


Computación Centrada en Redes

# Data centers, SDN, Wi-Fi y 5G

## Clos, ECMP y overlays; redes para IA y redes sin cables

**Gerson Garrido**

Semana 14

UNIVERSIDAD NACIONAL DE INGENIERÍA

# Hoy

## Sesión A: data centers

1. Por qué un data center es distinto
2. Clos y leaf-spine. Sobresuscripción
3. BGP dentro del data center
4. ECMP: por flujo, no por paquete
5. Cuando cae un spine
6. Overlays y VXLAN
7. SDN y match-action

## Sesión B: IA y redes sin cables

8. Redes para entrenar modelos
9. Qué cambia sin cables
10. Wi-Fi: CSMA/CA y ACK
11. 5G: RAN, núcleo y sesión
12. Movilidad
13. TCP sobre un enlace que pierde

**Lectura**

Kurose y Ross, secs. 4.4, 5.5, 6.5.2 y 6.6 (sesión A), y 7.1 a 7.5 (sesión B). CS168: *Topologies*, *Routing*, *Virtualization*, *Software-Defined Networking*, *Collective Operations*, *Wireless Links* y *Cellular*. Complementarias: RFC 7938, RFC 7348, y de CS168 *Congestion Control* y *Host Networking*.

SECCIÓN 01

# Data centers

01

# Un solo dueño, otro tipo de tráfico

**Data center**

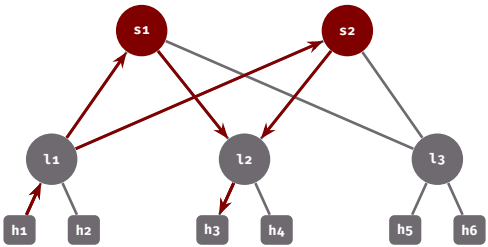
Miles de servidores en racks, con una red propia. El operador controla el hardware, la topología, los protocolos y a menudo los hosts.

- **Norte-sur:** entra y sale. Un cliente pide una página.
- **Este-oeste:** queda entre servidores. Una búsqueda se vuelve consultas a decenas de servicios.
- En un data center grande **domina el este-oeste**.

**Por qué un árbol con un núcleo enorme no alcanza**

Cuello de botella entre ramas; el costo crece peor que el tamaño; una falla arriba se lleva mucha capacidad.

# Leaf-spine: muchos caminos con switches baratos




- Cada leaf, a **cada** spine.
- Sin enlaces entre leaves ni entre spines.
- Dos leaves: siempre a dos saltos.
- Tantos caminos como spines.
- Cae un spine: se pierde 1/S de la capacidad, no la conectividad.
- Más servidores: más leaves.
  Más capacidad: más spines.

**Idea de 1953**

Clos la pensó para centrales telefónicas. Capacidad con cantidad, no con tamaño.

# Sobresuscripción, con un cálculo

**Sobresuscripción:** capacidad hacia los servidores dividida entre capacidad hacia los spines.

leaf con 48 x 25 Gbps abajo y 8 x 100 Gbps arriba; spines de 32 puertos

```
hacia servidores:  48 x 25 Gbps  = 1200 Gbps
hacia spines:       8 x 100 Gbps =  800 Gbps
sobresuscripción:  1200 / 800    = 1,5 : 1
spines:            8     (un uplink a cada uno)
leaves máximos:    32    (un puerto de cada spine)
servidores:        32 x 48 = 1536
```

**Error común**

Contar puertos: 48 abajo y 8 arriba no es 6:1.

Con tres capas: una fat-tree de $k$ puertos llega a $k^3/4$ servidores. Con $k = 48$, 27 648.

SECCIÓN 02

# BGP

02



# BGP dentro de un edificio

## ¿Por qué BGP? (RFC 7938)

- Más simple, y con menos inundación que estado de enlace (link-state).
- Control fino de qué se anuncia a quién.
- Un controlador puede sumarse con una sesión normal.

## El esquema

- eBGP de un salto, en cada enlace.
- AS privados.
- Todos los spines, el mismo AS: 65140.
- Cada leaf, el suyo: 65141, 65142, 65143.

**Dos consecuencias de compartir AS**

1. Los caminos tienen el mismo AS_PATH: el multipath sale solo.
2. Un camino que da la vuelta por un leaf no existe: lo impide la detección de ciclos.

# Dos rutas iguales, y las dos se usan

l1: tabla de BGP

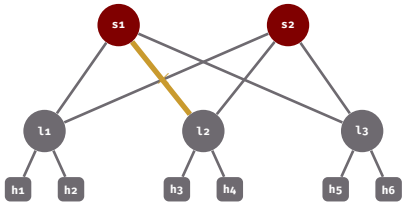
```
$ vtysh -c 'show bgp ipv4 unicast 10.214.2.0/24'
Paths: (2 available, best #1, table default)
  65140 65142
    10.214.21.0 from 10.214.21.0 (10.214.0.102)
      Origin IGP, valid, external, multipath, best (Older Path)
  65140 65142
    10.214.11.0 from 10.214.11.0 (10.214.0.101)
      Origin IGP, valid, external, multipath
```

l1: forwarding table

```
$ ip route show 10.214.2.0/24
10.214.2.0/24 nhid 23 proto bgp metric 20
        nexthop via 10.214.21.0 dev eth2 weight 1
        nexthop via 10.214.11.0 dev eth1 weight 1
```

BGP anuncia la mejor, pero instala todas las `multipath`. La decisión se toma en el leaf.

# La ruta que no aparece




se cae solo s1-l2

```
l1$ ip route show 10.214.2.0/24
10.214.2.0/24 nhid 18 via 10.214.21.0 dev eth2 ...

s1$ vtysh -c 'show bgp ipv4 unicast 10.214.2.0/24'
% Network not in table
```

- Físicamente, `s1` podría ir por `l1`, `s2`, `l2`.
- No lo aprende: el AS 65140 ya estaría en el AS_PATH.
- Y lo queremos así. Un leaf no debe ser tránsito entre spines.

SECCIÓN 03

# ECMP

03

# Dos next hop: ¿por cuál va cada paquete?

**ECMP**

Enviar el tráfico por varios caminos de igual costo, decidiendo con un **hash** de campos del encabezado.

| Se decide con | Problema |
|---|---|
| Ronda de paquetes | Un mismo flujo se desordena |
| IP de destino | Todos los que le hablan a un servidor, por un camino |
| Las dos IP | Dos servidores con mucho tráfico, por un camino |
| 5-tupla | Cada conexión por un camino, y se reparten |

**5-tupla**

IP y puerto de origen y destino, y protocolo. La misma 5-tupla es la misma conexión.

# El hash reparte al azar, no igual

ecmp_sim: el camino más cargado, respecto de lo justo

```
$ bin/ecmp_sim balance 2 8 20000
2 caminos, 8 flujos: el camino más cargado lleva 1.27 veces lo justo en promedio (peor ensayo: 2.00)
$ bin/ecmp_sim balance 2 6400 2000
2 caminos, 6400 flujos: el camino más cargado lleva 1.01 veces lo justo en promedio (peor ensayo: 1.05)
$ bin/ecmp_sim balance 8 64 20000
8 caminos, 64 flujos: el camino más cargado lleva 1.53 veces lo justo en promedio (peor ensayo: 3.12)
```

- Con miles de flujos pequeños, funciona muy bien.
- Con pocos flujos o muchos caminos, no tanto.

# La política de hash, en Linux

`net.ipv4.fib_multipath_hash_policy`, por espacio de red:

| | |
|---|---|
| **0** (de fábrica) | Capa 3: direcciones de origen y destino |
| **1** | Capa 4: la 5-tupla |
| **2** | Capa 3, o la interna si está encapsulado |
| **3** | Los campos que se pidan en `fib_multipath_hash_fields` |

```
python3 scenarios.py routeget: ip route get, 64 puertos de origen de h1 a h3

política 0: {'10.214.21.0': 64}
política 1: {'10.214.11.0': 30, '10.214.21.0': 34}
política 3: {'10.214.11.0': 39, '10.214.21.0': 25}
```

**Entre dos hosts**

Con el valor de fábrica, mil conexiones de `h1` a `h3` van todas por el mismo spine.

# ¿Se reparte igual? 16 flujos de iperf3

política 0, tres corridas

```
s1 hacia l2:        0.0 MB    0.0 %
s2 hacia l2:    21856.9 MB  100.0 %
s1 hacia l2:        0.0 MB    0.0 %
s2 hacia l2:    15496.3 MB  100.0 %
s1 hacia l2:        0.0 MB    0.0 %
s2 hacia l2:    13756.6 MB  100.0 %
```

política 1, tres corridas

```
s1 hacia l2:     5462.5 MB   54.9 %
s2 hacia l2:     4484.0 MB   45.1 %
s1 hacia l2:     7246.4 MB   42.9 %
s2 hacia l2:     9633.9 MB   57.1 %
s1 hacia l2:     4031.9 MB   35.2 %
s2 hacia l2:     7412.0 MB   64.8 %
```

Bytes medidos en los contadores de cada spine. Qué spine le toca a cada flujo depende de la semilla del hash.

**Error común**

Probar ECMP con `ping`. Sin puertos, va por un camino. Si el otro está roto, ni se enteran. Un `traceroute` muestra también solo uno de los caminos.

# La excepción: el hash que el paquete ya trae

```
python3 scenarios.py predict

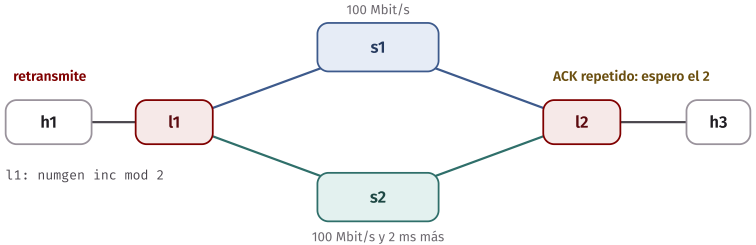
política 1: ip route get acertó 5 de 12 flujos
política 3: ip route get acertó 12 de 12 flujos
```

```
net/ipv4/route.c

1  /* short-circuit if we already have L4 hash present */
2  if (skb->l4_hash)
3      return skb_get_hash_raw(skb) >> 1;
```

- Con TCP de un socket local, por un veth, el paquete ya trae hash: sigue siendo por flujo, pero no se predice.
- La política 3, con `fields=0x37`, no toma ese atajo: la usamos para elegir camino.

# Por qué no por paquete




h3 guarda los adelantados y repite el ACK que pide el 2. h1 cree que hubo pérdida y retransmite lo que ya llegó.
Con el doble de capacidad disponible, rinde menos de un tercio.

# Por paquete: menos de un tercio

```
python3 scenarios.py perpacket, l1 forzado con nft: numgen inc mod 2 y dos tablas

por flujo:
      92 Mbit/s       0 retransmisiones      1513 fuera de orden en h3
por paquete:
      22 Mbit/s    1393 retransmisiones      4600 fuera de orden en h3
por flujo:
      95 Mbit/s      20 retransmisiones       392 fuera de orden en h3
por paquete:
      26 Mbit/s    1516 retransmisiones      5462 fuera de orden en h3
```

- Hay el doble de capacidad disponible, y rinde menos de un tercio.
- Más de mil retransmisiones sin ninguna pérdida programada.
- **ECMP reparte peor a cambio de no romper lo que corre encima.**

# Flujos grandes: el hash cuenta flujos, no bytes

```
python3 scenarios.py elephants: uplinks de 100 Mbit/s, tres corridas

  los dos por s1:       33.1 +   62.5 =   95.5 Mbit/s
  uno por cada spine:   95.6 +   95.6 =  191.2 Mbit/s
  los dos por s1:       53.1 +   42.5 =   95.6 Mbit/s
  uno por cada spine:   95.6 +   95.6 =  191.2 Mbit/s
  los dos por s1:       47.3 +   48.0 =   95.4 Mbit/s
  uno por cada spine:   95.4 +   95.4 =  190.9 Mbit/s
```

```
¿qué tan seguido chocan?

$ bin/ecmp_sim elephant 2 2 200000
... sin choques en 50.1 % de los ensayos; caminos ocupados en promedio 1.50; throughput total 75 % del ideal
$ bin/ecmp_sim elephant 8 8 200000
... sin choques en 0.2 % de los ensayos; caminos ocupados en promedio 5.25; throughput total 66 % del ideal
```

Salidas: sobrar capacidad, flowlets, mirar la carga.

SECCIÓN 04

# Fallas

04

# Apagar un spine: ¿avisa o no avisa?

`flow_probe`: 16 flujos UDP, un datagrama cada 20 ms, un solo sentido.

carrier: ip link set down en s1

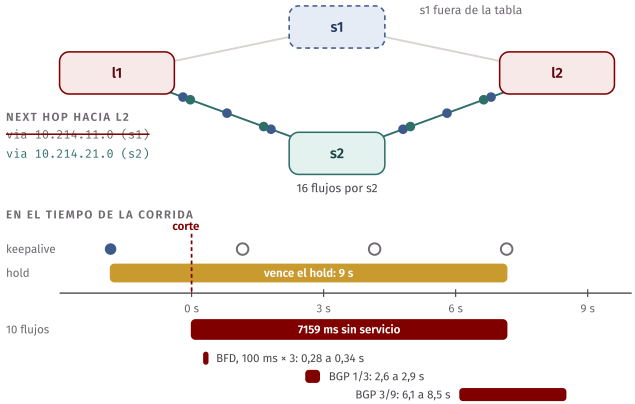
```
flujos: 16; con corte: 0; sin corte: 16
hueco máximo en los flujos sin corte: 34 ms
flujos: 16; con corte: 0; sin corte: 16
hueco máximo en los flujos sin corte: 29 ms
```

silent: netem loss 100 %, 3/9

```
flujos: 16; con corte: 10; sin corte: 6
hueco en los flujos con corte: 7159 a 7159 ms
hueco máximo en los flujos sin corte: 32 ms
flujos: 16; con corte: 6; sin corte: 10
hueco en los flujos con corte: 8500 a 8500 ms
hueco máximo en los flujos sin corte: 30 ms
```

- **No le pega a todos.** Se ve como un servicio que “a veces” se cuelga.
- Medimos un sentido. TCP usa los dos, y cada leaf reparte por su cuenta: con dos spines se afectan **3 de cada 4** conexiones.
- El tiempo es el hold: entre hold − keepalive y hold.

# La falla silenciosa, en el tiempo de una corrida




Según la edad del último keepalive, el corte cae entre 6 y 9 s: 7,2; 8,3 y 8,5 s en tres corridas.
Cuando BGP se entera, la tabla se corrige en milisegundos. Lo que separa 8 s de 0,3 s es solo cómo se detecta.

# Comparar con la semana 5

| Falla | Protocolo y temporizadores | Corte |
|---|---|---|
| Interfaz abajo | BGP, 3/9 | hueco de 29 a 34 ms |
| Silenciosa | BGP, 3/9 | 6 a 8,5 s |
| Silenciosa | BGP, 1/3 | 2,6 a 2,9 s |
| Silenciosa | BFD, 100 ms $\times$ 3 | 0,28 a 0,34 s |
| Silenciosa | BGP, 60/180 | 20 a 42 s (lo resuelve ARP; BGP, a unos 180 s) |
| Interfaz abajo | OSPF, semana 5 | 0 |
| Silenciosa | OSPF, 10/40, semana 5 | 35 a 38 s |
| Silenciosa | OSPF, 2/8, semana 5 | 7 a 8 s |

**Idea clave**

**Lo que tarda es enterarse.** Y apagar una interfaz para “probar” da números demasiado buenos.

SECCIÓN 05

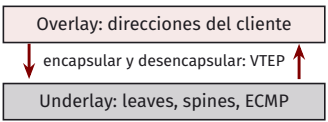
# Overlays

05

# Una red virtual encima de la física

**Overlay**

Red virtual cuyos paquetes viajan encapsulados dentro de los de otra red, el **underlay**, que los transporta sin mirar adentro.

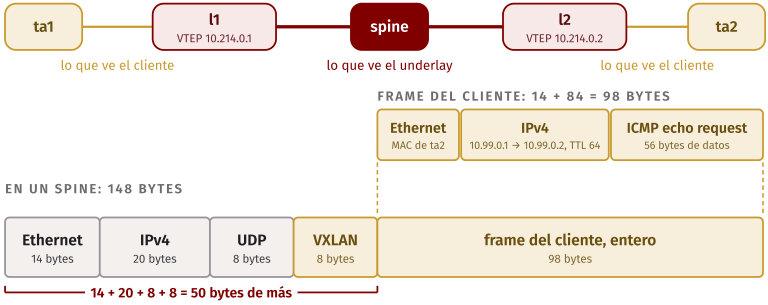



- **VPC:** sus direcciones solo existen en su red virtual.
- **Kubernetes:** los pods tienen direcciones que el underlay no conoce.

**La idea**

La dirección interna dice **quién es**. La externa dice **dónde está**.

# VXLAN: 50 bytes y un VNI




**Ethernet**
de salto en salto en el underlay

**IPv4**
de VTEP a VTEP

`10.214.0.1 → 10.214.0.2`

**UDP**
destino 4789; el origen sale de un hash

`55211 → 4789`

**VXLAN**
VNI, 24 bits: a qué red virtual pertenece

`VNI 100`

**Frame interno**
el underlay no lo mira; la MAC interna es la de ta2

tb1: igual, con VNI 200

Mismas direcciones en los dos clientes: lo único que separa sus paquetes es el VNI.

# VXLAN: las piezas

- **VTEP:** quien encapsula. Puede ser un leaf o un hipervisor.
- **VNI**, 24 bits: unos 16 millones de redes virtuales. Una VLAN tiene 12 bits, unas 4000.
- **UDP** porque el underlay ya sabe hacer ECMP con UDP. Puerto de destino 4789.
- **Aprender:** a cada MAC remota se le asocia la IP del VTEP. Se inunda y se escucha, o lo dice un controlador o BGP (EVPN).

# Dos clientes, las mismas direcciones

cada ping encuentra la MAC de su cliente

```
ta1$ ping -c 2 10.99.0.2     2 packets transmitted, 2 packets received, 0% packet loss
tb1$ ping -c 2 10.99.0.2     2 packets transmitted, 2 packets received, 0% packet loss

ta1$ ip neigh                10.99.0.2 dev eth1 lladdr aa:c1:ab:1c:27:7a REACHABLE
ta2$ cat .../eth1/address    aa:c1:ab:1c:27:7a
tb1$ ip neigh                10.99.0.2 dev eth1 lladdr aa:c1:ab:24:d6:6b REACHABLE
tb2$ cat .../eth1/address    aa:c1:ab:24:d6:6b

l1$ ip route show 10.99.0.0/24
(nada)
```

el bridge del VTEP: en cada MAC, también el VTEP remoto

```
l1$ bridge fdb show dev vx100
00:00:00:00:00:00 dst 10.214.0.2 self permanent
aa:c1:ab:1c:27:7a dst 10.214.0.2 self
```

Salidas resumidas. `l1` no tiene ruta: para él, esto es capa 2.

# El frame interno, en una captura

tshark -V en un spine, recortado

```
Frame 1: 148 bytes on wire (1184 bits), 148 bytes captured (1184 bits)
    [Protocols in frame: eth:ethertype:ip:udp:vxlan:eth:ethertype:ip:icmp:data]
Ethernet II, Src: aa:c1:ab:4f:e4:e4 (aa:c1:ab:4f:e4:e4), Dst: aa:c1:ab:a1:b2:b0 (aa:c1:ab:a1:b2:b0)
Internet Protocol Version 4, Src: 10.214.0.1, Dst: 10.214.0.2
    Time to Live: 64
User Datagram Protocol, Src Port: 55211, Dst Port: 4789
Virtual eXtensible Local Area Network
    VXLAN Network Identifier (VNI): 100
Ethernet II, Src: aa:c1:ab:e5:49:0f (aa:c1:ab:e5:49:0f), Dst: aa:c1:ab:1c:27:7a (aa:c1:ab:1c:27:7a)
Internet Protocol Version 4, Src: 10.99.0.1, Dst: 10.99.0.2
    Total Length: 84
Internet Control Message Protocol
```

- Frame interno: 14 + 84 = 98 bytes. Externo: 148. Los **50** del dibujo.
- Para el cliente, `ta2` está en su misma LAN.
- El de `tb1` es igual, salvo el VNI: 200.

# MTU: lo que cuestan 50 bytes

```
python3 scenarios.py mtu: underlay 1500, VXLAN 1450, cliente 1500

df_ping 1422 (paquete de 1450 bytes): pasa
df_ping 1423 (paquete de 1451 bytes): no pasa
df_ping 1472 (paquete de 1500 bytes): no pasa
TCP de ta1 a ta2: 0 Mbit/s
```

- El ping funciona. El handshake funciona. La conexión se abre. No pasa ni un byte.
- Cada lado anuncia MSS 1460. El bridge no cabe, descarta y no avisa: es capa 2.

**Error común**

“El ping anda, así que la red está bien.” Un ping de 56 bytes no prueba nada de la MTU.

# Dos salidas

### 1. Cliente con MTU 1450

tenant_mtu

```
df_ping 1423 ...: no pasa
TCP de ta1 a ta2: 4806 Mbit/s
```

Muchas nubes la entregan por DHCP.

### 2. Underlay con más MTU

jumbo: 1550 en los enlaces

```
df_ping 1472 ...: pasa
TCP de ta1 a ta2: 5232 Mbit/s
```

El cliente no se entera. Es lo que pide la RFC 7348: los VTEP no fragmentan.

**Con VXLAN en 1500, en Linux**

El VTEP le manda ICMP al cliente, que aprende `mtu 1450`. Es una cortesía de esta implementación: no cuenten con eso en un switch físico.

# Entropía: para qué sirve el puerto de origen

Afuera, todo lo de un VTEP a otro se ve igual: dos IP, UDP, 4789. ¿Se pierde ECMP?

```
python3 scenarios.py entropy: 16 flujos en un túnel de h1 a h3, política 3 en l1

puerto de origen externo según el flujo interno (49152 a 65535):
s1 hacia l2:    4170.9 MB   26.8 %
s2 hacia l2:   11388.5 MB   73.2 %
puerto de origen externo fijo (50000):
s1 hacia l2:   10708.8 MB  100.0 %
s2 hacia l2:       0.0 MB    0.0 %
```

**Idea clave**

El puerto de origen externo sale de un hash del frame interno. Le copia al underlay la identidad del flujo. Sin eso, una red virtual entera es un solo flujo enorme.

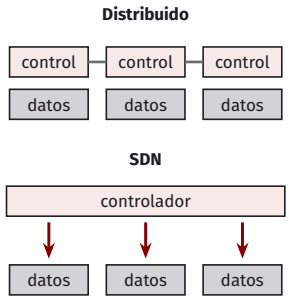
SECCIÓN 06

# SDN

06



# SDN: sacar la decisión del router




**SDN**

Plano de control separado de los equipos y **centralizado lógicamente**. Tiene vista global y programa las tablas.

"Lógicamente" no es una sola computadora: es un grupo replicado.

# Match-action: el forwarding generalizado

| Match | Acción | Se parece a |
|---|---|---|
| IP de destino en `10.0.0.0/8` | Enviar por el puerto 2 | Router |
| MAC de destino `aa:bb:...` | Enviar por el puerto 3 | Switch |
| TCP, puerto de destino 22 | Descartar | Firewall |
| IP de origen `10.1.0.0/16` | Reescribir y enviar | NAT |

**Forwarding generalizado**

Cada regla tiene un match sobre cualquier campo, una acción y una prioridad. Decide la primera que coincide. OpenFlow (2008) popularizó la idea.

# Un controlador de juguete

```
python3 sdn_ctl.py install policy.txt, y algo de tráfico: bytes por spine

5201 6153.14823459805 {'s1': 1609839048, 's2': 151}
5202 6071.025746016996 {'s1': 85, 's2': 1588739288}
```

```
lo que queda en l1: una tabla match-action con contadores

chain pre {
        ip saddr 10.214.1.20 tcp dport 5201 counter packets 2 bytes 120 drop
        tcp dport 5201 counter packets 1068718 bytes 1594876911 meta mark set 0x00000001 accept
        tcp dport 5202 counter packets 1575156 bytes 2350774917 meta mark set 0x00000002 accept
}
```

Elegir camino por puerto no se puede con una tabla que solo mira el destino.

# Lo que se gana y lo que se arriesga

Regla “5201 por `s1`” instalada, y `s1` se corta en silencio. BGP hace su trabajo:

```
l1$ ip route show 10.214.2.0/24
10.214.2.0/24 nhid 18 via 10.214.21.0 dev eth2 proto bgp metric 20
h1$ iperf3 -c 10.214.2.10 -p 5201 -t 1 --connect-timeout 1500
iperf3: error - unable to connect to server - server may have stopped running or ...
```

**Se gana**

- Vista global.
- Cambiar como se cambia un programa.
- Sin controlador, los equipos siguen.

**Se arriesga**

- La regla le gana a BGP.
- Sin controlador, nadie repara.
- ¿Y si la falla los separa? (Facebook, semana 7)

# SDN no quiere decir “sin routing distribuido”

**Error común**

Creer que con SDN ya no hay routing distribuido. Es una forma de organizar el plano de control. Muchos data centers usan BGP y controladores a la vez.

**Caso: Orion, de Google (NSDI 2021)**

Control plane de segunda generación de sus redes de data center (Jupiter), de campus y de larga distancia (B4).

- Microservicios en torno a una base de datos central, el NIB.
- Tiene que interoperar: ninguna red SDN vive aislada.
- Versiones cada dos semanas. Reportan hasta 40 veces en tiempo de convergencia, y 50 y 100 veces en disponibilidad. Son cifras de Google.

SECCIÓN 07

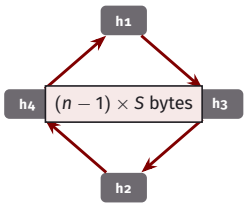
# Redes para IA

07

# Entrenar: todos esperan a todos

1. Cada GPU calcula sobre su parte.
2. **Todas intercambian resultados.** Nadie sigue antes de que termine.
3. Vuelta al paso 1.

> **All-reduce**
>
> Cada nodo tiene un vector. Al final, todos tienen la suma, elemento por elemento.




En un anillo, cada nodo envía lo último que recibió y va sumando. `ring_reduce.c` envía y recibe a la vez con `poll`: si todos hacen `send` y después `recv`, el anillo se traba.

# Un enlace lento atrasa a todos

```
python3 scenarios.py collective: 1 MB, 4 nodos, h4 con salida a 100 Mbit/s

  h1: mediana sin atraso     9.1 ms; con h4 a 100 Mbit/s   253.0 ms
  h3: mediana sin atraso     9.3 ms; con h4 a 100 Mbit/s   252.6 ms
  h2: mediana sin atraso     9.2 ms; con h4 a 100 Mbit/s   243.9 ms
  h4: mediana sin atraso     9.5 ms; con h4 a 100 Mbit/s   247.6 ms
```

- `h3` ni envía a `h4` ni recibe de él, y va igual de lento.
- Predicción: 3 MB $\times$ 8 / 100 Mbit/s = 0,24 s. Medimos 0,25 s.

> **Idea clave**
>
> La velocidad del trabajo completo es la del enlace más lento.

# Importa la peor latencia, no el promedio

tail_rounds.py: 1 ms normal, 100 ms con 1 % de probabilidad

```
10 trabajadores, 1.00 % lentos: promedio por trabajador 2.0 ms; promedio por ronda 10.4 ms; rondas lentas 9.5 %
100 trabajadores, 1.00 % lentos: promedio por trabajador 2.0 ms; promedio por ronda 63.8 ms; rondas lentas 63.4 %
1000 trabajadores, 1.00 % lentos: promedio por trabajador 2.0 ms; promedio por ronda 100.0 ms; rondas lentas 100.0 %
```

- Exacto: fracción de rondas lentas $= 1 - (1 - p)^N$.
- El promedio por trabajador es excelente. La ronda es 50 veces peor.
- Pocos flujos, enormes, entre las mismas direcciones: el peor caso para ECMP (8 elefantes y 8 caminos: un tercio de capacidad perdida).
- Por eso: reparto por paquete con transportes que toleran el desorden, RDMA y una red casi sin descartes, y el patrón elegido según la topología.

SECCIÓN 08

# Sin cables

08

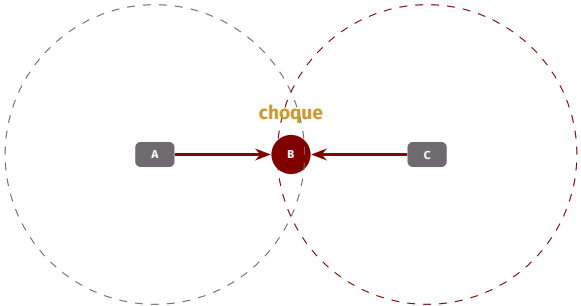
# Qué cambia sin cables

- **Atenuación:** en espacio libre, la potencia cae con el cuadrado de la distancia. Adentro, más rápido.
- **Multitrayecto:** la señal rebota y las copias se suman o se cancelan.
- **Ruido e interferencia:** el aire lo comparten todos.
- Todo se resume en la **SNR**: a mayor SNR, menos errores por bit. Por eso la radio **cambia de velocidad** según la señal.

**La consecuencia**

La tasa de error es mucho mayor que en un cable, y cambia de un segundo a otro.

# Terminal oculto y terminal expuesto




- **Oculto:** A y C llegan a B, pero no se oyen. Los dos escuchan silencio, transmiten y chocan en B.
- CSMA responde “¿hay silencio **aquí**?”. La pregunta correcta es “¿hay silencio **en el receptor**?”.
- **Expuesto:** C oye una transmisión de B hacia otro lado y se calla sin necesidad. No hay pérdida: hay desperdicio.

# Wi-Fi: asociarse y CSMA/CA

### Asociarse con el AP

1. Oír un beacon (SSID, BSSID), o preguntar.
2. Elegir un AP y asociarse.
3. Autenticarse, si se pide.
4. DHCP: ya es una LAN.

### CSMA/CA

1. Canal libre (DIFS): transmitir.
2. Ocupado: backoff al azar. Se cuenta solo con el canal libre.
3. Enviar el frame **entero**. No se aborta.
4. El receptor manda ACK tras un SIFS, más corto: nadie se adelanta.
5. Sin ACK: se duplica la ventana y se reintenta.

> **¿Por qué no detectar colisiones?**
>
> Una radio no oye más que a sí misma. Y en el terminal oculto, el choque ocurre en el receptor. Por eso cada colisión cuesta el frame completo.

# Un modelo de CSMA/CA

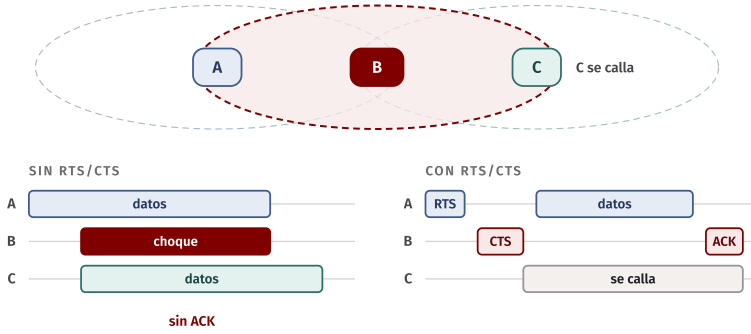
```
dcf_sim: sirve para comparar, no para predecir megabits

  1 estaciones que se oyen: tiempo útil  95.2 %; intentos que chocan   0.0 %
  5 estaciones que se oyen: tiempo útil  83.7 %; intentos que chocan  26.5 %
 20 estaciones que se oyen: tiempo útil  71.5 %; intentos que chocan  45.8 %
 50 estaciones que se oyen: tiempo útil  62.9 %; intentos que chocan  57.2 %

2 estaciones ocultas, sin RTS/CTS: tiempo útil  49.8 %; intentos que chocan  55.0 %
2 estaciones ocultas, con RTS/CTS: tiempo útil  92.0 %; intentos que chocan  11.3 %
```

- Con más estaciones chocan más, pero el tiempo útil baja gradualmente: la ventana que se duplica apaga la pelea.
- Dos ocultas: sin RTS/CTS se desperdicia la mitad del tiempo.

# RTS/CTS: reservar en el receptor




B confirma con un ACK. Si chocan dos RTS se pierde solo un frame muy corto.
Con el modelo y dos estaciones ocultas: tiempo útil de 49,8 % sin RTS/CTS y de 92,0 % con él.

Si chocan dos RTS, se pierde un frame corto. Cuesta dos frames más: se activa solo sobre cierto tamaño.

# ¿Por qué Wi-Fi usa ACK si TCP ya los tiene?

- **Tasa de error.** Si el enlace pierde el 10 % y lo intenta hasta 3 veces en total, se escapa $0{,}1^3 = 0{,}1\,\%$.
- **Tiempo.** Wi-Fi reintenta en micro o milisegundos. TCP se entera al menos un RTT después.
- **Malentendido.** TCP cree que una pérdida es congestión.

**Argumento end-to-end**

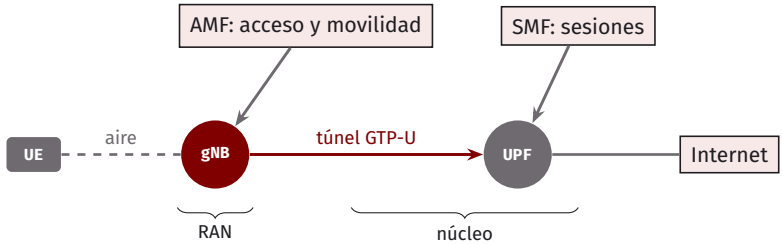
El extremo debe garantizar la entrega. Aun así, a la capa baja le conviene ayudar.

SECCIÓN 09

# 5G

09

# 5G: RAN y núcleo




- Control separado de datos. AUSF y UDM saben quién es cada suscriptor.
- **Distinto de Wi-Fi:** la torre reserva tiempo y frecuencia. No se compite.
- **Sesión PDU:** el SMF elige un UPF, da una IP y programa el túnel.
- Es **un overlay**: la IP de adentro dice quién es, la de afuera dónde está.

# Por qué la red se volvió software

- Las funciones del núcleo son **funciones de red**, que se ofrecen servicios entre sí y corren en servidores comunes.
- **Control y datos separados:** la idea de SDN, en celular. Se ponen UPF cerca del usuario sin mover el control.
- **Network slicing:** sobre la misma infraestructura, redes lógicas con garantías distintas.
- **La RAN también:** parte de su procesamiento pasa a servidores generales.

**Error común**

Creer que 5G es solo una radio más rápida. El cambio de arquitectura está en el núcleo.

# Movilidad: ¿qué tiene que cambiar?

**Wi-Fi, misma subred:** se conserva la IP. El AP nuevo envía un frame con la MAC de la estación como origen, y los switches la aprenden de nuevo. En otra subred, la conexión TCP se rompe.

**5G, handover:** el teléfono informa sus mediciones; la torre decide; la vieja reenvía a la nueva; el SMF mueve el extremo del túnel. El **UPF es el ancla** de la IP.

| | **Wi-Fi** | **5G** |
|---|---|---|
| Quién decide el cambio | La estación | La red, con mediciones del teléfono |
| Acceso al medio | CSMA/CA, compitiendo | Reservas de tiempo y frecuencia |
| Espectro | Sin licencia | Con licencia, del operador |
| Se conserva la IP | En la misma subred | Sí |

# TCP sobre un enlace que pierde

```
python3 scenarios.py wireless: emulado con netem, 100 Mbit/s, 40 ms de RTT, CUBIC

sin pérdida                          94.5 Mbit/s      96 retransmisiones
0,1 % de pérdida                     29.5 Mbit/s      18 retransmisiones
1 % de pérdida                        6.4 Mbit/s      43 retransmisiones
sin pérdida, con 4 ms de jitter      94.4 Mbit/s      34 retransmisiones
```

- Se pierde 1 de cada 100 paquetes, y TCP se queda con menos del 8 % del enlace.
- Mathis, de la semana 11: unos 3,5 Mbit/s con 1 %. CUBIC rinde algo más.
- La última fila imita un enlace que repara: un reintento es un paquete que demora.
- Salidas: recuperar en el enlace (lo común), partir la conexión, o no usar la pérdida como señal.

SECCIÓN 10

# Laboratorio

10

# Laboratorio: una red leaf-spine

1. **BGP:** completar `lab/l3-bgp.conf` y cargarlo con `scenarios.py l3`.
2. **Next hop:** `ip route` y `show bgp` en `l1`.
3. **Flujos:** `policy0`, varias veces. ¿Cuánto por cada spine?
4. **Hash:** `policy1`, `routeget`, `elephants`, `predict`.
5. **Apagar un spine:** `carrier`, `silent`, `silent_short`, `silent_bfd`. Comparar con la semana 5.
6. **VXLAN:** completar `lab/vxlan.sh`, y hallar el frame interno en una captura.
7. Avanzar con **P5**.

**No olviden**

`scenarios.py reset` entre pasos. `make test` levanta la red, comprueba lo que dicen los apuntes y destruye todo. Tarda unos seis minutos.

# Resumen

- Leaf-spine: muchos caminos iguales. Sobresuscripción: se cuenta capacidad.
- ECMP reparte por flujo para no desordenar. El hash cuenta flujos, no bytes.
- Lo que tarda en una falla es enterarse, y solo les pega a los flujos de ese camino.
- Overlay: la dirección interna dice quién es, la externa dónde está. VXLAN cuesta 50 bytes.
- SDN: control separado y centralizado lógicamente, con match-action. Convive con BGP.
- Trabajo colectivo: importa la peor latencia.
- Wi-Fi: CSMA/CA, backoff y ACK. 5G: reservas, control separado y túneles.
- TCP confunde pérdida inalámbrica con congestión; el enlace la esconde.

# Para la próxima clase

- Terminar el laboratorio y avanzar con **P5**, que se entrega la próxima semana.
- Próxima semana: seguridad de redes (firewalls, IPsec, WireGuard) y presentación de proyectos.
- Resolver los ejercicios de los apuntes.
- Lecturas: Kurose y Ross, secs. 4.4, 5.5, 6.5.2, 6.6 y 7.1 a 7.5.

**Pregunta**

Una aplicación en la nube abre conexiones sin problema, hace `ping` sin problema, pero se cuelga al bajar archivos grandes, y solo para algunos clientes. Con lo de esta semana: ¿qué dos explicaciones distintas revisarían primero, y cómo las distinguirían?