


Computación Centrada en Redes

# DNS y HTTP: los nombres y la web por dentro

## Transporte y aplicaciones

**Gerson Garrido**

Semana 12

UNIVERSIDAD NACIONAL DE INGENIERÍA

# Hoy

**Sesión A: DNS**

1. Nombres y direcciones
2. Jerarquía, zonas y resolver
3. Recursiva, iterativa y glue
4. Cache y TTL
5. Registros y el mensaje
6. UDP, TCP y ataques

**Sesión B: HTTP**

1. Petición y respuesta
2. Host y cookies
3. Cuántos RTT cuesta una página
4. Cache, ETag y proxies
5. HTTP/2
6. Lo que HTTP/2 no arregló

**Lectura**

Kurose y Ross, sec. 2.4 y secs. 2.2.1–2.2.6. CS168, capítulos *DNS* y *HTTP*.

# Dónde estamos

- Ya hay transporte: un programa le manda bytes a otro.
- Pero nadie escribe `45.80.81.43` en el navegador.

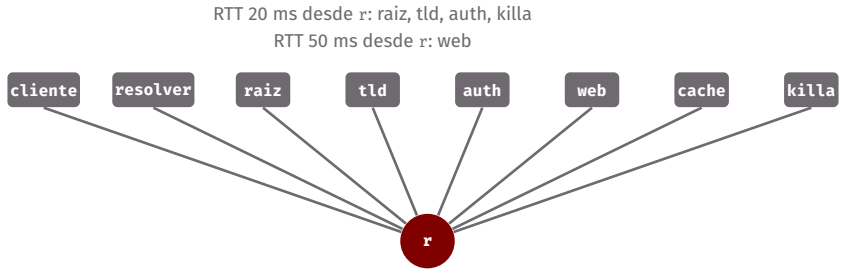
# Dónde estamos

- Ya hay transporte: un programa le manda bytes a otro.
- Pero nadie escribe `45.80.81.43` en el navegador.
- Dos protocolos hacen posible ese gesto:
  - **DNS**: de un nombre a una dirección.
  - **HTTP**: pedir y recibir una página.

# Dónde estamos

- Ya hay transporte: un programa le manda bytes a otro.
- Pero nadie escribe `45.80.81.43` en el navegador.
- Dos protocolos hacen posible ese gesto:
  - **DNS**: de un nombre a una dirección.
  - **HTTP**: pedir y recibir una página.
- Las mejoras de HTTP se entienden contando RTT.

# La red de esta semana




Un Internet de bolsillo: su propia raíz y su TLD `.test`, reservado para pruebas. `make up` la levanta y arranca los servidores.

SECCIÓN 01

# DNS: jerarquía y resolución

01

# Nombre y dirección son cosas distintas

- La **dirección** dice *dónde*: la usan los routers.
- El **nombre** dice *qué*: lo usa la gente.
- Separarlos permite cambiar uno sin tocar el otro.

**DNS**

Base de datos distribuida y jerárquica, y el protocolo para consultarla: dado un nombre y un tipo, devuelve registros.

# Nombre y dirección son cosas distintas

- La **dirección** dice *dónde*: la usan los routers.
- El **nombre** dice *qué*: lo usa la gente.
- Separarlos permite cambiar uno sin tocar el otro.

**DNS**

Base de datos distribuida y jerárquica, y el protocolo para consultarla: dado un nombre y un tipo, devuelve registros.

**Además de traducir:** alias (CNAME), correo (MX), reparto de carga, datos varios (TXT).

# Por qué no un solo servidor

- Al principio: un archivo, `HOSTS.TXT`, que cada máquina copiaba.

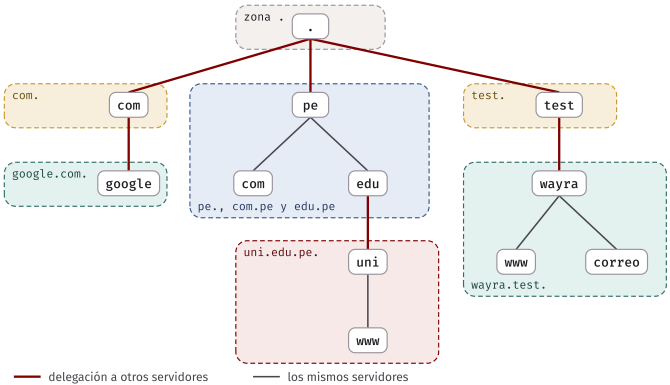
# Por qué no un solo servidor

- Al principio: un archivo, `HOSTS.TXT`, que cada máquina copiaba.
- Centralizar falla por cuatro lados: punto único de falla, todo el tráfico en un lugar, lejos de casi todos, y un solo administrador para los nombres del mundo.

# Por qué no un solo servidor

- Al principio: un archivo, `HOSTS.TXT`, que cada máquina copiaba.
- Centralizar falla por cuatro lados: punto único de falla, todo el tráfico en un lugar, lejos de casi todos, y un solo administrador para los nombres del mundo.
- DNS reparte dos cosas:
  - **Autoridad**: cada organización administra sus nombres.
  - **Carga**: la cache hace que casi ninguna pregunta llegue lejos.

# El árbol de nombres




**Zona**

La parte del árbol que administra una misma autoridad. Puede **delegar** una parte de abajo, que pasa a ser otra zona.

# Tres clases de servidores, y el resolver

| Servidor | De qué responde | Por `www.uni.edu.pe` |
|---|---|---|
| Raíz | La zona `.` | “los de `.pe` son estos” |
| TLD | `pe.`, `com.` | “los de `uni.edu.pe` son estos” |
| Autoritativo | Una organización | “es `45.80.81.43`” |

# Tres clases de servidores, y el resolver

| Servidor | De qué responde | Por `www.uni.edu.pe` |
|---|---|---|
| Raíz | La zona `.` | "los de `.pe` son estos" |
| TLD | `pe.`, `com.` | "los de `uni.edu.pe` son estos" |
| Autoritativo | Una organización | "es `45.80.81.43`" |

**Resolver recursivo**

Recibe preguntas, responde desde su cache si puede, y si no, recorre la jerarquía. No tiene autoridad sobre nada: busca y recuerda.

# Tres clases de servidores, y el resolver

| Servidor | De qué responde | Por `www.uni.edu.pe` |
|---|---|---|
| Raíz | La zona `.` | “los de `.pe` son estos” |
| TLD | `pe.`, `com.` | “los de `uni.edu.pe` son estos” |
| Autoritativo | Una organización | “es `45.80.81.43`” |

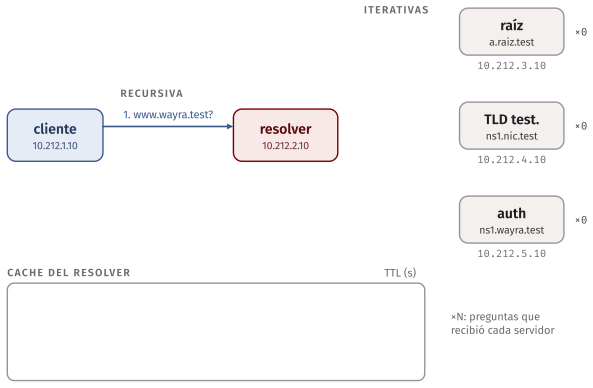
**Resolver recursivo**

Recibe preguntas, responde desde su cache si puede, y si no, recorre la jerarquía. No tiene autoridad sobre nada: busca y recuerda.

**Error común**

La raíz no conoce todos los nombres. Conoce los TLD y a quién preguntarles. Y las 13 letras son direcciones anycast con muchas copias.

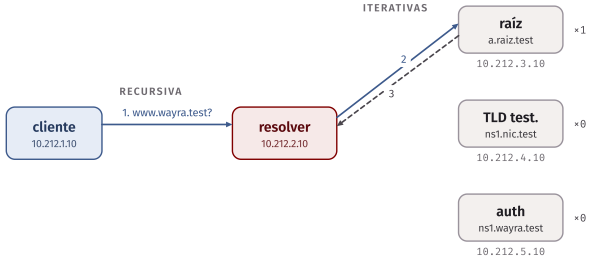
# Recursiva e iterativa




El cliente hace una consulta recursiva, con RD en 1: "tráigame la respuesta final".

**Recursiva** (RD=1): "tráigame la respuesta". **Iterativa**: "dígame lo que sepa". Si no sabe, da una **referral**.

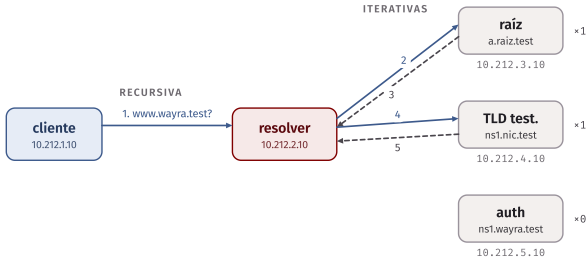
# Recursiva e iterativa




Consulta iterativa a la raíz.
No hay respuesta (ANSWER: 0): manda a ns1.nic.test y regala su dirección, con glue.

**Recursiva** (RD=1): “tráigame la respuesta”. **Iterativa**: “dígame lo que sepa”. Si no sabe, da una **referral**.

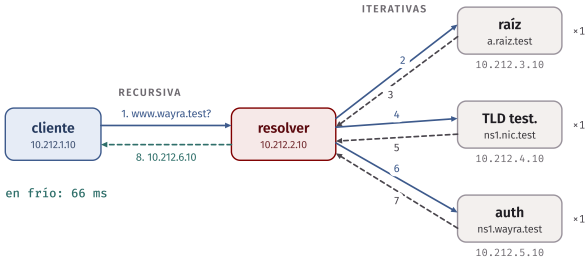
# Recursiva e iterativa




El TLD tampoco tiene la respuesta: manda a ns1.wayra.test, también con glue.
Cada salto cuesta unos 22 ms.

**Recursiva** (RD=1): “tráigame la respuesta”. **Iterativa**: “dígame lo que sepa”. Si no sabe, da una **referral**.

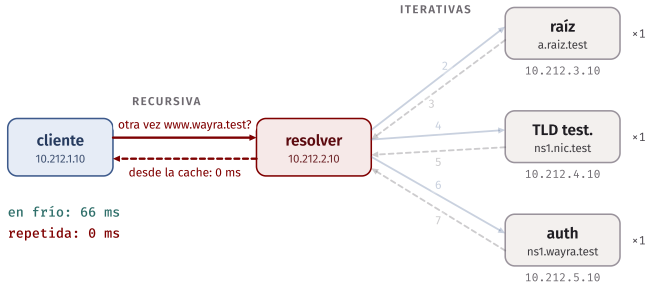
# Recursiva e iterativa




auth responde con el flag aa, y el resolver se lo entrega al cliente.
En frío fueron 66 ms: tres RTT de 20 ms y lo que tarda Python.

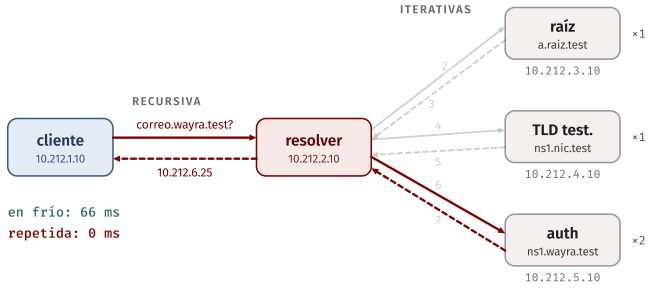
**Recursiva** (RD=1): “tráigame la respuesta”. **Iterativa**: “dígame lo que sepa”. Si no sabe, da una **referral**.

# La cache, en la misma resolución




La misma pregunta, repetida: sale de la cache. 0 ms, y ningún servidor recibe nada.

# La cache, en la misma resolución




correo.wayra.test nunca se preguntó, pero la delegación de wayra.test ya está en cache: va directo a auth.

# `dig +trace` en nuestra red

cliente

```
$ dig +trace www.wayra.test
.  86400  IN  NS  a.raiz.test.
;; Received 41 bytes from 10.212.2.10#53(10.212.2.10) in 44 ms

test.  86400  IN  NS  ns1.nic.test.
;; Received 70 bytes from 10.212.3.10#53(a.raiz.test) in 22 ms

wayra.test.  3600  IN  NS  ns1.wayra.test.
;; Received 66 bytes from 10.212.4.10#53(ns1.nic.test) in 22 ms

www.wayra.test.  300  IN  A  10.212.6.10
;; Received 48 bytes from 10.212.5.10#53(ns1.wayra.test) in 21 ms
```

Cada salto cuesta unos 20 ms de `netem`, más lo que tarda Python.

# A mano: la raíz no responde, deriva

cliente

```
$ dig @10.212.3.10 +norecurse www.wayra.test
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 48816
;; flags: qr; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 1

;; AUTHORITY SECTION:
test.                86400   IN      NS      ns1.nic.test.

;; ADDITIONAL SECTION:
ns1.nic.test.        86400   IN      A       10.212.4.10
```

`ANSWER: 0` sin error. **Authority** dice quién sabe; **additional** regala su dirección.

# El último paso lleva aa

cliente

```
$ dig @10.212.5.10 +norecurse www.wayra.test
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 31625
;; flags: qr aa; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; ANSWER SECTION:
www.wayra.test.       300     IN      A       10.212.6.10
```

aa, *authoritative answer*: lo da solo quien administra la zona. Una referral no lo lleva, ni la respuesta de un resolver.

# Glue: huevo y gallina

**Glue**

La dirección de un servidor de nombres, que la zona padre entrega con la delegación. Indispensable si el servidor está *dentro* de la zona delegada.

sin glue: el padre dice quién, pero no dónde

```
$ dig @10.212.4.10 +norecurse www.killa.test +noall +authority +additional
killa.test.           3600    IN      NS      ns1.killa.test.

$ dig www.killa.test
;; ->>HEADER<<- opcode: QUERY, status: SERVFAIL, id: 53848
```

`NXDOMAIN`: el nombre no existe. `SERVFAIL`: el resolver no pudo averiguarlo.

# En el Internet real

dig +trace +nodnssec www.uni.edu.pe, 2 de noviembre de 2026

```
.                   4502    IN      NS      e.root-servers.net.
                    (... las otras doce letras ...)
;; Received 420 bytes from 192.168.65.7#53(192.168.65.7) in 103 ms

pe.                 172800  IN      NS      a.lactld.org.
pe.                 172800  IN      NS      pch.rcp.pe.
pe.                 172800  IN      NS      pe1.dnsnode.net.
pe.                 172800  IN      NS      quipu.rcp.net.pe.
;; Received 296 bytes from 199.7.83.42#53(l.root-servers.net) in 70 ms

uni.edu.pe.         3600    IN      NS      rngcyt.uni.edu.pe.
uni.edu.pe.         3600    IN      NS      rpcyt.uni.edu.pe.
uni.edu.pe.         3600    IN      NS      dns.uni.edu.pe.
;; Received 178 bytes from 204.61.216.85#53(pch.rcp.pe) in 6 ms

www.uni.edu.pe.     7200    IN      A       45.80.81.43
;; Received 59 bytes from 45.80.81.12#53(rngcyt.uni.edu.pe) in 8 ms
```

`.pe` delega `uni.edu.pe` directamente: una zona no se corta en cada punto.

SECCIÓN 02

# Cache y TTL

02



# Cache y TTL

**TTL de un registro**

Segundos que un resolver puede guardarlo en cache. Lo decide quien administra la zona.

| En frío | Repetida |
|---|---|
| 66 ms | 0 ms |
| 66 ms | 0 ms |
| 71 ms | 1 ms |
| 66 ms | 0 ms |

Tres RTT de 20 ms dan 60 ms de piso. También se guardan las **delegaciones**: un nombre nuevo de la misma zona ya no molesta a la raíz.

# Cache y TTL

**TTL de un registro**

Segundos que un resolver puede guardarlo en cache. Lo decide quien administra la zona.

| En frío | Repetida |
|---|---|
| 66 ms | 0 ms |
| 66 ms | 0 ms |
| 71 ms | 1 ms |
| 66 ms | 0 ms |

Tres RTT de 20 ms dan 60 ms de piso. También se guardan las **delegaciones**: un nombre nuevo de la misma zona ya no molesta a la raíz. No es el TTL de IP: ese cuenta saltos, y este cuenta segundos.

# Cómo baja el TTL

TTL de 4 segundos en la zona

```
$ dig +noall +answer rapido.wayra.test
rapido.wayra.test.    4       IN      A       10.212.6.11
$ sleep 2; dig +noall +answer rapido.wayra.test
rapido.wayra.test.    2       IN      A       10.212.6.11
$ sleep 3; dig +noall +answer rapido.wayra.test
rapido.wayra.test.    4       IN      A       10.212.6.11
```

El log de `auth` muestra dos preguntas: la primera y la de después de vencer.

> **En un resolver público**
>
> `8.8.8.8` dio `7123` y 3 s después `2860`: una dirección anycast tiene muchas caches detrás.

# Elegir el TTL

| TTL | Ventajas | Riesgos |
|---|---|---|
| Corto | Un cambio se nota rápido; reparto de tráfico | Más carga y más lentitud. Si el autoritativo cae, deja de resolver pronto |
| Largo | Menos carga. Aguanta una caída del autoritativo | Un error tarda en corregirse: sigue en caches del mundo |

# Elegir el TTL

| TTL | Ventajas | Riesgos |
|---|---|---|
| Corto | Un cambio se nota rápido; reparto de tráfico | Más carga y más lentitud. Si el autoritativo cae, deja de resolver pronto |
| Largo | Menos carga. Aguanta una caída del autoritativo | Un error tarda en corregirse: sigue en caches del mundo |

**Error común**

"Propagación de DNS". Nada viaja: cada resolver usa su copia hasta que le vence. Por eso se baja el TTL *unos días antes* de una mudanza.

SECCIÓN 03

# Registros y mensajes

03

# Registros

| Tipo | Datos | Para qué |
|---|---|---|
| A | Dirección IPv4 | Traducir un nombre |
| AAAA | Dirección IPv6 | Lo mismo, en IPv6 |
| NS | Nombre de servidor | Quién responde por la zona |
| CNAME | Otro nombre | Este nombre es un alias |
| MX | Preferencia y servidor | A quién entregarle el correo |
| TXT | Texto | Datos para otros sistemas |
| SOA | Datos de la zona | Autoridad, y cache negativa |

# Registros

| Tipo | Datos | Para qué |
|---|---|---|
| A | Dirección IPv4 | Traducir un nombre |
| AAAA | Dirección IPv6 | Lo mismo, en IPv6 |
| NS | Nombre de servidor | Quién responde por la zona |
| CNAME | Otro nombre | Este nombre es un alias |
| MX | Preferencia y servidor | A quién entregarle el correo |
| TXT | Texto | Datos para otros sistemas |
| SOA | Datos de la zona | Autoridad, y cache negativa |

**Error común**

El MX da el *nombre* de un servidor, que necesita su propio A o AAAA. Menor preferencia significa más preferido.

# Registros en acción

cliente

```
$ dig +noall +answer wayra.test MX
wayra.test.           3600    IN      MX      10 correo.wayra.test.
$ dig +noall +answer blog.wayra.test
blog.wayra.test.      300     IN      CNAME   www.wayra.test.
www.wayra.test.       300     IN      A       10.212.6.10
$ dig noexiste.wayra.test
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 40207
wayra.test. 60 IN SOA ns1.wayra.test. admin.wayra.test. 2026110201 3600 600 86400 60
```

Con un CNAME, el resolver devuelve el alias y lo que encontró. Lo negativo también se guarda: el SOA dice cuánto.

# El mensaje: 12 bytes y cuatro secciones

| ID | flags |
|---|---|
| preguntas | respuestas |
| autoridad | adicionales |
| pregunta: nombre tipo clase | |
| respuestas: registros | |
| autoridad: registros | |
| adicionales: registros | |

El mismo formato para preguntar y para responder. `dns_ask.c` lo arma a mano, campo por campo y en orden de red.

# Byte por byte

respuesta, desde el fin del encabezado UDP

```
59 55 81 80 00 01 00 01 00 00 00 00
03 77 77 77 05 77 61 79 72 61 04 74 65 73 74 00 00 01 00 01
c0 0c 00 01 00 01 00 00 01 2b 00 04 0a d4 06 0a
```

- `59 55`: el ID, 22869. `81 80`: QR, RD y RA.
- `03 www 05 wayra 04 test 00`: largo y etiqueta, no puntos.

# Byte por byte

respuesta, desde el fin del encabezado UDP

```
59 55 81 80 00 01 00 01 00 00 00 00
03 77 77 77 05 77 61 79 72 61 04 74 65 73 74 00 00 01 00 01
c0 0c 00 01 00 01 00 00 01 2b 00 04 0a d4 06 0a
```

- `59 55`: el ID, 22869. `81 80`: QR, RD y RA.
- `03 www 05 wayra 04 test 00`: largo y etiqueta, no puntos.
- `c0 0c`: **compresión**. Dos bits en 1 son un puntero a la posición 12, donde ya estaba el nombre.
- `00 00 01 2b`: TTL 299. `0a d4 06 0a`: 10.212.6.10.

# Byte por byte

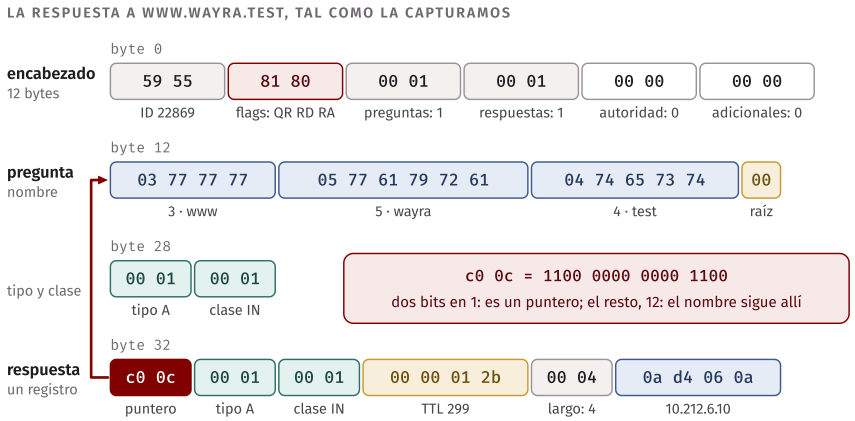
respuesta, desde el fin del encabezado UDP

```
59 55 81 80 00 01 00 01 00 00 00 00
03 77 77 77 05 77 61 79 72 61 04 74 65 73 74 00 00 01 00 01
c0 0c 00 01 00 01 00 00 01 2b 00 04 0a d4 06 0a
```

- `59 55`: el ID, 22869. `81 80`: QR, RD y RA.
- `03 www 05 wayra 04 test 00`: largo y etiqueta, no puntos.
- `c0 0c`: **compresión**. Dos bits en 1 son un puntero a la posición 12, donde ya estaba el nombre.
- `00 00 01 2b`: TTL 299. `0a d4 06 0a`: 10.212.6.10.

Un puntero a sí mismo cuelga a un programa descuidado: solo se aceptan hacia atrás.

# La misma respuesta, campo por campo




Dos bytes en vez de dieciséis. Las secciones de autoridad y adicionales van vacías: sus contadores están en 0.

# Por qué UDP, y cuándo TCP

- **UDP**: una pregunta y una respuesta chicas. Repetir es inofensivo.
- **Límite**: 512 bytes en el RFC 1035. Si no cabe, flag `TC`, y a repetir por TCP.
- Por TCP, cada mensaje lleva su **largo** delante: un stream no tiene límites.

cliente

```
$ dig grande.wayra.test TXT
;; Truncated, retrying in TCP mode.
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 46062
;; SERVER: 10.212.2.10#53(10.212.2.10) (TCP)
;; MSG SIZE  rcvd: 1774
```

**EDNS** permite anunciar respuestas más grandes por UDP. Bloquear TCP/53 es un error.

SECCIÓN 04

# Ataques a DNS

04

# Envenenamiento de cache

**Envenenamiento**

El resolver acepta y guarda una respuesta falsa: todos sus clientes reciben la dirección equivocada hasta que venza el TTL.

# Envenenamiento de cache

**Envenenamiento**

El resolver acepta y guarda una respuesta falsa: todos sus clientes reciben la dirección equivocada hasta que venza el TTL.

- Una respuesta no trae prueba de autoría, y UDP no tiene handshake.
- Es una carrera: la respuesta falsa debe llegar antes que la verdadera.

# Envenenamiento de cache

**Envenenamiento**

El resolver acepta y guarda una respuesta falsa: todos sus clientes reciben la dirección equivocada hasta que venza el TTL.

- Una respuesta no trae prueba de autoría, y UDP no tiene handshake.
- Es una carrera: la respuesta falsa debe llegar antes que la verdadera.
- El resolver solo acepta si coinciden servidor, **ID**, **puerto** y **pregunta**.
- Lo que protege es lo que un atacante fuera del camino no puede ver.

# Cuánto aporta cada campo

| Adivinar | Combinaciones | Con 1000 intentos |
|---|---|---|
| Solo el ID (16 bits) | 65 536 | 1,5 % |
| ID y puerto (32768–60999) | 1 850 212 352 | 0,000054 % |

# Cuánto aporta cada campo

| Adivinar | Combinaciones | Con 1000 intentos |
|---|---|---|
| Solo el ID (16 bits) | 65 536 | 1,5 % |
| ID y puerto (32768–60999) | 1 850 212 352 | 0,000054 % |

- **Bailiwick**, defensa antigua: solo se guardan direcciones de nombres dentro de la zona delegada.
- Kaminsky, 2008: nombres inventados de un mismo dominio nunca están en cache, y la delegación falsa respeta el bailiwick. El TTL ya no limita los intentos: de ahí el puerto al azar.

# Cuánto aporta cada campo

| Adivinar | Combinaciones | Con 1000 intentos |
|---|---|---|
| Solo el ID (16 bits) | 65 536 | 1,5 % |
| ID y puerto (32768–60999) | 1 850 212 352 | 0,000054 % |

- **Bailiwick**, defensa antigua: solo se guardan direcciones de nombres dentro de la zona delegada.
- Kaminsky, 2008: nombres inventados de un mismo dominio nunca están en cache, y la delegación falsa respeta el bailiwick. El TTL ya no limita los intentos: de ahí el puerto al azar.

**Idea clave**

El azar no autentica. Solo dificulta adivinar, y solo a quien *no* ve el tráfico.

# DNSSEC autentica

Firmas (RRSIG), claves (DNSKEY) y una cadena (DS) hasta la raíz.

del contenedor del curso

```
$ dig @1.1.1.1 +dnssec +noall +comments pe. SOA | grep flags
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1
$ delv @1.1.1.1 pe. SOA
; fully validated
$ delv @1.1.1.1 uni.edu.pe. A
; unsigned answer
```

# DNSSEC autentica

Firmas (RRSIG), claves (DNSKEY) y una cadena (DS) hasta la raíz.

del contenedor del curso

```
$ dig @1.1.1.1 +dnssec +noall +comments pe. SOA | grep flags
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1
$ delv @1.1.1.1 pe. SOA
; fully validated
$ delv @1.1.1.1 uni.edu.pe. A
; unsigned answer
```

Error común: DNSSEC **no** da privacidad. La consulta firmada se lee igual:

tcpdump

```
eth0  Out IP 192.168.65.3.61050 > 1.1.1.1.53: 29822+ [1au] SOA? pe. (43)
eth0  In  IP 1.1.1.1.53 > 192.168.65.3.61050: 29822$ 2/0/1 SOA, RRSIG (384)
```

# DoH y DoT cifran

| | Autentica | Cifra | Protege de |
|---|---|---|---|
| ID y puerto | No | No | Atacante fuera del camino |
| DNSSEC | Sí, de extremo a extremo | No | Respuestas falsas |
| DoT, DoH | Solo el canal | Hasta el resolver | Quien observa la red local |

- Por DoH, los 22 paquetes capturados fueron al puerto 443: ninguno al 53, y sin el nombre.
- El resolver sigue viendo todo: con DoH se elige en quién confiar.
- Son complementarios.

SECCIÓN 05

# HTTP: petición y respuesta

05

# HTTP/1.1 es texto

cliente

```
$ printf 'GET / HTTP/1.1\r\nHost: www.wayra.test\r\n\r\n' | nc -w1 10.212.6.10 80
HTTP/1.1 200 OK
Server: mini_web
ETag: "d8a8038111f4"
Cache-Control: max-age=60
Content-Type: text/html
Content-Length: 67

<html><img src=/a.png><img src=/b.png><img src=/c.png>wayra</html>
```

Línea inicial, headers, línea vacía y cuerpo. `Content-Length` marca el fin: TCP es un stream, y no hay otros límites.

# Métodos y códigos

| | Seguro | Idempot. |
|---|---|---|
| `GET` | sí | sí |
| `HEAD` | sí | sí |
| `POST` | no | no |
| `PUT` | no | sí |
| `DELETE` | no | sí |

| | |
|---|---|
| `2xx` | Salió bien: `200` |
| `3xx` | Algo más: `301`, `304` |
| `4xx` | Error de quien pidió: `400`, `404` |
| `5xx` | Error del servidor: `500` |

**Por qué importa**

En una red que pierde cosas, un `GET` se repite sin miedo. Repetir un `POST` puede pagar dos veces la misma compra.

# Para qué sirve `Host`

la misma dirección y el mismo puerto

```
$ printf 'GET / HTTP/1.1\r\nHost: blog.wayra.test\r\n\r\n' | nc -w1 10.212.6.10 80
<html>el blog, en la misma IP</html>                    (se omiten los headers)
```

# Para qué sirve `Host`

la misma dirección y el mismo puerto

```
$ printf 'GET / HTTP/1.1\r\nHost: blog.wayra.test\r\n\r\n' | nc -w1 10.212.6.10 80
<html>el blog, en la misma IP</html>                    (se omiten los headers)
```

**Sin `Host`**, el RFC 9112 exige un `400`:

un servidor real, y el nuestro

```
$ printf 'GET / HTTP/1.1\r\n\r\n' | nc -w3 example.com 80
HTTP/1.1 400 Bad Request
Server: cloudflare

$ printf 'GET / HTTP/1.1\r\n\r\n' | nc -w1 10.212.6.10 80
HTTP/1.1 200 OK
Server: mini_web
```

Que un servidor conteste no demuestra que cumpla el protocolo: `mini_web` es un juguete.

# Sin estado, y las cookies

**Sin estado**

Cada petición se entiende sola: el servidor no necesita recordar las anteriores.

el estado viaja en los mensajes

```
$ curl -s -D- http://www.wayra.test/visitas
HTTP/1.1 200 OK
Set-Cookie: visitas=1; Path=/
...
visita 1
$ curl -s -H 'Cookie: visitas=7' http://www.wayra.test/visitas
visita 8
```

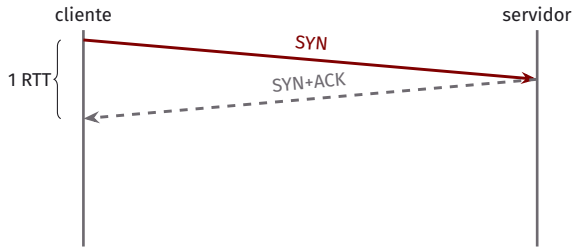
El servidor nos creyó el 7. Por eso una cookie real lleva un identificador al azar de una sesión guardada aparte.

SECCIÓN 06

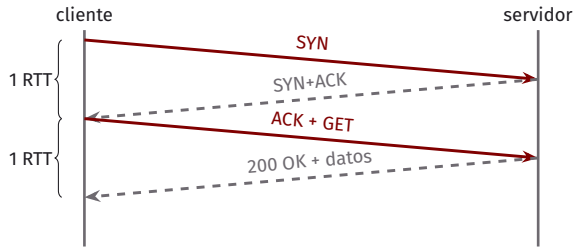
# Cuánto cuesta una página

06

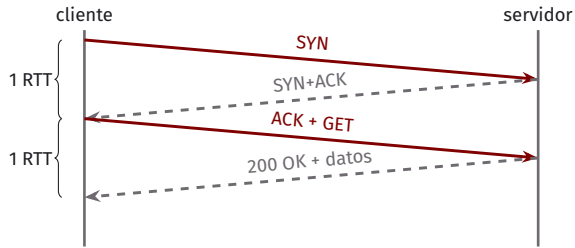
# Un objeto por conexión nueva: 2 RTT



# Un objeto por conexión nueva: 2 RTT



# Un objeto por conexión nueva: 2 RTT




Medido, con RTT de 50 ms: `con=0.052 ttfb=0.104`.

# No persistente contra persistente

| Modo | Cuenta | RTT | A 50 ms |
|---|---|---|---|
| No persistente | 4 objetos × 2 RTT | 8 | 400 ms |
| Persistente | 1 + 4 × 1 RTT | 5 | 250 ms |

medido, tres corridas

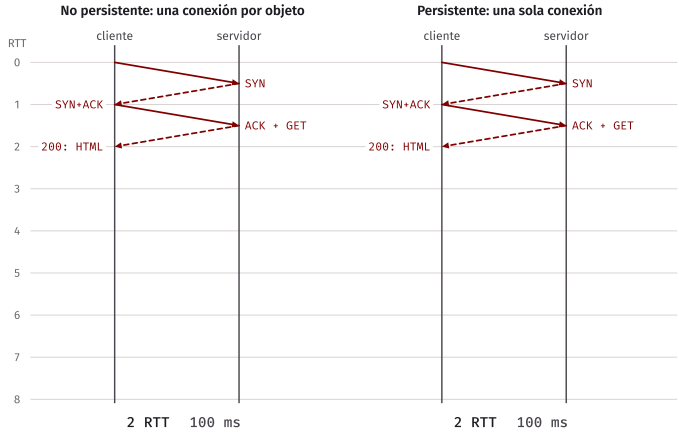
```
separadas=0.419766 persistente=0.259841
separadas=0.435746 persistente=0.268202
separadas=0.429272 persistente=0.262072
```

por objeto, en la conexión reutilizada

```
dns=0.000714 con=0.052259 ttfb=0.111525 total=0.111564 conexiones=1
dns=0.000000 con=0.000000 ttfb=0.055842 total=0.055871 conexiones=0
dns=0.000000 con=0.000000 ttfb=0.052721 total=0.052739 conexiones=0
dns=0.000000 con=0.000000 ttfb=0.051877 total=0.051896 conexiones=0
```

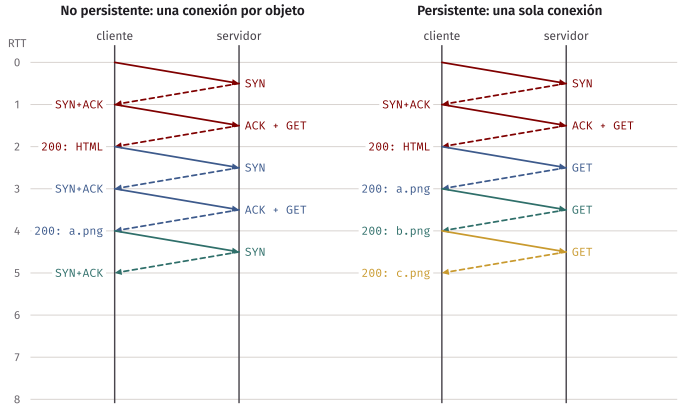
Además se conserva lo aprendido por TCP: una conexión nueva parte en slow start.

# 8 RTT contra 5 RTT



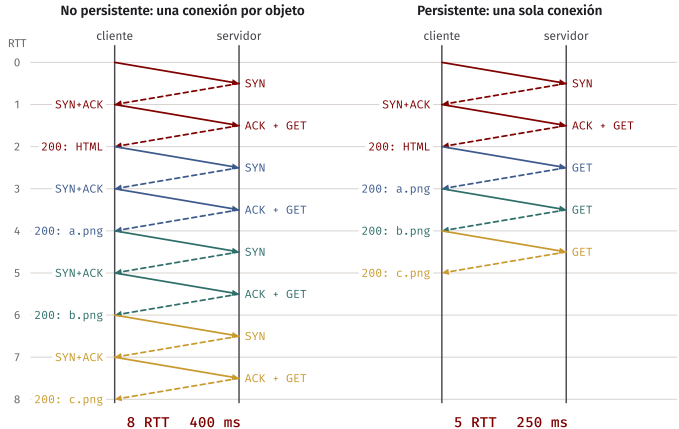

El primer objeto cuesta lo mismo en los dos: 1 RTT de conexión y 1 RTT para la petición.
Medido: con=0,052 s y ttfb=0,104 s.

# 8 RTT contra 5 RTT




En la conexión persistente, cada objeto siguiente cuesta 1 RTT: 1 + 4 × 1 = 5 RTT.
Medido: unos 0,26 s.

# 8 RTT contra 5 RTT




Sin reutilizar, cada objeto paga su handshake: 4 × 2 = 8 RTT.
Medido: unos 0,42 s. Además, cada conexión nueva parte en slow start.

# El problema de la fila

- Por una conexión persistente, las respuestas vuelven **en orden**.
- No llevan etiqueta: solo el orden dice a qué petición corresponden.

# El problema de la fila

- Por una conexión persistente, las respuestas vuelven **en orden**.
- No llevan etiqueta: solo el orden dice a qué petición corresponden.
- Si la primera tarda, las demás esperan aunque estén listas: head-of-line blocking de HTTP.

# El problema de la fila

- Por una conexión persistente, las respuestas vuelven **en orden**.
- No llevan etiqueta: solo el orden dice a qué petición corresponden.
- Si la primera tarda, las demás esperan aunque estén listas: head-of-line blocking de HTTP.
- El remedio de los navegadores: conexiones paralelas, hasta unas 6 por servidor. Cada una paga su handshake y compite por el enlace.

SECCIÓN 07

# Cache de HTTP

07



# Dos preguntas gobiernan la cache

1. **¿Sigue fresca?** Se usa sin preguntar: 0 RTT hasta el origen.
2. **¿Sigue siendo válida?** Se pregunta si cambió: 1 RTT y casi ningún byte.

| | |
|---|---|
| `max-age=60` | Fresca durante 60 segundos |
| `no-cache` | Se guarda, pero **siempre** se revalida |
| `no-store` | No se guarda |
| `private` | El navegador sí; un proxy compartido no |

# Dos preguntas gobiernan la cache

1. **¿Sigue fresca?** Se usa sin preguntar: 0 RTT hasta el origen.
2. **¿Sigue siendo válida?** Se pregunta si cambió: 1 RTT y casi ningún byte.

| | |
|---|---|
| `max-age=60` | Fresca durante 60 segundos |
| `no-cache` | Se guarda, pero **siempre** se revalida |
| `no-store` | No se guarda |
| `private` | El navegador sí; un proxy compartido no |

**Error común**

`no-cache` no significa “no guardar”. Eso es `no-store`.

# ETag y 304

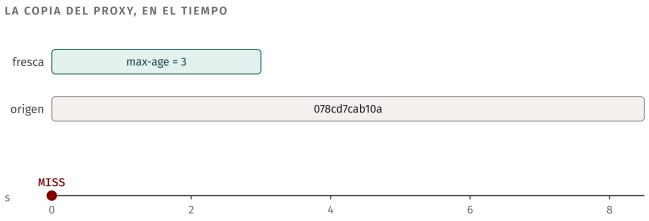
cliente

```
$ curl -sv -o /dev/null -H 'If-None-Match: "d8a8038111f4"' http://www.wayra.test/
> GET / HTTP/1.1
> Host: www.wayra.test
> If-None-Match: "d8a8038111f4"
>
< HTTP/1.1 304 Not Modified
< ETag: "d8a8038111f4"
< Cache-Control: max-age=60
<
```

Un 304 **no tiene cuerpo**. Se ahorran los bytes, pero no el RTT. (Recortamos algunos headers.)

# Un proxy en acción, paso a paso



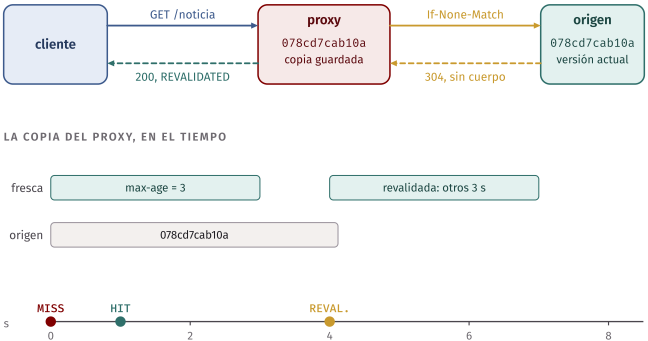

Primera vez: la cache no tiene nada.
Pide al origen, guarda y responde. X-Cache: MISS.

# Un proxy en acción, paso a paso



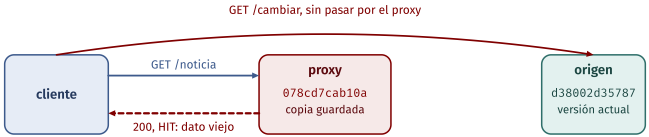

Un segundo después, la copia sigue fresca: se responde sin tocar al origen.
X-Cache: HIT, Age: 1.

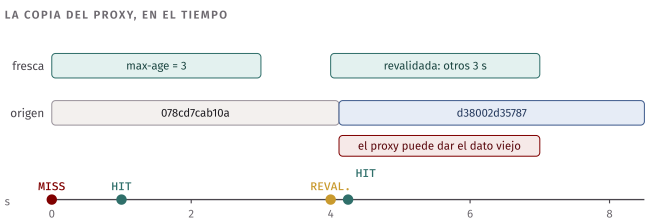
# Un proxy en acción, paso a paso




Pasaron 4 s y venció.
Pregunta con If-None-Match y recibe un 304 sin cuerpo: sigue valiendo. El cliente ve 200, REVALIDATED.

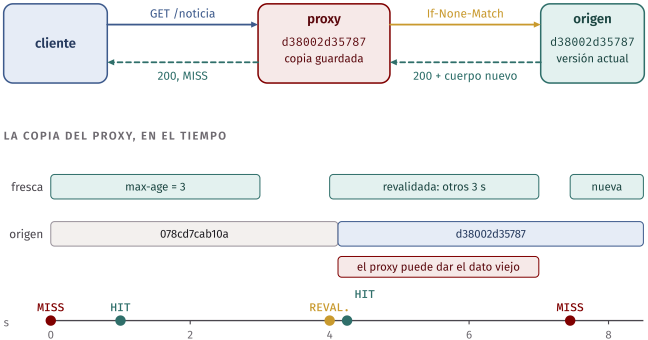
# Un proxy en acción, paso a paso




Cambia la noticia en el origen.
La copia sigue fresca, así que el proxy entrega el dato viejo, sin error alguno. Es el precio de la cache.

# Un proxy en acción, paso a paso




Venció y cambió: la pregunta condicional recibe un 200 con el cuerpo nuevo.
X-Cache: MISS, y queda la versión nueva.

# Un proxy en acción: `max-age=3`

`curl -x 10.212.7.10:3128 http://www.wayra.test/noticia`, resumido:

| **Cuándo** | ETag | Age | X-Cache |
|---|---|---|---|
| Primera vez | `078cd7cab10a` | | MISS |
| 1 s después | `078cd7cab10a` | 1 | HIT |
| 3 s después, ya vencida | `078cd7cab10a` | 0 | REVALIDATED |
| El origen cambia, y enseguida | `078cd7cab10a` | 0 | HIT |
| 3,2 s después | `d38002d35787` | | MISS |

# Un proxy en acción: `max-age=3`

`curl -x 10.212.7.10:3128 http://www.wayra.test/noticia`, resumido:

| **Cuándo** | ETag | Age | X-Cache |
|---|---|---|---|
| Primera vez | `078cd7cab10a` | | MISS |
| 1 s después | `078cd7cab10a` | 1 | HIT |
| 3 s después, ya vencida | `078cd7cab10a` | 0 | REVALIDATED |
| El origen cambia, y enseguida | `078cd7cab10a` | 0 | HIT |
| 3,2 s después | `d38002d35787` | | MISS |

**Miren la cuarta línea**

El origen ya cambió, y el cliente recibe el dato viejo sin error. Es el precio de la cache: dura como mucho `max-age`. Es el mismo compromiso que el TTL de DNS.

# Un proxy en acción: `max-age=3`

`curl -x 10.212.7.10:3128 http://www.wayra.test/noticia`, resumido:

| **Cuándo** | ETag | Age | X-Cache |
|---|---|---|---|
| Primera vez | `078cd7cab10a` | | MISS |
| 1 s después | `078cd7cab10a` | 1 | HIT |
| 3 s después, ya vencida | `078cd7cab10a` | 0 | REVALIDATED |
| El origen cambia, y enseguida | `078cd7cab10a` | 0 | HIT |
| 3,2 s después | `d38002d35787` | | MISS |

**Miren la cuarta línea**

El origen ya cambió, y el cliente recibe el dato viejo sin error. Es el precio de la cache: dura como mucho `max-age`. Es el mismo compromiso que el TTL de DNS.

El cliente ve `200` incluso al revalidar: el `304` es entre el proxy y el origen. Con `no-store`: `BYPASS`, siempre.

# Un proxy en acción: `max-age=3`

`curl -x 10.212.7.10:3128 http://www.wayra.test/noticia`, resumido:

| **Cuándo** | ETag | Age | X-Cache |
|---|---|---|---|
| Primera vez | `078cd7cab10a` | | MISS |
| 1 s después | `078cd7cab10a` | 1 | HIT |
| 3 s después, ya vencida | `078cd7cab10a` | 0 | REVALIDATED |
| El origen cambia, y enseguida | `078cd7cab10a` | 0 | HIT |
| 3,2 s después | `d38002d35787` | | MISS |

> **Miren la cuarta línea**
>
> El origen ya cambió, y el cliente recibe el dato viejo sin error. Es el precio de la cache: dura como mucho `max-age`. Es el mismo compromiso que el TTL de DNS.

El cliente ve `200` incluso al revalidar: el `304` es entre el proxy y el origen. Con `no-store`: `BYPASS`, siempre.

Con el proxy al lado: `HIT` en 0,0006 s; directo, 0,105 s.

SECCIÓN 08

# HTTP/2

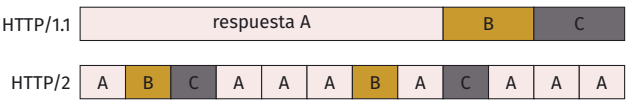
08



# HTTP/2: frames y streams

**HTTP/2**

Una conexión TCP lleva muchos **streams**. Cada mensaje se parte en **frames**, y cada frame lleva el número de su stream. Eso permite mezclarlos: **multiplexación**.




Encabezado binario de 9 bytes: largo (3), tipo, flags y stream (4). El cliente usa streams impares; el 0 es de la conexión. **Cambia el formato, no el significado**.

# Dos streams, una conexión

lo que anota h2_frames.py

```
<- HEADERS stream=1 largo=35
-> HEADERS stream=1 largo=1
<- HEADERS stream=3 largo=10
-> HEADERS stream=3 largo=1
-> DATA stream=1 largo=400
-> DATA stream=3 largo=400
-> DATA stream=1 largo=400
-> DATA stream=3 largo=400
-> DATA stream=1 largo=400
-> DATA stream=3 largo=400
-> DATA stream=1 largo=400
-> DATA stream=3 largo=400
-> DATA stream=1 largo=400 END_STREAM
-> DATA stream=3 largo=400 END_STREAM
```

- Los streams se **turnan**; ninguno espera al otro.
- `curl` reporta una sola conexión.
- Hay control de flujo por stream, sobre el de TCP. (Omitimos aquí los frames `SETTINGS` y `WINDOW_UPDATE`.)

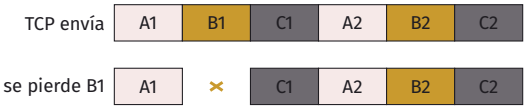
# Compresión de headers: HPACK

```
tshark -d tcp.port==8080,http2 ... -O http2, recortado

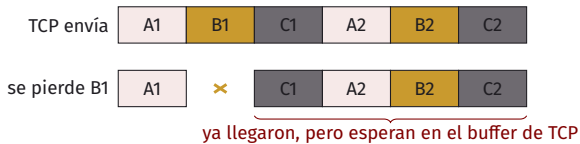
Stream: HEADERS, Stream ID: 1, Length 35, GET /uno
    Header: :method: GET
    Header: :scheme: http
    Header: :authority: 10.212.6.10:8080
    Header: :path: /uno
    Header: user-agent: curl/8.14.1
    Header: accept: */*
Stream: HEADERS, Stream ID: 3, Length 10, GET /dos
```

- Tabla estática, tabla dinámica y Huffman.
- Lo que ya se mandó, se dice con un índice: **35 bytes contra 10**.
- Nuestra respuesta mide un byte, `0x88`: la entrada 8 de la tabla, `:status: 200`.

# Lo que HTTP/2 no arregló



# Lo que HTTP/2 no arregló



# Lo que HTTP/2 no arregló




**Error común**

HTTP/2 eliminó el bloqueo de HTTP, no todo bloqueo. Una pérdida en TCP detiene **todos los streams**. TCP no sabe de streams. Por eso existe HTTP/3, sobre QUIC, la próxima semana.

SECCIÓN 09

# Laboratorio

09

# Laboratorio: DNS paso a paso y HTTP por dentro

En `code/week-12/`, con `make up`:

1. `dig +trace`, y a mano con `+norecurse`; luego un nombre `.pe`.
2. Registros y TTL: en nuestra red y con un resolver público.
3. Capturar una consulta y comparar ID, preguntas y respuestas con `dig`.
4. HTTP a mano con `nc`. ¿Qué pasa sin `Host`?
5. Con `curl -v`: un `200` y luego un `304` con `If-None-Match`.
6. `curl -w`: resolución, conexión y primera respuesta; con conexión reutilizada.
7. **Delegar `killa.test`**: completen `lab/test.zone` y `lab/killa.zone`.
8. Avanzar con **P4**.

`make test` comprueba la delegación, y se salta esas pruebas hasta que la completen.

# Resumen

- **DNS**: raíz, TLD y autoritativos. El resolver recorre por usted, y la cache evita casi todo.
- **TTL**: un compromiso entre rapidez y lo pronto que se nota un cambio.
- **UDP** por defecto; **TCP** cuando no cabe (`TC`).
- **ID y puerto al azar** dificultan adivinar. **DNSSEC** autentica. **DoH y DoT** cifran.
- **HTTP/1.1** es texto y sin estado. Una página con 3 imágenes: **8 RTT** contra **5 RTT**.
- **Cache**: `max-age` evita preguntar; `ETag` y `304` evitan volver a bajar.
- **HTTP/2** mezcla streams y comprime headers, pero una pérdida en TCP detiene a todos.

# Para la próxima clase

- Completen la delegación, y comprueben el `SERVFAIL` al quitar el glue.
- Hagan los ejercicios 1, 2 y 7 del capítulo.
- **P4**: se entrega la próxima semana.
- Lean Kurose y Ross, secs. 8.2, 8.3 y 8.6: TLS. Y secs. 2.2.7, 2.5 y 3.8: QUIC y CDN.

# Para la próxima clase

- Completen la delegación, y comprueben el `SERVFAIL` al quitar el glue.
- Hagan los ejercicios 1, 2 y 7 del capítulo.
- **P4**: se entrega la próxima semana.
- Lean Kurose y Ross, secs. 8.2, 8.3 y 8.6: TLS. Y secs. 2.2.7, 2.5 y 3.8: QUIC y CDN.

**Para pensar**

Vimos que hoy casi todo va cifrado, que quien mira la red ya no puede leer. ¿Qué es exactamente lo que TLS garantiza, y qué sigue viendo el observador?