

UNIVERSIDAD NACIONAL DE INGENIERÍA

Sistemas Operativos

LFS, SSDs, integridad y sistemas distribuidos

Escribir como un log, desconfiar del disco y compartir archivos por la red

Gerson Garrido

Semana 15

Hoy

Sesión A: persistencia, parte final

1. LFS: un sistema de archivos que solo agrega al final
2. SSDs: flash, FTL y recolección de basura
3. Integridad: errores latentes, corrupción y checksums

Sesión B: sistemas distribuidos

4. Comunicación no confiable, reintentos y RPC
5. NFS: un servidor sin estado
6. AFS: archivos completos y callbacks

> **Lectura**
>
> OSTEP cap. 43–46 (*Log-structured File Systems*, *Flash-based SSDs*, *Data Integrity and Protection*) y cap. 47–51 (*Distributed Systems*, NFS y AFS).

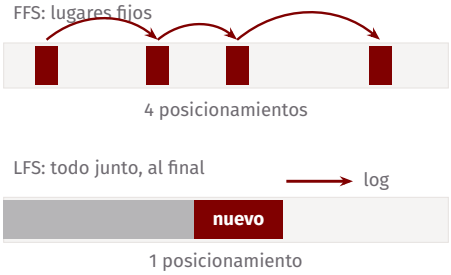
SECCIÓN 01

LFS

01

¿Por qué un sistema de archivos que solo agrega?

- La memoria crece y la caché absorbe casi todas las **lecturas**: al disco llegan, sobre todo, escrituras.
- En disco, lo secuencial es cientos de veces más rápido (semana 12).
- Crear un archivo pequeño en FFS escribe inodo, bitmap, directorio y datos: varias escrituras pequeñas en lugares fijos y distantes.



La idea de LFS (Rosenblum y Ousterhout, 1991)

Juntar *todas* las escrituras en memoria y mandarlas juntas, en una escritura grande y secuencial, **al final del log**. Nunca se sobrescribe en el lugar.

Segmentos: juntar en memoria, escribir de una vez



Cuando el buffer se llena (o pasa un tiempo), el **segmento** completo se escribe al final del log: un solo posicionamiento y luego transferencia pura. Datos e inodos van juntos.

¿Qué tan grande debe ser un segmento?

Escribir D MB cuesta un posicionamiento más la transferencia:

$$T = T_{\text{pos}} + \frac{D}{R_{\text{pico}}} \qquad R_{\text{efectiva}} = \frac{D}{T}$$

Queremos $R_{\text{efectiva}} = F \cdot R_{\text{pico}}$, con $0 < F < 1$. Despejando:

$$D = \frac{F}{1 - F} \cdot R_{\text{pico}} \cdot T_{\text{pos}}$$

Ejemplo

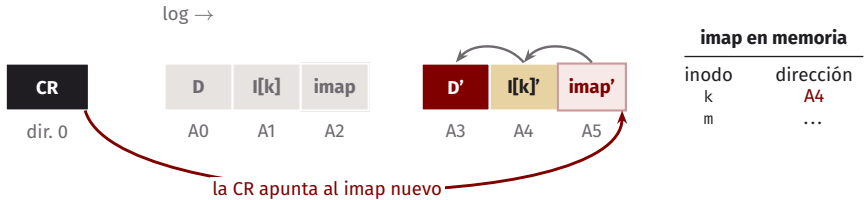
$T_{\text{pos}} = 10$ ms, $R_{\text{pico}} = 100$ MB/s y queremos el 90 % ($F = 0{,}9$):

$D = 9 \times 100 \text{ MB/s} \times 0{,}01 \text{ s} = 9 \text{ MB}$

F	0,90	0,95	0,99
D	9 MB	19 MB	99 MB

Cada punto cerca del 100 % cuesta muchísimo más buffer.

El problema: ¿dónde está ahora el inodo?



- Como nada se sobrescribe, cada versión nueva del inodo cae en otra dirección. El **mapa de inodos** *(inode map, imap)* traduce número de inodo → dirección actual.
- Sobrescribir `D`: se agregan `D'`, el inodo nuevo y el pedazo de imap que cambió. Lo viejo queda muerto.
- La **región de checkpoint** *(checkpoint region, CR)*, en un lugar fijo, apunta a los pedazos del imap. Se actualiza cada tanto (unos 30 s), no en cada escritura.

Leer un archivo en LFS

1. Al montar: leer la CR y, con ella, **todo el imap** a memoria.
2. El directorio da el número de inodo; el imap, su dirección.
3. Leer el inodo. Desde aquí, igual que FFS: punteros directos e indirectos.

Los directorios guardan números

Un directorio guarda (nombre, número de inodo). Si el inodo se mueve, solo cambia el imap y el directorio no se reescribe. Sin imap habría que reescribir la cadena entera hasta la raíz.

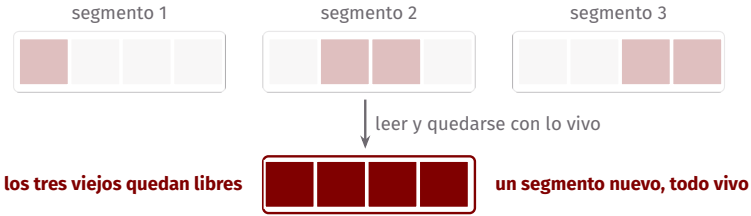
kernel/fs.h

```
1 #define IBLOCK(i, sb) \
2   ((i) / IPB + sb.inodestart)
```

En xv6

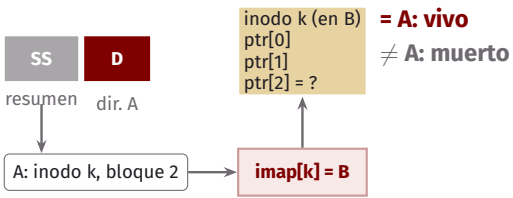
Los inodos están en posiciones **fijas**: su bloque sale de una división. Por eso xv6 (y FFS) no necesitan imap. LFS sí, porque sus inodos se mueven.

Limpieza: recuperar segmentos con basura



- Con el tiempo, los segmentos viejos se llenan de versiones muertas.
- El **limpiador** *(cleaner)* lee M segmentos, copia lo vivo a $N < M$ segmentos nuevos y libera los M viejos. Trabaja con segmentos completos: así siempre hay espacio contiguo para escribir rápido.
- Dos preguntas: ¿cómo sabe qué bloques están vivos? ¿Cuáles conviene limpiar?

¿Este bloque está vivo? El resumen del segmento



1. Cada segmento lleva un **resumen** *(segment summary)*: para cada bloque, de qué inodo y de qué posición es.
2. Buscar ese inodo en el imap.
3. Leer el inodo y mirar su puntero en esa posición.
4. Si apunta a A, el bloque está vivo; si no, es basura.

Atajo: un número de versión en el imap y en el resumen. Si no coinciden (el archivo se borró o truncó), es basura sin leer el inodo.

Qué limpiar, y cómo sobrevivir a un apagón

Política de limpieza

Cuándo: periódicamente, con el disco ocioso o cuando se llena.

Cuáles: separar segmentos *calientes* (se sobrescriben seguido) de *fríos*. Los fríos se limpian antes; a los calientes conviene esperarlos.

Recuperación ante fallos

Dos CR, en los extremos del disco, usadas por turnos. Marcas de tiempo al inicio y al final detectan una CR escrita a medias.

Roll forward: desde la última CR, recorrer el log y recuperar lo escrito después.

La idea sobrevivió

Escribir en un lugar nuevo *(copy-on-write)* está en WAFL, ZFS, btrfs… y dentro de cada SSD.

En xv6: el log no es LFS

kernel/log.c: install_trans()

```
1  for (tail = 0; tail < log.lh.n; tail++) {
2    // ...
3    struct buf *lbuf = bread(log.dev, log.start + tail + 1); // read log block
4    struct buf *dbuf = bread(log.dev, log.lh.block[tail]);   // read dst
5    memmove(dbuf->data, lbuf->data, BSIZE); // copy block to dst
6    bwrite(dbuf);                           // write dst to disk
```

xv6: journal de redo

Los bloques pasan primero por el log (semana 14); tras el commit se copian a su lugar fijo, `log.lh.block[tail]`.

LFS: el log es el sistema de archivos

No hay lugar fijo: los datos se escriben una vez y el imap dice dónde quedaron.

SECCIÓN 02

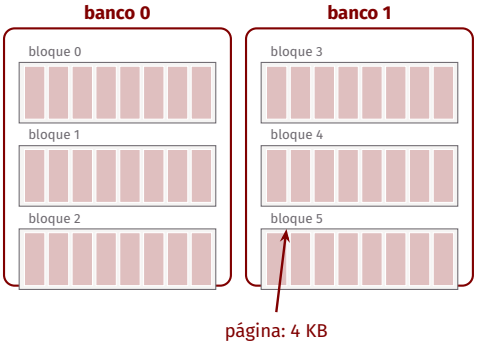
SSDs

02



Memoria flash: bancos, bloques y páginas

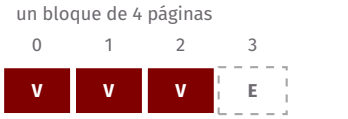
- Cada celda guarda carga: 1 bit (**SLC**), 2 (**MLC**) o 3 (**TLC**). Más bits: más capacidad, menos velocidad y duración.
- Un chip tiene **bancos** que trabajan en paralelo.
- Cada banco tiene **bloques** de 128–256 KB, y cada bloque, **páginas** de 4 KB.
- Ojo: un *bloque* flash es la unidad de borrado, y es grande. No es un bloque del sistema de archivos.



Tres operaciones, y hay que borrar antes de escribir

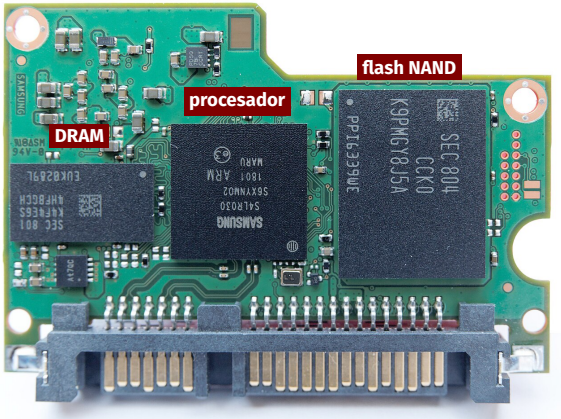
	Leer pág.	**Programar** pág.	**Borrar** bloque
SLC	25 µs	200–300 µs	1,5–2 ms
MLC	50 µs	600–900 µs	∼3 ms
TLC	∼75 µs	∼900–1350 µs	∼4,5 ms

- Leer es rápido y no depende de la posición: no hay brazo.
- **Programar** solo cambia 1s por 0s: la página tiene que estar borrada.
- **Borrar** pone todo el bloque en 1s. Es lento y desgasta: un bloque MLC aguanta unos 10 000 ciclos; uno SLC, unos 100 000.



¿Reescribir la página 0? Hay que borrar el bloque entero: antes, copiar 1 y 2.

Una SSD por dentro



Samsung SSD 860 Pro. Michael Bemmerl, CC BY 3.0 de, Wikimedia Commons.

- **Flash NAND**: los datos. Un solo encapsulado con varios chips apilados.
- **Procesador** (núcleos ARM): ejecuta el FTL *(flash translation layer)*.
- **DRAM**: caché de escrituras y tabla de mapeo del FTL.
- Hacia afuera, la interfaz de siempre: *leer y escribir bloques*.

Un FTL log-structured

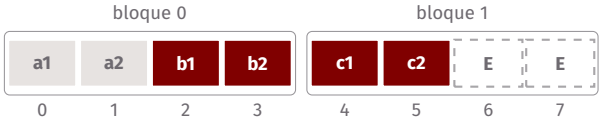


tabla de mapeo

lógica	física
100	4
101	5
2000	2
2001	3

Sobrescribir 100 y 101 con c1 y c2: no se toca la página 0. Lo nuevo va a la siguiente página libre (4 y 5), la tabla se actualiza y 0 y 1 quedan muertas.

La alternativa ingenua: mapeo directo

Página lógica *N* = física *N*. Cada sobrescritura obliga a leer el bloque, borrarlo y reprogramarlo entero: lentísimo, y gasta siempre los mismos bloques.

Recolección de basura



lógica	física
100	4
101	5
2000	6
2001	7

El bloque 0 ya no tiene nada vivo: se **borra** y vuelve al log. Es la limpieza de LFS, dentro del disco.

Amplificación de escritura ***(write amplification)***

WA = páginas programadas en la flash ÷ páginas que pidió escribir el sistema. Aquí el sistema escribió 2 (c1, c2) y la flash programó 4 (c1, c2, b1, b2): WA = 2.

Cuanto más llena, peor: un FTL de juguete

bash

```
$ for u in 50 70 90; do ./ftl $u; done
ocupación 50%: escrituras del host 40960,
               copias del GC 9436
  amplificación de escritura = 1.23
ocupación 70%: escrituras del host 57340,
               copias del GC 48265
  amplificación de escritura = 1.84
ocupación 90%: escrituras del host 73720,
               copias del GC 305283
  amplificación de escritura = 5.14
```

`code/week-15/ftl.c`: 128 bloques de 32 páginas, sobrescrituras al azar; el GC elige el bloque con menos páginas vivas. Salida partida en dos líneas.

¿Por qué?

Con el disco al 90 %, el bloque "más vacío" todavía está casi todo vivo: para liberar una página hay que copiar varias.

- **Sobreaprovisionamiento** *(overprovisioning)*: capacidad extra escondida para que la SSD nunca esté llena.
- **TRIM**: el sistema avisa qué bloques borró, y el GC deja de copiarlos.

¿Cuánto ocupa la tabla de mapeo?

Mapeo por página

SSD de 1 TB, páginas de 4 KB, 4 bytes por entrada:

$$\frac{2^{40}}{2^{12}} = 2^{28} \text{ entradas}$$

$$2^{28} \times 4 \text{ B} = 1 \text{ GB de DRAM}$$

solo para la tabla.

- **Mapeo por bloque**: una entrada por bloque de 256 KB, 64 veces menos. Pero sobrescribir 4 KB obliga a copiar el bloque entero.
- **Híbrido**: pocos *bloques de log* con mapeo por página y el resto por bloque; de vez en cuando se fusionan.
- **Caché de la tabla**: la tabla completa vive en la flash y en DRAM solo lo que se usa. Funciona si hay localidad, como una TLB.

Nivelación de desgaste

- Un bloque con datos **fríos** (que nunca cambian) no tiene basura: el GC nunca lo elige y nunca se borra.
- Los bloques calientes acumulan borrados y se gastan antes.
- **Nivelación de desgaste** *(wear leveling)*: el FTL mueve cada tanto los datos fríos para que esos bloques también entren a la rotación. Cuesta escrituras extra.

bash

```
$ ./ftl 90
  borrados por bloque: mín 80,
                  prom 92.4, máx 104
$ ./ftl 90 hot
  amplificación de escritura = 5.47
  borrados por bloque: mín 71,
                  prom 98.3, máx 124
```

`hot`: el 80 % de las escrituras va al 20 % de los datos. Este FTL no nivela: la brecha entre bloques crece. Salida resumida.

SECCIÓN 03

Integridad

03

Los discos fallan a medias

- El RAID de la semana 13 supone fallas **fail-stop**: el disco funciona o no.
- Pero un disco que “funciona” puede fallar en un solo bloque:
  - **Error latente de sector** *(latent sector error, LSE)*: el sector se dañó. El ECC del disco lo detecta y el disco devuelve un error.
  - **Corrupción silenciosa**: el disco devuelve datos incorrectos sin avisar. Firmware con errores, un bus que voltea bits, escrituras en el lugar equivocado.

	Baratos	Caros
LSE	9,40 %	1,40 %
Corrupción	0,50 %	0,05 %

Discos con al menos un problema en unos 3 años; 1,5 millones de discos (Bairavasundaram et al., 2007–2008). Baratos: SATA; caros: SCSI/FC.

Raro, pero no tanto

Con miles de discos, pasa todas las semanas.

Errores latentes: el disco avisa

- La lectura devuelve error: detectar es fácil.
- Recuperarse usa la redundancia que ya conocen: leer el espejo o reconstruir con la paridad (RAID 4/5).
- Problema: reconstruyendo un disco caído en RAID 4/5, un LSE en otro disco deja ese bloque sin arreglo. Por eso existen esquemas con **dos paridades** (RAID-6, RAID-DP).

kernel/virtio_disk.c

```
1  // en virtio_disk_intr()
2  if (disk.info[id].status != 0)
3    panic("virtio_disk_intr status");
```

En xv6

Si el disco virtual informa un error, xv6 no reintenta ni tiene otra copia: `panic`. La respuesta más simple posible a un LSE.

Checksums: un resumen pequeño de un bloque grande

Un **checksum** *(suma de verificación)* es una función $C(D)$ que da pocos bytes para un bloque D. Si D cambia, casi siempre cambia $C(D)$.

- **XOR** de todos los bytes: barato; no ve dos cambios en el mismo bit.
- **Suma** módulo 256: no ve bytes reordenados.
- **Fletcher**: dos sumas; la segunda acumula la primera y así depende de la posición. Adler-32 es una variante.
- **CRC**: división de polinomios; detecta ráfagas de errores.

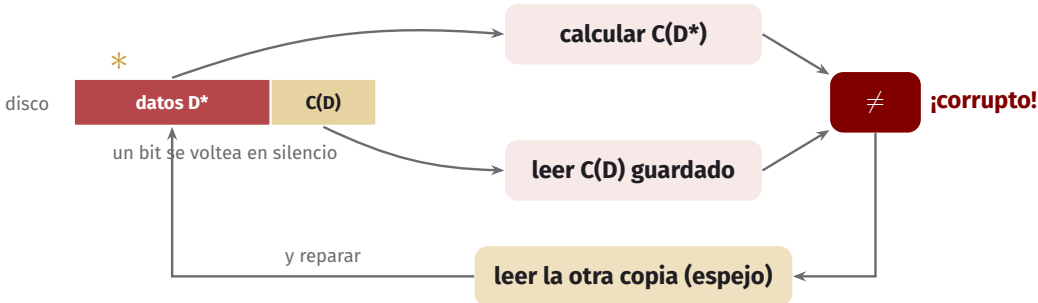
Fletcher de $1, 2, 3, 4$ y de $4, 3, 2, 1$

$s_1 \mathrel{+}= d_i$, $s_2 \mathrel{+}= s_1$ (mód 255)

d_i	s_1	s_2	d_i	s_1	s_2
1	1	1	4	4	4
2	3	4	3	7	11
3	6	10	2	9	20
4	10	20	1	10	30

	$1, 2, 3, 4$	$4, 3, 2, 1$
Suma	10	10
XOR	4	4
Fletcher	(10, 20)	(10, 30)

Un checksum atrapa la corrupción



Al leer, se recalcula el checksum y se compara con el guardado. Si no coinciden, se lee la otra copia y se repara el bloque; sin redundancia, al menos se devuelve un error.

Qué detecta cada checksum

```
bash

$ seq 1 1000 > num.txt
$ ./checksum num.txt 100
2. El mismo bit volteado en dos bytes seguidos
  XOR-8        00000031 -> 00000031  NO DETECTADO
  Fletcher-16  000084b9 -> 000080b9  detectado
  Adler-32     9e0f7a5c -> 9e0b7a5c  detectado
  CRC-32       8dc4565d -> 910792e7  detectado

3. Dos bytes seguidos intercambiados
  XOR-8        00000031 -> 00000031  NO DETECTADO
  Fletcher-16  000084b9 -> 000057b9  detectado
  Adler-32     9e0f7a5c -> 9de27a5c  detectado
  CRC-32       8dc4565d -> d7195ea5  detectado
```

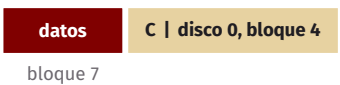
- `checksum.c` daña una copia en memoria y compara. Con un solo bit volteado (caso 1) los cuatro detectan.
- XOR falla cuando dos cambios se cancelan. Los demás dependen de la posición.
- Ninguno es perfecto: hay 2^{32768} bloques de 4 KB y 2^{32} valores de CRC-32. Hay colisiones; se busca que los errores *comunes* no las produzcan.

Escrituras mal dirigidas y escrituras perdidas

Mal dirigida ***(misdirected write)***

Datos correctos en el bloque equivocado. Su checksum viaja con ellos y calza: no se nota.

Solución: guardar también un **identificador físico** (disco, bloque). Si al leer el bloque 7 dice “soy el bloque 4”, está mal.



Perdida ***(lost write)***

El disco dice “listo” pero nunca escribió. Queda el dato *viejo* con su checksum *viejo*: coherentes entre sí, el checksum no lo detecta.

- **Verificar al escribir**: releer lo escrito. Duplica las E/S.
- Guardar el checksum **en otro lugar**: en el inodo o en el bloque que apunta al dato (ZFS). Si la escritura se perdió, el checksum nuevo no calza.

Scrubbing y costos

- Un checksum solo se revisa al **leer**. ¿Y los datos que nadie lee en meses?
- **Scrubbing**: recorrer todos los bloques cada cierto tiempo, verificar y reparar con la copia buena antes de que esa también falle.
- Costo: 8 bytes por bloque de 4 KB (menos del 0,2 %) y leer cada byte.

En xv6

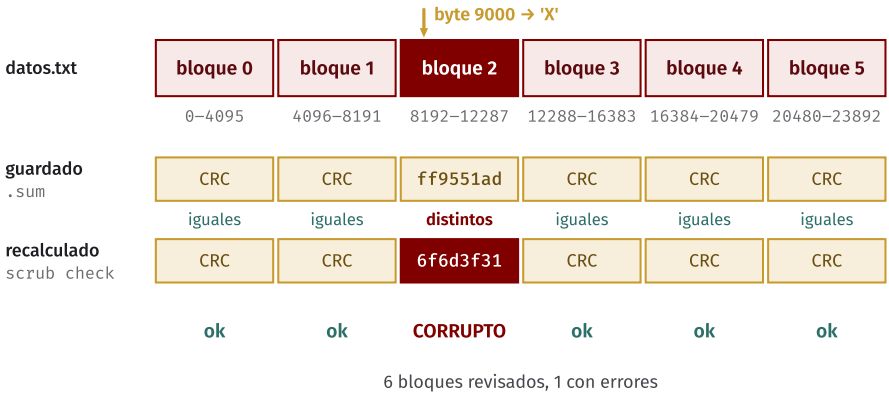
`bread` y `bwrite` (`kernel/bio.c`) no calculan ni guardan checksums: xv6 confía en que el disco devuelve lo que se escribió.

bash

```
$ seq 1 5000 > datos.txt
$ ./scrub sign datos.txt
6 bloques firmados en datos.txt.sum
$ ./scrub check datos.txt
6 bloques revisados, 0 con errores
$ printf 'X' | dd of=datos.txt bs=1 \
    seek=9000 conv=notrunc status=none
$ ./scrub check datos.txt
bloque 2: CORRUPTO (guardado ff9551ad,
                    leído 6f6d3f31)
6 bloques revisados, 1 con errores
```

`scrub.c`: un CRC-32 por bloque de 4 KB, en un archivo aparte. Salida partida en dos líneas.

`scrub check`, bloque por bloque



El byte 9000 cae en el bloque 2 (bytes 8192 a 12287): es el único CRC que no coincide.

SECCIÓN 04

Sistemas distribuidos

04

En un sistema distribuido, fallar es lo normal

- Muchas máquinas que cooperan por la red: la web, el correo, la nube. Con miles de máquinas, siempre hay alguna rota.
- Los mensajes **se pierden**: un router con el buffer lleno descarta paquetes. También se corrompen, se duplican o llegan tarde.
- Las máquinas se caen y reinician; a veces la red se parte en dos.

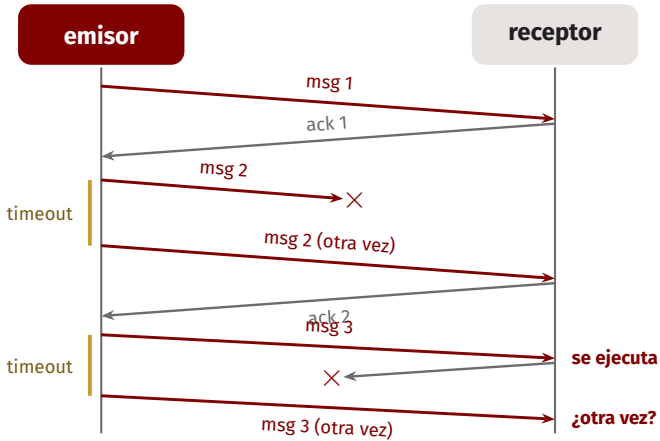
UDP: comunicación no confiable

Enviar un datagrama y olvidarse. Si se pierde, nadie avisa. Trae un checksum: al menos no entrega datos corruptos.

La pregunta de hoy

¿Cómo construir algo **confiable** sobre algo que no lo es? Confirmaciones, timeouts, reintentos y números de secuencia. Es lo que TCP hace por dentro.

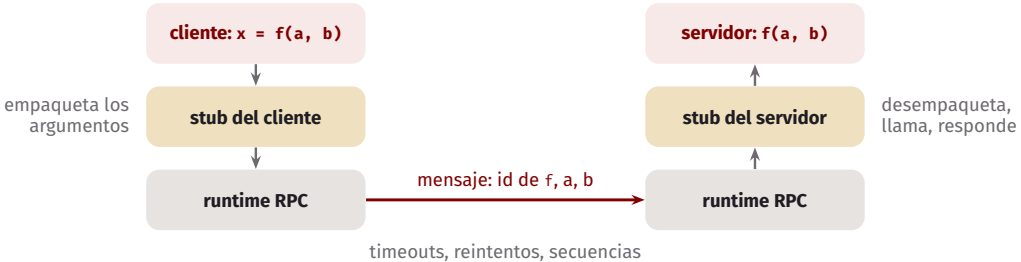
Confirmación, timeout y reintento



El receptor no distingue "el emisor reintenta" de "un mensaje nuevo igual".

Número de secuencia: el receptor recuerda el último que procesó. Si llega uno repetido, no lo vuelve a ejecutar: solo reenvía el ack.

Llamadas a procedimientos remotos (RPC)



- El **stub** lo genera un compilador a partir de la interfaz: empaqueta *(marshaling)* los argumentos, los envía y espera la respuesta.
- El servidor suele atender varias llamadas a la vez con un pool de hilos.
- El **runtime** da la confiabilidad. Si se pierde una respuesta y el cliente reintenta, ¿se ejecuta `f` dos veces?

Probémoslo: udprpc

udprpc.c: cliente

```
for (;;) {
  sendto(fd, m, sizeof(*m), 0, sa, slen);
  struct msg r;
  for (;;) {
    ssize_t n = recv(fd, &r, sizeof(r), 0);
    if (n < 0)
      break;  // timeout: reintentar
    if (n == sizeof(r) && r.seq == m->seq)
      return r.val;
    // respuesta vieja: ignorarla
  }
  (*retries)++;
}
```

udprpc.c: servidor

```
if (lost(&seed))  // pedido perdido
  continue;
// ...
} else if (dedup && m.seq <= last_seq) {
  if (m.seq < last_seq)  // copia atrasada
    continue;
  r = last_reply;  // duplicado: no re-ejecutar
} else {
  counter += m.val;  // NO idempotente
  // ... guardar last_seq y last_reply
}
if (lost(&seed))  // respuesta perdida
  continue;
```

udprpc: ¿cuántas veces se sumó?

```bash
$ ./udprpc 200 20
llamadas ADD: 200  reintentos: 120
contador esperado: 200
contador en el servidor: 253
$ ./udprpc 200 20 dedup
llamadas ADD: 200  reintentos: 120
contador esperado: 200
contador en el servidor: 200
```

`fork()` crea el servidor; hablan por `127.0.0.1`. Se pierde el 20 % de pedidos y respuestas. Salida resumida.

- 120 reintentos: unos por pedidos perdidos (inofensivos) y otros por respuestas perdidas.
- Sin `dedup`, cada respuesta perdida es una suma de más: 53. `ADD` **no es idempotente**.
- Con `dedup`, el servidor recuerda la última secuencia y su respuesta. Funciona, pero ahora el servidor tiene estado: ¿y si se reinicia?

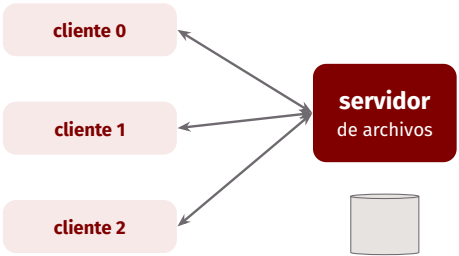
SECCIÓN 05

NFS

05



Un sistema de archivos en la red



- Los programas usan `open`, `read`, `write`. El **FS del cliente**, en el kernel, los convierte en mensajes al servidor.
- Sun NFSv2 es un protocolo abierto: cualquiera puede escribir un cliente o un servidor.
- Meta: que el servidor se recupere de una caída **rápido y sin complicaciones**.

El problema del estado

Si `open` devolviera un descriptor guardado en el servidor, tras una caída `read(fd)` no significaría nada. Y si se cae el cliente, el servidor no sabe cuándo liberar ese estado.

Sin estado: cada pedido trae todo lo que necesita

Mensaje	Lleva
`LOOKUP`	fh del directorio, nombre
`GETATTR`	fh
`READ`	fh, offset, cantidad
`WRITE`	fh, offset, cantidad, datos
`CREATE`, `MKDIR`	fh del directorio, nombre

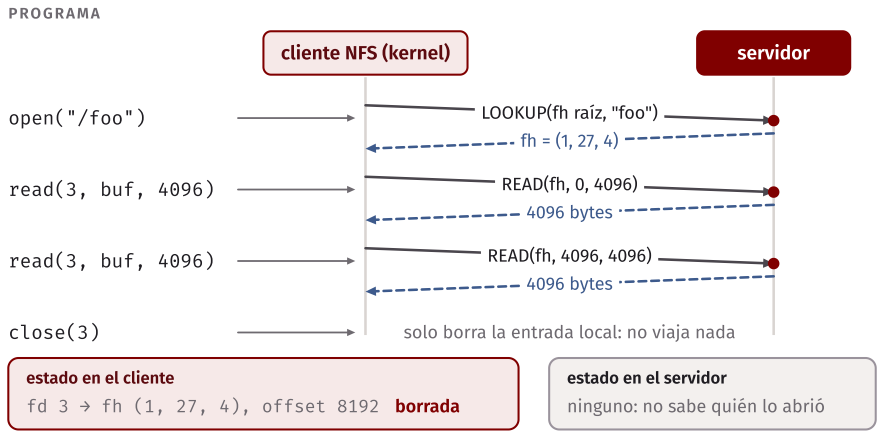
File handle (fh)

(volumen, número de inodo, generación). La generación cambia si el inodo se reutiliza: un fh viejo no abre otro archivo.

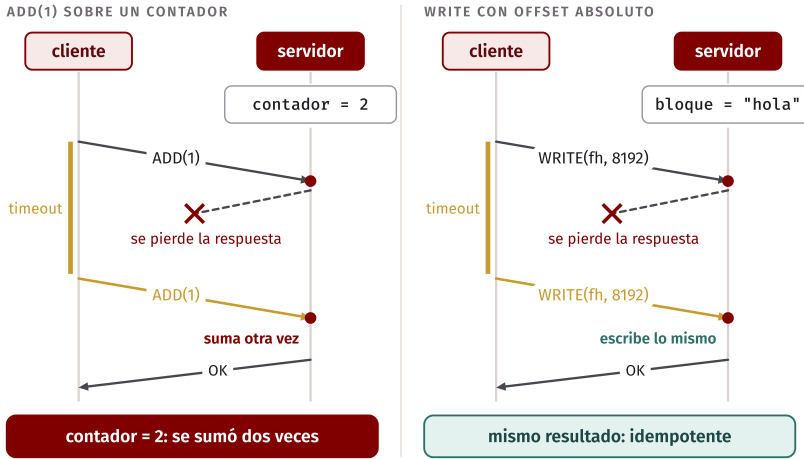
Leer un archivo

1. `open("/foo")`: `LOOKUP(fh raíz, "foo")` devuelve el fh y los atributos.
2. El **cliente** anota: fd 3 $\rightarrow$ (fh, offset 0).
3. `read(3, buf, 4096)`: envía `READ(fh, 0, 4096)` y el cliente avanza su offset.
4. `close(3)`: libera la entrada *local*. El servidor ni se entera.

Leer por NFS: el estado vive en el cliente



Idempotencia: reintentar sin miedo



Un remedio para tres fallas

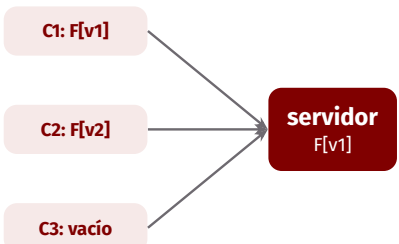
Idempotente

Hacerla dos veces deja lo mismo que hacerla una. `READ`, `WRITE` con offset absoluto, `GETATTR`, `LOOKUP`: sí. `ADD(1)`: no.

Reintentar y listo

Pedido perdido, respuesta perdida o servidor caído: el cliente **reintenta y listo**, sin tabla de secuencias. (Un `MKDIR` repetido responde “ya existe”.)

La caché del cliente y la consistencia

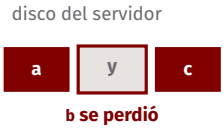


C2 escribió v2, pero sigue en su caché

- El cliente guarda en memoria los bloques leídos y escritos: si no, cada `read` cruza la red.
- **Visibilidad**: C3 lee v1 del servidor aunque C2 ya escribió v2. NFS: flush-on-close; al cerrar, el cliente envía lo que modificó.
- **Caché vencida**: C1 sigue leyendo v1. NFS: antes de usar la caché, `GETATTR` para ver si cambió la fecha de modificación.
- Tantos `GETATTR` saturaban al servidor: se agregó una caché de atributos de unos 3 s. Mientras dura, se pueden leer datos viejos.

El servidor también tiene caché, pero no puede mentir

- El cliente escribe `a`, `b`, `c` sobre un archivo que tenía `x`, `y`, `z`.
- El servidor responde OK a `b` antes de llevarlo al disco, y se cae…
- …el cliente no reintenta `b`. Al volver, el archivo dice `a`, `y`, `c`.



Regla

El servidor lleva cada `WRITE` a almacenamiento estable *antes* de responder. Por eso las escrituras en NFS son lentas, y los servidores usan memoria con batería o sistemas de archivos que escriben rápido, como WAFL.

SECCIÓN 06

AFS

06

AFS: diseñado para escalar

AFSv1 (Carnegie Mellon, años 80)

Caché de archivos completos en el *disco local*: al abrir, se trae el archivo entero; al cerrar, si cambió, se devuelve. Al reabrir, pregunta "¿cambió?" (`TestAuth`).

Un servidor aguantaba unos 20 clientes.

AFSv2

Callbacks: el servidor promete *avisar* si el archivo cambia. Ya no hay que preguntar.

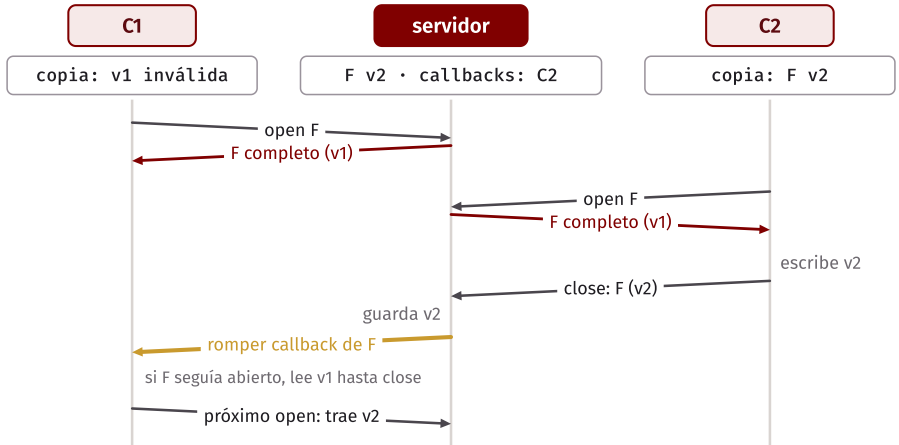
FID en vez de rutas: el cliente recorre los directorios.

Un servidor pasó a atender unos 50 clientes.

El cambio clave

De *preguntar* a ser *avisado*. Pero el servidor ahora tiene estado: sabe qué clientes tienen cada archivo.

Un callback se rompe



Consistencia: gana el último en cerrar

- Lo que escribe un cliente no se ve en otras máquinas hasta que cierra el archivo. En la misma máquina se ve al instante.
- Si dos clientes modifican el mismo archivo a la vez, gana el **último en cerrar** *(last writer wins)*: el archivo queda entero de uno o del otro.
- Si el servidor se reinicia, olvida los callbacks: los clientes deben tratar su caché como sospechosa y revalidarla.

	NFSv2	**AFSv2**
Caché	bloques, en memoria	archivos completos, en disco
¿Cambió?	`GETATTR` (+3 s)	callback
Visibilidad	al cerrar	al cerrar
Escrituras a la vez	bloques mezclados	gana el último
Servidor	sin estado	con estado
Tras una caída	reintentar	revalidar cachés

Resumen

- **LFS** junta las escrituras en segmentos grandes al final del log. Como los inodos se mueven, necesita un **imap** (y una CR fija que lo apunta) y un **limpiador** que usa el resumen del segmento para saber qué está vivo.
- Una **SSD** no puede sobrescribir: borra por bloques y programa por páginas. Su **FTL** es un LFS: tabla de mapeo, GC y nivelación de desgaste. Cuanto más llena, mayor la **amplificación de escritura**.
- Los discos fallan a medias: **LSE** (el disco avisa) y **corrupción silenciosa** (no avisa). Checksums con ID físico y **scrubbing**; las escrituras perdidas piden el checksum en otro lugar.
- Sobre una red que pierde mensajes: **timeouts**, **reintentos** y **secuencias**; RPC los esconde detrás de una llamada.
- **NFS**: servidor sin estado y operaciones **idempotentes**; caché con flush-on-close y `GETATTR`. **AFS**: archivos completos, **callbacks**, gana el último en cerrar.

Para la próxima clase

- **Laboratorio**: simuladores de OSTEP.
  - `lfs.py`: preguntas 1, 2, 5 y 6 del cap. 43.
  - `ssd.py`: preguntas 1, 2, 4 y 5 del cap. 44.
  - `checksum.py`: preguntas 1, 3 y 7 del cap. 45.
  - `afs.py`: preguntas 1–4 del cap. 50.
  - Cap. 48: cliente/servidor UDP con reintentos (preguntas 1–3).
- Semana 16, seguridad: OSTEP cap. 53–55; sesión B, cap. 56–57.

Entregas

P5 (archivos grandes y enlaces simbólicos en xv6) y el bono `cow` se entregan la semana 16. El examen final es en el período de exámenes.

Preguntas para pensar

¿Por qué LFS necesita un imap si FFS no? ¿Qué es la amplificación de escritura y por qué la empeora un disco casi lleno? ¿Por qué hacer idempotentes las operaciones de NFS simplifica la recuperación ante fallos?