

Computación Centrada en Redes

Principios de diseño y sockets

Fundamentos

Gerson Garrido

Semana 2

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

Sesión A: principios de diseño

1. Best effort: lo que IP promete
2. El argumento end-to-end
3. El reloj de arena y fate sharing
4. De ARPANET a ATM y Frame Relay
5. La IETF y los RFC

Sesión B: sockets en C

6. Procesos, puertos y orden de bytes
7. Un servidor y un cliente con TCP
8. `recv` y `send` no respetan mensajes
9. UDP
10. Varios clientes: `poll`

Lectura

Sesión A: Kurose y Ross, secs. 1.7 y 4.5; CS168, *Network Architecture*; Saltzer, Reed y Clark, *End-to-End Arguments in System Design*, secciones 1 y 2.
Sesión B: Kurose y Ross, secs. 2.1 y 2.6.1–2.6.2; Beej, *Guide to Network Programming*: *What is a Socket?*, *IP Addresses, structs, and Data Munging*, *System Calls or Bust* y *Client-Server Background*.

SECCIÓN 01

Best effort

01

El contrato de IP

Servicio best effort

La red intenta llevar cada datagrama a su destino, sin ninguna garantía sobre si llega, cuándo llega, en qué orden o cuántas copias llegan.

Puede pasar	Por qué
Se pierde	Una cola se llenó o un enlace cayó
Tarda mucho	Colas largas; el retardo no tiene un máximo
Llega al revés	Caminos distintos
Llega dos veces	Una capa de abajo lo retransmitió
Llega con bits cambiados	IPv4 revisa solo su cabecera

¿Quién compraría un servicio así?

Es fácil de cumplir

- Corre sobre casi cualquier enlace
- No exige circuitos ni reservas
- Une redes de dueños distintos

No decide por las aplicaciones

- La voz prefiere perder a llegar tarde
- Un archivo prefiere esperar a perder
- Un juego solo quiere el último dato

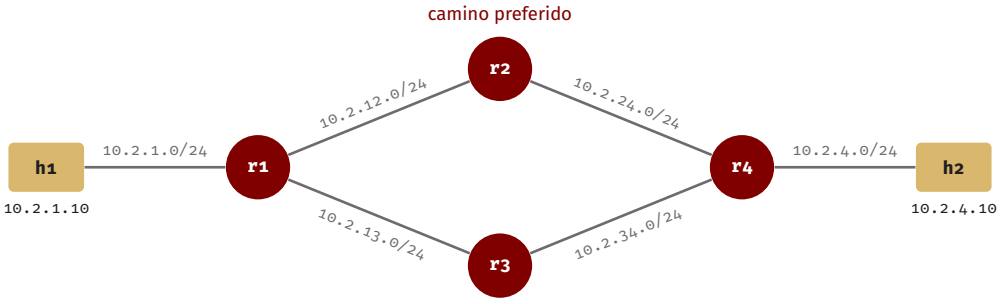
Comparen con la red telefónica

Núcleo elaborado y teléfono simple: un servicio nuevo requiere cambiar el núcleo. En Internet basta con instalar un programa en dos computadoras.

Idea clave

La inteligencia está en los extremos, y los extremos son de ustedes.

La topología de la semana



- `code/week-02/fate.clab.yml`, con MTU de 1500 bytes. Se levanta con `make deploy`.
- OSPF llena las tablas: por ahora es una caja negra (semana 5).
- Empeoramos `r1–r2` con `tc netem`: retardo variable y 10 % de pérdida.

La misma red, dos transportes

UDP: casi el servicio de IP tal cual

```
seq_recv en h2

recibidos: 903 de 1000
perdidos: 97
fuera de orden: 210
repetidos: 0
```

Nadie avisó. Otras corridas: 109 y 106.

TCP, sobre el mismo enlace

```
chunk_recv en h2

recv: 22 llamadas, 102400 bytes, menor 1040, mayor 20272
bytes fuera de lugar: 0
```

Todo, y en orden.

> **El reparto**
> Ningún router sabe qué es una retransmisión de TCP: TCP corre **solo en los dos hosts**.

SECCIÓN 02

End-to-end

02

El argumento end-to-end

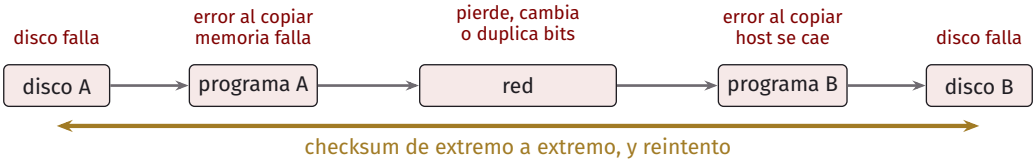
Saltzer, Reed y Clark (1981, revisado en 1984)

Una función que solo se puede hacer completa y correctamente con el conocimiento de la aplicación en los extremos no se puede ofrecer como una función de la red. La red puede dar una versión incompleta, pero solo como **mejora de desempeño**.

Ojo

Es un argumento, no un teorema: no prohíbe nada. Pide una pregunta antes de poner una función en la red: ¿de todas formas la tendrán que hacer los extremos?

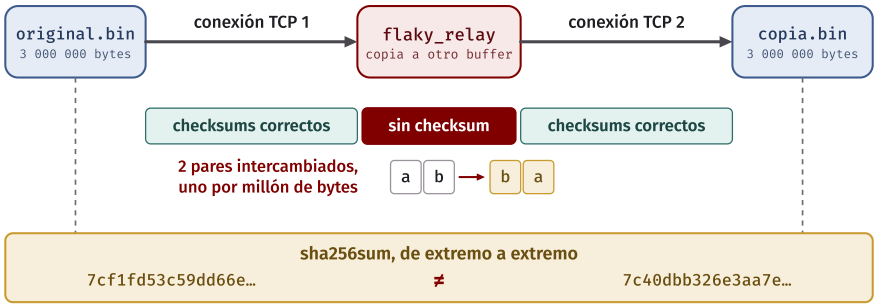
El caso: copiar un archivo con cuidado



- Una red perfecta elimina **una** amenaza de cinco.
- La comprobación de los extremos hace falta igual, y además cubre a la red.
- Lo que la red haga solo cambia **cuántas veces hay que reintentar**.

Lo reproducimos: el gateway de MIT

Un gateway con checksums en cada salto intercambiaba un par de bytes, una vez por millón. Los datos no estaban protegidos *dentro* del gateway.



Lo que dice el hash

archivo de 3 MB, dentro del contenedor

```
copiados 3000000 bytes, 2 pares intercambiados
$ sha256sum /tmp/original.bin /tmp/copia.bin
7cf1fd53c59dd66e65eb4aa9acb5eadfc6feae65cf5624410 8d79d8a0d8e1b94  /tmp/original.bin
7c40dbb326e3aa7e4a0b1e2ba754d529706f15dd1a870fa97b18fc695959d774  /tmp/copia.bin
```

Error común

“Uso TCP, así que mis datos llegan intactos.” TCP protege un tramo: por eso existen los SHA-256 de las descargas.

¿Entonces la red no debe ayudar?

Un archivo de N paquetes; cada uno se pierde con probabilidad p. Si los extremos solo reintentan el archivo completo:

$$\frac{1}{(1-p)^N} \overset{N=1000,\ p=0{,}01}{\approx} 23\,000 \text{ intentos}$$

- **TCP**: reintenta por pedazos. Sigue en los extremos.
- **Wi-Fi**: retransmite en el enlace, en milisegundos.

Si Wi-Fi fallara

Nada se rompe: solo va más lento. TCP sigue a cargo de la corrección.

El costo de ponerla abajo

Todas las aplicaciones la pagan, aunque no la necesiten. A la voz le llega tarde.

No es solo confiabilidad

Función	¿Por qué en los extremos?	Ayuda en la red
Entrega confiable	Solo el extremo sabe si llegó bien a su destino final	Retransmitir en un enlace malo
Cifrado	Si la red cifra, la red ve los datos	WPA, solo en ese tramo
Eliminar duplicados	Un extremo también duplica, si reintenta tras una caída	No hace falta
Confirmar la entrega	Llegar al host no es que la aplicación lo procesó	No hace falta

Un ACK de TCP

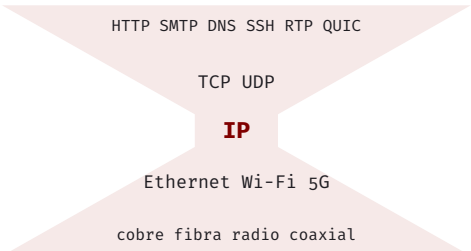
Dice que el kernel de B recibió los bytes. No dice que el programa los leyó, ni que el pago se registró.

SECCIÓN 03

Reloj de arena y estado

03

El reloj de arena



Narrow waist

Un solo protocolo en el centro. Todo el que quiera estar en Internet habla IP.

Con A aplicaciones y L enlaces: sin centro común, $A \times L$; con IP, $A + L$. Con 20 y 10: **200 contra 30**.

El precio

Cambiar la cintura es carísimo: mírenlo en IPv6.

Fate sharing: ¿dónde vive el estado?

Clark (1988)

Guardar el estado de una comunicación en los extremos que la usan, de modo que solo se pierda si muere la propia entidad que lo necesita.

	Router IP	Host
Routing y forwarding	Una entrada por **prefijo**	Una tabla pequeña
Estado de la conexión	**Ninguno**	Secuencia, ventana, buffers
Si se pierde	Se reconstruye con los vecinos	Se pierde la conexión

¿Qué sabe cada nodo de una conexión activa?

durante un envío de chunk_send

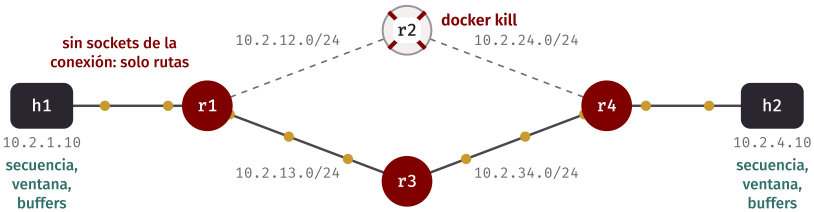
```
$ docker exec clab-w02-fate-h1 ss -tn
State Recv-Q Send-Q Local Address:Port  Peer Address:Port
ESTAB 0      0          10.2.1.10:58496    10.2.4.10:9000
$ docker exec clab-w02-fate-r1 ss -tn
State Recv-Q Send-Q Local Address:Port Peer Address:Port
```

`r1` está reenviando cada paquete de esa conexión y no tiene ningún socket que la describa. Todo lo que tiene es esto:

r1

```
$ docker exec clab-w02-fate-r1 ip route show 10.2.4.0/24
10.2.4.0/24 nhid 25 via 10.2.12.2 dev eth2 proto ospf metric 20
```

Matamos el router, a la mitad de la conexión




```
r1$ ip route show 10.2.4.0/24
10.2.4.0/24 nhid 27 via 10.2.13.3 dev eth3 proto ospf metric 20
```

intervalo (s)	0–1	1–2	2–3	3–4	4–5	5–6	6–7	7–8
Mbit/s	200	200	200	199	200	200	200	199
Retr	1	0	0	0	271	0	0	0
Cwnd (KB)	349	349	349	349	287	287	287	287

La misma conexión, los mismos sockets, la misma secuencia: el router muerto no tenía nada que restaurar.

La conexión siguió

iperf3 en h1

```
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   3.00-4.00   sec  23.8 MBytes   199 Mbits/sec    0    349 KBytes
[  5]   4.00-5.00   sec  23.9 MBytes   200 Mbits/sec  271    287 KBytes
[  5]   5.00-6.00   sec  23.9 MBytes   200 Mbits/sec    0    287 KBytes
```

Al morir `r2`, TCP reenvió 271 segmentos, con los mismos sockets y la misma secuencia. Las cifras dependen de la máquina.

¿Y si muere un **extremo**? Matamos el proceso receptor en `h2`:

h1

```
$ docker exec clab-w02-fate-h1 /lab/chunk_send -g 50 10.2.4.10 9003 1000 200
send: Broken pipe
```

Una máquina que se cuelga sin avisar

Con una regla de `nft`, `h2` descarta todo lo que entra. `chunk_send` sigue enviando:

seis segundos entre una consulta y otra

```
$ docker exec clab-w02-fate-h1 ss -tno dst 10.2.4.10
State Recv-Q Send-Q Local Address:Port  Peer Address:Port
ESTAB 0      40000      10.2.1.10:56400    10.2.4.10:9004 timer:(on,1.003ms,3)
$ docker exec clab-w02-fate-h1 ss -tno dst 10.2.4.10
State Recv-Q Send-Q Local Address:Port  Peer Address:Port
ESTAB 0      71056      10.2.1.10:56400    10.2.4.10:9004 timer:(on,4.665ms,5)
```

- Lo que no se ha confirmado crece, y el último número de `timer` cuenta retransmisiones.
- `send` sigue devolviendo éxito: solo significa “lo copié a un buffer local”.
- TCP tardará minutos en rendirse. Si necesitan saberlo antes, pongan sus propios plazos.

Excepciones: NAT, firewalls con estado y load balancers sí guardan estado por conexión. Si se reinician, cortan conexiones sanas (semanas 4 y 15).

SECCIÓN 04

Historia

04

De ARPANET a la Web

1961–65	Kleinrock, Baran y Davies: conmutación de paquetes
1969	ARPANET: UCLA, SRI, UCSB y Utah. Se publica el RFC 1
1974	Cerf y Kahn: redes distintas, unidas por gateways simples
1978	TCP se separa en TCP e IP: nace el espacio para UDP
1983	ARPANET pasa de NCP a TCP/IP el 1 de enero
1985–86	NSFNET conecta universidades
1989–91	Berners-Lee crea la Web, sin pedirle nada a la red
1995	NSFNET deja su red troncal: Internet es comercial

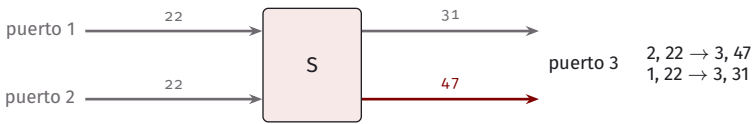
La fila de 1978

Es el argumento end-to-end antes de tener nombre: quienes probaban con voz no querían las retransmisiones de TCP.

Circuitos virtuales: ATM y Frame Relay

Circuito virtual

Un camino fijo, establecido antes de enviar datos. Cada switch guarda una entrada por circuito, y cada paquete lleva una etiqueta corta que se reescribe en cada salto.



La etiqueta 22 aparece dos veces porque solo tiene significado en su enlace.

Datagrama y circuito virtual

	Datagrama de IP	Circuito virtual
Antes de enviar	Nada	Señalización en todos los switches
Cada paquete lleva	Dirección completa	Etiqueta corta
Estado intermedio	Por prefijo	**Una entrada por circuito**
Búsqueda	Longest prefix match	Índice exacto
Orden	No garantizado	Mismo camino, en orden
Calidad garantizada	Difícil	Natural: se reserva al establecer
Cae un equipo intermedio	Toman otro camino	Se cae el circuito

La última fila

Es el experimento de `r2` en su otra versión. Con circuitos, la red guarda estado de cada conversación, y comparte su destino con ella: no hay fate sharing.

Celdas de 53 bytes



Por qué

- 48: compromiso entre 64 (América) y 32 (Europa).
- Voz a 64 kbit/s: llenar una celda toma $48 \times 8/64\,000 = 6$ ms.
- Tamaño fijo: switches rápidos en hardware.

Qué costó

- *Cell tax*: $5/53 \approx 9{,}4$ % en cabeceras.
- Un paquete de 1500 bytes: más de 30 celdas. Se pierde una, y el paquete es inútil.

Frame Relay: frames variables, circuitos contratados (DLCI), sin corrección en cada salto, a diferencia de X.25.

Por qué ATM y Frame Relay cedieron ante IP

- **Cada circuito cuesta en la red.** No tiene sentido uno por pedido de una página.
- **La Web llegó sin pedir permiso.** Su tráfico no necesitaba las garantías que ATM cobraba.
- **IP corre sobre cualquier cosa**, también sobre ATM. Y siguió cuando ATM se fue.
- **Ethernet creció más rápido y más barato**, sin celdas ni circuitos.
- **Las razones técnicas envejecieron.** A gigabits, un paquete largo tarda microsegundos.

Hoy

Frame Relay está retirado o en retiro. ATM sobrevivió en accesos como ADSL. La idea de las etiquetas sigue en **MPLS**, debajo de IP: se usa para ingeniería de tráfico y VPNs, dentro de un operador. IP sigue en el medio.

SECCIÓN 05

La IETF y los RFC

05



Quién define Internet

- **IETF**: abierta, sin membresía formal. Consenso aproximado y código que funciona.
- **IEEE**: Ethernet, Wi-Fi. **W3C** y **WHATWG**: la Web. **ICANN/IANA**: números y nombres.
- Un **RFC** publicado nunca cambia: se reemplaza.

MUST: obligatorio. **SHOULD:** salvo buena razón. **MAY:** opcional (RFC 2119 y 8174).

encabezado de RFC 9293

```
Internet Engineering Task Force (IETF)        W. Eddy, Ed.
STD: 7                                         MTI Systems
Request for Comments: 9293                     August 2022
Obsoletes: 793, 879, 2873, 6093, 6429, 6528,
           6691
Updates: 1011, 1122, 5961
Category: Standards Track
ISSN: 2070-1721
```

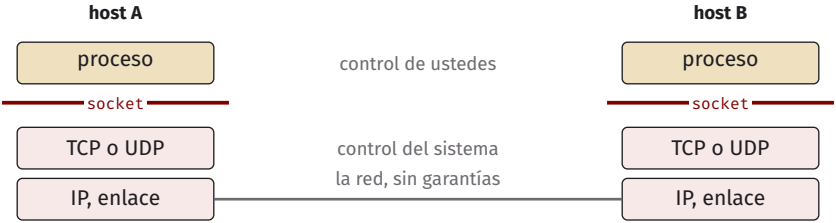
- `Obsoletes`: lo reemplaza. `Updates`: cambia una parte.
- `Category`: Informational no es un estándar.
- Antes de confiar en un RFC, miren si ya lo reemplazaron.

SECCIÓN 06

Procesos, puertos y bytes

06

El socket: la frontera



Socket

La interfaz entre un proceso y el transporte. Para el programa es un descriptor: un entero sobre el que llama a `send` y `recv`.

Un **puerto**, de 16 bits, identifica un socket dentro de un host. Una conexión TCP se identifica con cuatro números: dirección y puerto de cada extremo.

Qué le pide una aplicación al transporte

Necesidad	Quién	TCP	UDP
Confiabilidad	Archivos, Web	Sí: todo, en orden, una vez	No
Throughput	Video	Sin garantía	Sin garantía
Tiempo	Voz, juegos	Sin garantía; una retransmisión agrega retardo	Sin garantía; nada lo retrasa a propósito
Seguridad	Casi todo	No: TLS arriba	No: se agrega arriba

Es el best effort visto desde arriba

Una videollamada funciona porque la aplicación se adapta, no porque la red le prometa algo.

El orden de los bytes

- x86-64 y ARM actuales: **little-endian**.
- Los protocolos de Internet: **big-endian**, u *orden de red*.
- `htons`, `ntohs`: 16 bits.
  `htonl`, `ntohl`: 32 bits.
- Sin `htons`, el puerto 9002 (`0x232a`) se vuelve el 10787 (`0x2a23`). Sin error alguno.

byte_order en una máquina little-endian

```
$ build/byte_order
9002 en memoria:       2a 23
htons(9002):           23 2a
0x0a020110 en memoria: 10 01 02 0a
htonl(0x0a020110):     0a 02 01 10
sizeof(struct wrong_header) = 4, aunque sus campos suman 3
cabecera en 3 bytes:   07 23 2a
```

Error común

Copiar una `struct` a la red arrastra el relleno del compilador. Un formato se define byte a byte.

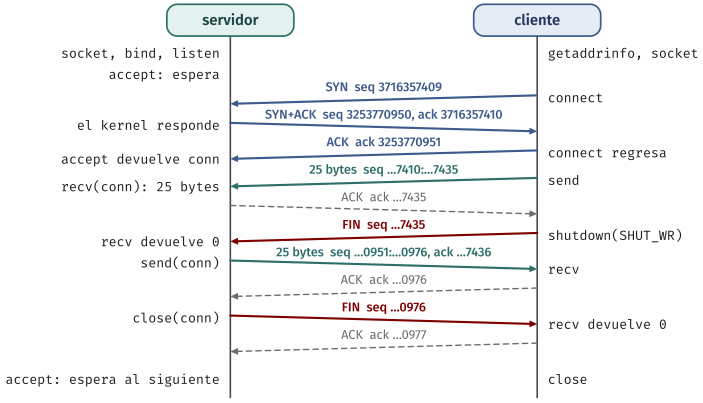
SECCIÓN 07

TCP en C

07



Un servicio de conteo: la secuencia de llamadas



close cierra la otra dirección, y el servidor vuelve a esperar. Diez segmentos para 25 bytes de ida y 25 de vuelta.

El cliente envía un texto; el servidor responde con bytes, palabras y líneas, como `wc`.

El servidor: preparar el socket

wc_server.c

```c
// 1. Un socket TCP sobre IPv4.
int listener = socket(AF_INET,
                      SOCK_STREAM, 0);
if (listener == -1) {
    perror("socket");
    return EXIT_FAILURE;
}
// 4. Marcarlo como pasivo: desde
// aquí el kernel completa handshakes.
if (listen(listener, 16) == -1) {
    perror("listen");
    return EXIT_FAILURE;
}
```

wc_server.c

```c
// 3. A un puerto, en todas las interfaces.
struct sockaddr_in addr;
memset(&addr, 0, sizeof addr);
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr.sin_port = htons(port);
if (bind(listener, (struct sockaddr *)&addr,
         sizeof addr) == -1) {
    perror("bind");
    return EXIT_FAILURE;
}
```

El paso 2 es `SO_REUSEADDR`, y lo vemos al final. `listen` convierte el socket en pasivo: ya no sirve para enviar, sino para recibir conexiones.

`accept` devuelve un socket nuevo

wc_server.c

```c
// 5. Un socket nuevo por cada conexión; listener sigue escuchando.
for (;;) {
    struct sockaddr_in peer;
    socklen_t peer_len = sizeof peer;
    int conn = accept(listener, (struct sockaddr *)&peer, &peer_len);
```

mientras atiende a un cliente, en la misma máquina

```
$ ss -tan "( sport = :9302 or dport = :9302 )"
State  Recv-Q Send-Q Local Address:Port  Peer Address:Port
LISTEN 0      16           0.0.0.0:9302       0.0.0.0:*
ESTAB  0      0          127.0.0.1:9302     127.0.0.1:58098
ESTAB  0      0          127.0.0.1:58098    127.0.0.1:9302
```

Mismo puerto local: lo distingue el otro extremo. El **kernel** hace el handshake; `accept` solo recoge el resultado de la cola.

Lo que devuelve `recv`

Devuelve	Significa	Qué hacer
`n > 0`	Hay `n` bytes; pueden ser menos de los pedidos	Procesar y seguir
`0`	El otro extremo cerró: **no habrá más datos**	Terminar
`-1`, `EINTR`	Una señal interrumpió la llamada	Repetir
`-1`, `ECONNRESET`	El otro extremo abortó	Cerrar e informar
`-1`, otro	Otro error	Informar y cerrar

0 no es un error

Es el final normal de un stream. Tratarlo como “nada todavía” es un bucle infinito que consume toda la CPU.

El cliente: `getaddrinfo`

wc_client.c

```c
hints.ai_family = AF_UNSPEC;      // IPv4 o IPv6
hints.ai_socktype = SOCK_STREAM;  // TCP
int err = getaddrinfo(host, port, &hints, &results);
if (err != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(err));
    return -1;
}
for (ai = results; ai != NULL; ai = ai->ai_next) {
    fd = socket(ai->ai_family, ai->ai_socktype,
                ai->ai_protocol);
    ...
    if (connect(fd, ai->ai_addr, ai->ai_addrlen) == 0) {
        break;
    }
    ...
}
```

- Una **lista**: se prueban en orden.
- No usa `errno`: por eso `gai_strerror`.
- El cliente no hace `bind`: el kernel elige un puerto.
- `shutdown(fd, SHUT_WR)`: cierra solo la dirección de envío, y envía un FIN.

El handshake, en una captura

tcpdump -i lo -nn -S 'tcp port 9202', solo puertos, flags, seq y length

```
.57616 > .9202: Flags [S], seq 3716357409, length 0
.9202 > .57616: Flags [S.], seq 3253770950, ack 3716357410, length 0
.57616 > .9202: Flags [.], ack 3253770951, length 0
.57616 > .9202: Flags [P.], seq 3716357410:3716357435, ack 3253770951, length 25
.9202 > .57616: Flags [.], ack 3716357435, length 0
.57616 > .9202: Flags [F.], seq 3716357435, ack 3253770951, length 0
.9202 > .57616: Flags [P.], seq 3253770951:3253770976, ack 3716357436, length 25
.57616 > .9202: Flags [.], ack 3253770976, length 0
.9202 > .57616: Flags [F.], seq 3253770976, ack 3716357436, length 0
.57616 > .9202: Flags [.], ack 3253770977, length 0
```

- Líneas 1 a 3: **handshake**. SYN, SYN+ACK, ACK. Sin datos. El ACK suma 1 a la secuencia.
- Línea 4: los 25 bytes. Línea 6: el FIN de `shutdown`, que cierra *una* dirección.
- Línea 7: el servidor responde después del FIN. Líneas 9 y 10: se cierra la otra dirección.
- MSS anunciado: 65 495 en `lo`; 1460 en la topología (1500 − 20 − 20).

SECCIÓN 08

Stream y framing

08



Error común: `recv` no devuelve un mensaje

Stream de bytes: los mismos bytes en el mismo orden, pero un `send` no corresponde a un `recv`.

Cinco `send` de 100 bytes, seguidos:

```
chunk_recv -v

recv 1: 500 bytes
recv: 1 llamadas, 500 bytes, menor 500, mayor 500
bytes fuera de lugar: 0
```

Con 50 ms entre uno y otro:

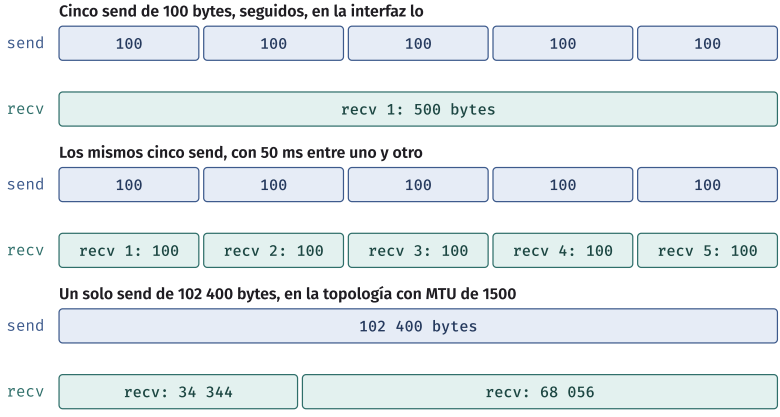
```
chunk_recv -v

recv 1: 100 bytes
recv 2: 100 bytes
recv 3: 100 bytes
recv 4: 100 bytes
recv 5: 100 bytes
```

Esta es la trampa

Un `send`, un `recv`: parece que TCP conserva mensajes. Pasa las pruebas en la laptop, y falla bajo carga.

Los cortes no son de quien envía



Un send grande también se parte, y otra corrida da 5 recv. El programa tiene que funcionar con cualquier reparto.

Un solo `send` de 102 400 bytes

Situación	Llamadas a `recv`	Menor	Mayor
Interfaz `lo`	1	102 400	102 400
Topología, MTU de 1500 bytes	2	34 344	68 056
Topología, otra corrida	5	7 240	40 544
Topología, 10 % de pérdida	22	1 040	20 272
Topología, 10 % de pérdida, otra corrida	25	1 448	23 168

- Mediciones reales, con un buffer de 1 MiB. Dependen de la máquina y del momento.
- Si un cliente envía 3000 bytes con un `send`: de **1 a 3000** llamadas a `recv`.
- Con un buffer de 4096 bytes, el mínimo es $102\,400/4096 = 25$, sin importar cómo lleguen.
- Hasta contar palabras se afecta: `hola mu` + `ndo`. `wc_update` guarda estado.

Framing: poner las fronteras

Método	Dónde termina	Ejemplo	Riesgo
Cerrar	`recv` devuelve 0	HTTP/1.0, `wc_server`	Un mensaje por conexión; si el emisor muere, parece completo
Tamaño fijo	Cada k bytes	Celdas de ATM	Todos deben medir igual
Delimitador	Un byte, como `\n`	SMTP, `tally_server`	Hay que escapar el delimitador
Largo al inicio	Lo dice la cabecera	TLS, HTTP/2	Hay que validar el largo

Idea clave

Cualquier cosa que analice un stream debe poder detenerse en cualquier byte y seguir después. Si el protocolo necesita mensajes, los pone **su programa**.

Leer exactamente *n* bytes

io_util.c

```c
ssize_t recv_all(int fd, void *buf, size_t n) {
    char *p = buf;
    size_t got = 0;
    while (got < n) {
        ssize_t r = recv(fd, p + got, n - got, 0);
        if (r == -1) {
            ...  // EINTR se repite; otro error
            return -1;
        }
        if (r == 0) {
            break;  // el otro extremo cerró
        }
        got += (size_t)r;
    }
    return (ssize_t)got;
}
```

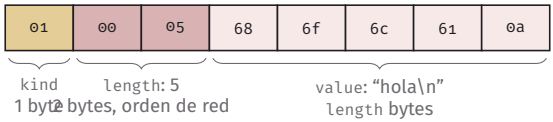
Con `-x 100`, registros de tamaño fijo. Mismo envío que llegó pegado:

chunk_recv -v -x 100

```
registro 1: 100 bytes
registro 2: 100 bytes
registro 3: 100 bytes
registro 4: 100 bytes
registro 5: 100 bytes
```

Si devuelve menos de `n`, el otro extremo cerró a la mitad: no es un mensaje completo.

Un formato con largo al inicio: TLV



1. Leer la cabecera con `recv_all`, convertir con `ntohs`, y leer `value`.
2. **El tamaño de `length` fija el máximo**: con 2 bytes, 65 535.
3. **El largo viene de la red: no se le cree.** Con 4 bytes, alguien anuncia 4 GiB. Se valida antes de reservar memoria.
4. **Se define qué pasa si algo no calza**: un `kind` desconocido, un cliente que se va a la mitad.

Error común: `send` también escribe menos

`send` solo **copia** al buffer del kernel. Un receptor que nunca lee, y se piden 8 MiB:

build/short_send: el número depende del sistema

```
no bloqueante, 1.ª llamada: se pidieron 8388608 bytes y send devuelve 2625195
no bloqueante, 2.ª llamada: send devuelve -1, errno = Resource temporarily unavailable
bloqueante con SO_SNDTIMEO: se pidieron 8388608 bytes y send devuelve 2625195
```

io_util.c

```c
while (left > 0) {
    ssize_t sent = send(fd, p, left, MSG_NOSIGNAL);
    if (sent == -1) {
        ...  // EINTR se repite; otro error: return -1
    }
    p += sent;
    left -= (size_t)sent;
}
```

- Bloqueante: suele esperar a copiar todo, salvo una señal o un plazo.
- Sin `MSG_NOSIGNAL`, una conexión cerrada genera `SIGPIPE`, que mata al proceso.
- Que `send` tenga éxito no significa que los datos llegaron.

SECCIÓN 09

UDP

09



UDP: sin conexión

time_server.c

```c
// No hay listen ni accept: un solo socket recibe de todos los clientes.
for (;;) {
    ...
    ssize_t n = recvfrom(fd, request, sizeof request, 0,
                         (struct sockaddr *)&peer, &peer_len);
    ...
    // Cada datagrama lleva su destino: la dirección de quien preguntó.
    if (sendto(fd, reply, strlen(reply), 0,
               (struct sockaddr *)&peer, peer_len) == -1) {
        perror("sendto");
    }
}
```

- Sin `accept`: sabe a quién responder porque cada datagrama trae su origen.
- **UDP sí conserva las fronteras**: un `sendto` es un datagrama, y `recvfrom` lo devuelve entero.

Con el servidor apagado

sin connect

```
$ build/time_client -w 500 127.0.0.1 9207
enviados 5 bytes
sin respuesta después de 500 ms
```

con connect

```
$ build/time_client -c -w 500 127.0.0.1 9207
enviados 5 bytes
recv: Connection refused
```

lo que viajó en las dos corridas: la red hizo lo mismo

```
IP 127.0.0.1.59814 > 127.0.0.1.9207: UDP, length 5
IP 127.0.0.1 > 127.0.0.1: ICMP 127.0.0.1 udp port 9207 unreachable, length 41
IP 127.0.0.1.63717 > 127.0.0.1.9207: UDP, length 5
IP 127.0.0.1 > 127.0.0.1: ICMP 127.0.0.1 udp port 9207 unreachable, length 41
```

- **Sin `connect`**: el socket habla con muchos, y la API no diría a cuál se refiere. Linux descarta el error.
- **Con `connect`**: un solo destino. El error llega en la *siguiente* llamada.

Error común: `connect` no da confiabilidad a UDP: no envía nada, y muchos firewalls filtran el ICMP.

TCP y UDP

	TCP	UDP
Unidad	Stream de bytes	Datagrama
Fronteras de mensaje	No	Sí
Conexión	Handshake; un socket por cliente	No; un socket para todos
Pérdida	La oculta, retransmitiendo	La deja pasar
Orden	Garantizado	No
Control de congestión	Sí	Responsabilidad del programa
Plazo para esperar	Opcional	Casi siempre hace falta

¿Por qué elegir UDP? DNS se ahorra un RTT. La voz prefiere perder a llegar tarde. QUIC construye su propio transporte sin tocar kernel ni routers. En P3, `rimay` hará lo mismo.

SECCIÓN 10

Varios clientes

10

Un cliente lento detiene a todos

`wc_server` atiende a uno a la vez. A se queda conectado 3 segundos; mientras, B envía `hola`:

wc_server: un cliente a la vez

```
$ (sleep 3) | nc -N 127.0.0.1 9302 &
cliente 127.0.0.1:55196
$ time (echo hola | build/wc_client 127.0.0.1 9302)
bytes=0 words=0 lines=0
cliente 127.0.0.1:55206
bytes=5 words=1 lines=1
real    0m2.698s
```

El `connect` de B regresó de inmediato: esperó en la cola de `accept`. Con un proceso por cliente:

wc_fork_server

```
$ time (echo hola | build/wc_client 127.0.0.1 9302)
bytes=5 words=1 lines=1
real    0m0.002s
```

Un solo hilo con `poll`

`poll`
Recibe un arreglo de descriptores y se bloquea hasta que alguno tenga algo. Marca en cada `revents` lo que ocurrió.

tally_server.c

```c
// Lo que el servidor recuerda de cada conexión.
struct client {
    char line[MAX_LINE];
    size_t used;
};
```

Sin pila por cliente, si llega `SU`, hay que guardarlo y volver a `poll`: es una **máquina de estados**.

dos clientes a la vez

```
$ (echo 'SUMA 5'; sleep 1; echo TOTAL; sleep 1) \
    | nc -N 127.0.0.1 9302       # cliente A
total 5
total 15
$ (sleep 0.5; echo 'SUMA 10'; sleep 1) \
    | nc -N 127.0.0.1 9302       # cliente B
total 15
```

Un total común, sin mutex: hay un solo hilo.

El ciclo

tally_server.c

```c
for (;;) {
    // Dormir hasta que algún socket tenga algo.
    if (poll(fds, count, -1) == -1) { ... }
    // De atrás hacia adelante: quitar un cliente mueve al último a su lugar.
    for (nfds_t i = count; i-- > 1;) {
        if ((fds[i].revents & (POLLIN | POLLERR | POLLHUP)) == 0) { continue; }
        ssize_t n = recv(fds[i].fd, buf, sizeof buf, 0);
        if (n <= 0 || handle_bytes(fds[i].fd, &clients[i], buf, (size_t)n) == -1) {
            close_checked(fds[i].fd);
            count--;
            fds[i] = fds[count];
            clients[i] = clients[count];
        }
    }
    // fds[0], el socket que escucha: POLLIN es una conexión nueva.
}
```

Cada `recv` lee **una vez**, lo que haya, y regresa.

El peligro: un cliente que no lee

tally_server.c

```c
static int reply_to(int fd, const char *text) {
    size_t len = strlen(text);
    ssize_t sent = send(fd, text, len, MSG_DONTWAIT | MSG_NOSIGNAL);
    ...
    return (size_t)sent == len ? 0 : -1;     // -1: hay que cerrar
}
```

- Un `send` normal con el buffer lleno se bloquearía: detiene el único hilo.
- El marcador corta al cliente: sirve para respuestas de pocos bytes.
- Otra política: guardar lo pendiente y pedir `POLLOUT`. Es la que necesita P0.

Error común

Con `poll`, cualquier llamada que se bloquee detiene a todos: `recv_all`, un `send` bloqueante, un `sleep`.

Las tres opciones

	Un proceso	Un hilo	Un hilo con `poll`
Estado del cliente	Pila y memoria	Pila del hilo	Una estructura propia
Costo por cliente	Alto	Medio	Lo que ocupe su estructura
Estado común	Difícil	Con mutex	Sin mutex
Un error de memoria	Afecta a uno	Afecta a todos	Afecta a todos
Varios núcleos	Sí	Sí	No, salvo varios procesos
Una llamada bloqueada	Afecta a uno	Afecta a uno	**Detiene a todos**

En la práctica

Con miles de conexiones se usa `epoll` o `kqueue`, con la misma idea. Nginx y Redis siguen este modelo; PostgreSQL usa un proceso por conexión.

Address already in use

reuse_demo cierra primero y termina

```
$ build/reuse_demo -r 9222 &
escuchando en el puerto 9222
$ nc -d 127.0.0.1 9222
chau
$ ss -tan state time-wait '( sport = :9222 )'
Recv-Q Send-Q Local Address:Port Peer Address:Port
0      0          127.0.0.1:9222    127.0.0.1:57682
$ build/reuse_demo 9222
bind: Address already in use
$ build/reuse_demo -r 9222 &
escuchando en el puerto 9222
$ build/reuse_demo -r 9222
bind: Address already in use
```

- Quien cierra primero queda en **TIME_WAIT**: 60 s en Linux.
- `SO_REUSEADDR` permite el `bind` durante ese tiempo.
- **Los dos servidores** deben tenerla: si el anterior no la tuvo, hay que esperar.
- **No** comparte un puerto en uso: la última línea tiene otro servidor escuchando.

`ss -ltnp 'sport = :9222'` muestra qué proceso tiene el puerto.

SECCIÓN 11

Laboratorio

11

Laboratorio: un cliente y un servidor en C

En `code/week-02/`. Se comprueba con `make test-lab`, que falla hasta que terminen.

1. Correr `wc_server` y `wc_client`. Capturar con `tcpdump` y hallar el handshake.
2. Enviar 100 KB y contar cuántas veces devuelve datos `recv`.
3. Definir registros `kind`, `length`, `value`, y atender varios pedidos por conexión.
4. Servicio de conteo con UDP. Con el servidor apagado, ¿por qué el cliente no se entera, salvo con `connect`?
5. Una sala con `poll`: cada línea de un cliente les llega a los demás.
6. Avanzar con **P0**.

> **Sobre P0**
>
> Pide las mismas ideas con otro programa: un formato con largo y `poll` juntos. Allí no sirve `recv_all`: dejaría esperando a los demás.

Para la próxima clase

- Leer Kurose y Ross, secciones 1.2.2 (*Physical Media*), 6.1 y 6.2.
- Terminar el laboratorio y entregar **P0**.
- Hacer los ejercicios del capítulo.

Pregunta

Un cliente envía 3000 bytes con una sola llamada a `send`. ¿Cuántas llamadas a `recv` puede necesitar el servidor? ¿Y si fuera UDP?