

UNIVERSIDAD NACIONAL DE INGENIERÍA

Computación Centrada en Redes

Enlace, Ethernet y VLANs

Redes locales e IP

Gerson Garrido

Semana 3

Hoy

Sesión A

1. La capa física en lo esencial
2. Qué hace la capa de enlace
3. Detección de errores: paridad, checksum y CRC

Sesión B

4. Acceso múltiple: ALOHA y CSMA/CD
5. Ethernet y direcciones MAC
6. ARP
7. Switches
8. Spanning tree
9. VLANs

Lectura

Sesión A: Kurose y Ross, secs. 1.2.2, 6.1 y 6.2; CS168, *Links*. Sesión B: Kurose y Ross, secs. 6.3 y 6.4; CS168, *Ethernet*, *Layer 2 Routing* y *ARP*. Complementaria: RFC 826.

SECCIÓN 01

La capa física

01

Medios guiados y no guiados

Medio	Órdenes de magnitud	Qué lo limita
Par trenzado	1 a 10 Gb/s, hasta 100 m	Atenuación e interferencia
Coaxial	Varios Gb/s compartidos	Atenuación, medio compartido
Fibra multimodo	10 a 100 Gb/s, cientos de metros	Dispersión
Fibra monomodo	100 Gb/s o más, decenas de km	Atenuación, muy baja
Wi-Fi	Hasta varios Gb/s, decenas de metros	Paredes, interferencia, compartido
Celular	Hasta más de 1 Gb/s, km	Distancia, espectro, usuarios
Satélite	Decenas a cientos de Mb/s	Propagación, clima

Idea clave

El medio decide la tasa de errores. Por eso Ethernet solo *detecta* errores y Wi-Fi, además, retransmite en la capa de enlace.

Ancho de banda, modulación y codificación

- **Bandwidth** tiene dos sentidos: hertz (señales) y bits por segundo (redes).
- **Modulación:** más estados por símbolo, más bits, pero el ruido los confunde más fácil.
- **Codificación:** reglas para que el receptor mantenga su reloj. Cuesta: 10 Gb/s útiles usan 10,3125 Gb/s en la fibra (64b/66b).

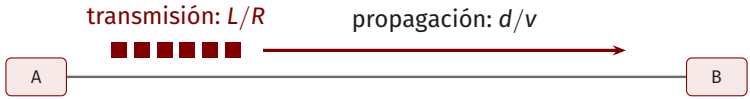
Límite de Shannon

$C = B \log_2(1 + S/N)$

20 MHz con 30 dB ($S/N = 1000$):
$20 \times 10^6 \cdot \log_2(1001) \approx 199$ Mb/s.

Más capacidad: más espectro o mejor señal.

Dos tiempos que no hay que confundir



Transmisión

Lo que tarda en salir el frame completo. 1500 bytes a 1 Gb/s: 12 µs.

Propagación

Lo que tarda un bit en llegar. Con $v \approx 2 \times 10^8$ m/s, 100 m: 0,5 µs.

Un enlace más rápido pone los bits más juntos, no los hace viajar más rápido.

SECCIÓN 02

La capa de enlace

02

Servicios de la capa de enlace

- **Framing:** dónde empieza y termina, con direcciones.
- **Acceso al medio:** quién transmite.
- **Entrega entre vecinos:** solo hasta el otro extremo de este enlace.
- **Detección de errores:** descartar lo dañado.
- **Entrega confiable:** opcional. Wi-Fi sí, Ethernet no.

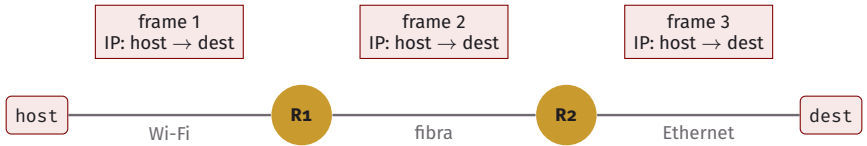
Frame

La unidad de datos del enlace. Encapsula un datagrama, como un sobre a una carta.

Dónde vive

Sobre todo en la tarjeta de red (NIC): el CRC es casi gratis en hardware.

De un salto a la vez



Idea clave

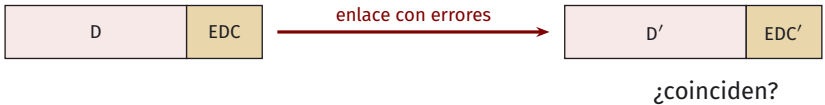
El frame nace y muere en cada enlace. Cada router desenvuelve el frame, mira el datagrama y lo envuelve en un frame nuevo. El datagrama sigue siendo el mismo.

SECCIÓN 03

Detección de errores

03

El modelo



- Si la comprobación falla, hubo un error.
- Si pasa, probablemente no lo hubo: siempre quedan patrones que no se notan.
- Los errores suelen llegar en **ráfagas**, no sueltos.

Paridad

Paridad par: un bit para que los unos sean pares.

una dimensión

```
datos:    1 0 1 1 0 0 1
paridad:  0
```

Detecta solo una cantidad impar de errores.

Bidimensional: por fila y por columna.

se daña un bit

```
1 0 1 0 | 0
1 1 0 1 | 0   <- esta fila falla
0 1 1 0 | 0
---------
0 0 1 1 | 0
    ^ esta columna falla
```

El cruce dice cuál es: **corrige** un error (FEC).

Checksum de Internet

- Palabras de 16 bits, suma con acarreo circular, complemento a uno.
- El receptor suma todo y debe obtener `0xFFFF`.
- Lo usan IPv4, UDP y TCP.

```
          0x4500
        + 0x0054
  ---------------
          0x4554
complemento: 0xBAAB
```

Débil

La suma no depende del orden: si dos palabras se intercambian, el checksum es el mismo.

Es barato en software. Vuelve en las semanas de transporte.

CRC: una división

CRC

Emisor y receptor acuerdan un generador G de $r+1$ bits (el primero es 1). Para los datos D, se eligen r bits R para que D seguido de R sea **divisible** entre G. Si el receptor obtiene residuo distinto de cero, hubo un error.

Aritmética módulo 2

- Sin acarreos: sumar y restar son XOR.
- El generador "cabe" en cualquier grupo que empiece en 1.

¿Por qué el residuo?

- Agregar r ceros es multiplicar por 2^r.
- $T = D \cdot 2^r \oplus R$, con R el residuo de $D \cdot 2^r / G$.
- Sumar el residuo es quitarlo: la división queda exacta.

Ejemplo resuelto: $D = 1101011$, $G = 1001$

```
build/crc 1101011 1001

  1101011000
  1001  (XOR en la posición 0)
  0100011000
   1001  (XOR en la posición 1)
  0000111000
      1001  (XOR en la posición 4)
  0000011100
       1001  (XOR en la posición 5)
  0000001110
        1001  (XOR en la posición 6)
  0000000111
CRC (r = 3 bits): 111
se transmite: 1101011111
```

1. G tiene 4 bits: $r = 3$.
2. Se agregan 3 ceros.
3. XOR donde quede un 1 adelante.
4. Los 3 bits finales son el CRC.

El receptor

Divide `1101011111` sin agregar nada, y el residuo es `000`.

Qué pasa con los errores

un bit dañado

```
$ build/crc -c 1101011011 1001
  ...
residuo: 100
ERROR detectado
```

el error es múltiplo de G

```
$ build/crc -c 1100010111 1001
  ...
residuo: 000
sin errores detectados
```

La regla

Si el patrón de error es *E*, el error pasa inadvertido **si y solo si** *E* es múltiplo de *G*.

En la segunda palabra cambiaron los bits 3 y 6: el patrón `1001` desplazado. El generador `1001` sirve para aprender, pero es malo.

Qué detecta siempre un CRC

Con un CRC de r bits, con generador que empieza y termina en 1:

- **Cualquier ráfaga de hasta r bits**, y por tanto cualquier error de un bit.
- **Cualquier cantidad impar de errores**, si G tiene a $x + 1$ como factor.
  $x^3 + 1 = (x + 1)(x^2 + x + 1)$
- De los patrones al azar, deja pasar cerca de $1/2^r$.

CRC-32

Ethernet usa un generador de 32 bits, `0x04C11DB7` sin el bit alto: detecta toda ráfaga de hasta 32 bits y deja pasar del orden de 1 de cada 2^{32} patrones al azar. En hardware es un registro de desplazamiento y unos XOR.

Cuál detecta qué

```
python3 detect.py
```

```
20000 intentos por fila, frames de 64 bytes. Daños NO detectados:
daño                        paridad  checksum   crc32
1 bit                             0         0       0
2 bits                        20000       616       0
ráfaga de 16 bits              9957         0       0
ráfaga de 32 bits             10033         1       0
ráfaga de 100 bits             9958         1       0
intercambio de palabras       20000     20000       0
```

- El checksum falla con 2 bits cerca de 1/32 de las veces: misma columna de 16 bits (1/16) y sentidos opuestos (1/2).
- El 0 del CRC-32 no significa infalible: a 1 en 2^{32}, lo esperable en 20 000 intentos es 0.

¿Por qué el transporte no usa CRC?

	Enlace: CRC	Transporte: checksum
Se calcula en	Hardware: casi gratis	Software: una suma es barata
Protege	Un enlace; cambia en cada salto	De extremo a extremo
Ve errores de	Transmisión en el cable	También dentro de los routers

Error común

Un CRC correcto no garantiza datos correctos para la aplicación. Si un router daña un paquete en su memoria, calcula un CRC nuevo y correcto sobre datos dañados. Es el argumento end-to-end de la semana 2.

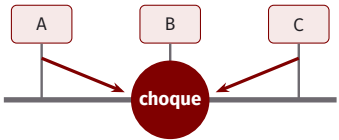
SECCIÓN 04

Acceso múltiple

04

El problema

- Varios nodos comparten un medio: Wi-Fi, coaxial, la Ethernet original.
- Dos a la vez: **colisión** *(collision)*.
- Las reglas se coordinan por el mismo medio.



Lo ideal, con un canal de R b/s:

1. Un nodo solo transmite a R.
2. M nodos obtienen R/M en promedio.
3. Sin coordinador central.
4. Simple.

Tres familias

Familia	Ejemplos	Un solo activo	Muchos activos	Colisiones
Particionar	TDM, FDM, CDMA	Mal: R/N	Bien	No
Aleatorio	ALOHA, CSMA/CD, CSMA/CA	Bien: R	Peor	Sí
Turnos	Polling, token passing	Casi R	Bien	No

Tráfico en ráfagas

Particionar reserva y desperdicia, como la conmutación de circuitos de la semana 2. Las redes de datos prefieren ideas de las otras familias.

ALOHA ranurado

- Universidad de Hawái, alrededor de 1970.
- Tiempo en ranuras. Si hay colisión, se reintenta con probabilidad p.
- Sirve la ranura con **exactamente un** intento:

$$S = N\,p\,(1-p)^{N-1}$$

- Máximo en $p = 1/N$. Con N grande y $G = Np$:

$$S = G\,e^{-G}, \quad \text{máx. en } G = 1: \ 1/e \approx 0{,}368$$

ALOHA puro

Sin ranuras: el período de riesgo se duplica.

$$S = G\,e^{-2G}$$

Máximo en $G = 1/2$: $1/(2e) \approx 0{,}184$.

La simulación coincide con la fórmula

build/aloha

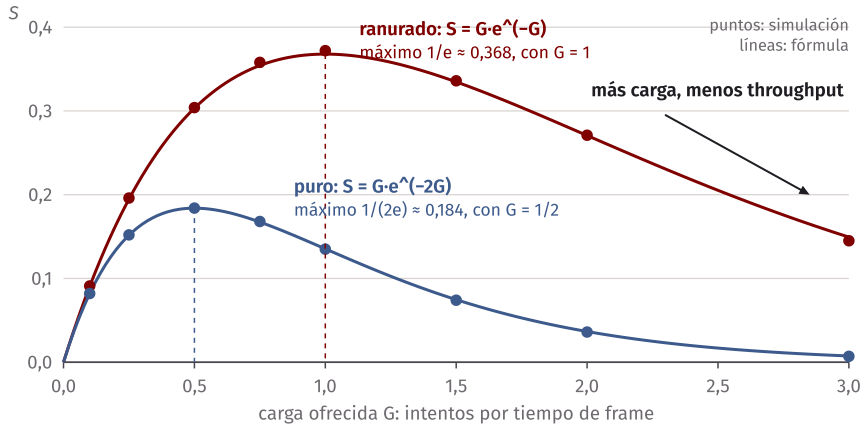
G	ranurado sim	G*e^-G	puro sim	G*e^-2G
0.10	0.091	0.090	0.082	0.082
0.25	0.196	0.195	0.152	0.152
0.50	0.304	0.303	0.184	0.184
0.75	0.358	0.354	0.168	0.167
1.00	0.372	0.368	0.135	0.135
1.50	0.336	0.335	0.074	0.075
2.00	0.271	0.271	0.036	0.037
3.00	0.145	0.149	0.007	0.007

Se cae

A la derecha del máximo, **más carga da menos throughput**.

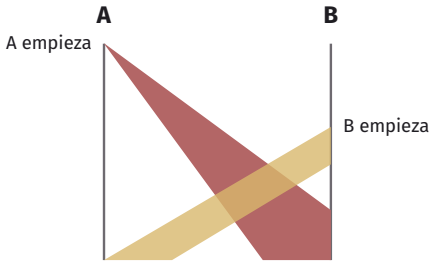
Una red de acceso aleatorio sobrecargada no se queda en su máximo: colapsa. Veremos esta forma en control de congestión.

Más carga, menos throughput



CSMA: escuchar antes de hablar

- **CS:** si el canal está ocupado, se espera.
- **CD:** si otro empieza mientras transmito, corto.
- ¿Por qué sigue habiendo colisiones? Por el retardo de propagación: B sigue oyendo silencio hasta que le llega la señal de A.



CSMA/CD: retroceso exponencial binario

Tras la colisión n del frame: K al azar en $\{0, \ldots, 2^m - 1\}$, con $m = \text{mín}(n, 10)$. Se espera $K \times 512$ tiempos de bit. Tras 16 intentos, se descarta.

Sin saber cuántos

Nadie necesita saber cuántos compiten: el rango crece hasta que uno gana.

Col.	K	**Máx. a 10 Mb/s**
1.ª	0, 1	51,2 µs
2.ª	0 a 3	153,6 µs
3.ª	0 a 7	358,4 µs
10.ª y más	0 a 1023	52,4 ms

De dónde salen los 64 bytes

- El emisor debe seguir transmitiendo cuando vuelva la señal de una colisión: dos tiempos de propagación.
- 512 bits a 10 Mb/s tardan 51,2 µs.
- A 2×10^8 m/s dan unos 5 km de ida; los repetidores bajaban ese máximo a unos 2.5 km.

Eficiencia aproximada

$1/(1+5a)$ con $a = d_{\text{prop}}/d_{\text{trans}}$. Con 1500 bytes a 10 Mb/s y $d_{\text{prop}} = 10$ µs: 96 %.

¿Y hoy?

Cada host tiene un cable propio hasta un switch, **full-duplex**. No hay colisiones posibles y CSMA/CD no se usa.

En el aire sigue vivo

Wi-Fi usa CSMA/CA: no puede oír mientras transmite. Semana 14.

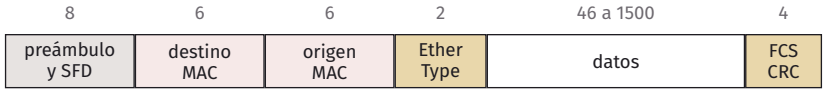
SECCIÓN 05

Ethernet

05



El frame de Ethernet



- **EtherType:** multiplexa hacia arriba. `0x0800` IPv4, `0x0806` ARP, `0x86DD` IPv6.
- **Datos:** se rellena hasta 46. Los 1500 son la MTU clásica.
- **Tamaño:** 64 a 1518 bytes, sin contar el preámbulo.
- **Servicio:** sin conexión y no confiable. Sin handshake y sin ACK.
- **Sin TTL.** Un frame en un ciclo no muere. Ya volveremos sobre esto.

Direcciones MAC

- 48 bits, por interfaz, no por equipo.
- Primeros 24 bits: el fabricante (OUI).
- Primer byte, bit menos significativo: grupo.
- El siguiente: administrada localmente.
- Broadcast: `ff:ff:ff:ff:ff:ff`.

Nuestros hosts tienen `02:00:00:00:00:01`: el `02` es local y unicast.

	MAC	**IP**
Capa	Enlace	Red
Estructura	Plana	Jerárquica
Alcance	Red local	Internet
Al mudarse	No cambia	Cambia
Se parece a	DNI	Dirección postal

SECCIÓN 06

ARP

06



Se conoce la IP, falta la MAC

1. Se busca en la tabla ARP (una cache).
2. Si no está, se pregunta por **broadcast**: "¿quién tiene 10.203.0.2? Avísenle a 10.203.0.1".
3. Todos lo reciben. Solo el owner responde.
4. La respuesta va por **unicast**.
5. Se guarda en la tabla, por un tiempo.

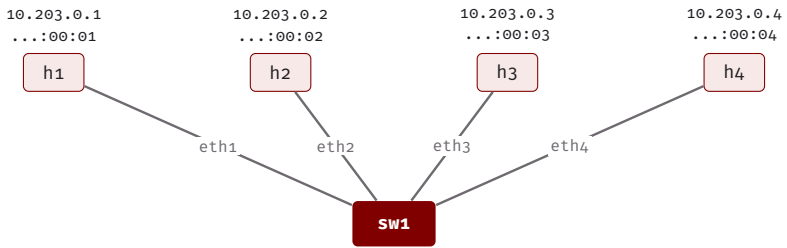
¿Por qué así?

Se pregunta a gritos porque no se sabe a quién dirigirse. Se responde en privado porque la pregunta ya traía la MAC.

Alcance

ARP llega hasta donde llega un broadcast: el borde de la red local.

La topología del laboratorio



- `make lan` levanta un bridge de Linux y cuatro hosts, con MAC fijas.
- `h1$ cmd` quiere decir `docker exec clab-w03-lan-h1 cmd`.
- `eth0` es la red de administración: no es parte del laboratorio.

Un intercambio real

```
h1$ tcpdump -t -e -n -r /tmp/arp.pcap

02:00:00:00:00:01 > ff:ff:ff:ff:ff:ff, ethertype ARP (0x0806), length 42: Request who-has 10.203.0.2 tell 10.203.0.1, length 28
02:00:00:00:00:02 > 02:00:00:00:00:01, ethertype ARP (0x0806), length 42: Reply 10.203.0.2 is-at 02:00:00:00:00:02, length 28
02:00:00:00:00:01 > 02:00:00:00:00:02, ethertype IPv4 (0x0800), length 98: 10.203.0.1 > 10.203.0.2: ICMP echo request, id 83, seq 0, length 64
02:00:00:00:00:02 > 02:00:00:00:00:01, ethertype IPv4 (0x0800), length 98: 10.203.0.2 > 10.203.0.1: ICMP echo reply, id 83, seq 0, length 64
```

```
las dos tablas

h1$ ip neigh show dev eth1
10.203.0.2 lladdr 02:00:00:00:00:02 REACHABLE
h2$ ip neigh show dev eth1
10.203.0.1 lladdr 02:00:00:00:00:01 DELAY
```

`h2` nunca preguntó y ya conoce a `h1`: la consulta traía sus datos, y el RFC 826 le deja aprovecharlos.

Los bytes de la consulta

```
tcpdump -xx

0x0000:  ffff ffff ffff 0200 0000 0001 0806 0001
0x0010:  0800 0604 0001 0200 0000 0001 0acb 0001
0x0020:  0000 0000 0000 0acb 0002
```

Bytes	Campo
0–5	Destino: broadcast
6–11	Origen: `h1`
12–13	EtherType: ARP
14–17	Hardware y protocolo
18, 19	Largos: 6 y 4
20–21	Operación: 1 consulta
22–31	MAC e IP de quien pregunta
32–37	MAC buscada: ceros
38–41	IP buscada

length 42

El mínimo de 64 incluye relleno y FCS, que pone la tarjeta. Entre interfaces veth no hay ni uno ni otro: 42 + 18 + 4 = 64 en un cable real.

ARP a mano, en C

arp_probe.c

```c
memset(frame, 0xff, 6);
memcpy(frame + 6, my_mac, 6);
put_u16(frame + 12, ETHERTYPE_ARP);

uint8_t *arp = frame + ETH_HEADER_LEN;
put_u16(arp + 0, 1);
put_u16(arp + 2, ETHERTYPE_IP);
arp[4] = 6;
arp[5] = 4;
put_u16(arp + 6, ARP_OP_REQUEST);
memcpy(arp + 8, my_mac, 6);
memcpy(arp + 14, my_ip, 4);
memset(arp + 18, 0, 6);
memcpy(arp + 24, target_ip, 4);
```

Socket AF_PACKET: lo que se envía ya es un frame. Cada campo, en su posición y orden de bytes.

h1$ arp_probe eth1 10.203.0.4

```
frame enviado (42 bytes):
ff ff ff ff ff ff 02 00 00 00 01 08 06
00 01 08 00 06 04 00 01 02 00 00 00 00 01
0a cb 00 01 00 00 00 00 00 00 0a cb 00 04
respuesta en 0,047 ms, la MAC del
vecino es 02:00:00:00:00:04
```

(La línea de la respuesta está partida en dos para que quepa.)

Hacia otra subred: se busca al gateway

`10.203.110.1/24` quiere enviar a `10.203.120.2`:

1. Compara con su máscara: otra subred.
2. Su ruta tiene un gateway: `10.203.110.254`.
3. Busca con ARP la MAC **del gateway**.
4. Arma un frame con la MAC del router y un datagrama con la IP final.

Idea clave

La IP dice a dónde va el paquete al final. La MAC dice a qué vecino se lo paso ahora.

SECCIÓN 07

Switches

07

Aprender mirando el origen

- **Hub:** capa física. Repite bits por todos los puertos. Todos colisionan. Historia.
- **Switch:** guarda el frame y lo envía solo por donde hace falta. Una cola por puerto.
- **Transparente:** ningún host sabe que existe.
- Un frame que entra por el puerto 3 con origen X prueba que X está por el puerto 3.

Por eso

No hace falta configurar un switch. La tabla se llena sola. Una entrada olvidada se quita a los 300 s, para adaptarse a cambios.

El algoritmo

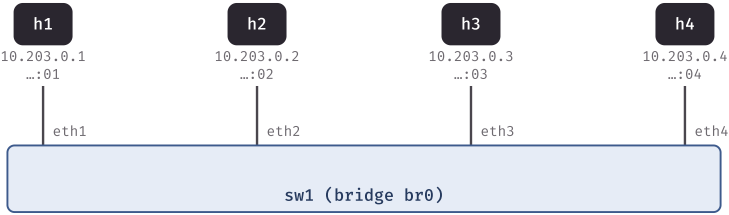
al llegar un frame por el puerto p, con origen S y destino D

```
aprender: anotar que S está por el puerto p, con tiempo 0
buscar D en la tabla:
  si D es broadcast o no está:   enviarlo por todos, menos p     (flooding)
  si D está por el puerto p:     descartarlo                     (filtering)
  si D está por otro puerto q:   enviarlo solo por q             (forwarding)
periódicamente: borrar las entradas que no se han visto hace mucho
```

Antes de seguir

`h1` hace ping a `h2`, con todo vacío. ¿Qué ve `h3`? ¿Qué queda en la tabla?

Un ping, frame por frame



LOS SEIS FRAMES

```
…:01 > ff:ff:ff:ff:ff:ff  ARP Request
…:02 > …:01  ARP Reply
…:01 > …:02  ICMP echo request, seq 0
…:02 > …:01  ICMP echo reply, seq 0
…:01 > …:02  ICMP echo request, seq 1
…:02 > …:01  ICMP echo reply, seq 1
```

TABLA DEL SWITCH

```
02:00:00:00:00:01 dev eth1
02:00:00:00:00:02 dev eth2
```

frames que vio h3: 1

h3 vio uno de los seis frames: el broadcast. En la tabla no están h3 ni h4, porque un switch no conoce a un host que no ha hablado.

El switch aprende

```
h3$ tcpdump -U -t -e -n -l -i eth1 "arp or icmp"

02:00:00:00:00:01 > ff:ff:ff:ff:ff:ff, ethertype ARP (0x0806), length 42: Request who-has 10.203.0.2 tell 10.203.0.1, length 28
```

```
sw1

$ bridge fdb show br br0 | grep -v permanent
02:00:00:00:00:01 dev eth1 master br0
02:00:00:00:00:02 dev eth2 master br0
```

- Hubo seis frames y `h3` vio uno: el broadcast.
- En la tabla no están `h3` ni `h4`: un switch no conoce a quien no ha hablado.

Flooding de un destino desconocido, y un hub

Vaciamos la tabla del switch, no la tabla ARP de `h1`. Con `ping -c 3`, `h3` ve:

h3

```
02:00:00:00:00:01 > 02:00:00:00:00:02, ethertype IPv4 (0x0800), length 98: 10.203.0.1 > 10.203.0.2: ICMP echo request, id 119, seq 0, length 64
```

Con `ageing_time 0`, el bridge olvida todo, y `ping -c 2` se ve completo:

h3

```
02:00:00:00:00:01 > 02:00:00:00:00:02, ethertype IPv4 (0x0800), length 98: 10.203.0.1 > 10.203.0.2: ICMP echo request, id 127, seq 0, length 64
02:00:00:00:00:02 > 02:00:00:00:00:01, ethertype IPv4 (0x0800), length 98: 10.203.0.2 > 10.203.0.1: ICMP echo reply, id 127, seq 0, length 64
02:00:00:00:00:01 > 02:00:00:00:00:02, ethertype IPv4 (0x0800), length 98: 10.203.0.1 > 10.203.0.2: ICMP echo request, id 127, seq 1, length 64
02:00:00:00:00:02 > 02:00:00:00:00:01, ethertype IPv4 (0x0800), length 98: 10.203.0.2 > 10.203.0.1: ICMP echo reply, id 127, seq 1, length 64
```

La respuesta de `h2` le enseña al switch dónde está, y `h3` deja de ver esa conversación.

Hub, switch y router

Equipo	Capa	Dominio de colisión	Dominio de broadcast	Reenvía según
Hub	Física	Uno solo	Uno solo	Nada: repite bits
Switch	Enlace	Uno por enlace	Uno solo	MAC de destino
Router	Red	Uno por enlace	Uno por interfaz	IP de destino

Por eso hay subredes

Una tabla plana y un solo dominio de broadcast sirven para una red pequeña. Una red grande se divide en subredes unidas por routers.

SECCIÓN 08

Spanning tree

08

Un ciclo por redundancia

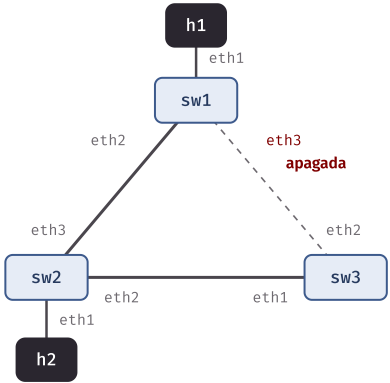
- Un broadcast se reenvía por todos los puertos.
- En un ciclo, las copias vuelven y siguen dando vueltas.
- **Ethernet no tiene TTL**.

Topología `make loop`, con 5 ms de retardo en cada enlace entre switches. Así la tormenta no se come la laptop.

No quiten el retardo

Con retardo en un solo sentido, `h2` recibió más de un millón de frames por segundo.

Una sola consulta ARP, para siempre



FRAMES QUE RECIBIÓ H2

1369

TABLA DE SW1

```
02:00:00:00:00:02 dev eth2
02:00:00:00:00:01 dev eth3
```

h1 está por eth1, no por eth3

ping -c 5: 1 recibido, 80 % de pérdida

`ciclo abierto otra vez, 3 s: 1369 -> 1369`

La tormenta solo se acaba abriendo el ciclo: con eth3 apagada, el contador se queda en 1369.

Un solo ARP

contador de frames en h2

```
ciclo abierto, 5 s sin tráfico: 8 -> 9
sw1$ ip link set eth3 up
ciclo cerrado, 5 s sin tráfico: 9 -> 9

h1$ ping -c 1 -W 1 10.203.0.2
1 packets transmitted, 0 packets received
rx de h2 = 208
rx de h2 = 308
rx de h2 = 411
rx de h2 = 509
rx de h2 = 613
```

- El ciclo no daña mientras nadie transmita.
- Una consulta ARP: unos 100 frames por segundo, sin que nadie envíe nada más.
- Las cifras dependen del retardo y de la máquina. En un enlace real, satura.

Tormenta de broadcast

Las copias de un broadcast circulan sin fin.

Las tablas quedan envenenadas

sw1 durante la tormenta

```
$ bridge fdb show br br0 | grep 02:00
02:00:00:00:00:02 dev eth2 master br0
02:00:00:00:00:01 dev eth3 master br0
```

- `h1` está conectado a `eth1` de `sw1`, pero `sw1` cree que está por `eth3`.
- La copia que dio la vuelta llegó con origen `h1`, y el switch aprendió, de buena fe, algo falso.

h1

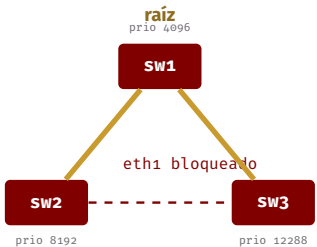
```
$ ping -c 5 -W 1 10.203.0.2
5 packets transmitted, 1 packets received, 80% packet loss
```

Solo se acaba si se abre el ciclo: `ciclo abierto otra vez, 3 s: 1369 -> 1369`.

STP: apagar enlaces, sin quitarlos

Spanning Tree Protocol (802.1D), Radia Perlman, 1985. Con mensajes BPDU:

1. **Raíz:** gana el menor identificador: prioridad y MAC.
2. **Puerto raíz:** en cada switch, el de menor costo hacia la raíz.
3. **Puerto designado:** en cada enlace, el extremo más cercano a la raíz.
4. **Todo lo demás se bloquea.** Sigue escuchando BPDU.



La predicción sale tal cual

bridge link e ip -d link show br0, con las líneas cortadas

```
sw1$ 3585: eth2@if3584:  mtu 9500 master br0 state forwardin
     3582: eth3@if3583:  mtu 9500 master br0 state forwardin
     bridge_id 1000.AA:C1:AB:5D:BD:C7 designated_root 1000.AA:C1:AB:5D:BD:C7 root_port 0 root_path_cost 0
sw2$ 3584: eth3@if3585:  mtu 9500 master br0 state forwardin
     3580: eth2@if3581:  mtu 9500 master br0 state forwardin
     bridge_id 2000.AA:C1:AB:0F:A2:F9 designated_root 1000.AA:C1:AB:5D:BD:C7 root_port 3 root_path_cost 2
sw3$ 3581: eth1@if3580:  mtu 9500 master br0 state blocking
     3583: eth2@if3582:  mtu 9500 master br0 state forwardin
     bridge_id 3000.AA:C1:AB:A1:D1:2D designated_root 1000.AA:C1:AB:5D:BD:C7 root_port 2 root_path_cost 2
```

- `1000` es 4096 en hexadecimal. Los tres coinciden en la raíz.
- Enlace `sw2`–`sw3`: empate a costo 2, gana `sw2`, y `sw3:eth1` se bloquea.
- Con STP, el mismo ping: `con stp: 1373 -> 1377`. Sin tormenta.

Cuando un enlace falla

```
se corta sw1–sw2

t=0 s   sw3:eth1 state blocking
t=21 s  sw3:eth1 state listening
t=23 s  sw3:eth1 state learning
t=25 s  sw3:eth1 state forwarding
t=25 s  ping OK
```

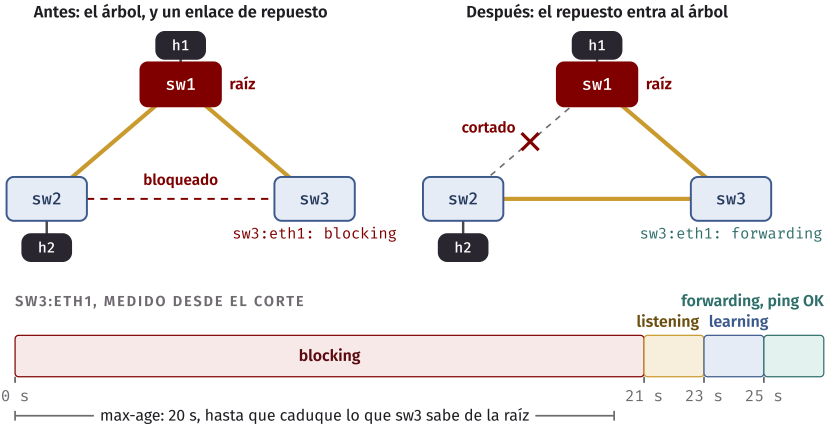
STP clásico, del kernel de Linux. Por defecto serían 50 s.

- **max-age (20 s):** `sw3` guarda la información de la raíz hasta que caduque. Los BPDU peores de `sw2` no cuentan.
- **forwarding-delay (2 s aquí, 15 s por defecto):** dura cada etapa, listening y learning. Evita un ciclo durante la transición.

Hoy

RSTP (802.1w) converge en segundos o menos. Un data center prefiere routing de capa 3, con todos los enlaces activos: semana 14.

El repuesto entra al árbol



Teoría: 20 + 2 + 2 = 24 s, con forwarding-delay de 2 s en cada etapa. Por defecto, con 15 s por etapa, serían 50 s.

SECCIÓN 09

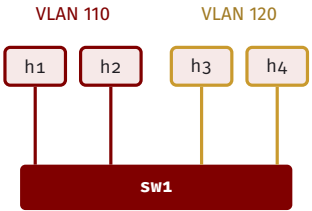
VLANs

09

Una red física, varias lógicas

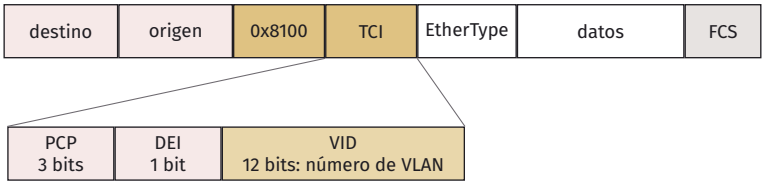
VLAN

Una red lógica separada dentro de una misma red física. El switch trata a cada VLAN como un switch aparte: cada una es su propio **dominio de broadcast**.



- **Access:** una VLAN, hacia un host. El host no sabe nada.
- **Trunk:** varias VLANs, entre equipos. Cada frame lleva una marca.

La etiqueta 802.1Q



- `0x8100` ocupa el lugar del EtherType: un equipo sin VLANs no toca el resto.
- 4096 valores; 0 y 4095 están reservados: quedan 4094 VLANs.
- Se agregan 4 bytes y se recalcula el FCS. El máximo pasa a 1522 bytes.
- La forwarding table es por pareja **(VLAN, MAC)**.

VLANs en un bridge de Linux

vlan/lan_sw1.sh

```
1  ip link set bro type bridge vlan_filtering 1
2
3  for port in eth1 eth2; do
4    bridge vlan add dev $port vid 110 pvid
         untagged
5    bridge vlan del dev $port vid 1
6  done
7
8  for port in eth3 eth4; do
9    bridge vlan add dev $port vid 120 pvid
         untagged
10   bridge vlan del dev $port vid 1
11 done
```

- `vid`: la VLAN se permite en el puerto.
- `pvid`: lo que entra sin etiqueta es de esta VLAN.
- `untagged`: lo que sale va sin etiqueta.

sw1$ bridge vlan show

```
port              vlan-id
eth1              110 PVID Egress Untagged
eth2              110 PVID Egress Untagged
eth3              120 PVID Egress Untagged
eth4              120 PVID Egress Untagged
bro               1 PVID Egress Untagged
```

Qué deja de funcionar

Los cuatro hosts siguen en la subred `10.203.0.0/24`, a propósito.

h1 y h3

```
h1$ ping -c 2 -W 1 10.203.0.2 | tail -2
2 packets transmitted, 2 packets received, 0% packet loss
h1$ ping -c 2 -W 1 10.203.0.3 | tail -2
2 packets transmitted, 0 packets received, 100% packet loss
h1$ ip neigh show dev eth1
10.203.0.2 lladdr 02:00:00:00:00:02 REACHABLE
10.203.0.3 FAILED
```

sw1$ bridge fdb show br br0 | grep -v permanent

```
02:00:00:00:00:01 dev eth1 vlan 110 master br0
02:00:00:00:00:02 dev eth2 vlan 110 master br0
02:00:00:00:00:03 dev eth3 vlan 120 master br0
02:00:00:00:00:04 dev eth4 vlan 120 master br0
```

Una VLAN separa, no conecta

Error común

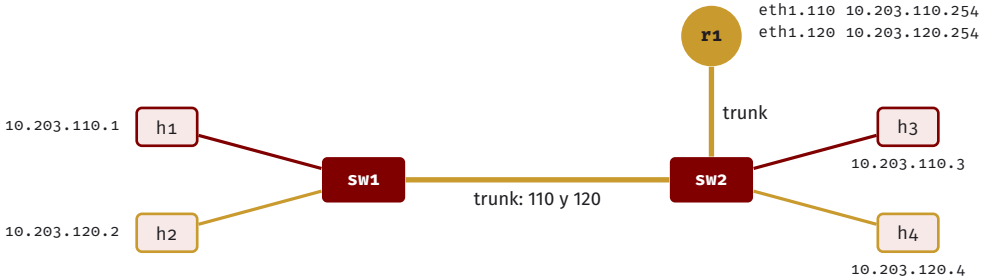
Creer que una VLAN, por sí sola, da comunicación entre redes. Hace lo contrario: dos VLANs son dos redes tan separadas como si estuvieran en switches sin conexión.

Para pasar de una a otra

Hace falta un **router**, y cada VLAN necesita su propia subred IP.

Las VLANs tampoco son un firewall: una vez conectadas por un router, pasa lo que el router deje pasar.

Router-on-a-stick



`make campus`: una VLAN repartida entre dos switches y un solo cable hasta el router, con una **subinterfaz** por VLAN.

Los trunks

vlan/campus_sw2.sh

```
1  ip link set br0 type bridge vlan_filtering
       1
2
3  bridge vlan add dev eth1 vid 110 pvid
       untagged
4  bridge vlan del dev eth1 vid 1
5  bridge vlan add dev eth2 vid 120 pvid
       untagged
6  bridge vlan del dev eth2 vid 1
7
8  for port in eth3 eth4; do
9    bridge vlan add dev $port vid 110
10   bridge vlan add dev $port vid 120
11   bridge vlan del dev $port vid 1
12 done
```

sw2$ bridge vlan show

```
port              vlan-id
eth1              110 PVID Egress Untagged
eth2              120 PVID Egress Untagged
eth3              110
                  120
eth4              110
                  120
br0               1 PVID Egress Untagged
```

Un trunk no lleva `pvid` ni `untagged`. Sus frames salen con etiqueta, y uno que entre sin etiqueta se descarta.

Las subinterfaces del router

vlan/campus_r1.sh

```
1  ip link add link eth1 name eth1.110 type vlan
       id 110
2  ip link add link eth1 name eth1.120 type vlan
       id 120
3  ip addr add 10.203.110.254/24 dev eth1.110
4  ip addr add 10.203.120.254/24 dev eth1.120
5  ip link set eth1.110 up
6  ip link set eth1.120 up
```

Lo que envía `eth1.110` sale por `eth1` con la etiqueta 110. Lo que llega con la etiqueta 110 aparece en `eth1.110` sin ella.

r1$ ip -br addr show | grep eth1

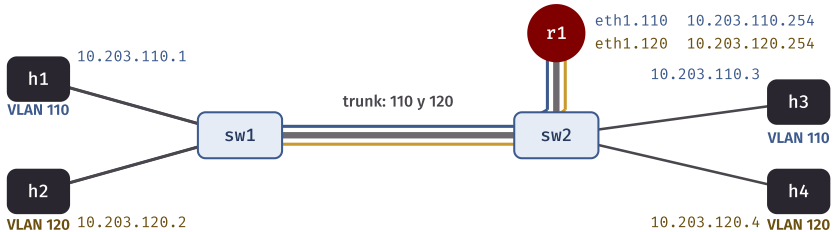
```
eth1.110@eth1    UP             10.203.110.254/24
eth1.120@eth1    UP             10.203.120.254/24
eth1@if3549      UP
```

r1$ vtysh -c 'show ip route connected'

```
C>* 10.203.110.0/24 is directly connected, eth1.110
C>* 10.203.120.0/24 is directly connected, eth1.120
```

Para el router son interfaces comunes, con rutas directas. (Salida abreviada.)

Un ping de h1 a h2: el camino



tramo	MAC origen	MAC destino	etiqueta 802.1Q	bytes	TTL
h1 a sw1	…:01 (h1)	…:fe (r1)	sin etiqueta	98	64
trunk, de ida	…:01 (h1)	…:fe (r1)	110	102	64
trunk, de vuelta	…:fe (r1)	…:02 (h2)	120	102	63
sw1 a h2	…:fe (r1)	…:02 (h2)	sin etiqueta	98	63

En los cuatro tramos: IP de origen 10.203.110.1, IP de destino 10.203.120.2.

sw1 quita la etiqueta y entrega a h2. Están en el mismo switch, pero el ping cruzó el trunk dos veces.

En el trunk: el mismo paquete, dos etiquetas

```
sw1$ tcpdump -t -e -n -r /tmp/c.pcap | cut -c1-116

02:00:00:00:00:01 > ff:ff:ff:ff:ff:ff, ethertype 802.1Q (0x8100), length 46: vlan 110, p 0, ethertype ARP (0x0806), R
02:00:00:00:00:fe > 02:00:00:00:00:01, ethertype 802.1Q (0x8100), length 46: vlan 110, p 0, ethertype ARP (0x0806), R
02:00:00:00:00:01 > 02:00:00:00:00:fe, ethertype 802.1Q (0x8100), length 102: vlan 110, p 0, ethertype IPv4 (0x0800),
02:00:00:00:00:fe > ff:ff:ff:ff:ff:ff, ethertype 802.1Q (0x8100), length 46: vlan 120, p 0, ethertype ARP (0x0806), R
02:00:00:00:00:02 > 02:00:00:00:00:fe, ethertype 802.1Q (0x8100), length 46: vlan 120, p 0, ethertype ARP (0x0806), R
02:00:00:00:00:fe > 02:00:00:00:00:02, ethertype 802.1Q (0x8100), length 102: vlan 120, p 0, ethertype IPv4 (0x0800),
02:00:00:00:00:02 > 02:00:00:00:00:fe, ethertype 802.1Q (0x8100), length 102: vlan 120, p 0, ethertype IPv4 (0x0800),
02:00:00:00:00:fe > 02:00:00:00:00:01, ethertype 802.1Q (0x8100), length 102: vlan 110, p 0, ethertype IPv4 (0x0800),
```

- Los tamaños son 46 y 102, no 42 y 98: los 4 bytes de 802.1Q.
- La tercera línea sube con `vlan 110`; la sexta baja con `vlan 120`. El cambio ocurre solo en `r1`.
- En la subinterfaz `eth1.110` de `r1`, el mismo tráfico aparece sin etiqueta.

Los bytes de la etiqueta

tcpdump -xx: el echo request

```
0x0000:  0200 0000 00fe 0200 0000 0001 8100 006e
0x0010:  0800 4500 0054 38fe 4000 4001 0612 0acb
```

tshark -r c.pcap -Y 'icmp.type == 8' -O vlan

```
802.1Q Virtual LAN, PRI: 0, DEI: 0, ID: 110
    000. .... .... .... = Priority: Best Effort (default) (0)
    ...0 .... .... .... = DEI: Ineligible
    .... 0000 0110 1110 = ID: 110
    Type: IPv4 (0x0800)
```

`8100 006e`: `0x6e` es 110. Sigue `0800`, el EtherType real.

Cuidado con el filtro

Al leer este archivo, `icmp` no encuentra nada: hay que usar `vlan and icmp`. En vivo sí aparece, porque el kernel entrega la etiqueta aparte del frame.

La IP no cambia, la MAC sí

Echo request	MAC origen	MAC destino	IP origen	IP destino	TTL
Visto en `h1`	`...:01` (h1)	`...:fe` (r1)	10.203.110.1	10.203.120.2	64
Visto en `h2`	`...:fe` (r1)	`...:02` (h2)	10.203.110.1	10.203.120.2	63

Es lo que veníamos diciendo

En `h1`, `ip neigh` solo muestra a `10.203.110.254`: buscó la MAC del gateway, nunca la de `h2`. El frame nació y murió en cada enlace; el datagrama siguió, con un TTL menos.

Tabla por VLAN, y una VLAN que falta

sw1$ bridge fdb show br br0 | grep -v permanent

```
02:00:00:00:00:02 dev eth2 vlan 120 master br0
02:00:00:00:00:01 dev eth1 vlan 110 master br0
02:00:00:00:00:04 dev eth3 vlan 120 master br0
02:00:00:00:00:fe dev eth3 vlan 120 master br0
02:00:00:00:00:fe dev eth3 vlan 110 master br0
```

La MAC de `r1` aparece dos veces: una en cada VLAN.

sin la VLAN 120 en el trunk de sw1

```
sw1$ bridge vlan del dev eth3 vid 120
h1$ ping -c 1 -W 1 10.203.120.254 | tail -2
1 packets transmitted, 1 packets received, 0 % packet loss
h2$ ping -c 1 -W 1 10.203.120.254 | tail -2
1 packets transmitted, 0 packets received, 100 % packet loss
```

Error común

Un número de VLAN solo conecta si esa VLAN está permitida en **todos** los trunks del camino.

SECCIÓN 10

Laboratorio

10

Capturar ARP y configurar VLANs

En `code/week-03/`:

1. `make lan`. Mirar `ip neigh`, vaciar la tabla, hacer ping y capturar ARP.
2. Mirar `bridge fdb show`. Predecir qué puertos reciben el primer frame. Comprobar con `tcpdump`.
3. Aplicar `vlan/lan_sw1.sh`. ¿Qué pings dejan de funcionar?
4. `make campus`. Aplicar los scripts de `vlan/` y correr `make check`. Encontrar la etiqueta en Wireshark.
5. **Reto:** `h4` a una VLAN 130, sin llevarla hacia `sw1`. `make check-reto`.
6. Entregar **P0** y leer **P1**.

> **Al terminar**
>
> `make test` corre todos los experimentos. `make clean` destruye las topologías y sus redes de Docker.

Resumen

- El enlace entrega frames entre vecinos, un salto a la vez.
- Paridad, checksum y CRC: cuanto mejor, más caro en software. El CRC es una división módulo 2.
- Acceso múltiple: particionar, aleatorio, turnos. Sobre un switch full-duplex no hace falta.
- MAC: a qué vecino. IP: a dónde al final. ARP traduce, y en otra subred busca al gateway.
- Un switch aprende de las direcciones de origen. Sin un árbol, un solo broadcast tumba la red.
- Una VLAN separa dominios de broadcast. Para cruzar de una a otra hace falta un router.

Para la próxima clase

- Terminar el laboratorio, entregar **P0** y leer el enunciado de **P1**.
- Leer Kurose y Ross, secs. 4.1, 4.2 y 4.3.1, y la parte de CIDR de 4.3.2. CS168, *IP Header*, *Addressing* y *Router Hardware*.

Pregunta

Un host con `10.203.110.1/24` envía a `10.203.120.2`. Con esa máscara, ¿cómo decide que el destino no es vecino? ¿Cómo elige un router entre miles de prefijos?