

Computación Centrada en Redes

TLS, QUIC y CDNs

Transporte y aplicaciones

Gerson Garrido

Semana 13

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

Sesión A: criptografía esencial y TLS 1.3

1. Qué se quiere proteger
2. Simétrica, clave pública y firmas
3. Certificados
4. El handshake de TLS 1.3
5. Forward secrecy
6. Lo que ve quien está en el medio

Sesión B: QUIC, CDNs y streaming

7. Por qué no se cambia TCP
8. Streams independientes
9. Handshake integrado y 0-RTT
10. Migración de conexión
11. CDN: DNS y anycast
12. DASH: el cliente decide

Lectura

Kurose y Ross, secs. 8.2, 8.3 y 8.6 (sesión A); 2.2.7, 2.5 y 3.8 (sesión B). CS168: *TLS*. RFC 9000, secciones 1 y 2. Complementarias: RFC 8446, secciones 1 y 2; RFC 9001 y RFC 9114.

SECCIÓN 01

Criptografía esencial

01

Hasta ahora, todo viaja a la vista

El atacante está **en el camino**: un router, un access point, un enlace.

- Puede leer todo lo que pasa
- Puede borrar, cambiar y repetir paquetes
- Puede inventar paquetes nuevos

No puede:

- Romper las matemáticas
- Entrar a los extremos

Lo que queremos

Confidencialidad: solo el destinatario lo lee.

Integridad: llegó tal como salió.

Autenticación: hablo con quien creo.

Idea clave

TLS no hace que la red sea confiable. Hace que la red no importe. Lo que no impide es que le corten la conexión.

Cifrado simétrico

Definición

Cifrado *(encryption)* con una misma clave K, la clave simétrica *(symmetric key)*, compartida por las dos partes. Quien no tiene K no debería sacar nada del texto cifrado, más allá de su tamaño.

- AES y ChaCha20. Muy rápidos: unos 3 GB/s con AES-128-GCM en la máquina donde se preparó esta clase.
- El problema es otro: ¿cómo se ponen de acuerdo en K dos máquinas que nunca se vieron, si todo pasa por el atacante?

Cifrar no es proteger la integridad

AES-CTR hace un XOR con una secuencia que parece azar. Un XOR sobre el cifrado es el mismo XOR sobre el texto.

```
python3 ctr_flip.py

mensaje:          pagar 0100 soles a Ana
cifrado:          2e79b69f843d384ff0931ea45bcbf4fc3aa39f9df994
cifrado alterado: 2e79b69f843d314ff0931ea45bcbf4fc3aa39f9df994
se descifra como: pagar 9100 soles a Ana
MAC del original sobre el alterado: NO coincide, se descarta
```

Error común

Creer que si está cifrado, nadie lo pudo cambiar. Por eso TLS 1.3 solo usa **cifrado autenticado** (AEAD): AES-GCM o ChaCha20-Poly1305.

Clave pública

Definición

Cada parte tiene una clave **pública** *(public key)*, que le da a cualquiera, y una **privada**, que nunca sale de su máquina. Lo que hace una lo deshace la otra.

Dos usos que nos importan:

- **Ponerse de acuerdo** en un secreto hablando en público.
- **Probar quién soy**, sin decir ningún secreto.

Los números que siguen son de juguete, a propósito: `crypto_toys.py` no protege nada, solo deja seguir las cuentas.

RSA en una cuenta

$n = p \times q$. Pública: (n, e). Privada: d. La misma operación con distinto exponente.

```
python3 crypto_toys.py

== RSA con p=61, q=53, e=17
n=3233 e=17 d=2753
cifrar con la pública:  65^17 mod 3233 = 2790
descifrar con la privada: 2790^2753 mod 3233 = 65
firmar con la privada:  65^2753 mod 3233 = 588
verificar con la pública: 588^17 mod 3233 = 65
```

- Seguridad: factorizar un n de miles de bits es muy difícil. El 3233 lo hacen a mano.
- Las dos últimas líneas: lo que hizo la **privada** lo comprueba cualquiera con la **pública**. Esa es la firma digital *(digital signature)*.
- El RSA real siempre agrega relleno aleatorio. El de libro es determinista y maleable.

Diffie-Hellman en una cuenta

Un secreto común, sin haber compartido nada antes:

```
python3 crypto_toys.py

== Diffie-Hellman con p=2087, g=5
Ana elige a=731 y publica A = g^a mod p = 114
Beto elige b=1289 y publica B = g^b mod p = 2029
Ana calcula B^a mod p = 1778
Beto calcula A^b mod p = 1778
quien solo vio A y B, probando exponentes, halla a=731
```

- Coinciden porque $(g^b)^a = (g^a)^b$.
- Sacar el exponente es el **logaritmo discreto**. Con p pequeño se prueba. Con un p de miles de bits, o con curvas elípticas como X25519, no se conoce forma práctica.

Diffie-Hellman solo no autentica



M descifra con K_1, lee o cambia, y cifra con K_2

El problema

Ana cree que acordó una clave con Beto. Acordó una con el man-in-the-middle, sin que este rompiera nada.

Falta saber **con quién** se habla: de eso se encargan las firmas.

¿Por qué no usar la clave pública para todo?

openssl speed: depende de cada computadora

```
$ openssl speed -seconds 1 rsa2048
                              keygen     signs    verify keygens/s    sign/s  verify/s
                  rsa2048 0.055556s 0.000820s 0.000019s      18.0    1219.6   51373.2
```

1220 firmas por segundo, de 256 bytes: unos 0,3 MB/s, contra 3 GB/s de AES-GCM. Unas 10 000 veces.

Todo protocolo real es híbrido

Diffie-Hellman efímero	acordar el secreto de la conexión
Firma (RSA, ECDSA, Ed25519)	probar quién es el servidor
AEAD (AES-GCM, ChaCha20)	proteger todos los datos

Hash, MAC y firma

	Clave	Quién lo crea	Quién lo verifica
Hash (SHA-256)	Ninguna	Cualquiera	Cualquiera
MAC (HMAC)	Compartida	Quien tenga la clave	Quien tenga la clave
Firma	Privada y pública	Solo el owner	Cualquiera

- Un **hash** no autentica: el atacante cambia el mensaje y lo recalcula.
- Un **MAC** no le prueba nada a un tercero: cualquiera de los dos pudo crearlo.
- Una **firma** solo pudo hacerla uno. Por eso sirve para lo que sigue.
- Ojo: este MAC nada tiene que ver con la dirección MAC de Ethernet.

Una firma, y un byte cambiado

openssl con una clave Ed25519 (recortado)

```
$ openssl pkeyutl -sign -inkey firma.key -rawin -in msg.txt -out msg.sig
$ wc -c msg.sig
64 msg.sig
$ openssl pkeyutl -verify -pubin -inkey firma.pub -rawin -in msg.txt -sigfile msg.sig
Signature Verified Successfully
$ printf "pagar 900 soles\n" > msg.txt
$ openssl pkeyutl -verify -pubin -inkey firma.pub -rawin -in msg.txt -sigfile msg.sig
Signature Verification Failure
```

Falta algo: para verificar la firma del servidor necesito su clave pública. Si me llega por la red, ¿cómo sé que no es la del atacante?

SECCIÓN 02

Certificados

02

Certificados y autoridades de certificación

Definición

Un **certificado** *(certificate)* dice “esta clave pública es de este nombre, en estas fechas”. Lo firma una **CA**, que antes comprueba que quien lo pide controla el nombre.

la CA de la clase: make y pki.sh (recortado)

```
$ openssl x509 -in good.crt -noout -subject -issuer -dates -ext subjectAltName,basicConstraints
subject=CN=demo.redes.test
issuer=O=Curso de Redes, CN=CA de la clase
X509v3 Subject Alternative Name:
    DNS:demo.redes.test
X509v3 Basic Constraints:
    CA:FALSE
```

La confianza nace en la lista de **CA raíz** del sistema. `CA:FALSE`: esta clave no puede firmar otros.

Una cadena real

python3 internet.py chain (recortado a sujeto y emisor)

```
0 s:CN=www.cloudflare.com
  i:C=US, O=Let's Encrypt, CN=YE1
1 s:C=US, O=Let's Encrypt, CN=YE1
  i:C=US, O=ISRG, CN=Root YE
2 s:C=US, O=ISRG, CN=Root YE
  i:C=US, O=Internet Security Research Group, CN=ISRG Root X2
3 s:C=US, O=Internet Security Research Group, CN=ISRG Root X2
  i:C=US, O=Internet Security Research Group, CN=ISRG Root X1
```

Cada `i:` (emisor) es el `s:` (sujeto) de la línea siguiente.

Idea clave

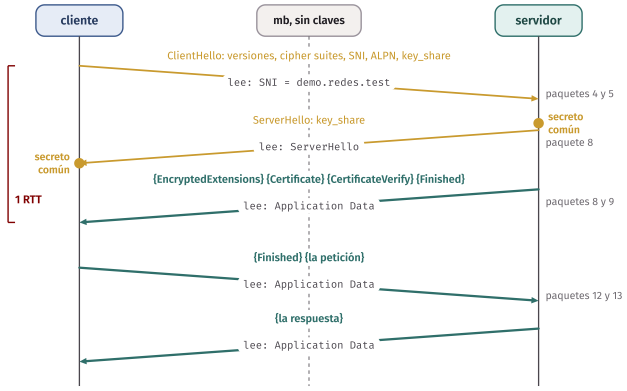
Un certificado es **público**: cualquiera puede presentar el de su banco. Lo que no puede es probar que tiene la clave **privada**. Esa prueba es una firma, durante el handshake.

SECCIÓN 03

TLS 1.3

03

TLS 1.3: una sola ida y vuelta



La respuesta llega 2 RTT después del primer mensaje.
Con el handshake de TCP son 3. Medido: 1,1 RTT de TLS y 3,5 hasta el primer byte.

Normalmente, el servidor no autentica al cliente: para eso están la contraseña y la cookie.

Contar los RTT

Protocolo	RTT al primer byte
TCP solo	2
TCP + TLS 1.3	3
TCP + TLS 1.2	4

Red `c - mb - s`: 40 ms de RTT con `netem`.
Medianas de 7 conexiones, en ms.

```
python3 scenarios.py rtts (recortado)

          TCP listo  TLS listo  primer byte  RTT del TLS  RTT al byte
TLS 1.3        44.1       88.7        138.0          1.1          3.5
TLS 1.2        43.2      129.1        170.9          2.1          4.3
```

Error común

Creer que TLS 1.3 es más rápido porque cifra mejor. Cambió el handshake: el `key_share` va primero.

Forward secrecy

Alguien graba por años el tráfico cifrado de un servidor. Un día, roba la clave privada del certificado. ¿Puede leer lo grabado?

- **Intercambio RSA de TLS 1.2:** sí. El secreto de cada conexión iba cifrado con esa clave.
- **TLS 1.3:** no. El Diffie-Hellman es **efímero**: sus números se crean para esa conexión y se borran.

Forward secrecy

Se da si robar las claves de larga duración no permite descifrar sesiones pasadas.

Idea clave

La clave de larga duración **firma**, y la efímera **acuerda**. Robarla permite suplantar desde ese día, no abrir lo grabado.

Lo que ve el router del medio, sin claves

python3 scenarios.py observer (recortado)

```
4       10.213.1.10
5       10.213.1.10     Client Hello (SNI=demo.redes.test)
8       10.213.2.10     Server Hello, Change Cipher Spec, Application Data
9       10.213.2.10     Application Data, Application Data, Application Data
12      10.213.1.10     Change Cipher Spec, Application Data
13      10.213.1.10     Application Data

Mensajes de certificado que se pueden leer: 0
```

- `ClientHello` y `ServerHello` van en claro. Tienen que ir así: llevan el Diffie-Hellman.
- Los registros cifrados se declaran **todos** como Application Data. El tipo real va adentro, y el certificado también.
- `Change Cipher Spec` no hace nada: es un disfraz para middleboxes viejos.

`sni_peek`: sin clave y sin biblioteca de TLS

python3 scenarios.py peek (recortado)

```
conexión de 10.213.1.10
  registro: tipo 22 (handshake), versión 3.1, 1570 bytes
  cipher suites ofrecidas: 30
  SNI: demo.redes.test
  ALPN: h2 http/1.1
  versiones: TLS1.3 TLS1.2
  key_share: grupo 0x11ec (1216 bytes) grupo 0x001d (32 bytes)
```

- `versión 3.1` es otro disfraz. La negociación real va en `supported_versions`.
- `0x001d` es X25519. `0x11ec` es X25519MLKEM768, un híbrido pensado para quien graba hoy y espera una computadora cuántica. Por eso el mensaje ya no entra en un paquete.
- Cada largo viene del otro extremo, y puede mentir: el parser revisa los límites en cada lectura.

HTTPS no oculta todo

Error común

Creer que HTTPS oculta todo.

Se ve

- Direcciones IP de los dos extremos
- Tiempos y tamaños de cada mensaje
- Normalmente, el nombre en el SNI
- La consulta de DNS, si no va cifrada

No se ve

- La ruta: `/banco/transferir`
- Headers y cookies
- El contenido
- El certificado, en TLS 1.3

¿Por qué no se cifra el SNI? Porque el servidor lo necesita para elegir el certificado, antes de que exista ninguna clave. *Encrypted Client Hello* lo resuelve con una clave publicada en DNS; su adopción todavía es parcial.

Con `SSLKEYLOGFILE`, la misma captura

python3 scenarios.py keylog (recortado)

```
5       10.213.1.10     Client Hello (SNI=demo.redes.test)
8       10.213.2.10     Server Hello, Change Cipher Spec, Encrypted Extensions
9       10.213.2.10     Certificate, Certificate Verify, Finished
12      10.213.1.10     Change Cipher Spec, Finished
13      10.213.1.10     GET / HTTP/1.1
14      10.213.2.10     New Session Ticket
15      10.213.2.10     New Session Ticket
23      10.213.1.10     Alert (Level: Warning, Description: Close Notify)
```

- Es el diagrama del handshake, ahora en paquetes reales.
- Ese archivo es tan sensible como una contraseña. Sirve para depurar, y nunca en producción.
- En Wireshark: *Preferences, Protocols, TLS, (Pre)-Master-Secret log filename*.

Qué protege cada error de `curl`

```
python3 scenarios.py certs

certificado    salida  mensaje de curl
good                0  bien
selfsigned         60  curl: (60) SSL certificate problem: self-signed certificate
wrongname          60  curl: (60) SSL: no alternative certificate subject name matches target hostname 'demo.redes.test'
expired            60  curl: (60) SSL certificate problem: certificate has expired

Con la CA del sistema, sin la CA de la clase:
good               60  curl: (60) SSL certificate problem: unable to get local issuer certificate
```

- **selfsigned:** justo lo que presentaría un man-in-the-middle.
- **wrongname:** firmado y válido, para otro sitio.
- **Sin la CA:** el servidor no cambió. Falta la confianza del cliente.
- `curl -k`: conserva el cifrado y pierde la autenticación.

SECCIÓN 04

QUIC

04

¿Por qué no se arregla TCP?

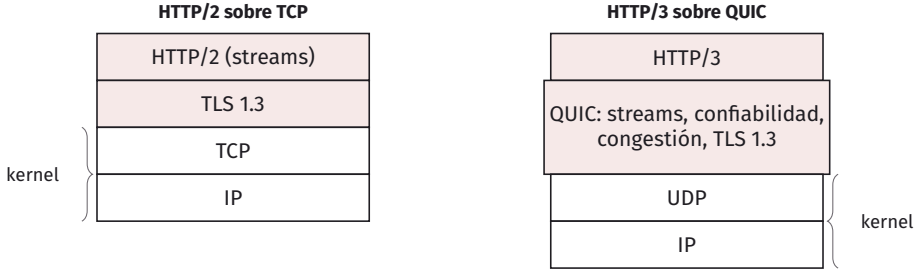
Los problemas se conocen: un RTT más al inicio, un solo orden de bytes, conexiones atadas a cuatro campos.

- **El kernel.** TCP vive en el sistema operativo. Una mejora tarda años en llegar a todos, y algunos equipos no se actualizan nunca.
- **Los middleboxes.** Firewalls y NAT miran TCP, y cortan o “corrigen” lo que no reconocen. El protocolo se **osificó**.

Ya lo vimos: TLS 1.3 tiene que disfrazarse de TLS 1.2 para pasar.

> **La respuesta de QUIC**
>
> En **espacio de usuario**, dentro de la aplicación. **Sobre UDP**, que los middleboxes ya dejan pasar. Y cifrando casi todo, para que nada se vuelva a osificar.

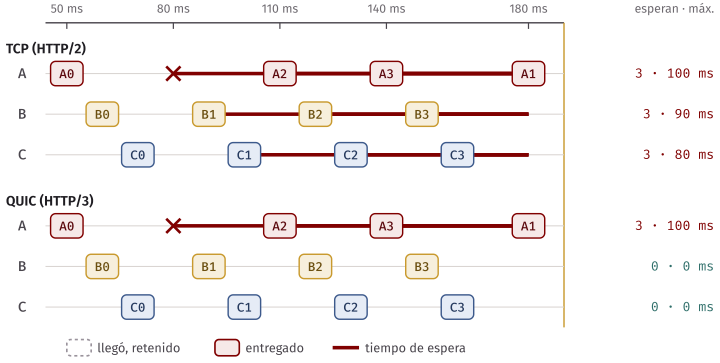
Dónde está cada cosa



Error común

Creer que UDP hace a QUIC poco confiable. QUIC implementa secuencia, ACK, retransmisión, control de flujo y de congestión. UDP es solo el sobre, igual que en su `rimay`.

Una pérdida, tres streams



A los 180 ms llega la retransmisión. TCP suelta todo de golpe: nueve paquetes esperaron, hasta 100 ms. En QUIC, solo esperó el stream que perdió algo.

Con un solo stream, no hay ninguna diferencia.

El simulador: tiempos exactos, no una medición

python3 hol_sim.py (recortado)

```
TX=10 ms, retardo=50 ms, recuperación=100 ms, se pierde el paquete 3
paq stream llega  entrega(TCP)  entrega(QUIC)
  3 A1       180          180            180  <- perdido, llega retransmitido
  4 B1        90          180             90
  5 C1       100          180            100
  6 A2       110          180            180

          TCP: esperan, máx.     QUIC: esperan, máx.
A               3    100 ms             3   100 ms
B               3     90 ms             0     0 ms
C               3     80 ms             0     0 ms
```

- El contenedor tiene un cliente de HTTP/3, pero no un servidor en el que se pueda provocar pérdidas. Por eso, aquí se simula.
- Gana una página con muchos recursos en una red con pérdidas. Una descarga única, no.

Los streams, en una conexión real

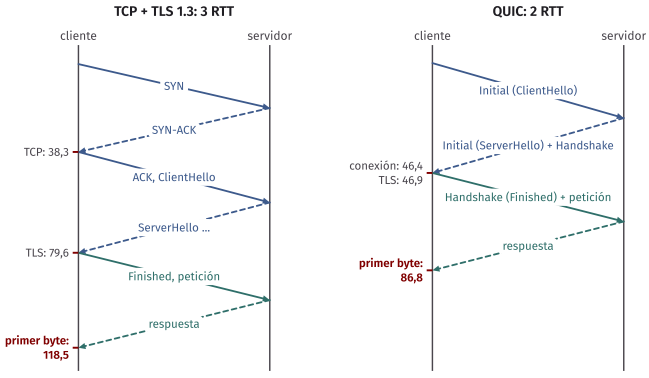
python3 internet.py quic, con SSLKEYLOGFILE (recortado)

```
10 HTTP3 Protected Payload (KP0), DCID=01f9a53171c3575aa4fd80317dc2b55aa4c2f59d, PKN: 0, STREAM(2), SETTINGS
11 HTTP3 Protected Payload (KP0), DCID=01f9a53171c3575aa4fd80317dc2b55aa4c2f59d, PKN: 1, STREAM(10)
12 HTTP3 Protected Payload (KP0), DCID=01f9a53171c3575aa4fd80317dc2b55aa4c2f59d, PKN: 2, STREAM(6)
13 HTTP3 Protected Payload (KP0), DCID=01f9a53171c3575aa4fd80317dc2b55aa4c2f59d, PKN: 3, STREAM(0), HEADERS: GET /cdn-cgi/trace
15 HTTP3 Protected Payload (KP0), PKN: 7, ACK, STREAM(3), SETTINGS
```

Bits bajos	Lo abre	Sentido	En la captura
`00`	Cliente	Dos	Stream 0: la petición `GET`
`10`	Cliente	Uno	2, 6 y 10: control y QPACK
`11`	Servidor	Uno	3, 7, 11, …

Un paquete retransmitido lleva un **número de paquete nuevo**. Desaparece la ambigüedad del RTT de la semana 10.

Handshake integrado: 2 RTT contra 3



Mediana de 11 conexiones nuevas a www.cloudflare.com, en ms, desde el inicio. La altura es proporcional al tiempo.

Los mensajes de TLS 1.3 viajan en frames CRYPTO. TLS entrega los secretos y QUIC protege sus paquetes (RFC 9001).

HTTP/2 frente a HTTP/3, en Internet

```
python3 internet.py h2h3: medianas de 11, en ms (recortado)

          versión    DNS  conexión  TLS listo  primer byte   total
HTTP/2          2    2.3      38.3       79.6        118.5   118.6
HTTP/3          3    1.9      46.4       46.9         86.8    86.9
```

- **HTTP/2:** con un RTT cercano a 40 ms, cada fase suma uno: 38, 80, 118.
- **HTTP/3:** conexión y TLS son casi iguales, porque son el mismo handshake.
- No comparen el 38,3 con el 46,4: el segundo ya incluye TLS.

> **Idea clave**
>
> QUIC gana en conexiones nuevas, en redes con pérdidas y muchos streams, y al cambiar de red. En una descarga larga por una red buena, no hay milagro.

El cifrado que no protege

Sin ninguna clave, `tshark` leyó el SNI de un paquete “cifrado”. Porque las claves de los Initial salen de:

```
HKDF-Extract(sal fija publicada en el RFC 9001, DCID del primer paquete)
```

El DCID viaja a la vista. Cualquiera repite la cuenta:

```
python3 quic_initial.py --catch secreto.redes.test

datagrama 0: 1200 bytes, DCID=514057c04c133f34, número de paquete 0, bytes de CRYPTO: 1158
datagrama 1: 1200 bytes, DCID=514057c04c133f34, número de paquete 1, bytes de CRYPTO: 350
ClientHello de 1508 bytes
SNI:  secreto.redes.test
ALPN: h3
```

`curl` intentó HTTP/3 contra un socket UDP nuestro. No hay servidor QUIC en ninguna parte, y no hace falta: el cliente habla primero. Las claves coinciden con el apéndice A del RFC 9001.

Qué protege QUIC, y de quién

Paquetes	Claves	Lo leen
Initial (`ClientHello`, `ServerHello`)	DCID y sal pública	Cualquiera en el camino
Handshake (certificado, `Finished`)	Diffie-Hellman	Solo los extremos
1-RTT (datos, ACK, números de paquete)	Diffie-Hellman	Solo los extremos

- ¿Para qué cifrar Initial? No da confidencialidad. Evita que un middlebox se osifique sobre el formato, y solo quien vio el primer paquete puede inyectar respuestas válidas.
- En TCP, secuencia y ACK van en claro, incluso con TLS. En QUIC, no. Un middlebox no ve el progreso, ni puede forzar un cierre con un `RST` falso.
- Un datagrama con Initial mide **al menos 1200 bytes**, y el servidor no manda más de **3 veces** lo recibido de una dirección sin validar: defensa contra la amplificación.

0-RTT: un RTT menos, y datos que se pueden repetir

Con un ticket de una visita anterior, el cliente cifra datos en su primer vuelo. El servidor todavía no aportó nada nuevo: quien los grabe puede mandarlos otra vez.

```
python3 zero_rtt.py build/pki

== protección por defecto
cliente legítimo: Early data was accepted
veces que el servidor procesó la petición: 1 antes de repetir, 1 después
== -no_anti_replay
cliente legítimo: Early data was accepted
veces que el servidor procesó la petición: 1 antes de repetir, 2 después
```

TLS 1.3 real, con `POST /transferir?monto=100`. El atacante no lee nada. No le hace falta.

> **Idea clave**
>
> Solo debe ir lo seguro de repetir. Un `GET` de una página, sí. Un pago, no.

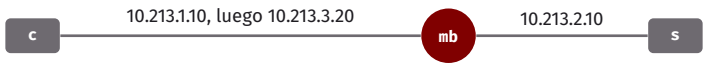
Migración: el connection ID

TCP

La conexión se identifica por cuatro campos. Si el celular deja el Wi-Fi, cambia su IP: **esa conexión muere**, aunque todo lo demás esté bien.

QUIC

Un **connection ID** en el encabezado, que no depende de la dirección ni del puerto. Aparece desde otro lado con un ID conocido: es la misma conexión.



Un juguete que no es QUIC: `cid_echo.py`

python3 scenarios.py migration (recortado)

```
== TCP
5: mensaje-5
-- el cliente cambia a 10.213.3.20; el socket sigue atado a 10.213.1.10
6: sin respuesta (timed out)
respondidos: 6 de 12
== UDP con connection ID
5: c0ffee 6 mensaje-5
-- el cliente cambia a 10.213.3.20, con otro puerto y otro socket
6: c0ffee 7 mensaje-6
respondidos: 12 de 12
```

Lo que QUIC agrega, sin lo cual esto sería inseguro (RFC 9000, sec. 9):

- **Validar el camino** con `PATH_CHALLENGE`: una dirección se puede falsificar.
- **Varios IDs**: con uno solo, un observador sigue al celular de una red a otra.
- **Todo va cifrado**. Nuestro `c0ffee` no: cualquiera podría secuestrar la sesión.

SECCIÓN 05

CDNs

05



CDN: acercar el contenido

Definición

Una **CDN** es un conjunto de servidores repartidos por el mundo, con copias del contenido de sus clientes, que atiende cada petición desde una réplica cercana.

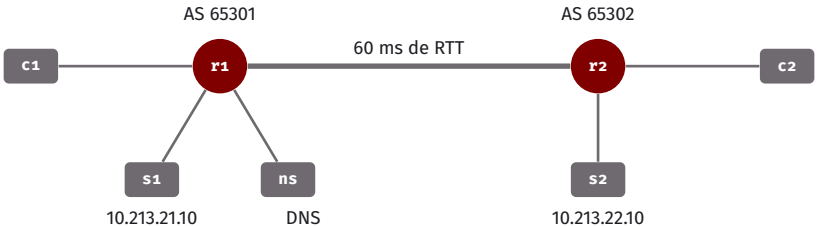
¿Por qué ayuda aun con el mismo ancho de banda? Casi todo se mide en RTT:

- Cada handshake lo paga, y cada ida y vuelta de slow start.
- El throughput de TCP es inversamente proporcional al RTT (semana 11).

Además: menos carga sobre el origen, menos tránsito que pagar y más capacidad contra picos y DoS. La réplica funciona como una caché: si no tiene el objeto, se lo pide al origen, respetando `Cache-Control`.

La pregunta difícil: ¿cómo se lleva a cada usuario a la réplica adecuada?

Nuestra CDN mínima



s1 y s2: AS 65310, y las dos con `10.213.50.1`

Selección por DNS

El autoritativo de la CDN contesta según quién pregunta. `geo_dns.py` lo reduce a una tabla:

python3 cdn_demo.py dns (recortado)

```
c1 pregunta cdn.redes.test a 10.213.30.10: 10.213.21.10
   curl a esa dirección: réplica s1, RTT 0.1 ms
   la otra réplica, 10.213.22.10: RTT 61.7 ms
c2 pregunta cdn.redes.test a 10.213.30.10: 10.213.22.10
```

python3 internet.py dns: un mismo nombre, varios resolvers (recortado)

```
www.akamai.com          @1.1.1.1   2.18.21.198 2.18.21.202
www.akamai.com          @8.8.8.8   190.98.161.48 190.98.161.67
www.cloudflare.com      @1.1.1.1   104.16.123.96 104.16.124.96
```

En Internet, el autoritativo no ve al usuario sino **a su resolver**, que puede estar lejos.

Selección por anycast

Definición

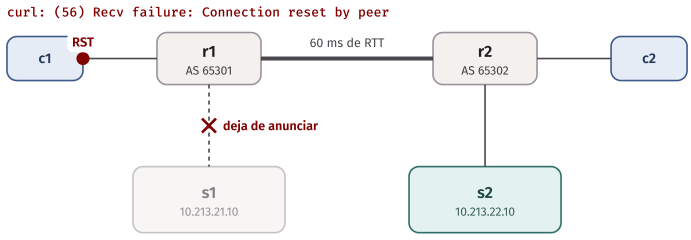
Varios servidores anuncian el **mismo prefijo** por BGP. Cada usuario llega al que su red prefiera.

python3 cdn_demo.py anycast (recortado)

```
Paths: (2 available, best #1, table default)
  65310
      Origin IGP, metric 0, valid, external, best (AS Path)
  65302 65310
      Origin IGP, valid, external
c1 -> 10.213.50.1: réplica s1, RTT 0.0 ms
c2 -> 10.213.50.1: réplica s2, RTT 0.0 ms
```

Gana el AS path más corto, como en la semana 6.

Anycast: una réplica deja de anunciar



Otra corrida, con una descarga abierta en s1. Sus paquetes pasan a llegar a s2, que nunca oyó de esa conexión, y responde con un RST: curl termina con el error 56, tras 2 463 792 bytes.

Si se cae una réplica

python3 cdn_demo.py failover (recortado)

```
antes:   c1 -> réplica s1, RTT 0.0 ms
s1 deja de anunciar; c1 llega a otra réplica tras 0.4 s
después: c1 -> réplica s2, RTT 61.7 ms
```

python3 cdn_demo.py cut: una descarga abierta (recortado)

```
curl: (56) Recv failure: Connection reset by peer
2463792 bytes
```

La trampa

Esa conexión llega a `s2`, que nunca oyó de ella, y responde con `RST`. Lo mismo si BGP cambia de ruta.

DNS o anycast

	Por DNS	Por anycast
Quién decide	El autoritativo de la CDN	La red, con BGP
Información	Del resolver, carga, mediciones	Política y AS path
Ante una falla	Lento: TTL y cachés	Rápido en la red
Riesgo	Resolver lejos del usuario	Cambio de ruta corta TCP

Idea clave

BGP no sabe de latencia ni de carga. “Más cercana” significa “camino de BGP preferido”. Las CDN reales combinan las dos técnicas.

SECCIÓN 06

Streaming

06

DASH: el cliente decide

Definición

El video se corta en **segmentos** de pocos segundos, cada uno en **varias calidades**. Son archivos comunes, por HTTP, y un **manifiesto** los describe.

```
<SegmentTemplate timescale="1" duration="2" startNumber="0"
                 media="$RepresentationID$/seg-$Number$.m4s"/>
<Representation id="400k" bandwidth="400000"/>
<Representation id="1000k" bandwidth="1000000"/>
<Representation id="2500k" bandwidth="2500000"/>
<Representation id="5000k" bandwidth="5000000"/>
```

- Servidor sin estado de sesión, y cualquier caché o CDN lo sirve sin esfuerzo.
- La decisión queda en el extremo, que es quien ve lo que le pasa a su red.

El bucle del cliente

1. Si el buffer está lleno, esperar.
2. Elegir la calidad con `choose`.
3. Bajar el segmento por HTTP.
4. Medir: bits / tiempo.
5. Buffer: menos el tiempo que tardó, más la duración del segmento.

Buffer

Segundos de video ya bajados y sin reproducir. Si llega a cero, la imagen se congela.

Tres metas que chocan

Más calidad, sin congelarse y sin saltar de una calidad a otra. Toda la dificultad está en `choose`: el ABR (*adaptive bitrate*).

En el simulador: la red cae entre los segundos 40 y 70

python3 dash_sim.py fixed (recortado)

```
 seg  inicio  calidad  tarda  medido    buffer  congelado
  24   46.7s   5000k   8.3s  1.20M     2.0s      5.3s
  25   55.0s   5000k   8.3s  1.20M     2.0s      6.3s
bitrate promedio: 5.00 Mbit/s; cambios: 0; congelado: 16.7 s
```

python3 dash_sim.py last: la última medición, sin margen (recortado)

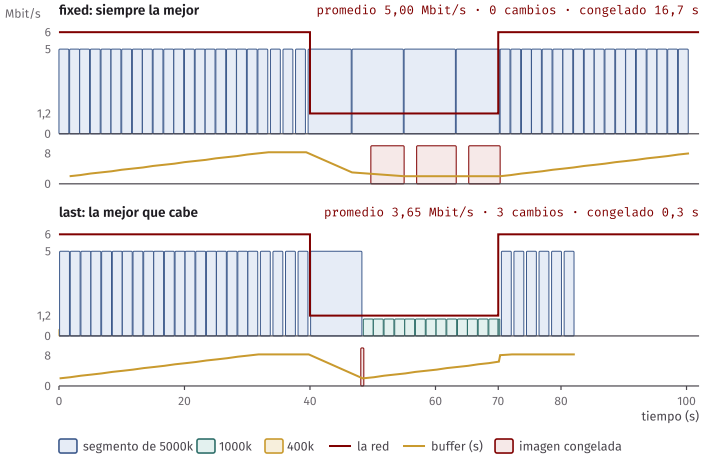
```
bitrate promedio: 3.65 Mbit/s; cambios: 3; congelado: 0.3 s
```

python3 dash_sim.py last –oscillating: la red oscila cada 3 s (recortado)

```
bitrate promedio: 3.56 Mbit/s; cambios: 27; congelado: 0.0 s
```

Promedio perfecto y experiencia horrible; o sin congelarse, saltando 27 veces.

Las dos corridas completas



Un cliente real: de 6 a 1,2 Mbit/s, por 12 s

scenarios.py dash last (recortado)

seg	t	calidad	medido	buffer	congelado
0	0.3s	400k	3.15M	2.0s	0.0s
5	8.7s	5000k	5.40M	3.6s	0.0s
6	12.3s	5000k	2.75M	2.0s	0.0s
7	16.7s	2500k	1.12M	2.0s	2.4s
8	18.6s	1000k	1.09M	2.2s	0.0s
12	23.6s	2500k	5.09M	5.1s	0.0s
13	25.5s	5000k	5.40M	5.3s	0.0s

congelado total: 2.5 s

scenarios.py dash fixed (recortado)

seg	t	calidad	medido	buffer	congelado
4	9.3s	5000k	5.39M	2.6s	0.0s
5	15.2s	5000k	1.70M	2.0s	3.3s
6	23.2s	5000k	1.24M	2.0s	6.1s
7	25.1s	5000k	5.36M	2.1s	0.0s

congelado total: 9.4 s

- Del lado del servidor: `python3 -m http.server`. Nada más.
- El segmento 0 mide 3,15 M en un enlace de 6: es pequeño y va por una conexión TCP nueva, en slow start. **Las calidades bajas se miden peor**, y eso dificulta subir.
- Abajo está el control de congestión: dos lazos de control, uno dentro del otro.

SECCIÓN 07

Laboratorio

07

Laboratorio: un handshake TLS y HTTP/2 frente a HTTP/3

Trabajen en `code/week-13/`. Con `make` y `make up` tienen certificados y red.

1. Con `openssl s_client`, lean la cadena de un sitio con TLS 1.3.
2. Capturen el handshake. Encuentren `ClientHello`, `ServerHello` y el SNI. ¿Qué ya va cifrado?
3. Abran esa captura en Wireshark con `SSLKEYLOGFILE`.
4. `curl –http2` frente a `–http3-only`. Cuenten RTT.
5. Capturen QUIC. ¿Qué se lee sin claves? ¿Por qué ven tan poco los middleboxes?
6. Certificado propio: ¿qué ataque detiene cada error de `curl`?
7. Una CDN: un nombre, varios resolvers. Luego `make up-cdn`.
8. Entreguen **P4** y lean el enunciado de **P5**.

Para completar: `lab/abr.py` (su propio ABR) y `lab/mkchain.sh` (una cadena de tres niveles). Sin la intermedia, `curl` debe rechazar al servidor.

Algunas actividades se basan en el laboratorio de TLS de Kurose y Ross.

Resumen

- Cifrar solo da confidencialidad. TLS 1.3 solo usa AEAD.
- Clave pública para acordar y autenticar; simétrica para los datos.
- El certificado es público. La prueba es `CertificateVerify`.
- TLS 1.3: 1 RTT. DH efímero: forward secrecy.
- HTTPS deja ver IP, tamaños y SNI.
- QUIC: espacio de usuario, sobre UDP y confiable.
- Una pérdida solo detiene su stream. 2 RTT contra 3.
- Los Initial se abren sin secretos. El resto, no.
- 0-RTT se puede repetir. El connection ID sobrevive al cambio de red.
- CDN: DNS o anycast. DASH: el cliente decide.

Para la próxima clase

- Entregar **P4**. Leer el enunciado de **P5**, `wasi`: se entrega en la semana 15.
- Leer para la semana 14: Kurose y Ross, secs. 4.4, 5.5, 6.5.2 y 6.6, sobre data centers y SDN.
- Terminar `lab/abr.py` y `lab/mkchain.sh`.

Pregunta

Un compañero dice: “QUIC cifra todo desde el primer paquete, así que nadie en el camino sabe a qué sitio entro”. Con lo que vieron hoy, ¿qué le responden? Muestren qué parte de su frase es falsa, con una captura.