tres enlaces: $d = N \cdot L/R = 3 \cdot 0{,}12\ \text{ms} = 0{,}36\ \text{ms}$.

Store-and-forward, paso a paso



Varios paquetes avanzan en fila, como en una línea de montaje: después del primero, llega uno cada 0,12 ms.

Store-and-forward, paso a paso



Con 1 MB: como un solo mensaje, 3 · 80 = 240 ms. Como 667 paquetes, (3 + 666) · 0,12 = 80,3 ms.

¿Por qué dividir en paquetes?

Enviar **1 MB** por 3 enlaces de 100 Mbps, ignorando los encabezados.

Un solo mensaje

8 000 000 bits. Cada enlace: 80 ms.

Total: $3 \times 80 = \mathbf{240\ ms}$.

¿Por qué dividir en paquetes?

Enviar **1 MB** por 3 enlaces de 100 Mbps, ignorando los encabezados.

Un solo mensaje

8 000 000 bits. Cada enlace: 80 ms.

Total: $3 \times 80 = \mathbf{240\ ms}$.

667 paquetes de 1500 bytes

El primero llega a los 0,36 ms.
Luego, uno cada 0,12 ms.

Total: $(3 + 666) \times 0{,}12 = \mathbf{80{,}3\ ms}$.

¿Por qué dividir en paquetes?

Enviar **1 MB** por 3 enlaces de 100 Mbps, ignorando los encabezados.

Un solo mensaje

8 000 000 bits. Cada enlace: 80 ms.

Total: $3 \times 80 = \mathbf{240\ ms}$.

667 paquetes de 1500 bytes

El primero llega a los 0,36 ms.
Luego, uno cada 0,12 ms.

Total: $(3 + 666) \times 0{,}12 = \mathbf{80{,}3\ ms}$.

Dos ganancias

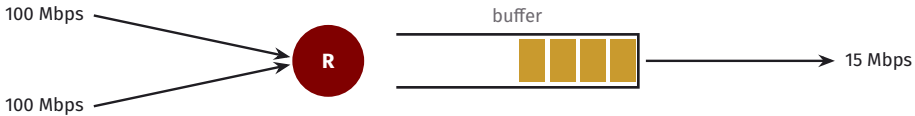
Los paquetes avanzan en fila, como en una línea de montaje. Y si se pierde uno, solo se repite ese.

Colas y pérdida



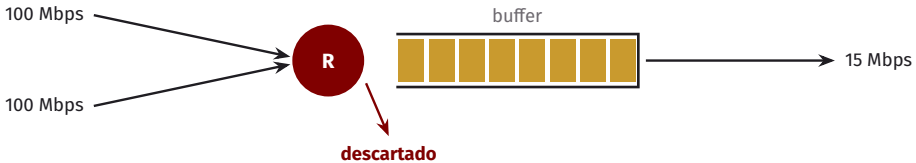
- Sin reserva, si llegan más rápido de lo que salen, esperan: **retardo de cola**.

Colas y pérdida



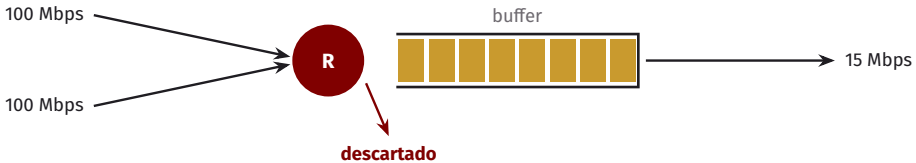
- Sin reserva, si llegan más rápido de lo que salen, esperan: **retardo de cola**.

Colas y pérdida



- Sin reserva, si llegan más rápido de lo que salen, esperan: **retardo de cola**.
- Si el buffer se llena, el router descarta: **pérdida**.

Colas y pérdida



- Sin reserva, si llegan más rápido de lo que salen, esperan: **retardo de cola**.
- Si el buffer se llena, el router descarta: **pérdida**.
- No es una falla: es el precio de no reservar. Y TCP existe para convivir con ello.

Forwarding y routing

r2

```
$ ip route
default via 172.90.1.1 dev eth0
10.0.12.0/24 dev eth1 proto kernel scope link src 10.0.12.2
10.0.23.0/24 dev eth2 proto kernel scope link src 10.0.23.2
10.1.0.0/24 nhid 19 via 10.0.12.1 dev eth1 proto 196 metric 20
10.2.0.0/24 nhid 20 via 10.0.23.3 dev eth2 proto 196 metric 20
172.90.1.0/24 dev eth0 proto kernel scope link src 172.90.1.6
```

Para llegar a `10.2.0.0/24`, salir por `eth2` hacia `10.0.23.3`.

Forwarding

Mover cada paquete usando la tabla. Decisión local, muy rápida.

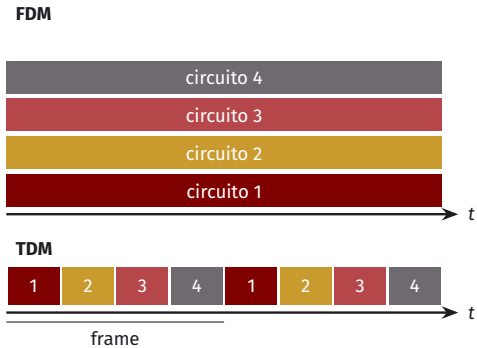
Routing

Decidir caminos y llenar tablas. Trabajo de toda la red, más lento.

Conmutación de circuitos

Antes de enviar, se reserva capacidad en cada enlace del camino. Si no se usa, se pierde.

- **FDM**: una banda de frecuencias, todo el tiempo.
- **TDM**: un slot de cada frame, con toda la capacidad.



Comparación, con un ejemplo

100 Mbps con TDM de 10 slots; 25 MB. Circuito: 200 Mbit/10 Mbps + 0,2 s = **20,2 s**, aunque nadie más use el enlace. Paquetes, con el enlace libre: **2 s**.

	Paquetes	**Circuitos**
Reserva previa	Ninguna	En cada enlace
Capacidad	La toma quien la necesita	Fija: si no se usa, se pierde
Retardo	Variable: colas	Constante, ya establecido
Pérdida por congestión	Posible	No, si se admitió
Si un enlace cae	Puede tomar otro camino	Se cae el circuito
Estado en el núcleo	Tablas de forwarding	Por cada circuito

Comparación, con un ejemplo

100 Mbps con TDM de 10 slots; 25 MB. Circuito: 200 Mbit/10 Mbps + 0,2 s = **20,2 s**, aunque nadie más use el enlace. Paquetes, con el enlace libre: **2 s**.

	Paquetes	**Circuitos**
Reserva previa	Ninguna	En cada enlace
Capacidad	La toma quien la necesita	Fija: si no se usa, se pierde
Retardo	Variable: colas	Constante, ya establecido
Pérdida por congestión	Posible	No, si se admitió
Si un enlace cae	Puede tomar otro camino	Se cae el circuito
Estado en el núcleo	Tablas de forwarding	Por cada circuito

El tráfico de datos llega a ráfagas *(bursts)*: reservar para el pico es un desperdicio.

Multiplexación estadística

Enlace de **1 Gbps**. Cada usuario activo transmite a **100 Mbps**.

- Con circuitos: caben 10.

Multiplexación estadística

Enlace de **1 Gbps**. Cada usuario activo transmite a **100 Mbps**.

- Con circuitos: caben 10.
- Con paquetes, si cada uno está activo el 10 % del tiempo, **de forma independiente**:

Multiplexación estadística

Enlace de **1 Gbps**. Cada usuario activo transmite a **100 Mbps**.

- Con circuitos: caben 10.
- Con paquetes, si cada uno está activo el 10 % del tiempo, **de forma independiente**:

$X \sim \text{Binomial}(N, p)$

$P(X = k) = \binom{N}{k} p^k (1-p)^{N-k}$

El problema es $P(X > 10)$.

Multiplexación estadística

Enlace de **1 Gbps**. Cada usuario activo transmite a **100 Mbps**.

- Con circuitos: caben 10.
- Con paquetes, si cada uno está activo el 10 % del tiempo, **de forma independiente**:

$X \sim \text{Binomial}(N, p)$

$P(X = k) = \binom{N}{k} p^k (1-p)^{N-k}$

El problema es $P(X > 10)$.

Usuarios	Promedio activos	$P(X > 10)$
10	1	0
20	2	$7{,}1 \times 10^{-7}$
30	3	$8{,}9 \times 10^{-5}$
40	**4**	$\mathbf{1{,}5 \times 10^{-3}}$
50	5	$9{,}4 \times 10^{-3}$
100	10	0,42

`./multiplex 1000 100 0.1 N`

Multiplexación estadística

Enlace de **1 Gbps**. Cada usuario activo transmite a **100 Mbps**.

- Con circuitos: caben 10.
- Con paquetes, si cada uno está activo el 10 % del tiempo, **de forma independiente**:

$X \sim \text{Binomial}(N, p)$

$P(X = k) = \binom{N}{k} p^k (1-p)^{N-k}$

El problema es $P(X > 10)$.

Usuarios	Promedio activos	$P(X > 10)$
10	1	0
20	2	$7{,}1 \times 10^{-7}$
30	3	$8{,}9 \times 10^{-5}$
40	**4**	$\mathbf{1{,}5 \times 10^{-3}}$
50	5	$9{,}4 \times 10^{-3}$
100	10	0,42

```
./multiplex 1000 100 0.1 N
```

Con 40 usuarios se pasa del límite el 0,15 % del tiempo, y entonces los paquetes primero esperan en el buffer.

Lo que hay detrás de esa tabla

Idea clave

Cada usuario pide su **pico** solo una fracción del tiempo. Los paquetes no crean capacidad: usan la que queda libre. Con p cercano a 1 no hay ganancia.

Error común: olvidar la independencia

Si todos se activan por la misma causa (una final del mundial, la matrícula en línea, un sismo), la probabilidad real es mucho mayor.

En la práctica: los ISP venden más capacidad de la que tienen (**oversubscription**). Por eso el servicio es barato, y por eso se pone lento en horas de mayor uso.

SECCIÓN 05

Una red de redes

05

¿Quién conecta a quién?

Hay miles de redes de acceso. Armemos la estructura como un razonamiento de diseño.

¿Quién conecta a quién?

Hay miles de redes de acceso. Armemos la estructura como un razonamiento de diseño.

1. **Todos con todos.** $N(N-1)/2$ enlaces: para decenas de miles de redes, cientos de millones de enlaces. Imposible.

¿Quién conecta a quién?

Hay miles de redes de acceso. Armemos la estructura como un razonamiento de diseño.

1. **Todos con todos.** $N(N-1)/2$ enlaces: para decenas de miles de redes, cientos de millones de enlaces. Imposible.
2. **Un solo ISP global.** Funciona, y justamente por eso aparecen competidores.

¿Quién conecta a quién?

Hay miles de redes de acceso. Armemos la estructura como un razonamiento de diseño.

1. **Todos con todos.** $N(N-1)/2$ enlaces: para decenas de miles de redes, cientos de millones de enlaces. Imposible.
2. **Un solo ISP global.** Funciona, y justamente por eso aparecen competidores.
3. **Lo que tenemos.** Varios ISP globales conectados entre sí. Debajo, ISP regionales que les compran acceso y venden a los de acceso.

La jerarquía, y quién le paga a quién



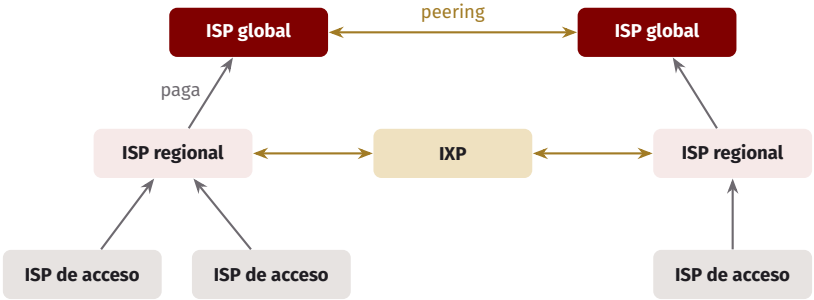
La jerarquía, y quién le paga a quién



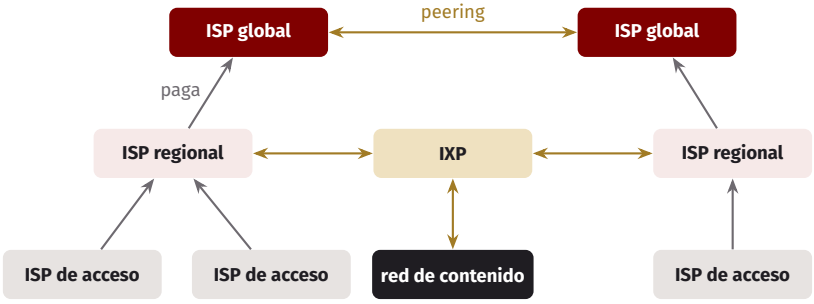
La jerarquía, y quién le paga a quién



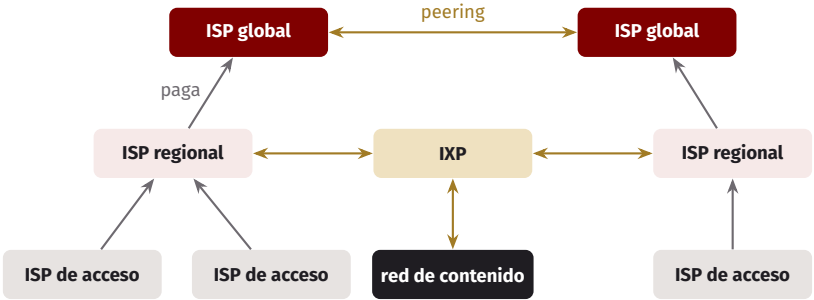
La jerarquía, y quién le paga a quién



La jerarquía, y quién le paga a quién



La jerarquía, y quién le paga a quién



IXP: lugar donde muchas redes se conectan a un switch para hacer peering. En el Perú, el NAP Perú.

Sin dueño y sin centro

Idea clave

Internet son decenas de miles de redes independientes, los **sistemas autónomos**, cada una con su negocio. Los caminos los deciden, en buena medida, contratos entre empresas, no la distancia.

- **Cliente y proveedor:** el cliente paga.
- **Peering:** conexión directa, normalmente sin pago, cuando ambos ganan.
- **Multi-homing:** dos o más proveedores, para resistir fallas.
- En la semana 6 lo configuran ustedes, con BGP.

SECCIÓN 06

Retardo

06

Las cuatro componentes

$$d_{\text{nodal}} = d_{\text{proc}} + d_{\text{cola}} + d_{\text{trans}} + d_{\text{prop}}$$



	Depende de	Valor típico
Procesamiento	del router	microsegundos o menos
Cola	de la carga del enlace	de cero a milisegundos, o más
Transmisión	del tamaño y de la velocidad del enlace	microsegundos a milisegundos
Propagación	de la distancia y del medio	nanosegundos a cientos de ms

Las cuatro componentes

$$d_{\text{nodal}} = d_{\text{proc}} + d_{\text{cola}} + d_{\text{trans}} + d_{\text{prop}}$$



	Depende de	Valor típico
Procesamiento	del router	microsegundos o menos
Cola	de la carga del enlace	de cero a milisegundos, o más
Transmisión	del tamaño y de la velocidad del enlace	microsegundos a milisegundos
Propagación	de la distancia y del medio	nanosegundos a cientos de ms

Las cuatro componentes

$$d_{\text{nodal}} = d_{\text{proc}} + d_{\text{cola}} + d_{\text{trans}} + d_{\text{prop}}$$



	Depende de	Valor típico
Procesamiento	del router	microsegundos o menos
Cola	de la carga del enlace	de cero a milisegundos, o más
Transmisión	del tamaño y de la velocidad del enlace	microsegundos a milisegundos
Propagación	de la distancia y del medio	nanosegundos a cientos de ms

Las cuatro componentes

$$d_{\text{nodal}} = d_{\text{proc}} + d_{\text{cola}} + d_{\text{trans}} + d_{\text{prop}}$$



	Depende de	Valor típico
Procesamiento	del router	microsegundos o menos
Cola	de la carga del enlace	de cero a milisegundos, o más
Transmisión	del tamaño y de la velocidad del enlace	microsegundos a milisegundos
Propagación	de la distancia y del medio	nanosegundos a cientos de ms

Transmisión no es propagación

Cien personas salen por una puerta angosta y caminan a otro edificio.

- **Transmisión**: que crucen la puerta todas. Depende de cuántas son y del ancho de la puerta.
- **Propagación**: lo que camina cada una. Depende de la distancia.

Error común

"Mi enlace es de 1 Gbps" no dice nada de la propagación. Entre Lima y Tokio siguen siendo decenas de milisegundos.

Ninguno domina siempre

Lima a Miami: 1500 bytes, 100 Mbps, unos 4200 km en línea recta.

delay.c

```
$ ./delay 1500 100 4200
transmision L/R =      0.120 ms
propagacion d/s =     21.000 ms
domina la propagacion, 175.0 veces mayor
```

Enlace lento y corto: 1500 bytes, 1 Mbps, 1 km.

delay.c

```
$ ./delay 1500 1 1
transmision L/R =     12.000 ms
propagacion d/s =      0.005 ms
domina la transmision, 2400.0 veces mayor
```

Ninguno domina siempre

Lima a Miami: 1500 bytes, 100 Mbps, unos 4200 km en línea recta.

delay.c

```
$ ./delay 1500 100 4200
transmision L/R =      0.120 ms
propagacion d/s =     21.000 ms
domina la propagacion, 175.0 veces mayor
```

Enlace lento y corto: 1500 bytes, 1 Mbps, 1 km.

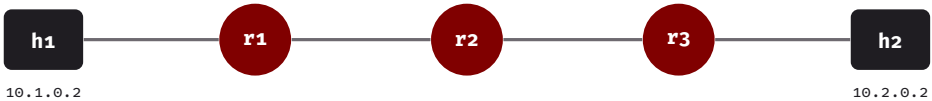
delay.c

```
$ ./delay 1500 1 1
transmision L/R =     12.000 ms
propagacion d/s =      0.005 ms
domina la transmision, 2400.0 veces mayor
```

Producto bandwidth-delay

Bits “en vuelo” en un enlace: $R \times d_{\text{prop}}$. Hacia Miami, 100 Mbps $\times$ 21 ms = 2,1 Mbit, unos 262 KB. Un paquete de 1500 bytes ocupa menos del 1 % del cable. Volverá en la semana 9.

Una red para medir



r2/frr.conf

```
1  frr defaults traditional
2  hostname r2
3  interface eth1
4   ip address 10.0.12.2/24
5  interface eth2
6   ip address 10.0.23.2/24
7  ip route 10.1.0.0/24 10.0.12.1
8  ip route 10.2.0.0/24 10.0.23.3
```

- Cada nodo es un contenedor. Cada enlace, un par de interfaces virtuales.
- **netem** frena lo que sale por una interfaz. Un enlace necesita una regla en cada extremo.

r2

```
$ tc qdisc replace dev eth2 root netem delay 21ms
```

El experimento: RTT según el tamaño

`ping` de cada tamaño, 40 veces; nos quedamos con el **mínimo**, que descarta colas y ruido.

```
python3 experiments.py size
```

```
RTT mínimo en ms, según los bytes de datos de ping

caso                                  64     400     800    1200    1472   R estimada
un enlace a 5 Mbps                  1.56    1.88    3.36    5.61    5.67      4.8 Mbps
tres enlaces a 5 Mbps               5.63    8.88   11.66   16.84   18.05      5.3 Mbps
un enlace a 5 Mbps y cable de 21 ms 44.89   45.15   46.87   48.54   49.80      4.4 Mbps
```

El experimento: RTT según el tamaño

`ping` de cada tamaño, 40 veces; nos quedamos con el **mínimo**, que descarta colas y ruido.

```
python3 experiments.py size
```

```
RTT mínimo en ms, según los bytes de datos de ping

caso                                  64     400     800    1200    1472   R estimada
un enlace a 5 Mbps                  1.56    1.88    3.36    5.61    5.67      4.8 Mbps
tres enlaces a 5 Mbps               5.63    8.88   11.66   16.84   18.05      5.3 Mbps
un enlace a 5 Mbps y cable de 21 ms 44.89   45.15   46.87   48.54   49.80      4.4 Mbps
```

- **Crece en línea recta:** $\mathrm{RTT}_0 + 2(S + 42) \cdot 8/R$. De 64 a 1472 bytes: teoría 4,5 ms, medido 4,1. La pendiente mide R.

El experimento: RTT según el tamaño

`ping` de cada tamaño, 40 veces; nos quedamos con el **mínimo**, que descarta colas y ruido.

```
python3 experiments.py size

RTT mínimo en ms, según los bytes de datos de ping

caso                                  64     400     800    1200    1472   R estimada
un enlace a 5 Mbps                  1.56    1.88    3.36    5.61    5.67      4.8 Mbps
tres enlaces a 5 Mbps               5.63    8.88   11.66   16.84   18.05      5.3 Mbps
un enlace a 5 Mbps y cable de 21 ms 44.89   45.15   46.87   48.54   49.80      4.4 Mbps
```

- **Crece en línea recta:** $\mathrm{RTT}_0 + 2(S+42)\cdot 8/R$. De 64 a 1472 bytes: teoría 4,5 ms, medido 4,1. La pendiente mide R.
- **Tres enlaces, tres veces la pendiente:** 12,4 ms. Es store-and-forward, medido.

El experimento: RTT según el tamaño

`ping` de cada tamaño, 40 veces; nos quedamos con el **mínimo**, que descarta colas y ruido.

```
python3 experiments.py size

RTT mínimo en ms, según los bytes de datos de ping

caso                                  64     400     800    1200    1472   R estimada
un enlace a 5 Mbps                  1.56    1.88    3.36    5.61    5.67      4.8 Mbps
tres enlaces a 5 Mbps               5.63    8.88   11.66   16.84   18.05      5.3 Mbps
un enlace a 5 Mbps y cable de 21 ms 44.89   45.15   46.87   48.54   49.80      4.4 Mbps
```

- **Crece en línea recta:** $\mathrm{RTT}_0 + 2(S + 42) \cdot 8/R$. De 64 a 1472 bytes: teoría 4,5 ms, medido 4,1. La pendiente mide R.
- **Tres enlaces, tres veces la pendiente:** 12,4 ms. Es store-and-forward, medido.
- **El cable suma lo mismo a todos:** unos 43 ms. La propagación no depende del tamaño.

El experimento: RTT según el tamaño

`ping` de cada tamaño, 40 veces; nos quedamos con el **mínimo**, que descarta colas y ruido.

```
python3 experiments.py size

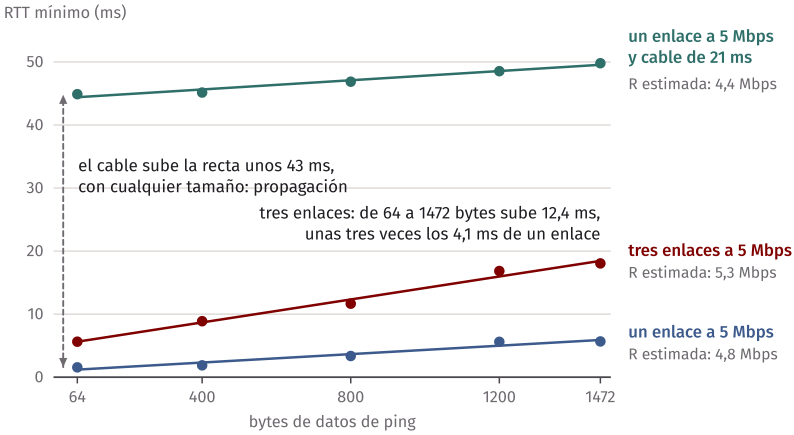
RTT mínimo en ms, según los bytes de datos de ping

caso                                  64     400     800    1200    1472   R estimada
un enlace a 5 Mbps                  1.56    1.88    3.36    5.61    5.67      4.8 Mbps
tres enlaces a 5 Mbps               5.63    8.88   11.66   16.84   18.05      5.3 Mbps
un enlace a 5 Mbps y cable de 21 ms 44.89   45.15   46.87   48.54   49.80      4.4 Mbps
```

- **Crece en línea recta:** $\mathrm{RTT}_0 + 2(S + 42) \cdot 8/R$. De 64 a 1472 bytes: teoría 4,5 ms, medido 4,1. La pendiente mide R.
- **Tres enlaces, tres veces la pendiente:** 12,4 ms. Es store-and-forward, medido.
- **El cable suma lo mismo a todos:** unos 43 ms. La propagación no depende del tamaño.

Las cifras dependen de la máquina. RTT_0 no es cero porque `tc` también agrega algo: comparamos diferencias.

Pendiente y altura: transmisión y propagación



SECCIÓN 07

Colas y pérdida

07

Intensidad de tráfico

Intensidad de tráfico

Con paquetes de L bits, a paquetes por segundo y un enlace de R bits/s:

$$\rho = \frac{L\,a}{R}$$

Intensidad de tráfico

Intensidad de tráfico

Con paquetes de L bits, a paquetes por segundo y un enlace de R bits/s:

$$\rho = \frac{L\,a}{R}$$

- $\rho > 1$: llegan más bits de los que salen. La cola crece sin límite. Ningún buffer lo arregla.

Intensidad de tráfico

Intensidad de tráfico

Con paquetes de L bits, a paquetes por segundo y un enlace de R bits/s:

$$\rho = \frac{L\,a}{R}$$

- $\rho > 1$: llegan más bits de los que salen. La cola crece sin límite. Ningún buffer lo arregla.
- $\rho \leq 1$: la cola depende de **cómo** llegan, no solo de cuántos.

Cómo llegan importa

Simulador de una cola, con el tiempo en unidades de L/R (`queue_sim.c`).

Periódico, un paquete cada $1/\rho$: `./queue_sim periodic 0.99` da retardo 0,00. Sin cola, al 99 %.

Cómo llegan importa

Simulador de una cola, con el tiempo en unidades de L/R (`queue_sim.c`).

Periódico, un paquete cada $1/\rho$: `./queue_sim periodic 0.99` da retardo 0,00. Sin cola, al 99 %.

ráfagas de 10: esperan 0, 1, ..., 9

```
$ for r in 0.10 0.50 0.90; do ./queue_sim burst:10 $r; done
rho=0.10  retardo_medio=      4.50  retardo_max=      9.00  perdida=  0.0%
rho=0.50  retardo_medio=      4.50  retardo_max=      9.00  perdida=  0.0%
rho=0.90  retardo_medio=      4.50  retardo_max=      9.00  perdida=  0.0%
```

Cómo llegan importa

Simulador de una cola, con el tiempo en unidades de L/R (`queue_sim.c`).

Periódico, un paquete cada $1/\rho$: `./queue_sim periodic 0.99` da retardo 0,00. Sin cola, al 99 %.

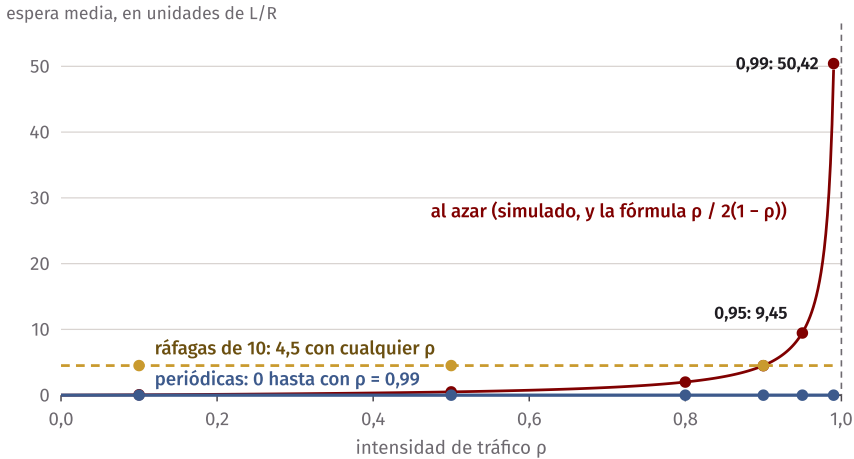
ráfagas de 10: esperan 0, 1, ..., 9

```
$ for r in 0.10 0.50 0.90; do ./queue_sim burst:10 $r; done
rho=0.10  retardo_medio=      4.50  retardo_max=      9.00  perdida=  0.0%
rho=0.50  retardo_medio=      4.50  retardo_max=      9.00  perdida=  0.0%
rho=0.90  retardo_medio=      4.50  retardo_max=      9.00  perdida=  0.0%
```

Error común

Creer que con $\rho < 1$ no hay cola. Solo garantiza que no crezca sin límite. Con ráfagas hay cola incluso en un enlace casi vacío.

Cómo llegan importa: la curva que hay que recordar



Llegadas al azar: simulación y fórmula

ρ	**simulado**	$\frac{\rho}{2(1-\rho)}$
0,10	0,06	0,06
0,50	0,50	0,50
0,80	1,99	2,00
0,90	4,48	4,50
0,95	9,45	9,50
0,99	50,42	49,50

Fórmula de la cola M/D/1, usada sin demostrar.

De 0,95 a 0,99: 4 % más de tráfico, más de 5 veces el retardo.

Si $\rho > 1$

Sin límite de buffer: no converge.

poisson, rho = 1.2

```
$ for n in 1000 10000 100000 1000000; do ...
rho=1.20  retardo_medio=     56.87 ...
rho=1.20  retardo_medio=    805.84 ...
rho=1.20  retardo_medio=   8339.84 ...
rho=1.20  retardo_medio=  82595.93 ...
```

Con buffer de 100: retardo acotado y pérdida.

poisson, buffer 100

```
$ for r in 0.50 0.90 1.00 1.20 2.00; do ...
rho=0.50  medio=   0.50  perdida=  0.0%
rho=0.90  medio=   4.48  perdida=  0.0%
rho=1.00  medio=  49.12  perdida=  0.5%
rho=1.20  medio=  96.34  perdida= 16.7%
rho=2.00  medio=  98.37  perdida= 50.0%
```

Diez veces más paquetes, diez veces más retardo.

Dos predicciones exactas

Se pierde una fracción $1 - 1/\rho$: con 1,2, un 16,7 %. El retardo no pasa de $B \cdot L/R$. Un buffer enorme cambia pérdida por retardo: bufferbloat, en la semana 11.

Ahora con paquetes de verdad

```
python3 experiments.py queue

Enlace r1-r2 a 50 Mbps, buffer de 200 paquetes, UDP de h1 a h2

carga                 rho   RTT prom.   RTT max.   pérdida UDP
sin carga            0.00     2.04 ms    2.23 ms             -
regular              0.50     1.58 ms    2.56 ms          0.0%
regular              0.80     2.04 ms    5.25 ms          0.0%
regular              0.95     2.18 ms    5.52 ms          0.0%
regular              1.20    46.59 ms   47.05 ms         16.1%
ráfagas de 100       0.50     9.40 ms   23.79 ms          0.0%
```

Ahora con paquetes de verdad

```
python3 experiments.py queue

Enlace r1-r2 a 50 Mbps, buffer de 200 paquetes, UDP de h1 a h2

carga                 rho   RTT prom.   RTT max.   pérdida UDP
sin carga            0.00     2.04 ms    2.23 ms             -
regular              0.50     1.58 ms    2.56 ms          0.0%
regular              0.80     2.04 ms    5.25 ms          0.0%
regular              0.95     2.18 ms    5.52 ms          0.0%
regular              1.20    46.59 ms   47.05 ms         16.1%
ráfagas de 100       0.50     9.40 ms   23.79 ms          0.0%
```

- `iperf3` envía casi periódicamente: con 0,95 casi no hay cola.

Ahora con paquetes de verdad

```
python3 experiments.py queue

Enlace r1-r2 a 50 Mbps, buffer de 200 paquetes, UDP de h1 a h2

carga                  rho   RTT prom.   RTT max.   pérdida UDP
sin carga             0.00     2.04 ms    2.23 ms             -
regular               0.50     1.58 ms    2.56 ms          0.0%
regular               0.80     2.04 ms    5.25 ms          0.0%
regular               0.95     2.18 ms    5.52 ms          0.0%
regular               1.20    46.59 ms   47.05 ms         16.1%
ráfagas de 100        0.50     9.40 ms   23.79 ms          0.0%
```

- `iperf3` envía casi periódicamente: con 0,95 casi no hay cola.
- Con 1,2, buffer lleno: $200 \cdot 1442 \cdot 8/50$ Mbps $= 46{,}1$ ms; se midieron 46,6. Pérdida predicha: 16,7 %.

Ahora con paquetes de verdad

```
python3 experiments.py queue

Enlace r1-r2 a 50 Mbps, buffer de 200 paquetes, UDP de h1 a h2

carga                  rho   RTT prom.   RTT max.   pérdida UDP
sin carga             0.00     2.04 ms    2.23 ms             -
regular               0.50     1.58 ms    2.56 ms          0.0%
regular               0.80     2.04 ms    5.25 ms          0.0%
regular               0.95     2.18 ms    5.52 ms          0.0%
regular               1.20    46.59 ms   47.05 ms         16.1%
ráfagas de 100        0.50     9.40 ms   23.79 ms          0.0%
```

- `iperf3` envía casi periódicamente: con 0,95 casi no hay cola.
- Con 1,2, buffer lleno: $200 \cdot 1442 \cdot 8/50$ Mbps $= 46{,}1$ ms; se midieron 46,6. Pérdida predicha: 16,7 %.
- Ráfagas con 0,5: $100 \cdot 1442 \cdot 8/50$ Mbps $= 23{,}1$ ms. El peor RTT fue 23,8.

Ahora con paquetes de verdad

```
python3 experiments.py queue

Enlace r1-r2 a 50 Mbps, buffer de 200 paquetes, UDP de h1 a h2

carga                  rho   RTT prom.   RTT max.   pérdida UDP
sin carga             0.00     2.04 ms    2.23 ms             -
regular               0.50     1.58 ms    2.56 ms          0.0%
regular               0.80     2.04 ms    5.25 ms          0.0%
regular               0.95     2.18 ms    5.52 ms          0.0%
regular               1.20    46.59 ms   47.05 ms         16.1%
ráfagas de 100        0.50     9.40 ms   23.79 ms          0.0%
```

- `iperf3` envía casi periódicamente: con 0,95 casi no hay cola.
- Con 1,2, buffer lleno: $200 \cdot 1442 \cdot 8/50$ Mbps $= 46{,}1$ ms; se midieron 46,6. Pérdida predicha: 16,7 %.
- Ráfagas con 0,5: $100 \cdot 1442 \cdot 8/50$ Mbps $= 23{,}1$ ms. El peor RTT fue 23,8.

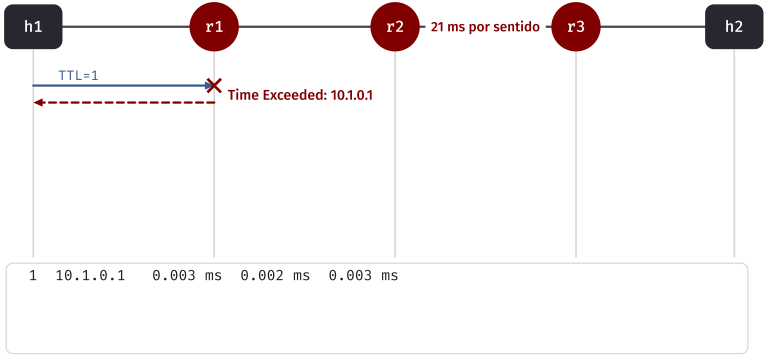
En la práctica: por esto una videollamada se corta si alguien inicia una descarga grande. Un milisegundo de diferencia es ruido.

SECCIÓN 08

traceroute

08

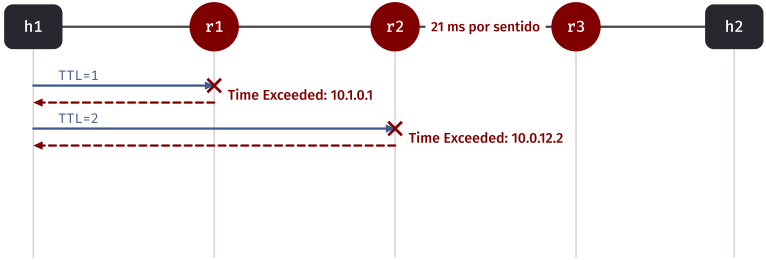
Cómo funciona `traceroute`



TTL = 1. r1 resta 1, llega a 0, descarta el paquete y avisa con ICMP. Responde 10.1.0.1, la interfaz que mira hacia h1.

Cada router resta 1 al **TTL**. Si llega a 0, descarta el paquete y avisa con ICMP. El tiempo hasta esa respuesta es el RTT a ese salto.

Cómo funciona `traceroute`

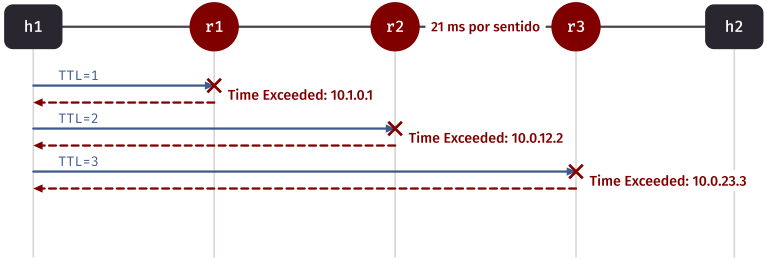



```
1  10.1.0.1   0.003 ms  0.002 ms  0.003 ms
2  10.0.12.2  0.001 ms  0.002 ms  0.002 ms
```

TTL = 2. r1 lo deja en 1 y lo reenvía; r2 lo deja en 0. r2 responde como 10.0.12.2, no como 10.0.23.2.

Cada router resta 1 al **TTL**. Si llega a 0, descarta el paquete y avisa con ICMP. El tiempo hasta esa respuesta es el RTT a ese salto.

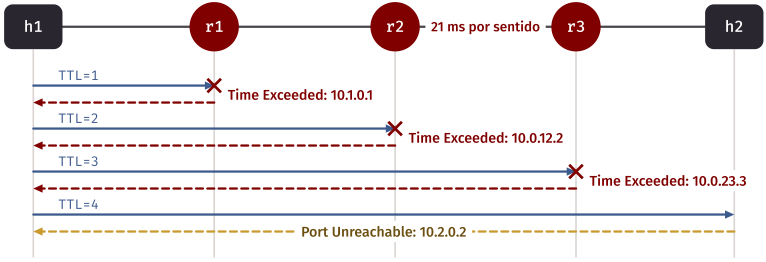
Cómo funciona `traceroute`



TTL = 3. r3 lo descarta, y el RTT sube unos 44 ms: esa respuesta sí cruzó el enlace largo, de ida y de vuelta.

Cada router resta 1 al **TTL**. Si llega a 0, descarta el paquete y avisa con ICMP. El tiempo hasta esa respuesta es el RTT a ese salto.

Cómo funciona `traceroute`




```
1  10.1.0.1   0.003 ms  0.002 ms  0.003 ms
2  10.0.12.2  0.001 ms  0.002 ms  0.002 ms
3  10.0.23.3  44.178 ms  54.962 ms  49.396 ms
4  10.2.0.2   47.938 ms  54.995 ms  58.627 ms
```

TTL = 4. El paquete llega a h2, a un puerto UDP que nadie usa, y h2 responde Port Unreachable: ya llegamos.

Cada router resta 1 al **TTL**. Si llega a 0, descarta el paquete y avisa con ICMP. El tiempo hasta esa respuesta es el RTT a ese salto. Ustedes escriben el suyo en P0.

En la topología: un enlace largo

```
python3 experiments.py cable

Enlace r2-r3 con 21 ms por sentido

traceroute to 10.2.0.2 (10.2.0.2), 30 hops max, 46 byte packets
 1  10.1.0.1  0.003 ms  0.002 ms  0.003 ms
 2  10.0.12.2  0.001 ms  0.002 ms  0.002 ms
 3  10.0.23.3  44.178 ms  54.962 ms  49.396 ms
 4  10.2.0.2  47.938 ms  54.995 ms  58.627 ms
```

- El salto 3 cruza el enlace largo: el RTT sube y ya no baja.
- Responde la interfaz **por donde sale** el error; en esta línea es la misma por donde entró el paquete: `r2` aparece como `10.0.12.2`, no como `10.0.23.2`. Un router tiene una dirección por interfaz.

En Internet: hacia Ciudad del Cabo

en la terminal de la computadora, desde una conexión doméstica en el Perú

```
$ traceroute -q 3 -w 2 -m 14 www.uct.ac.za
traceroute to cms-vip-prd.uct.ac.za (137.158.159.192), 14 hops max, 40 byte packets
 1  192.168.18.1 (192.168.18.1)  3.211 ms  2.693 ms  2.269 ms
 2  10.178.0.1 (10.178.0.1)  4.838 ms  38.222 ms  5.658 ms
 3  10.10.7.38 (10.10.7.38)  5.392 ms  5.528 ms  5.332 ms
 4  10.10.7.37 (10.10.7.37)  5.588 ms  5.875 ms  6.289 ms
 5  45.5.68.102 (45.5.68.102)  6.232 ms  5.754 ms  5.746 ms
 6  8.243.120.5 (8.243.120.5)  10.167 ms  6.497 ms  6.092 ms
 7  * * *
 8  4.68.38.70 (4.68.38.70)  82.491 ms  83.350 ms  82.815 ms
 9  ce-2-0-2.a02.miamfl02.us.ce.gin.ntt.net (129.250.200.114)  82.401 ms  82.054 ms  82.488 ms
10  et-0-0-1-66.rt04.bb.ampath.net (170.39.8.30)  82.700 ms  82.417 ms  82.540 ms
11  187-185-103-190.ampath.net (190.103.185.187)  241.758 ms  241.256 ms  241.834 ms
12  et-0-0-1-0-cpt7-pe1.net.tenet.ac.za (155.232.64.70)  242.202 ms  241.673 ms  241.905 ms
13  154.114.124.1 (154.114.124.1)  242.017 ms  241.578 ms  242.574 ms
14  * * *
```

Cómo leerlo

- **Saltos 1 a 4:** la casa y el ISP, con direcciones privadas.

Cómo leerlo

- **Saltos 1 a 4:** la casa y el ISP, con direcciones privadas.
- **Del 6 al 8, de 6 a 82 ms.** El salto 9 dice `miamfl`. RTT mínimo teórico a Miami: 42 ms. Se suman 76 ms: el cable no va en línea recta. La física da un piso, no una predicción.

Cómo leerlo

- **Saltos 1 a 4:** la casa y el ISP, con direcciones privadas.
- **Del 6 al 8, de 6 a 82 ms.** El salto 9 dice `miamfl`. RTT mínimo teórico a Miami: 42 ms. Se suman 76 ms: el cable no va en línea recta. La física da un piso, no una predicción.
- **Del 10 al 11, de 82 a 241 ms.** Miami–Ciudad del Cabo son unos 12 300 km: mínimo 123 ms. Se suman 159 ms, y se mantienen. Ahí hay un cable submarino.

Cómo leerlo

- **Saltos 1 a 4:** la casa y el ISP, con direcciones privadas.
- **Del 6 al 8, de 6 a 82 ms.** El salto 9 dice `miamfl`. RTT mínimo teórico a Miami: 42 ms. Se suman 76 ms: el cable no va en línea recta. La física da un piso, no una predicción.
- **Del 10 al 11, de 82 a 241 ms.** Miami–Ciudad del Cabo son unos 12 300 km: mínimo 123 ms. Se suman 159 ms, y se mantienen. Ahí hay un cable submarino.
- **Los nombres cuentan el negocio:** `ntt.net` es tránsito global, `ampath.net` un punto de intercambio académico y `tenet.ac.za` la red académica de Sudáfrica.

Cómo leerlo

- **Saltos 1 a 4:** la casa y el ISP, con direcciones privadas.
- **Del 6 al 8, de 6 a 82 ms.** El salto 9 dice `miamfl`. RTT mínimo teórico a Miami: 42 ms. Se suman 76 ms: el cable no va en línea recta. La física da un piso, no una predicción.
- **Del 10 al 11, de 82 a 241 ms.** Miami–Ciudad del Cabo son unos 12 300 km: mínimo 123 ms. Se suman 159 ms, y se mantienen. Ahí hay un cable submarino.
- **Los nombres cuentan el negocio:** `ntt.net` es tránsito global, `ampath.net` un punto de intercambio académico y `tenet.ac.za` la red académica de Sudáfrica.
- **Por Miami:** más de 16 000 km, cuando en línea recta son unos 9800. Lo deciden los contratos y los cables que existen.

La física también pone un techo

hacia www.ox.ac.uk

```
traceroute to www.ox.ac.uk.cdn.cloudflare.net (104.20.34.13), 22 hops max, 40 byte packets
 6  8.243.120.5  7.020 ms  6.027 ms  6.591 ms
 7  200.189.213.218  55.616 ms  55.279 ms  54.944 ms
 8  * 8.243.135.46  56.057 ms  54.746 ms
 9  198.41.228.7  57.195 ms  53.699 ms
    198.41.228.5  56.785 ms
10  104.20.34.13  55.218 ms  53.657 ms  55.314 ms
```

La física también pone un techo

hacia www.ox.ac.uk

```
traceroute to www.ox.ac.uk.cdn.cloudflare.net (104.20.34.13), 22 hops max, 40 byte packets
 6  8.243.120.5  7.020 ms  6.027 ms  6.591 ms
 7  200.189.213.218  55.616 ms  55.279 ms  54.944 ms
 8  * 8.243.135.46  56.057 ms  54.746 ms
 9  198.41.228.7  57.195 ms  53.699 ms
    198.41.228.5  56.785 ms
10  104.20.34.13  55.218 ms  53.657 ms  55.314 ms
```

- Lima–Oxford: unos 10 100 km, así que el RTT no puede bajar de 101 ms. Se midieron 55.

La física también pone un techo

hacia www.ox.ac.uk

```
traceroute to www.ox.ac.uk.cdn.cloudflare.net (104.20.34.13), 22 hops max, 40 byte packets
 6  8.243.120.5  7.020 ms  6.027 ms  6.591 ms
 7  200.189.213.218  55.616 ms  55.279 ms  54.944 ms
 8  * 8.243.135.46  56.057 ms  54.746 ms
 9  198.41.228.7  57.195 ms  53.699 ms
    198.41.228.5  56.785 ms
10  104.20.34.13  55.218 ms  53.657 ms  55.314 ms
```

- Lima–Oxford: unos 10 100 km, así que el RTT no puede bajar de 101 ms. Se midieron 55.
- El servidor que respondió no está en Inglaterra: es una CDN.

La física también pone un techo

hacia www.ox.ac.uk

```
traceroute to www.ox.ac.uk.cdn.cloudflare.net (104.20.34.13), 22 hops max, 40 byte packets
 6  8.243.120.5  7.020 ms  6.027 ms  6.591 ms
 7  200.189.213.218  55.616 ms  55.279 ms  54.944 ms
 8  * 8.243.135.46  56.057 ms  54.746 ms
 9  198.41.228.7  57.195 ms  53.699 ms
    198.41.228.5  56.785 ms
10  104.20.34.13  55.218 ms  53.657 ms  55.314 ms
```

- Lima–Oxford: unos 10 100 km, así que el RTT no puede bajar de 101 ms. Se midieron 55.
- El servidor que respondió no está en Inglaterra: es una CDN.
- Al revés: `www.pucp.edu.pe` resolvió a un nombre en `us-east-1.elb.amazonaws.com`.

¿Puede bajar el RTT en un salto posterior?

hacia www.uni.edu.pe

```
 7  45.183.47.179  5.896 ms  4.633 ms  6.118 ms
 8  10.2.17.222  50.047 ms  39.485 ms  40.203 ms
 9  10.2.18.14  6.389 ms  6.353 ms  6.635 ms
```

¿Puede bajar el RTT en un salto posterior?

```
hacia www.uni.edu.pe

 7  45.183.47.179  5.896 ms  4.633 ms  6.118 ms
 8  10.2.17.222  50.047 ms  39.485 ms  40.203 ms
 9  10.2.18.14  6.389 ms  6.353 ms  6.635 ms
```

- Cada línea es **un viaje distinto**. El paquete del salto 8 obligó a ese router a generar un ICMP.

¿Puede bajar el RTT en un salto posterior?

hacia www.uni.edu.pe

```
7  45.183.47.179  5.896 ms  4.633 ms  6.118 ms
8  10.2.17.222  50.047 ms  39.485 ms  40.203 ms
9  10.2.18.14  6.389 ms  6.353 ms  6.635 ms
```

- Cada línea es **un viaje distinto**. El paquete del salto 8 obligó a ese router a generar un ICMP.
- Reenviar lo hace hardware rápido. Generar errores, un procesador común, con baja prioridad.

¿Puede bajar el RTT en un salto posterior?

```
hacia www.uni.edu.pe

7  45.183.47.179  5.896 ms  4.633 ms  6.118 ms
8  10.2.17.222  50.047 ms  39.485 ms  40.203 ms
9  10.2.18.14  6.389 ms  6.353 ms  6.635 ms
```

- Cada línea es **un viaje distinto**. El paquete del salto 8 obligó a ese router a generar un ICMP.
- Reenviar lo hace hardware rápido. Generar errores, un procesador común, con baja prioridad.
- La respuesta puede volver por otro camino.

¿Puede bajar el RTT en un salto posterior?

```
hacia www.uni.edu.pe

 7  45.183.47.179  5.896 ms  4.633 ms  6.118 ms
 8  10.2.17.222  50.047 ms  39.485 ms  40.203 ms
 9  10.2.18.14  6.389 ms  6.353 ms  6.635 ms
```

- Cada línea es **un viaje distinto**. El paquete del salto 8 obligó a ese router a generar un ICMP.
- Reenviar lo hace hardware rápido. Generar errores, un procesador común, con baja prioridad.
- La respuesta puede volver por otro camino.

Idea clave

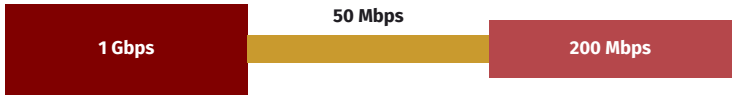
Si el RTT sube y **sigue alto**, hay distancia o cola real. Si vuelve a bajar, es ese router tardando en contestar.

SECCIÓN 09

Throughput

09

El cuello de botella



Throughput y cuello de botella
(bottleneck)

Tasa a la que el receptor recibe los datos. Un camino, sin nadie más: $\min(R_1, \ldots, R_n)$.

bottleneck.c

```
$ ./bottleneck 1000 1000 50 200
cuello de botella: enlace 2, a 50 Mbps
tiempo para 1000 MB: 160.00 s
```

A mano: 8×10^9 bits / 5×10^7 bits/s = 160 s. Cambiar el enlace de 1 Gbps por uno de 10 Gbps no acelera nada.

Medido: no importa dónde esté

```
python3 experiments.py bottleneck

TCP de h1 a h2, 5 s, en el receptor

 r1-r2  r2-r3  r3-h2       medido
    50    200    200    47.6 Mbps
   200     50    200    47.7 Mbps
   200    200     50    45.8 Mbps
   200    200    200   187.9 Mbps
    20    200    200    18.9 Mbps
```

- Lo limita el mínimo, esté donde esté.

Medido: no importa dónde esté

```
python3 experiments.py bottleneck

TCP de h1 a h2, 5 s, en el receptor

 r1-r2  r2-r3  r3-h2       medido
    50    200    200    47.6 Mbps
   200     50    200    47.7 Mbps
   200    200     50    45.8 Mbps
   200    200    200   187.9 Mbps
    20    200    200    18.9 Mbps
```

- Lo limita el mínimo, esté donde esté.
- ¿Por qué 47,6 y no 50? En un frame de 1514 bytes hay 1448 de datos. $\frac{1448}{1514} \times 50 = 47{,}8$ Mbps.

Medido: no importa dónde esté

```
python3 experiments.py bottleneck

TCP de h1 a h2, 5 s, en el receptor

 r1-r2  r2-r3  r3-h2       medido
    50    200    200    47.6 Mbps
   200     50    200    47.7 Mbps
   200    200     50    45.8 Mbps
   200    200    200   187.9 Mbps
    20    200    200    18.9 Mbps
```

- Lo limita el mínimo, esté donde esté.
- ¿Por qué 47,6 y no 50? En un frame de 1514 bytes hay 1448 de datos. $\frac{1448}{1514} \times 50 = 47{,}8$ Mbps.
- Esa tasa de datos útiles es el **goodput**.

Medido: no importa dónde esté

```
python3 experiments.py bottleneck

TCP de h1 a h2, 5 s, en el receptor

 r1-r2  r2-r3  r3-h2       medido
    50    200    200    47.6 Mbps
   200     50    200    47.7 Mbps
   200    200     50    45.8 Mbps
   200    200    200   187.9 Mbps
    20    200    200    18.9 Mbps
```

- Lo limita el mínimo, esté donde esté.
- ¿Por qué 47,6 y no 50? En un frame de 1514 bytes hay 1448 de datos. $\frac{1448}{1514} \times 50 = 47{,}8$ Mbps.
- Esa tasa de datos útiles es el **goodput**.

En un enlace compartido

$\min(R_s, R_c, R/n)$. Cliente de 100 Mbps, servidor de 1 Gbps, troncal de 10 Gbps. Con 50 flujos, el cuello es el acceso (100 Mbps); con 200, se mueve al núcleo (50 Mbps).

“Rápido” significa dos cosas

Tiempo $\approx$ RTT + tamaño / throughput. Con RTT de 42 ms y 50 Mbps:

Qué se envía	Tamaño / throughput	RTT	Domina
1 KB, una respuesta DNS	0,16 ms	42 ms	el retardo
1 MB, una foto	160 ms	42 ms	el throughput
1 GB, una película	160 s	42 ms	el throughput

Error común

Pasar de 100 Mbps a 1 Gbps no cambia casi nada al abrir páginas o en un videojuego en línea. Esos intercambios son muchos y pequeños, y los domina el RTT.

SECCIÓN 10

Capas

10



Cinco capas

Capa	Servicio	Unidad
Aplicación	Lo que el usuario quiere	mensaje
Transporte	Entre **procesos** de dos hosts	segmento
Red	Entre **hosts**, por muchas redes	datagrama
Enlace	Entre **nodos vecinos**	frame
Física	Mover **bits** por el medio	bit

Por qué capas

Cada capa usa el servicio de la de abajo y da uno a la de arriba. Se cambia una sin tocar las otras.

Cinco capas

Capa	Servicio	Unidad
Aplicación	Lo que el usuario quiere	mensaje
Transporte	Entre **procesos** de dos hosts	segmento
Red	Entre **hosts**, por muchas redes	datagrama
Enlace	Entre **nodos vecinos**	frame
Física	Mover **bits** por el medio	bit

Por qué capas

Cada capa usa el servicio de la de abajo y da uno a la de arriba. Se cambia una sin tocar las otras.

Error común

Las capas son responsabilidades, no un programa. Aplicación: su programa. Transporte y red: el sistema operativo. Enlace y física: la tarjeta y su driver.

OSI como vocabulario

OSI	Capa
7	Aplicación
6	Presentación
5	Sesión
4	Transporte
3	Red
2	Enlace de datos
1	Física

OSI como vocabulario

OSI	Capa
7	Aplicación
6	Presentación
5	Sesión
4	Transporte
3	Red
2	Enlace de datos
1	Física

- Internet no lo sigue: TCP/IP ya se había extendido, y no toda aplicación necesita presentación y sesión.

OSI como vocabulario

OSI	Capa
7	Aplicación
6	Presentación
5	Sesión
4	Transporte
3	Red
2	Enlace de datos
1	Física

- Internet no lo sigue: TCP/IP ya se había extendido, y no toda aplicación necesita presentación y sesión.
- Si la aplicación las necesita, las resuelve ella. TLS, por ejemplo, no cae limpio en ninguna capa de OSI.

OSI como vocabulario

OSI	Capa
7	Aplicación
6	Presentación
5	Sesión
4	Transporte
3	Red
2	Enlace de datos
1	Física

- Internet no lo sigue: TCP/IP ya se había extendido, y no toda aplicación necesita presentación y sesión.
- Si la aplicación las necesita, las resuelve ella. TLS, por ejemplo, no cae limpio en ninguna capa de OSI.
- Sobrevive en la forma de hablar: switch de capa 2, problema de capa 3, load balancer de capa 4 o de capa 7.

Encapsulamiento

Aplicación
mensaje

```
GET / HTTP/1.1 …
```

La aplicación entrega 72 bytes: el GET con sus encabezados de HTTP.

Al bajar, cada capa toma lo de arriba **sin mirarlo** y le pone adelante su encabezado *(header)*.

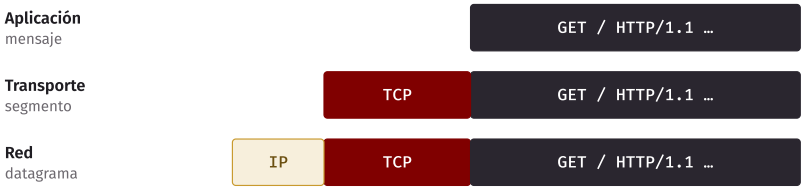
Encapsulamiento



TCP les pone adelante 32 bytes, sin mirar lo que llevan: puertos 50028 y 80.

Al bajar, cada capa toma lo de arriba **sin mirarlo** y le pone adelante su encabezado *(header)*.

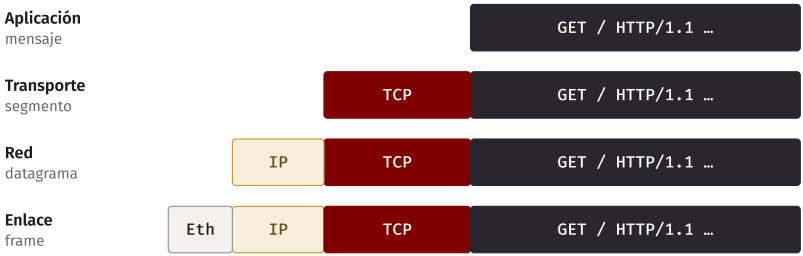
Encapsulamiento



IP pone 20 bytes más: 10.1.0.2 a 10.2.0.2, TTL 64, y el 06 que dice "adentro va TCP".

Al bajar, cada capa toma lo de arriba **sin mirarlo** y le pone adelante su encabezado *(header)*.

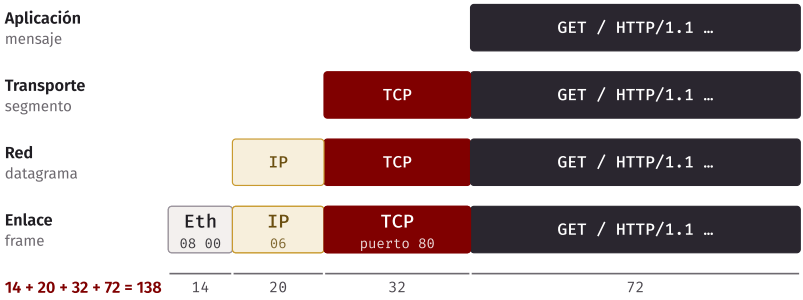
Encapsulamiento



Ethernet pone 14 bytes: la MAC de r1, el siguiente salto, y el 08 00 que dice "adentro va IPv4".

Al bajar, cada capa toma lo de arriba **sin mirarlo** y le pone adelante su encabezado *(header)*.

Encapsulamiento



138 bytes, y 66 son encabezados. Cada uno dice qué viene adentro: 08 00, IPv4; 06, TCP; puerto 80, HTTP.

Al bajar, cada capa toma lo de arriba **sin mirarlo** y le pone adelante su encabezado *(header)*. Al subir, cada capa quita el suyo, y lo usa para decidir qué hacer con lo demás.

Un paquete real, byte por byte

```
tshark -r captures/near-h1.pcap -Y http.request -x

0000  aa c1 ab b6 cc 2d aa c1 ab 1a 08 3d 08 00 45 00   .....-.....=..E.
0010  00 7c 8c ca 40 00 40 06 99 ab 0a 01 00 02 0a 02   .|..@.@.........
0020  00 02 c3 6c 00 50 88 69 7a 1a d1 e7 d6 00 80 18   ...l.P.iz.......
0030  01 f6 65 c8 00 00 01 01 08 0a 2d 86 82 31 59 9a   ..e.......-..1Y.
0040  b1 64 47 45 54 20 2f 20 48 54 54 50 2f 31 2e 31   .dGET / HTTP/1.1
0050  0d 0a 48 6f 73 74 3a 20 31 30 2e 32 2e 30 2e 32   ..Host: 10.2.0.2
0060  0d 0a 55 73 65 72 2d 41 67 65 6e 74 3a 20 63 75   ..User-Agent: cu
0070  72 6c 2f 38 2e 31 34 2e 31 0d 0a 41 63 63 65 70   rl/8.14.1..Accep
0080  74 3a 20 2a 2f 2a 0d 0a 0d 0a                     t: */*....
```

Un paquete real, byte por byte

```
tshark -r captures/near-h1.pcap -Y http.request -x

0000  aa c1 ab b6 cc 2d aa c1 ab 1a 08 3d 08 00 45 00   .....-.....=..E.
0010  00 7c 8c ca 40 00 40 06 99 ab 0a 01 00 02 0a 02   .|..@.@.........
0020  00 02 c3 6c 00 50 88 69 7a 1a d1 e7 d6 00 80 18   ...l.P.iz.......
0030  01 f6 65 c8 00 00 01 01 08 0a 2d 86 82 31 59 9a   ..e.......-..1Y.
0040  b1 64 47 45 54 20 2f 20 48 54 54 50 2f 31 2e 31   .dGET / HTTP/1.1
0050  0d 0a 48 6f 73 74 3a 20 31 30 2e 32 2e 30 2e 32   ..Host: 10.2.0.2
0060  0d 0a 55 73 65 72 2d 41 67 65 6e 74 3a 20 63 75   ..User-Agent: cu
0070  72 6c 2f 38 2e 31 34 2e 31 0d 0a 41 63 63 65 70   rl/8.14.1..Accep
0080  74 3a 20 2a 2f 2a 0d 0a 0d 0a                     t: */*....
```

- **Bytes 0–13, Ethernet:** MAC de `r1`, el siguiente salto, MAC de `h1`, y `08 00`: adentro va IPv4.

Un paquete real, byte por byte

```
tshark -r captures/near-h1.pcap -Y http.request -x

0000  aa c1 ab b6 cc 2d aa c1 ab 1a 08 3d 08 00 45 00   .....-.....=..E.
0010  00 7c 8c ca 40 00 40 06 99 ab 0a 01 00 02 0a 02   .|..@.@.........
0020  00 02 c3 6c 00 50 88 69 7a 1a d1 e7 d6 00 80 18   ...l.P.iz.......
0030  01 f6 65 c8 00 00 01 01 08 0a 2d 86 82 31 59 9a   ..e.......-..1Y.
0040  b1 64 47 45 54 20 2f 20 48 54 54 50 2f 31 2e 31   .dGET / HTTP/1.1
0050  0d 0a 48 6f 73 74 3a 20 31 30 2e 32 2e 30 2e 32   ..Host: 10.2.0.2
0060  0d 0a 55 73 65 72 2d 41 67 65 6e 74 3a 20 63 75   ..User-Agent: cu
0070  72 6c 2f 38 2e 31 34 2e 31 0d 0a 41 63 63 65 70   rl/8.14.1..Accep
0080  74 3a 20 2a 2f 2a 0d 0a 0d 0a                     t: */*....
```

- **Bytes 0–13, Ethernet:** MAC de `r1`, el siguiente salto, MAC de `h1`, y `08 00`: adentro va IPv4.
- **14–33, IP:** `45`: 20 bytes. TTL `40` = 64. `06`: adentro va TCP. Luego, 10.1.0.2 y 10.2.0.2.

Un paquete real, byte por byte

```
tshark -r captures/near-h1.pcap -Y http.request -x

0000  aa c1 ab b6 cc 2d aa c1 ab 1a 08 3d 08 00 45 00   .....-.....=..E.
0010  00 7c 8c ca 40 00 40 06 99 ab 0a 01 00 02 0a 02   .|..@.@.........
0020  00 02 c3 6c 00 50 88 69 7a 1a d1 e7 d6 00 80 18   ...l.P.iz.......
0030  01 f6 65 c8 00 00 01 01 08 0a 2d 86 82 31 59 9a   ..e.......-..1Y.
0040  b1 64 47 45 54 20 2f 20 48 54 54 50 2f 31 2e 31   .dGET / HTTP/1.1
0050  0d 0a 48 6f 73 74 3a 20 31 30 2e 32 2e 30 2e 32   ..Host: 10.2.0.2
0060  0d 0a 55 73 65 72 2d 41 67 65 6e 74 3a 20 63 75   ..User-Agent: cu
0070  72 6c 2f 38 2e 31 34 2e 31 0d 0a 41 63 63 65 70   rl/8.14.1..Accep
0080  74 3a 20 2a 2f 2a 0d 0a 0d 0a                     t: */*....
```

- **Bytes 0–13, Ethernet:** MAC de `r1`, el siguiente salto, MAC de `h1`, y `08 00`: adentro va IPv4.
- **14–33, IP:** `45`: 20 bytes. TTL `40` = 64. `06`: adentro va TCP. Luego, 10.1.0.2 y 10.2.0.2.
- **34–65, TCP:** puertos 50028 (`c3 6c`) y 80 (`00 50`). `80`: 32 bytes.

Un paquete real, byte por byte

```
tshark -r captures/near-h1.pcap -Y http.request -x

0000  aa c1 ab b6 cc 2d aa c1 ab 1a 08 3d 08 00 45 00   .....-.....=..E.
0010  00 7c 8c ca 40 00 40 06 99 ab 0a 01 00 02 0a 02   .|..@.@.........
0020  00 02 c3 6c 00 50 88 69 7a 1a d1 e7 d6 00 80 18   ...l.P.iz.......
0030  01 f6 65 c8 00 00 01 01 08 0a 2d 86 82 31 59 9a   ..e.......-..1Y.
0040  b1 64 47 45 54 20 2f 20 48 54 54 50 2f 31 2e 31   .dGET / HTTP/1.1
0050  0d 0a 48 6f 73 74 3a 20 31 30 2e 32 2e 30 2e 32   ..Host: 10.2.0.2
0060  0d 0a 55 73 65 72 2d 41 67 65 6e 74 3a 20 63 75   ..User-Agent: cu
0070  72 6c 2f 38 2e 31 34 2e 31 0d 0a 41 63 63 65 70   rl/8.14.1..Accep
0080  74 3a 20 2a 2f 2a 0d 0a 0d 0a                     t: */*....
```

- **Bytes 0–13, Ethernet:** MAC de `r1`, el siguiente salto, MAC de `h1`, y `08 00`: adentro va IPv4.
- **14–33, IP:** `45`: 20 bytes. TTL `40` = 64. `06`: adentro va TCP. Luego, 10.1.0.2 y 10.2.0.2.
- **34–65, TCP:** puertos 50028 (`c3 6c`) y 80 (`00 50`). `80`: 32 bytes.
- **66–137, HTTP:** 72 bytes.

Un paquete real, byte por byte

```
tshark -r captures/near-h1.pcap -Y http.request -x

0000  aa c1 ab b6 cc 2d aa c1 ab 1a 08 3d 08 00 45 00   .....-.....=..E.
0010  00 7c 8c ca 40 00 40 06 99 ab 0a 01 00 02 0a 02   .|..@.@.........
0020  00 02 c3 6c 00 50 88 69 7a 1a d1 e7 d6 00 80 18   ...l.P.iz.......
0030  01 f6 65 c8 00 00 01 01 08 0a 2d 86 82 31 59 9a   ..e.......-..1Y.
0040  b1 64 47 45 54 20 2f 20 48 54 54 50 2f 31 2e 31   .dGET / HTTP/1.1
0050  0d 0a 48 6f 73 74 3a 20 31 30 2e 32 2e 30 2e 32   ..Host: 10.2.0.2
0060  0d 0a 55 73 65 72 2d 41 67 65 6e 74 3a 20 63 75   ..User-Agent: cu
0070  72 6c 2f 38 2e 31 34 2e 31 0d 0a 41 63 63 65 70   rl/8.14.1..Accep
0080  74 3a 20 2a 2f 2a 0d 0a 0d 0a                     t: */*....
```

- **Bytes 0–13, Ethernet:** MAC de `r1`, el siguiente salto, MAC de `h1`, y `08 00`: adentro va IPv4.
- **14–33, IP:** `45`: 20 bytes. TTL `40` = 64. `06`: adentro va TCP. Luego, 10.1.0.2 y 10.2.0.2.
- **34–65, TCP:** puertos 50028 (`c3 6c`) y 80 (`00 50`). `80`: 32 bytes.
- **66–137, HTTP:** 72 bytes.
- $14 + 20 + 32 + 72 = 138$. Cada encabezado dice qué viene adentro.

Leerlo desde C

```
./layers captures/near-h1.pcap

1 total=74    enlace=14 ip=20 tcp=40 app=0 ttl=64 10.1.0.2 > 10.2.0.2 [S]
2 total=74    enlace=14 ip=20 tcp=40 app=0 ttl=61 10.2.0.2 > 10.1.0.2 [SA]
3 total=66    enlace=14 ip=20 tcp=32 app=0 ttl=64 10.1.0.2 > 10.2.0.2 [A]
4 total=138   enlace=14 ip=20 tcp=32 app=72 ttl=64 10.1.0.2 > 10.2.0.2 [PA]
5 total=66    enlace=14 ip=20 tcp=32 app=0 ttl=61 10.2.0.2 > 10.1.0.2 [A]
6 total=252   enlace=14 ip=20 tcp=32 app=186 ttl=61 10.2.0.2 > 10.1.0.2 [PA]
7 total=66    enlace=14 ip=20 tcp=32 app=0 ttl=64 10.1.0.2 > 10.2.0.2 [A]
8 total=163   enlace=14 ip=20 tcp=32 app=97 ttl=61 10.2.0.2 > 10.1.0.2 [PA]
```

- Lee cada campo por su posición, sin copiar a una `struct`: una `struct` no garantiza formato ni orden de bytes. Los campos van con el byte más significativo primero: por eso, `htons` en P0.
- Handshake, pedido, respuesta y cierre: 12 paquetes por 97 bytes de página. TCP, en la semana 10.

Qué lee cada equipo

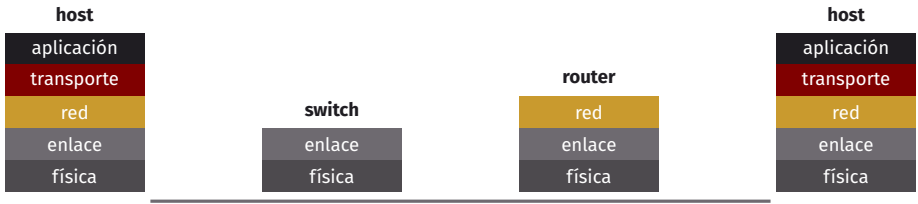


Qué lee cada equipo



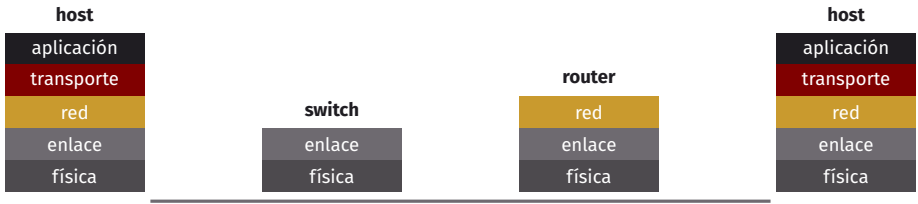
- **Switch:** mira la MAC de destino y elige el puerto. No sabe qué es una dirección IP.

Qué lee cada equipo



- **Switch:** mira la MAC de destino y elige el puerto. No sabe qué es una dirección IP.
- **Router:** abre el frame, mira la IP de destino, busca en su tabla, resta 1 al TTL y arma un frame nuevo. No lee TCP ni HTTP.

Qué lee cada equipo



- **Switch:** mira la MAC de destino y elige el puerto. No sabe qué es una dirección IP.
- **Router:** abre el frame, mira la IP de destino, busca en su tabla, resta 1 al TTL y arma un frame nuevo. No lee TCP ni HTTP.
- **Middleboxes** (NAT, firewalls, load balancers de capa 7): sí leen más arriba, y rompen la limpieza del modelo.

Comprobarlo: el mismo paquete a cada lado

```
python3 experiments.py web
```

campo	junto a h1	junto a h2	
eth.src	aa:c1:ab:1a:08:3d	aa:c1:ab:41:cc:1f	cambia
eth.dst	aa:c1:ab:b6:cc:2d	aa:c1:ab:91:f6:4b	cambia
ip.ttl	64	61	cambia
ip.src	10.1.0.2	10.1.0.2	
ip.dst	10.2.0.2	10.2.0.2	
ip.checksum	0x99ed	0x9ced	cambia
tcp.seq_raw	2288613913	2288613913	
tcp.len	0	0	

Comprobarlo: el mismo paquete a cada lado

```
python3 experiments.py web

campo         junto a h1          junto a h2
eth.src       aa:c1:ab:1a:08:3d   aa:c1:ab:41:cc:1f     cambia
eth.dst       aa:c1:ab:b6:cc:2d   aa:c1:ab:91:f6:4b     cambia
ip.ttl        64                  61                    cambia
ip.src        10.1.0.2            10.1.0.2
ip.dst        10.2.0.2            10.2.0.2
ip.checksum   0x99ed              0x9ced                cambia
tcp.seq_raw   2288613913          2288613913
tcp.len       0                   0
```

- **MAC:** se reemplazan. Valen un salto.

Comprobarlo: el mismo paquete a cada lado

```
python3 experiments.py web

campo         junto a h1          junto a h2
eth.src       aa:c1:ab:1a:08:3d   aa:c1:ab:41:cc:1f     cambia
eth.dst       aa:c1:ab:b6:cc:2d   aa:c1:ab:91:f6:4b     cambia
ip.ttl        64                  61                    cambia
ip.src        10.1.0.2            10.1.0.2
ip.dst        10.2.0.2            10.2.0.2
ip.checksum   0x99ed              0x9ced                cambia
tcp.seq_raw   2288613913          2288613913
tcp.len       0                   0
```

- **MAC:** se reemplazan. Valen un salto.
- **IP:** no cambian (salvo NAT, semana 4). El **TTL** baja en 3, y el **checksum** sube `0x0300`: lo protege.

Comprobarlo: el mismo paquete a cada lado

```
python3 experiments.py web

campo          junto a h1          junto a h2
eth.src        aa:c1:ab:1a:08:3d   aa:c1:ab:41:cc:1f     cambia
eth.dst        aa:c1:ab:b6:cc:2d   aa:c1:ab:91:f6:4b     cambia
ip.ttl         64                  61                    cambia
ip.src         10.1.0.2            10.1.0.2
ip.dst         10.2.0.2            10.2.0.2
ip.checksum    0x99ed              0x9ced                cambia
tcp.seq_raw    2288613913          2288613913
tcp.len        0                   0
```

- **MAC:** se reemplazan. Valen un salto.
- **IP:** no cambian (salvo NAT, semana 4). El **TTL** baja en 3, y el **checksum** sube `0x0300`: lo protege.
- **TCP:** intacto. Ningún router lo tocó.

Lo que verán en su captura

tcpdump -i any, curl http://example.com, ./layers web.pcap

```
1 total=80    enlace=20 ip=20 tcp=40 app=0 ttl=64 192.168.65.3 > 104.20.23.154 [S]
2 total=80    enlace=20 ip=20 tcp=40 app=0 ttl=64 104.20.23.154 > 192.168.65.3 [SA]
3 total=72    enlace=20 ip=20 tcp=32 app=0 ttl=64 192.168.65.3 > 104.20.23.154 [A]
4 total=147   enlace=20 ip=20 tcp=32 app=75 ttl=64 192.168.65.3 > 104.20.23.154 [PA]
```

Lo que verán en su captura

tcpdump -i any, curl http://example.com, ./layers web.pcap

```
1 total=80    enlace=20 ip=20 tcp=40 app=0 ttl=64 192.168.65.3 > 104.20.23.154 [S]
2 total=80    enlace=20 ip=20 tcp=40 app=0 ttl=64 104.20.23.154 > 192.168.65.3 [SA]
3 total=72    enlace=20 ip=20 tcp=32 app=0 ttl=64 192.168.65.3 > 104.20.23.154 [A]
4 total=147   enlace=20 ip=20 tcp=32 app=75 ttl=64 192.168.65.3 > 104.20.23.154 [PA]
```

- **Enlace de 20 bytes:** Linux cooked capture, con `-i any`. Ese encabezado nunca viajó por un cable: lo pone el sistema al capturar.

Lo que verán en su captura

```
tcpdump -i any, curl http://example.com, ./layers web.pcap

1 total=80    enlace=20 ip=20 tcp=40 app=0 ttl=64 192.168.65.3 > 104.20.23.154 [S]
2 total=80    enlace=20 ip=20 tcp=40 app=0 ttl=64 104.20.23.154 > 192.168.65.3 [SA]
3 total=72    enlace=20 ip=20 tcp=32 app=0 ttl=64 192.168.65.3 > 104.20.23.154 [A]
4 total=147   enlace=20 ip=20 tcp=32 app=75 ttl=64 192.168.65.3 > 104.20.23.154 [PA]
```

- **Enlace de 20 bytes:** Linux cooked capture, con `-i any`. Ese encabezado nunca viajó por un cable: lo pone el sistema al capturar.
- **TTL 64 desde el otro lado, y un ACK en 0,1 ms:** indica que el otro extremo de esta conexión es Docker Desktop. Aquí se ven las capas, no el camino.

Lo que verán en su captura

```
tcpdump -i any, curl http://example.com, ./layers web.pcap

1 total=80    enlace=20 ip=20 tcp=40 app=0 ttl=64 192.168.65.3 > 104.20.23.154 [S]
2 total=80    enlace=20 ip=20 tcp=40 app=0 ttl=64 104.20.23.154 > 192.168.65.3 [SA]
3 total=72    enlace=20 ip=20 tcp=32 app=0 ttl=64 192.168.65.3 > 104.20.23.154 [A]
4 total=147   enlace=20 ip=20 tcp=32 app=75 ttl=64 192.168.65.3 > 104.20.23.154 [PA]
```

- **Enlace de 20 bytes:** Linux cooked capture, con `-i any`. Ese encabezado nunca viajó por un cable: lo pone el sistema al capturar.
- **TTL 64 desde el otro lado, y un ACK en 0,1 ms:** indica que el otro extremo de esta conexión es Docker Desktop. Aquí se ven las capas, no el camino.
- **Frames de más de 1500 bytes:** la MTU de esa interfaz es de 65 535. Lo capturado no siempre es lo que pasa por el cable.

Lo que verán en su captura

```
tcpdump -i any, curl http://example.com, ./layers web.pcap

1 total=80    enlace=20 ip=20 tcp=40 app=0 ttl=64 192.168.65.3 > 104.20.23.154 [S]
2 total=80    enlace=20 ip=20 tcp=40 app=0 ttl=64 104.20.23.154 > 192.168.65.3 [SA]
3 total=72    enlace=20 ip=20 tcp=32 app=0 ttl=64 192.168.65.3 > 104.20.23.154 [A]
4 total=147   enlace=20 ip=20 tcp=32 app=75 ttl=64 192.168.65.3 > 104.20.23.154 [PA]
```

- **Enlace de 20 bytes:** Linux cooked capture, con `-i any`. Ese encabezado nunca viajó por un cable: lo pone el sistema al capturar.
- **TTL 64 desde el otro lado, y un ACK en 0,1 ms:** indica que el otro extremo de esta conexión es Docker Desktop. Aquí se ven las capas, no el camino.
- **Frames de más de 1500 bytes:** la MTU de esa interfaz es de 65 535. Lo capturado no siempre es lo que pasa por el cable.
- **Otros contenedores:** `-i any` también los ve. Filtren con `host example.com`.

SECCIÓN 11

Laboratorio

11

Entorno, `traceroute` y captura

1. Instalar Docker y Wireshark.
2. `env/run.sh` y `env/smoke/check.sh`: todas las líneas deben decir `ok`.
3. **Fuera del contenedor:** `traceroute` al Perú y a otro continente. ¿Hay un cable submarino?
4. Capturar una petición web. Abrirla en Wireshark.
5. Anotar el tamaño de cada encabezado.
6. A mano: 1500 bytes, 100 Mbps, Lima–Miami.
7. Leer el enunciado de **P0**.

captura, dentro del contenedor

```
tcpdump -i any -w web.pcap port 80 or port 443 &
sleep 1
curl -s -o /dev/null http://example.com
sleep 2
kill %1
```

Las actividades de captura se basan en los laboratorios de Wireshark de Kurose y Ross.

Para repetir las mediciones

code/week-01

```
make                              # compila los programas
make up                           # levanta la topología
python3 experiments.py cable      # también: size, bottleneck, queue, web
make down                         # destruye la topología y su red
make test                         # todo lo anterior, con pruebas
```

Antes de medir, predigan

Cambien un parámetro en `experiments.py`. Escriban primero lo que esperan, y luego midan. Si no coincide, averigüen qué parte explica el instrumento.

Para la próxima clase

- Leer Kurose y Ross, secs. 1.7 y 4.5, y CS168, *Network Architecture*.
- Leer Saltzer, Reed y Clark, *End-to-End Arguments in System Design*, secciones 1 y 2.
- Dejar el entorno funcionando, y leer el enunciado de **P0**.
- Hacer los ejercicios de los apuntes.

Pregunta

IP no promete entregar nada. ¿Por qué una red con una promesa tan débil le ganó a redes que sí garantizaban la entrega?